Techniques algorithmiques pour l'extraction de formules minimales inconsistantes

THÈSE

soutenue le 21 novembre 2007

pour l'obtention du

Doctorat de l'Université d'Artois

Spécialité Informatique

par

Cédric Piette

Composition du jury

Rapporteurs :	Thomas Schiex (INRA Toulouse)	
	Gérard Verfaillie (Office National d'Études et de Recherches Aérospatiales)	
Examinateurs :	Éric Grégoire (Université d'Artois)	*directeur de thèse*
	Youssef Hamadi (Microsoft Research Cambridge)	
	Pierre Marquis (Université d'Artois)	
	Bertrand Mazure (Université d'Artois)	*co-directeur de thèse*
	Lakhdar Sais (Université d'Artois)	
	Laurent Simon (INRIA Futurs Orsay)	

Centre de Recherche en Informatique de Lens – CNRS FRE 2499
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.fr

Sans contrainte, on tourne en rond.

[Mathieu Chedid]

Remerciements

Mes premiers remerciements sont adressés aux personnes qui m'ont fait la joie et l'honneur de faire partie du jury, que je n'osais espérer d'aussi grande qualité. Messieurs Thomas Schiex et Gérard Verfaillie, rapporteurs de ce mémoire, pour les conseils et remarques éclairés qu'ils m'ont prodigués et qui ont permis l'amélioration de ce manuscrit. Lakhdar Saïs pour avoir suivi de près mes travaux sans être mon encadrant officiel, et pour les très nombreuses références à des articles anciens dont lui seul a le souvenir. Pierre Marquis, pour sa patience face aux innombrables questions dont j'ai pu l'assommer. Youssef Hamadi, qui avec Lucas Bordeaux m'a chaleureusement accueilli dans le laboratoire de Microsoft à Cambridge, durant le séjour que j'ai eu la chance de faire en Angleterre. Laurent Simon, pour l'intérêt qu'il porte à mes travaux, et pour la carte de nouvel an dont il est l'instigateur.

Il n'est pas dans mon intention de mentionner ici mes encadrants pour la forme. Je leur adresse au contraire mes plus sincères remerciements pour m'avoir initié à cette passionnante thématique, pour cette traversée de la thèse qu'ils ont rendue croisière, et pour cette liberté octroyée avec (non, ce n'est pas paradoxal) une disponibilité à toute épreuve. Éric et Bertrand, merci, merci et merci.

De manière générale, merci à tous les membres du CRIL à qui je tiens à signifier l'immense plaisir que j'ai pris à travailler avec eux. Merveilleuse équipe que celle de ce laboratoire dans lequel j'ai eu l'honneur d'officier ces quelques années. Une mention spéciale à tous les doctorants pour la constante bonne ambiance qui règne dans nos bureaux. Évidemment, comment oublier les membres du CRIL « MIT » et nos débats de pause café ? Merci à vous tous.

Merci à mes parents de m'avoir doté de ces initiales annonciatrices. Plus précisément, merci à ma mère pour son indécrottable soutien de toutes sortes (impossible à détailler, il faudrait y consacrer une thèse) et à mon père de m'avoir initié à la programmation à l'âge de 8 ans. Il est vrai que ça manque cruellement au programme de l'école primaire. Merci également à mon frère, à ma grand-mère botaniste à qui je n'ai jamais osé avouer que mon travail consiste en partie à élaguer des arbres, et à mon grand-père-partenaire-qui-n-avait-pas-les-pieds-plats. Soutien plus grand eût été difficile.

Un merci déformé de démesure à Julie pour sa patience et son soutien inaltérables, pour sa curiosité et sa persévérance à (essayer de) comprendre mon thème de recherche, et globalement pour ces dernières années assez réussies.

Je tiens également à remercier mes amis et ex-collègues du *bibinôme*, pour m'avoir encouragé à poursuivre cette thèse il y a quelques années, alors que les contours obscurs de la NP-complétude m'étaient encore inconnus.

Merci à Chdem le filstool-ien pour la promesse tenue qui signifie beaucoup, à François pour les vendredi *philo* de la « bonne époque », à Dejan pour cette folle nuit de malchance qui a nous conduit à Gravelines (ah, cet improbable hurluberlu que je revois nous tendre des bouteilles vides), aux carvinnois pour toutes les bonnes soirées à la BA et ailleurs, et au bloc 51 pour les souvenirs immuables.

Enfin, merci à Django « Mini » Reinhardt, qui sans jamais faiblir m'accompagne dans mes longues nuits de travail ...

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Les problèmes de satisfaisabilité d'une formule propositionnelle (SAT) et de satisfaction de contraintes (CSP) sont des domaines de recherche majeurs en intelligence artificielle. Leur appartenance à la classe de complexité NP-complet les rend calculatoirement équivalents à de nombreux problèmes dont on recherche encore aujourd'hui des algorithmes efficaces, comme par exemple en théorie des graphes, au problème du parcours hamiltonien, de la clique maximum, ou du coloriage de graphes pour ne citer qu'eux. En outre, ces problèmes possèdent d'autres applications telles que la validation de circuits, la résolution de puzzles logiques, ou la recherche de clés cryptographiques, etc. On comprend alors la motivation d'une grande communauté de chercheurs pour ces problèmes.

Si de nombreuses raisons pratiques expliquent l'intérêt de ces problèmes, ils représentent également un enjeu important de la théorie de la complexité. En effet, l'extrême simplicité de leur formalisme en fait des instances idéales des problèmes non-déterministes polynomiaux. La résolution de ces problèmes par un algorithme déterministe en un temps polynomial ou la preuve de son impossibilité aurait pour corollaire une réponse à une question centrale de la théorie de la complexité. Ainsi, au-delà des intérêts pratiques, ces problèmes sont intrinsèquement une ressource pour la discipline informatique.

Pour toutes ces raisons, on constate depuis quelques années un important regain d'intérêt pour ces problèmes. En particulier, de nombreuses approches sont proposées dans le but d'élargir la classe des instances traitables en pratique. Ces améliorations permettent aujourd'hui de résoudre certains problèmes possédant des milliers, voire des millions de variables. Ainsi, les progrès algorithmiques considérables de la programmation par contraintes (PPC) rendent solubles en quelques secondes des problèmes considérés hors de portée il y a peu.

Une conséquence naturelle de ces progrès incessants est la modélisation de problèmes de taille de plus en plus importante. Or, quand des formules de grande taille censées être satisfaisables sont prouvées incohérentes, il peut être extrêmement difficile de déterminer la cause du problème, dans l'espoir de le réparer. En effet, toutes les contraintes d'un système ne sont pas nécessairement à l'origine de son incohérence, seules quelques-unes d'entre elles peuvent être conflictuelles et rendre l'ensemble sans solution. Cependant, contrairement aux problèmes cohérents pour lesquels un modèle est généralement produit, une réponse négative au test de la consistance d'une formule n'apporte que peu d'informations. Il semble intéressant de déterminer *pourquoi* un problème est sans solution ; précisément, dans ce cas, on cherche à effectuer l'extraction des plus petits ensembles de contraintes contradictoires. La connaissance de tels ensembles donne à l'utilisateur de nombreuses informations sur le problème et permet d'en restaurer la consistance avec un éventuel ordre préférentiel sur les contraintes. Il s'agit donc d'un diagnostic assisté par ordinateur de l'insatisfaisabilité. Malheureusement, il s'agit d'un problème d'une complexité extrêmement élevée dans le pire des cas, pour lequel les approches complètes et exactes sont souvent inapplicables en pratique. Il semble alors naturel de se diriger vers des méthodes heuristiques.

Les travaux présentés dans cette thèse ont donc pour objectif l'extraction de systèmes minimalement incohérents, dans divers formalismes. Plus précisément, notre attention s'est portée sur le développement de techniques algorithmiques capables de circonscrire une ou plusieurs causes d'incohérence en pratique, pour des problèmes issus d'applications réelles.

Ce document est composé de trois parties. Tout d'abord, certaines notions essentielles à la compréhension de ce manuscrit sont énoncées : les définitions usuelles de la logique propositionnelle, les bases de la théorie de la complexité, ainsi que la définition formelle du problème SAT et de quelques unes de ses variantes sont présentées. Un état de l'art concernant les meilleures méthodes connues pour l'extraction de formules booléennes minimalement inconsistantes (MUS) dans le cadre de la logique propositionnelle est également établi.

La deuxième partie est consacrée à notre contribution au cadre clausal booléen. Celle-ci est triple : en effet, dans un premier temps, nous présentons un nouveau concept permettant de prendre en compte un voisinage partiel des interprétations parcourues par une recherche locale, dans le but de faire l'approximation voire le calcul exact d'un MUS. Ces travaux ont abouti au développement d'un algorithme, baptisé `(A)OMUS`, qui est présenté et comparé expérimentalement aux meilleures méthodes connues. Une variante constructive est également étudiée d'un point de vue théorique puis évaluée à son tour empiriquement. Ensuite, nous montrons comment l'hybridation d'une méthode incomplète à la meilleure approche connue pour le calcul exhaustif de tous les MUS peut permettre à cette dernière d'éviter des appels à des oracles NP et CoNP, lui faisant gagner un ordre de grandeur en temps de calcul. Cependant, même avec cette amélioration, le nombre de MUS au sein d'une CNF (exponentiel dans le pire cas) peut rendre ce calcul irréalisable en pratique. Aussi, plusieurs méthodes efficaces permettant l'approximation de cet ensemble sont introduites. Cette partie se termine par une mise en application des techniques présentées à la gestion de l'incohérence des bases de connaissances stratifiées.

Enfin, la troisième partie porte sur le problème de satisfaction de contraintes (CSP). Dans ce formalisme également, de nombreuses approches ont été proposées par le passé pour le calcul d'un problème minimalement insatisfaisable (MUC). Dans une première contribution, nous revisitons le meilleur algorithme connu et proposons des améliorations qui accélèrent son calcul, et étendent en conséquence la classe des problèmes traitables. Ensuite, un nouveau concept qui raffine celui de MUC, défini classiquement par la communauté, est présenté. Celui-ci permet de prendre en considération les tuples du problème et non ses contraintes. La viabilité de ce concept est démontrée à travers des propriétés qui démontrent que le procédé mis au point permet de *réparer* un problème sans retrait de contraintes. Des résultats expérimentaux confirment l'intérêt de l'approche, d'un point de vue pratique.

Première partie

État de l'art

Chapitre 1

Logique propositionnelle : définitions, propriétés et formes normales

Sommaire

1.1 Définitions préliminaires

Grâce au langage, l'homme est capable de décrire des choses, d'exprimer des ordres, des souhaits, des émotions. Il peut également obtenir des énoncés à partir d'autres énoncés : ce procédé est appelé **raisonnement**.

La formalisation d'un tel procédé conduit à la naissance de logiques mathématiques. Parmi elles, la **logique classique des propositions** en est le fragment le plus simple. C'est cette logique qui fait l'objet de ce chapitre.

1.1.1 Syntaxe

Définition 1.1 (atome)
Un atome, également appelé **variable propositionnelle**, *est une variable booléenne utilisée pour représenter des propriétés de base. Il ne peut prendre que deux valeurs de vérités possibles :* vrai *ou* faux *(également notées par 1 ou 0, respectivement).*

Définition 1.2 (connecteur logique)
Un connecteur logique est un opérateur permettant de combiner une ou plusieurs variables propositionnelles pour en faire un énoncé complexe. Il est généralement muni d'un entier positif, son **arité**.

Définition 1.3 (formule propositionnelle)
Une formule propositionnelle est définie inductivement comme le plus petit ensemble pour l'inclusion tel que :

- *Toute variable propositionnelle α est une formule propositionnelle.*
- *Soient $\Sigma_1, \Sigma_2, ..., \Sigma_n$ des formules propositionnelles et μ un connecteur logique d'arité n. Alors $\mu(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ est une formule propositionnelle.*

Note 1.1
On note *Var(*Σ*)* l'ensemble des variables propositionnelles qui apparaissent dans une formule propositionnelle Σ.

Remarque 1.1
Dans la suite, nous considérons les connecteurs logiques suivants :

- les symboles *vrai* et *faux* d'arité 0 ;
- la négation ($\neg$) d'arité 1 ;
- la conjonction ($\wedge$), la disjonction ($\vee$), l'implication matérielle ($\Rightarrow$) et l'équivalence ($\Leftrightarrow$) d'arité 2.

Définition 1.4 (formules indépendantes)
On désigne par formules indépendantes tout ensemble de formules qui n'ont aucun atome en commun. Autrement dit, $\Sigma_1, \Sigma_2, ..., \Sigma_n$ sont des formules indépendantes si et seulement si

$$\forall i, j \text{ tel que } 1 \leq i < j \leq n, \ \ Var(\Sigma_i) \cap Var(\Sigma_j) = \emptyset$$

Définition 1.5 (taille d'une formule)
La taille d'une formule Σ, notée $|\Sigma|$, est le nombre d'occurrences de symboles (connecteurs et symboles propositionnels) utilisés pour l'écrire.

Définition 1.6 (littéral)
On désigne par littéral tout atome l ou sa négation $\neg l$. l est appelé littéral positif et $\neg l$ littéral négatif. On appelle **littéral complémentaire** *de l la négation de l.*

Définition 1.7 (clause)
On appelle clause toute disjonction finie de littéraux.

Remarque 1.2
Une clause est dite unaire si elle contient un seul littéral. De la même manière, elle est dite binaire si elle contient deux littéraux, ternaire si elle en contient 3, et n-aire si elle contient n littéraux. Une clause de longueur 0 est dite **vide**. Elle est généralement notée $\bot$.

Définition 1.8 (cube)
Un cube est une conjonction finie de littéraux. Ces conjonctions sont également appelées **termes** *ou* **monômes**.

Remarque 1.3
Les clauses et les cubes peuvent être représentés comme des ensembles de littéraux. On autorise donc l'utilisation des symboles ensemblistes sur les littéraux par rapport aux clauses et aux cubes, par abus de notation.

Définition 1.9 (subsomption)
On dit qu'une clause α est sous-sommée (ou subsumée) par une clause β si et seulement si $\beta \subseteq \alpha$.

Exemple 1.1
$c_1 = a \vee \neg b \vee \neg c \vee d$ est sous-sommée par $c_2 = a \vee \neg c$.

Définition 1.10 (clause tautologique)
On désigne par clause tautologique toute clause qui contient un littéral a et son complémentaire $\neg a$. La clause $\top$, symbole du vrai, est également une clause tautologique.

Définition 1.11 (clause fondamentale)
On nomme clause fondamentale toute clause non tautologique.

Définition 1.12 (résolution)
Soient c_1 et c_2 deux clauses fondamentales ayant au moins une variable v en commun, présente dans ces 2 clauses sous la forme de littéraux complémentaires. On appelle **résolvante** *issue de c_1 et c_2 la disjonction des littéraux présents dans ces clauses et à laquelle on ôte v et $\neg v$.*
Ce procédé est appelé **résolution**.

Définition 1.13 (longueur d'une clause)
On appelle longueur d'une clause c le nombre de littéraux distincts qui apparaissent dans c. On note cette longueur $l(c)$.

Exemple 1.2
$c_1 = a \vee \neg b \vee \neg c \vee d$
$c_2 = c \vee d \vee \neg e \vee c$
$l(c_1) = 4\,;\, l(c_2) = 3$

Définition 1.14 (clause positive, négative, mixte)
On nomme par clause positive (respectivement négative) toute clause qui ne contient aucun littéral négatif (respectivement positif). Une clause qui contient à la fois des littéraux positifs et négatifs est appelée clause mixte.

Définition 1.15 (clause de Horn)
Une clause de Horn est une clause qui contient au plus un littéral positif. Elle est définie comme strictement de Horn si et seulement si elle contient exactement un littéral positif.

Définition 1.16 (CNF)
*Une formule Σ est sous forme CNF (*conjunctive normal form*) si et seulement si Σ est une conjonction de disjonctions de littéraux :*

$$\bigwedge_{i=1}^{nbClauses} \left(\bigvee_{j_i=1}^{l(c_i)} a_{j_i} \right)$$

Définition 1.17 (DNF)
*Une formule Σ est sous forme DNF (*disjunctive normal form*) si et seulement si Σ est une disjonction de conjonctions de littéraux :*

$$\bigvee_{i=1}^{nbCubes} \left(\bigwedge_{j_i=1}^{l(c_i)} a_{j_i} \right)$$

$\omega(\Sigma)$	$\omega(\Gamma)$	$\omega(\neg\Sigma)$	$\omega(\Sigma \wedge \Gamma)$	$\omega(\Sigma \vee \Gamma)$	$\omega(\Sigma \Rightarrow \Gamma)$	$\omega(\Sigma \Leftrightarrow \Gamma)$	$\omega(\Sigma \oplus \Gamma)$
faux	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*	*faux*
faux	*vrai*	*vrai*	*faux*	*vrai*	*vrai*	*faux*	*vrai*
vrai	*faux*	*faux*	*faux*	*vrai*	*faux*	*faux*	*vrai*
vrai	*vrai*	*faux*	*vrai*	*vrai*	*vrai*	*vrai*	*faux*

TAB. 1.1 – Table de vérité d'une formule

Propriété 1.1
Toute formule propositionnelle peut être mise sous forme CNF ou DNF. Cependant, la transformation peut nécessiter une croissance exponentionnelle en la taille de la formule initiale.

Remarque 1.4
De la même manière que les clauses et les cubes peuvent être représentés comme des ensembles de littéraux, les formules CNF et DNF sont parfois notées comme des ensembles de clauses et de cubes, respectivement.

Définition 1.18 (littéral pur)
Un littéral l est dit pur pour une formule Σ sous forme CNF si et seulement si il apparaît dans Σ exclusivement positivement ou exclusivement négativement.

1.1.2 Sémantique

Définition 1.19 (interprétation)
En logique propositionnelle, une interprétation ω est une application de l'ensemble V *des variables propositionnelles vers l'ensemble des valeurs de vérité {vrai,faux}.*

$$\omega : \mathrm{V} \rightarrow \{vrai,faux\}$$

La sémantique des connecteurs logiques usuels est définie dans la table 1.1. Il existe d'autres connecteurs logiques (tel que le `NAND` de Sheffer), mais ceux présentés ici sont fonctionnellement complets du point de vue du pouvoir d'expression.

Note 1.2
La sémantique d'une formule Σ dans l'interprétation ω est définie de manière inductive. On dit que les connecteurs sont **vérifonctionnels**. La valeur de vérité donnée à une formule Σ par une interprétation ω est notée $\omega(\Sigma)$.

Note 1.3
Pour une formule Σ, il existe $2^{|Var(\Sigma)|}$ interprétations différentes.

Définition 1.20 (distance de Hamming)
Soient ω et ω' deux interprétations d'une formule Σ. On note $\delta_{H(\omega,\omega')} = \{x \mid x \in Var(\Sigma)$ et $\omega(x) \neq \omega'(x)\}$. La distance de Hamming entre deux interprétations ω et ω', notée $d_H(\omega,\omega')$, est alors définie par $|\delta_{H(\omega,\omega')}|$.

Exemple 1.3
Soient $\Sigma = a \vee (b \wedge c),\ \omega = \{a, \neg b, \neg c\},\ \omega' = \{\neg a, b, \neg c\}$
On a alors $d_H(\omega, \omega') = 2$.

Remarque 1.5
Il est intéressant de remarquer que la notion de distance entre interprétations définie par Hamming est une véritable distance mathématique, c'est-à-dire qu'elle vérifie les propriétés suivantes :

- symétrie
$$\forall \omega, \omega',\ d_H(\omega, \omega') = d_H(\omega', \omega)$$
- séparation
$$\forall \omega, \omega',\ d_H(\omega, \omega') = 0 \Leftrightarrow \omega = \omega'$$
- inégalité triangulaire
$$\forall \omega, \omega', \omega'',\ d_H(\omega, \omega'') \leq d_H(\omega, \omega') + d_H(\omega', \omega'')$$

Définition 1.21 (interprétation complémentaire)
ω' est l'interprétation complémentaire de ω si et seulement si $\forall \alpha,\ \omega(\alpha) = \neg\omega'(\alpha)$, autrement dit si $d_H(\omega, \omega') = |\omega|$.

Définition 1.22 (interprétation partielle, incomplète et complète)
Soit Σ une formule propositionnelle. On désigne par :

- *interprétation partielle toute interprétation ω telle que $|\omega| \leq |Var(\Sigma)|$*
- *interprétation incomplète toute interprétation ω telle que $|\omega| < |Var(\Sigma)|$*
- *interprétation complète toute interprétation ω telle que $|\omega| = |Var(\Sigma)|$*

Remarque 1.6
Quand aucune précision n'est apportée, lorsqu'on parle d'interprétation, on désigne implicitement une interprétation complète.

Définition 1.23 (formules équivalentes)
On dit que deux formules sont équivalentes (noté $\equiv$) si et seulement si elles prennent la même valeur de vérité pour toutes les interprétations. En d'autres termes, Σ et Γ sont équivalentes si et seulement si $\forall \omega$, $\omega(\Sigma) = \omega(\Gamma)$.

Note 1.4
On peut déduire de la table 1.1 l'équivalence des formules suivantes :

- $\Sigma \Rightarrow \Gamma$ est équivalent à $\neg\Sigma \vee \Gamma$
- $\Sigma \Leftrightarrow \Gamma$ est équivalent à $(\Sigma \Rightarrow \Gamma) \wedge (\Gamma \Rightarrow \Sigma)$ et à $(\neg\Sigma \vee \Gamma) \wedge (\neg\Gamma \vee \Sigma)$
- $\Sigma \oplus \Gamma$ est équivalent à $\neg(\Sigma \Leftrightarrow \Gamma)$
- $\neg(\Sigma \vee \Gamma)$ est équivalent à $\neg\Sigma \wedge \neg\Gamma$
- $\neg(\Sigma \wedge \Gamma)$ est équivalent à $\neg\Sigma \vee \neg\Gamma$

Note 1.5
L'équivalence permet la manipulation syntaxique des formules tout en préservant la sémantique. Ainsi, il existe de nombreuses règles de réécriture sur les formules : soient Σ, Γ et Ψ trois formules propositionnelles.

- la double négation :
$$\neg(\neg\Sigma) \equiv \Sigma$$

– l'associativité :

$$(\Sigma \vee \Gamma) \vee \Psi \equiv \Sigma \vee (\Gamma \vee \Psi)$$
$$(\Sigma \wedge \Gamma) \wedge \Psi \equiv \Sigma \wedge (\Gamma \wedge \Psi)$$

– la commutativité :

$$\Sigma \vee \Gamma \equiv \Gamma \vee \Sigma$$
$$\Sigma \wedge \Gamma \equiv \Gamma \wedge \Sigma$$

– l'idempotence :

$$\Sigma \vee \Sigma \equiv \Sigma$$
$$\Sigma \wedge \Sigma \equiv \Sigma$$

– la distributivité :

$$\Sigma \vee (\Gamma \wedge \Psi) \equiv (\Sigma \vee \Gamma) \wedge (\Sigma \vee \Psi)$$
$$\Sigma \wedge (\Gamma \vee \Psi) \equiv (\Sigma \wedge \Gamma) \vee (\Sigma \wedge \Psi)$$

Définition 1.24 (modèle)
Soient Σ une formule et ω une interprétation. On dit que ω est un modèle de Σ si $\omega(\Sigma) = vrai$. On dit aussi que ω satisfait Σ (noté $\omega \models \Sigma$). Toute interprétation qui n'est pas un modèle de Σ est appelée **contre-modèle** *de Σ.*

Définition 1.25 ((in)satisfaisabilité)
On dit qu'une formule Σ est **satisfaisable** *(ou* **consistante***) (ou* **cohérente***) si et seulement si il existe une interprétation ω telle que $\omega(\Sigma) = vrai$. Si Σ n'admet aucun modèle, elle est alors dite* **insatisfaisable** *(ou* **inconsistante***) (ou* **incohérente***).*

Définition 1.26 (validité)
On désigne par formule valide toute formule Σ telle que $\forall \omega, \omega(\Sigma) = vrai$. Une telle formule est également appelée **tautologie***. Une formule valide est notée $\models \Sigma$.*

Note 1.6
Si Σ est une formule valide, alors $\neg\Sigma$ est incohérent.

Définition 1.27 (conséquence logique)
Une formule Σ est une conséquence logique de Γ si et seulement si tout modèle de Γ est un modèle de Σ. On note cela $\Gamma \models \Sigma$.

Note 1.7
En notant $M(\Sigma)$ l'ensemble des modèles de Σ, on a : si Σ est une conséquence logique de Γ, alors $\forall m \in M(\Gamma),\ m \models \Sigma$.

Note 1.8
Si $\Sigma \equiv \Gamma$, alors ($\Sigma \models \Gamma$ et $\Gamma \models \Sigma$).

Propriété 1.2
Soient deux formules propositionnelles Σ et Γ. On a $\Sigma \models \Gamma$ si et seulement si la formule $\Sigma \wedge \neg\Gamma$ est inconsistante.

Définition 1.28 (impliqué (premier))
Soient Σ une formule CNF et c une clause. c est un impliqué *de Σ si et seulement si $\Sigma \models c$. De plus, si aucun sous-ensemble de c n'est un impliqué de Σ, alors c est un* impliqué premier *de Σ.*

Définition 1.29 (impliquant (premier))
Soient Σ une formule CNF et m un monôme. m est un impliquant *de Σ si et seulement si $m \models \Sigma$. De plus, si aucun sous-ensemble de m n'est un impliquant de Σ, alors m est un* impliquant premier *de Σ.*

Définition 1.30 (formules équi-satisfaisables)
Deux formules Σ et Γ sont équi-satisfaisables si et seulement si :

$$(\exists\ \omega_1 \text{ tel que } \omega_1 \models \Sigma) \Leftrightarrow (\exists\ \omega_2 \text{ tel que } \omega_2 \models \Gamma)$$

Propriété 1.3
Deux formules équivalentes sont équi-satisfaisables. La réciproque n'est pas vérifiée.

Exemple 1.4
Soient Σ et Γ deux formules et l un littéral. Les formules $(\Sigma \vee \Gamma)$ et $((\Sigma \vee \neg l) \wedge (\Gamma \vee l))$ sont équi-satisfaisables mais pas équivalentes.

Propriété 1.4
Soient Σ, Γ et Ψ trois formules propositionnelles. On a alors les propriétés suivantes :
- *réflexivité :* $\Sigma \models \Sigma$
- *équivalence à gauche : si* $(\Sigma \equiv \Gamma$ *et* $\Sigma \models \Psi)$*, alors* $\Gamma \models \Psi$
- *transitivité : si* $(\Sigma \models \Gamma$ *et* $\Gamma \models \Psi)$*, alors* $\Sigma \models \Psi$
- *coupure : si* $(\Sigma \wedge \Gamma \models \Psi$ *et* $\Sigma \models \Gamma)$ *alors* $\Sigma \models \Psi$
- *monotonie : si* $\Sigma \models \Gamma$ *alors* $\Sigma \wedge \Psi \models \Gamma$

Définition 1.31 (clause satisfaite, falsifiée, active)
Soit Σ une formule CNF, c une clause de Σ et ω une interprétation partielle de Σ. On dit que :
- *c est* satisfaite *par rapport à ω si et seulement si* $(\exists\ l \in c$ *tel que* $\omega(l) = vrai)$ *ou* $(\exists\ \neg l \in c$ *tel que* $\omega(l) = faux)$
- *c est* falsifiée *par rapport à ω si et seulement si* $(\forall\ l \in c$ *, on a* $\omega(l) = faux)$ *et* $(\forall\ \neg l \in c$ *tel que* $\omega(l) = vrai)$
- *si c n'est ni satisfaite ni falsifiée par rapport à ω, elle est dite* active.

1.2 Complexité

La théorie de la complexité est un des fondements majeurs de l'informatique. Son objectif est de discriminer différents problèmes en fonction de leur « difficulté ». Celle-ci est estimée à partir de la consommation en ressources informatiques requises à la résolution du problème.

Il existe **deux grands types de ressources** : **le temps de calcul** et **la mémoire consommée**. Elles permettent de définir respectivement la complexité temporelle et spatiale. De manière générale, on considère le temps de calcul comme la ressource la plus significative pour un algorithme, la mémoire consommée lui étant très liée. Ainsi, quand on parle de complexité, sauf mention contraire, il sera admis que c'est la complexité temporelle qui est désignée.

1.2.1 Notions de base

Définition 1.32 (complexité d'un algorithme)
Soit n la taille de la donnée en entrée. On dit qu'un algorithme a une complexité en $\mathcal{O}(P(n))$ si à partir d'un certain rang n_0, le nombre d'opérations élémentaires (noté f(n)) nécessaires à sa réalisation est majoré par un multiple de P(n).

Formellement, un algorithme K est en $\mathcal{O}(P(n))$ si

$$\exists\alpha\ \exists n_0\ \ \forall n > n_0\ , f_K(n)\ \leq\ \alpha \times P(n)$$

Définition 1.33 (algorithme polynomial)
Un algorithme est dit polynomial si dans le pire des cas, le nombre d'opérations nécessaires à sa terminaison est majoré par un polynôme en la taille n de l'entrée. Clairement, un algorithme K est polynomial s'il existe un entier i tel que K soit en $\mathcal{O}(n^i)$.

Remarque 1.7
Soit P un polynôme et $deg(P)$ le degré maximal de P. On dit qu'un algorithme en $\mathcal{O}(P(n))$ est d'ordre $deg(P)$. Ceci est justifié par le fait que le comportement asymptotique de P ne dépend que de son terme de plus haut degré.

Définition 1.34 (algorithme exponentiel)
Un algorithme est simplement exponentiel *si, en fonction de la taille de son entrée notée n, sa complexité peut s'écrire $\mathcal{O}(e^{P(n)})$ où e est un réel strictement supérieur à 1 et $P(n)$ est un polynôme en n.*

Remarque 1.8
Dans la suite de document, on utilisera le terme *exponentiel* plutôt que *simplement exponentiel* par soucis de concision.

Exemple 1.5
$f_S(n) = n!$
$f_T(n) = 2^n$
Si f_S et f_T sont deux fonctions représentant le nombre d'opérations de deux algorithmes S et T en fonction de la taille de leur entrée n, alors ces deux algorithmes sont exponentiels.

Définition 1.35 (complexité d'un problème)
La complexité d'un problème est la complexité dans le pire des cas du meilleur algorithme pour résoudre le problème.

Par exemple, la complexité du tri interne de données est en $\mathcal{O}(n \times log(n))$ bien que la plupart des algorithmes de tri (comme le tri à bulle, le tri sélection ou le tri insertion) soient d'ordre 2 (ou **quadratiques**). Cependant, il existe des algorithmes de tri en $\mathcal{O}(n \times log(n))$ tel que le tri fusion et la complexité du problème de tri interne est donc en $\mathcal{O}(n \times log(n))$.

Définition 1.36 (problème de décision)
Un problème de décision est un problème qui ne peut avoir que deux solutions possibles : oui *ou* non, vrai *ou* faux, 0 *ou* 1, *etc.*

Exemple 1.6
Soit E un ensemble. Déterminer si $x \in E$ est un problème de décision. En effet, soit $x \in E$, auquel cas la réponse est *oui*, soit $x \notin E$ et la réponse est *non.*

Définition 1.37 (problème complémentaire)
Soit Ω un problème de décision. On appelle problème complémentaire à Ω le problème qui pour toute entrée x, retourne vrai si et seulement si $\Omega(x)$=faux.

1.2.2 Machine de Turing

Une machine de Turing est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur et sa mémoire, créé en vue de donner une définition au concept d'algorithme ou de « procédure mécanique ». Ce modèle, essentiel en informatique théorique, est utilisé pour résoudre les problèmes de complexité algorithmique et de calculabilité.

Définition 1.38 (machine de Turing)
Une machine de Turing [Turing 1937] est un dispositif de calcul constitué de :

- *un ruban de longueur infinie discrétisé en cases, chacune d'entre elles pouvant contenir un symbole appartenant à un alphabet fini Σ ou être éventuellement vide ($\sqcup$) ;*
- *une tête de lecture/écriture pouvant se déplacer d'une case à la fois : vers la gauche (G), la droite (D), ou pas de déplacement ($\downarrow$) ;*
- *un registre d'état mémorisant l'état courant de la machine de Turing. L'ensemble d'états possibles E est toujours fini et contient entre autre un état initial $s \in E$ et un ensemble d'états d'arrêt $H \subset E$, ceux-ci pouvant éventuellement être munis d'une sémantique*
- *une table de transitions δ décrivant l'exécution du programme sur la machine. Cette table indique, en fonction de l'état courant de la machine et du caractère lu par la tête de lecture sur le case courante, comment déplacer la tête de lecture/écriture (une case vers la gauche, une case vers la droite ou pas de déplacement) dans quel état se situe la machine et quel symbole écrire sur le ruban. Formellement, on a :*

$$\delta : E \backslash H \times \Sigma \rightarrow (E \times \Sigma \times \{G, D, \downarrow\})$$

On effectue un calcul sur la machine de Turing en plaçant la donnée d'entrée sur le ruban, la tête de lecture/écriture pointant sur la première case contenant la donnée. Le programme se déroule en suivant la table de transition et ne se termine que lorsque l'exécution conduit à un état d'arrêt. Le résultat peut alors être connu en consultant l'état d'arrêt dans lequel l'exécution s'est arrêtée (ces états peuvent être munis d'une sémantique) et en lisant le ruban.

La thèse Church-Turing postule que tout problème de calcul basé sur une procédure algorithmique peut être résolu par une machine de Turing. Cette thèse n'est pas un énoncé mathématique, puisqu'elle ne suppose pas une définition précise de procédure algorithmique. En revanche, il est possible de définir une notion de « système acceptable de programmation » et de démontrer que le pouvoir de tels systèmes est équivalent à celui des machines de Turing : on parle alors de *Turing complétude*.

On voit ici l'intérêt qu'apporte la machine de Turing : un outil mathématique simple permettant la simulation de toutes les fonctions calculables par un algorithme, bien que nos ordinateurs actuels soient munis d'une mémoire à accès aléatoire. Ce modèle fait en outre abtraction de langage de programmation et se révèle un outil essentiel en informatique théorique, en particulier pour résoudre les problèmes de complexité algorithmique et de calculabilité.

Définition 1.39 (machine de Turing déterministe/non déterministe)
Une machine de Turing est dite déterministe si sa table de transition est composée de transitions déterministes, c'est-à-dire si c'est une application. A l'inverse, si la table de transitions possède au moins une transition permettant d'atteindre deux états différents, la machine de Turing est dite non déterministe.

Définition 1.40 (machine de Turing à oracle)
Le dispositif de calcul d'une machine de Turing peut être muni d'un élément qui lui permet de décider si un élément x appartient à un langage L en une étape de calcul. Une telle machine est appelée machine de Turing à oracle pour le langage L.

1.2.3 Classes de complexité

Définition 1.41 (classe P)
Un problème appartient à la classe P s'il peut être décidé par une machine de Turing déterministe dont la complexité temporelle est majorée par une fonction polynomiale.

Définition 1.42 (classe NP)
Un problème appartient à la classe NP s'il peut être codé par une machine de Turing non déterministe dont la complexité temporelle est majorée par une fonction polynomiale.

Cette classe peut paraître à première vue « aussi difficile » que la classe P, mais la difficulté est ici dissimulée derrière le non-déterminisme de la machine de Turing. Pour l'heure, on ne sait construire que des ordinateurs déterministes et simuler le fonctionnement d'une machine non déterministe sur une machine déterministe a été prouvé de complexité exponentielle. On comprend alors la grande différence entre ces deux classes de complexité.

Remarque 1.9
Trivialement, on a $P \subseteq NP$. Il suffit de considérer le déterminisme comme un non-déterminisme particulier où il n'y a à chaque étape qu'un chemin possible. Par contre, la théorie de la complexité ne permet pas pour l'heure de déterminer s'il y a égalité entre les deux classes ou une inclusion stricte entre P et NP. Cependant, même si la théorie ne possède pas encore de réponse définitive à cette question, en pratique, il existe des problèmes de classe NP pour lesquels on ne connaît aucun algorithme polynomial permettant de le résoudre dans le pire des cas. On peut alors faire la conjecture suivante :

Conjecture 1.1
On conjecture que $P \neq NP$.

Définition 1.43 (classe CoNP)
La classe de complexité CoNP est l'ensemble des problèmes dont les problèmes complémentaires appartiennent à la classe NP.

Conjecture 1.2
On conjecture que $NP \neq CoNP$.

Remarque 1.10
On peut prouver facilement que l'on a :

$$NP \neq CoNP \quad \Rightarrow \quad P \neq NP.$$

Ainsi, il « suffit » de prouver que la classe NP n'est pas égale à son complémentaire pour également conclure quant à l'inégalité entre les classes P et NP.

Définition 1.44 (classe DP)
Un problème Ω appartient à la classe DP s'il peut être écrit $\Omega = \Omega_1 \cap \Omega_2$ avec $\Omega_1 \in NP$ et $\Omega_2 \in CoNP$.

Définition 1.45 (classe C^L)
On note C^L l'ensemble des problèmes qui peuvent être codés par une machine de Turing à oracle L et dont la complexité temporelle est polynomiale en fonction de la taille de l'entrée.

Note 1.9
Soient C_1 et C_2 deux classes de complexité. On note $C_1^{C_2}=\bigcup_{L\in C_2} C_1^L$

Note 1.10
Soit i un entier positif. On note :
- $\Sigma_0^p=\Pi_0^p=\mathrm{P}$
- $\Sigma_{i+1}^p=\mathrm{NP}^{\Sigma_i^p}$
- $\Pi_{i+1}^p=\mathrm{Co}\Sigma_{i+1}^p$

1.2.4 Complétude

Définition 1.46 (réduction fonctionnelle polynomiale)
Soient deux problèmes Ω et Ψ, et un algorithme f qui prend en entrée des instances de Ω et qui retourne des instances de Ψ.
f est appelée réduction fonctionnelle polynomiale *si et seulement si :*
- *f est un algorithme polynomial ;*
- *ω est une instance positive de Ω si et seulement si $f(\omega)$ est une instance positive de Ψ.*

Définition 1.47 (complétude)
On dit qu'un problème Ω est complet pour sa classe C, si et seulement si pour tout problème $\Psi \in C$, il existe une réduction fonctionnelle de Ψ vers Ω. Un tel problème est alors dit C-complet.

Remarque 1.11
Sous l'hypothèse de la conjecture 1.1, il n'existe aucun algorithme déterministe qui puisse résoudre un problème NP-complet en temps polynomial dans le pire des cas. Cette classe de problème possède de nombreux intérêts puisque de nombreux problèmes pratiques sont NP-complets. Parmi eux, on trouve certains problèmes de la théorie des graphes, de planification classique, d'ordonnancement, etc.

Chapitre 2

Résolution pratique du problème SAT

Sommaire

2.1 SAT : définition

Le problème SAT, abréviation de *test de SATisfaisabilité d'une formule propositionnelle sous forme normale conjonctive*, est un axe de recherche majeur en intelligence artificielle. Cet intérêt est non seulement théorique, car l'éventuelle résolution de ce problème en temps polynomial induit une réponse essentielle à l'une des plus grande question de la théorie de la complexité, mais également pratique, car **SAT est la pierre angulaire de nombreuses applications en intelligence artificielle** telles que la démonstration automatique ou l'inférence non monotone. Par ailleurs, les algorithmes dédiés à SAT sont également utilisés pour traiter des problèmes issus de la vérification formelle, de la recherche opérationnelle, de puzzles logiques, etc. On peut donc comprendre pourquoi une grande communauté de chercheurs s'intéressent à ce problème.

Dans ce chapitre est défini formellement le problème SAT, ainsi que différents problèmes de recherche et d'optimisation qui lui sont liés, puis sont présentées les différentes techniques utilisées en pratique pour leur résolution. Bien que les travaux présentés dans cette thèse n'ont pas pour objectif premier la résolution pratique de SAT, les techniques algorithmiques développées s'appuient en grande partie sur les approches existantes pour SAT. Celles-ci sont par conséquent dépeintes dans ce chapitre.

Définition 2.1 (problème SAT)
Soit Σ une formule sous forme normale conjonctive (CNF). Le problème de décision SAT consiste à déterminer s'il existe un modèle de Σ.

Propriété 2.1
SAT est un problème NP-complet [Cook 1971].

Définition 2.2 (k-SAT)
k-SAT est le problème qui consiste à déterminer la consistance d'une conjonction de clauses de longueur k.

Propriété 2.2
2-SAT appartient la classe P.

Preuve
Un algorithme de décision de satisfaisabilité pour *2-SAT* se base sur la résolution entre les clauses de l'instance. Or, la résolution de clauses binaires produit des clauses de taille au plus 2 et le nombre de clauses binaires pouvant être construites est une fonction quadratique du nombre de variables de l'instance. Un tel algorithme est donc clairement polynomial et montre l'appartenance de *2-SAT* à la classe P. □

Propriété 2.3
k-SAT (avec $k \geq 3$) est NP-complet.

Remarque 2.1
Dès que l'instance n'est pas composée exclusivement de clauses binaires, le problème passe donc à la classe de complexité supérieure. Il existe cependant de nombreuses autres classes d'instances SAT qui possèdent un algorithme de reconnaissance de l'appartenance à la classe et de décision de la satisfaisabilité polynomial. Cette propriété est souvent due à la structure particulière de ces instances (comme les instances composées exclusivement de clauses de Horn ou les formules bien équilibrées), mais leurs champs d'application en sont d'autant restreints.

Définition 2.3 (MaxSAT)
MaxSAT est le problème d'optimisation associé à SAT. Il s'agit, pour une formule Σ donnée, de déterminer le nombre maximal de clauses de Σ qui peuvent être satisfaites par une même interprétation.

Remarque 2.2
Pour une formule CNF Σ satisfaisable composée de n clauses, MaxSAT(Σ)=n.

Définition 2.4 (#SAT)
Soit une formule Σ sous forme normale conjonctive. #SAT est le problème qui consiste à déterminer le nombre de modèles de Σ.

Définition 2.5 (conséquence)
Soient Σ et Γ deux formules de la logique propositionnelle classique. Ce problème consiste à déterminer si $\Sigma \models \Gamma$.

Remarque 2.3
Comme indiqué dans le chapitre précédent, tester si « $\Sigma \models \Gamma$ » est équivalent à tester si « $\Sigma \wedge \neg\Gamma$ est inconsistant ». Cette question est donc réductible au problème complémentaire à SAT, ce qui explique son appartenance à la classe de complexité CoNP.

Fonction DP(Σ : **formule CNF**) : **booléen**
  Si ($\Sigma = \emptyset$) **Alors Retourner** *vrai* **Fin Si**
  Si ($\bot \in \Sigma$) **Alors Retourner** *faux* **Fin Si**
  Choisir une variable $x \in \Sigma$;
  Faire la résolution avec les clauses contenant x ou $\neg x$;
  Ajouter les clauses résolvantes à Σ ;
  Supprimer de Σ les clauses contenant x ou $\neg x$;
  Retourner DP(Σ) ;
Fin

Algorithme 1 – La procédure DP

2.2 Approches complètes : des algorithmes basiques aux solveurs modernes

On appelle méthode complète de résolution de SAT toute technique qui permet de déterminer en temps fini la consistance de n'importe quelle formule CNF. Un algorithme naïf de méthode complète, appelé « force brute », consiste à énumérer explicitement les $2^{|Var(\Sigma)|}$ interprétations possibles pour une formule Σ et de tester pour chacune d'entre elles si elle est un modèle de Σ. Si aucun modèle n'a été trouvé, on peut alors affirmer que Σ est insatisfaisable. Il est clair qu'une telle méthode ne peut fonctionner que pour des instances de très petite taille. Même muni du plus puissant supercalculateur actuel, cet algorithme ne pourrait déterminer à échelle de vie humaine l'inconsistance d'une formule construite sur 100 variables.

De nombreuses méthodes plus efficaces ont été proposées : la méthode des tableaux de [Smullyan 1968], celle des diagrammes de décision binaire de [Akers 1978], mais les méthodes les plus utilisées en pratique restent indubitablement des variantes des techniques initiées par Davis, Putnam, Logemann et Loveland. Nous étudions donc dans ce paragraphe les approches DP et DLL ainsi que leurs principales améliorations algorithmiques.

2.2.1 La procédure DP

Cet algorithme, proposé par [Davis & Putnam 1960], en une **résolution dirigée** qui permet de déterminer si la formule donnée en entrée possède un modèle. Soit Σ la formule sous forme CNF donnée en entrée. Concrètement, l'algorithme vérifie en premier lieu les cas triviaux suivants :

- Σ est la formule vide. Dans ce cas, Σ est satisfaisable.
- Σ contient la clause vide, donc elle est insatisfaisable.

Si aucun de ces cas ne se présente, l'algorithme sélectionne alors une variable x de Σ, génère toutes les clauses résolvantes en utilisant x puis supprime toutes les clauses contenant une occurrence de cette variable. L'algorithme itère jusqu'à produire la formule vide (SAT), ou une formule contenant une clause vide (UNSAT). Un de ces cas se produit nécessairement en un temps fini. La procédure DP est décrite plus précisément dans l'Algorithme 1.

Exemple 2.1
Soient Σ la formule suivante : $\Sigma = (a \vee c \vee b)$

$$\begin{aligned}&\wedge (\neg a \vee b)\\ &\wedge (c \vee \neg b)\\ &\wedge (\neg c \vee \neg b \vee \neg a)\\ &\wedge (a \vee \neg c)\end{aligned}$$

Appliquons la procédure DP en sélectionnant les variables dans l'ordre alphabétique. Lors de la première itération, les cas triviaux n'étant pas rencontrés, la procédure choisit la variable a, génère les clauses résolvantes, et supprime les clauses contenant a. La formule devient alors :

$$\begin{aligned}\Sigma = &(c \vee b)\\ &\wedge (c \vee \neg b)\\ &\wedge (\neg c \vee \neg b)\\ &\wedge (b \vee \neg c)\end{aligned}$$

Aucun des cas triviaux ne se présente dans cette formule, l'algorithme est réitéré en choisissant comme variable b. On obtient : $\Sigma = c \ \wedge \neg c$

A la prochaine itération, c sera choisie, et la résolution des deux clauses unitaires « c » et « $\neg c$ » produit la clause vide. En supprimant les clauses où apparaît c, on obtient $\Sigma = \bot$. A l'itération suivante, la procédure DP renvoie donc *faux* ; Σ est ainsi prouvée incohérente.

Malgré sa simplicité conceptuelle, l'approche telle qu'elle est présentée ici est rarement utilisée en pratique. En effet, l'inconvénient majeur de l'algorithme DP est sa complexité exponentielle en temps mais aussi en espace due à la génération des clauses résolvantes. Il est facile de comprendre pourquoi cette méthode n'est pas utilisée sous sa forme originelle et pourquoi des améliorations ont été faites dans le but d'accroître l'efficacité de cet algorithme.

2.2.2 La procédure DLL

La procédure DLL, proposée en 1962 par Martin Davis, George Logemann et Donald Loveland [Davis *et al.* 1962], est une amélioration de la procédure DP. Alors que cette dernière utilise la résolution pour prouver ou réfuter la consistance d'une instance, la procédure DLL se fonde sur la **séparation**.

Il s'agit d'un algorithme de recherche **en profondeur d'abord** dans un arbre binaire où chaque nœud correspond à une sous-formule, sa branche gauche à l'affection d'une de ses variables à *faux* et sa branche droite à son affectation à *vrai*. Les feuilles correspondent, elles, soit à une contradiction, soit à une solution.

L'idée est qu'une formule Σ est satisfaisable si et seulement si $\Sigma \wedge l$ est satisfaisable ou $\Sigma \wedge \neg l$ est satisfaisable, $\forall l \in Var(\Sigma)$. Cette procédure choisit donc **heuristiquement** une variable l de Σ, puis détermine la consistance des sous-formules obtenues en affectant l à *vrai* puis à *faux*. Ce procédé est synthétisé dans l'algorithme 2.

L'intérêt de cette méthode réside principalement dans sa complexité spatiale qui n'est pas exponentielle, comme dans la procédure DP, mais polynomiale. Encore aujourd'hui, de nombreux solveurs « modernes », utilisés pour résoudre des instances de grandes tailles, sont basés sur cette procédure.

Définition 2.6 (espace de recherche)
On appelle espace de recherche l'ensemble des interprétations parcourues pour décider si une formule Σ est consistante ou non.

Fonction DLL(Σ : **formule CNF**) : **booléen**
  Appliquer la propagation unitaire ;
  Si ($\Sigma = \emptyset$) **Alors** **Retourner** *vrai* **Fin Si**
  Si ($\bot \in \Sigma$) **Alors** **Retourner** *faux* **Fin Si**
  Choisir une variable $x \in \Sigma$;
  Retourner (DLL($\Sigma \wedge x$) **OU** DLL($\Sigma \wedge \neg x$)) ;
Fin

Algorithme 2 – La procédure DLL

Fonction Propagation_unitaire(Σ : **formule CNF**) : **formule CNF simplifiée**
  Tant que ($\exists c \in \Sigma$ tel que taille(c)=1) **faire**
    Soit l l'unique littéral non assigné de c ;
    Supprimer toutes les clauses où apparaît l ;
    Supprimer toutes les occurrences de $\neg l$;
  Fait
  Retourner Σ
Fin

Algorithme 3 – La propagation unitaire

Clairement, la méthode « force brute » énumère plus d'interprétations qu'il ne faut. Dans la plupart des cas, de nombreuses interprétations peuvent être évitées car on peut « deviner », par exemple grâce à la structure de l'instance, qu'elles ne la satisfont trivialement pas. Éviter de considérer ces interprétations permet de réduire l'espace de recherche et ainsi accélérer les algorithmes.

La plus élémentaire des simplifications qui peut être faite pour éviter de parcourir des interprétations inutiles est appelée **propagation unitaire**. Cette opération consiste, tant qu'il existe une clause unitaire, à satisfaire son unique littéral. Clairement, les interprétations parcourues sont les seules qui peuvent potentiellement satisfaire la formule car affecter à son opposé un littéral qui apparaît dans une clause unitaire conduit nécessairement à une clause vide et donc à une contradiction.

Exemple 2.2
Soit la formule $\Sigma = (a \vee \neg b \vee c) \wedge (a \vee b) \wedge (a \vee \neg c) \wedge (\neg a \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$. En supposant le choix des variables fait par ordre alphabétique, l'arbre binaire de recherche de la procédure DLL est présentée en figure 2.1. On y voit clairement l'intérêt de la propagation unitaire quant à la réduction de l'espace de recherche.

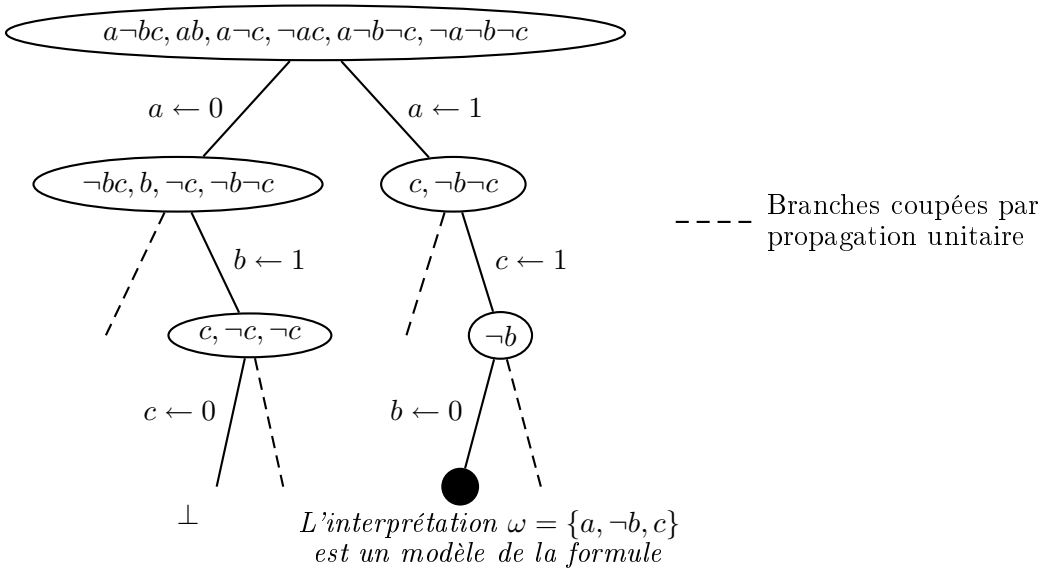


FIG. 2.1 – Arbre binaire de recherche

2.2.3 Heuristiques de choix de variables

Réduire l'espace de recherche permet d'accélérer les méthodes complètes : ceci est clairement visible par des simplifications telle que la propagation unitaire. Mais un autre critère permet de réduire le nombre d'interprétations à parcourir : la sélection de variables de branchement. Jusqu'ici, nous avons supposé l'ordre de sélection des variables arbitraire et fixé par avance, mais il faut savoir qu'il s'agit d'un facteur déterminant pour l'efficacité de la procédure DLL, puisqu'un arbre de recherche peut être exponentiellement plus « grand » (en nombre de nœuds) en fonction de l'ordre des variables affectées. Cependant, sélectionner la variable optimale à brancher à un certain niveau de l'arbre est un problème *NP-difficile*.

Il apparaît dès lors qu'il convient plutôt d'estimer le plus précisément possible la variable à affecter plutôt que de la calculer : on fait alors appel à une **heuristique**. Les heuristiques de choix de variables font l'objet de nombreux travaux dans la littérature, mais elles convergent toutes dans leur objectif : faire une affectation qui produise le plus grand nombre de propagations unitaires possibles, afin d'obtenir le sous-problème le plus simple possible à résoudre.

Une première heuristique développée dans ce sens a été baptisée « **Maximum occurrences in clauses of Minimum Size** » ou **MOMS** [Goldberg 1979]. Le but de cette heuristique est de choisir la variable qui apparaît le plus souvent dans les clauses les plus courtes, afin de provoquer, autant que possible, des propagations unitaires en cascade. Cette approche purement syntaxique se révèle relativement efficace en pratique, bien qu'elle ne se fonde que sur la structure du problème. Une amélioration de cette heuristique consiste à lui associer un critère sur l'équilibre de la taille des deux sous-arbres résultants des deux affectations potentielles de la variable choisie. Dans ce but, les variables ayant une fréquence d'apparition équilibrée sous forme négative et positive sont privilégiées par l'heuristique [Jeroslow & Wang 1990].

On peut également, pour déterminer la variable à brancher, avoir recours à la notion de **backbone** [Dequen & Dubois 2001]. Une variable appartient au backbone si et seulement si sa valeur de vérité est constante dans tous les modèles de l'instance. C'est alors un impliqué premier unaire de l'instance. L'heuristique tente de choisir, en fonction de l'instance et d'une interpré-

tation partielle, une variable appartenant au backbone du sous-ensemble de clauses satisfait par l'interprétation partielle. La probabilité pour une variable d'appartenir au backbone est donc estimée en comptant les occurrences de ce littéral dans les clauses satisfaites. Cette heuristique, assez peu utilisée en pratique, se comporte pourtant très bien sur de nombreuses instances.

Une autre alternative pour l'heuristique de choix de variable est d'**utiliser directement la propagation unitaire** pour décider de la variable à « brancher » [Freeman 1995]. Il s'agit, pour chaque variable potentiellement choisie, de l'affecter successivement à *faux* puis à *vrai*, et de déclencher la propagation unitaire dans chaque cas en comptant le nombre de propagations engendrées. Ce nombre permet de discriminer parmi les variables, celle dont l'affectation provoque le plus grand nombre de propagations unitaires. Contrairement à une heuristique comme MOMS, cette méthode est exacte quant au nombre de simplications qui pourront être faites après l'affection de chaque variable, et se révèle donc plus efficace. Bien sûr, elle est aussi beaucoup coûteuse en terme de temps de calcul, puisqu'il faut, pour chaque variable, l'affecter puis exécuter la procédure de propagation unitaire. Il faut donc choisir l'heuristique en évaluant le **compromis** que l'on souhaite faire entre sa précision et son coût. On peut également utiliser cette heuristique uniquement sur un sous-ensemble de variables sélectionnées par une heuristique moins coûteuse, comme MOMS, dans le but d'en réduire le coût [Li & Anbulagan 1997].

Toutefois, la plupart des solveurs modernes utilisent une nouvelle heuristique estimant dynamiquement la contrainte exercée par les clauses sur chaque variable de la formule. En pratique, un compteur est associé à chaque variable et quand un conflit apparaît lors de la recherche, celui-ci est analysé et les compteurs des variables impliquées sont incrémentées. Cette heuristique de type **conflict driven**, connue sous le nom de VSID (pour *Variable State Independent Decaying*) [Moskewicz *et al.* 2001], choisit dès lors la variable non affectée ayant le compteur le plus élevé. En effet, celle-ci est la plus impliquée dans les conflits, et est donc jugée fortement contrainte. Un des inconvénients de cette heuristique est qu'au début de la recherche, aucune indication sémantique sur les variables de la formule n'est disponible ; des choix arbitraires sont alors effectués. Pourtant les premiers choix de variables sont sans doute les plus importants pour la taille de l'arbre de recherche. Ce problème est géré par les techniques de « *restart* », qui sont présentées dans le paragraphe suivant.

2.2.4 Le *restart*

Les premiers choix effectués lors d'une recherche complète sont prépondérants. Ceux-ci influencent en grande partie la taille de l'espace de recherche : **un mauvais choix en début de recherche peut donc conduire à un parcours exponentiellement plus long**. Avec l'avènement de nouvelles heuristiques dynamiques, choisissant généralement les variables en fonction de leur implication dans les conflits rencontrés, **on aimerait pouvoir reconsidérer ces premiers choix**. C'est dans ce cadre qu'a été proposée la technique de redémarrage, ou *restart*. Sommairement, son principe est de recommencer la recherche après un avoir rencontré un certain nombre de conflits [Gomes *et al.* 1998]. L'idée générale est qu'après une certaine quantité de retours en arrière (ou **backtracks**), on considère que la recherche ne peut aboutir en un temps raisonnable, parce que par exemple de mauvais choix ont été faits. Cette dernière est alors stoppée, puis redémarrée, dans l'espoir qu'avec l'expérience acquise lors des recherches précédentes, l'heuristique dynamique fasse des choix plus judicieux, permettant de parcourir un espace de recherche réduit.

Malheureusement, cette stratégie, si elle peut s'avérer efficace, ôte la propriété de complétude à la recherche DLL. Pour pallier cet inconvénient, il a été par la suite proposé d'incrémenter le nombre de conflits à rencontrer avant chaque restart [Baptista & Marques-Silva 2000], afin

d'assurer la terminaison de l'algorithme. De surcroît, il est également préconisé d'effectuer parcimonieusement des choix de manière aléatoire, de façon à varier l'espace parcouru, et ne pas intensifier systématiquement la recherche en choisissant toujours les mêmes variables d'un restart à l'autre. La combinaison de ces techniques permet de manière générale d'**augmenter la robustesse des solveurs actuels**, en leur « interdisant » de s'engouffrer dans un espace de recherche démesurément trop grand, s'il existe des choix permettant de le réduire.

2.2.5 Apprentissage

Il existe de nombreuses techniques permettant de réduire l'espace de recherche et donc d'accélérer la résolution pratique d'instances. Parmi elles, une méthode utilisée par la plupart des solveurs modernes est appelée *apprentissage* (cf. par exemple [de Kleer & Williams 1987, Zhang & Malik 2002]). Elle a pour but, lors d'une interprétation partielle infructueuse, d'analyser la clause du conflit, afin de stocker de nouvelles contraintes (sous forme de clauses appelées *nogoods*) qui permettent d'éviter à la méthode de recherche de parcourir plusieurs fois le même sous-problème.

Plus précisément, le but est d'analyser le chemin courant et les littéraux choisis par l'heuristique pour déterminer ceux qui, parmi eux, sont responsables de la contradiction. Quand les littéraux en question sont identifiés, on effectue un saut en arrière non chronologique (**non-chronological backtracking**) au niveau du dernier littéral affecté qui a causé la production de la clause vide, et non au niveau du dernier littéral *choisi* comme dans le schéma classique de DLL. Dans le but d'éviter de parcourir plusieurs fois le même sous-problème, une clause contenant la négation des littéraux responsables de la contradiction est ajoutée. Il est clair que les clauses ajoutées par ce procédé sont des conséquences logiques de la formule. Ce sont en fait des résolvantes particulières de la CNF, des contraintes non exprimées qui sont exhibées à travers ces *nogoods*. Historiquement, depuis l'introduction de cette notion d'apprentissage, l'algorithme classique DLL a été rebaptisé DPLL, en référence à l'hybridation implicite avec la procédure DP.

Exemple 2.3
Soit Σ une formule CNF construite sur 11 variables et contenant 9 clauses telle que :

$$\begin{array}{lll} \Sigma = & (\ a \vee b\ ) & \mathbf{c_1} \\ \wedge & (\neg b \vee c \vee d) & \mathbf{c_2} \\ \wedge & (\neg b \vee e) & \mathbf{c_3} \\ \wedge & (\neg d \vee \neg e \vee f) & \mathbf{c_4} \\ \wedge & (a \vee c \vee f) & \mathbf{c_5} \\ \wedge & (\neg a \vee g) & \mathbf{c_6} \\ \wedge & (\neg g \vee b) & \mathbf{c_7} \\ \wedge & (\neg h \vee j) & \mathbf{c_8} \\ \wedge & (\neg i \vee k) & \mathbf{c_9} \end{array}$$

Supposons qu'un solveur muni d'une technique d'apprentissage et de retour-en-arrière non chronologique soit exécuté avec Σ donnée en entrée. On suppose également que les variables c, f, h et i soient successivement affectées à *faux*. Une série de propagations unitaires est alors déclenchée (a par c_5, g par c_6, b par c_7, d par c_2 et enfin e par c_3), rendant la clause c_4 falsifiée par l'interprétation courante $\omega = \{a, b, \neg c, d, e, \neg f, g, \neg h, \neg i\}$. Une analyse du graphe d'implication, qui est une structure dédiée à l'enregistrement des causes (*reason*) de chaque propagation unitaire, permet de déterminer que ni l'affectation h ni celle de i n'est en cause dans ce conflit. Ceci est

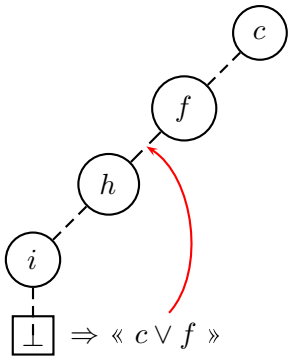


FIG. 2.2 – Élagage par apprentissage et retour-en-arrière non chronologique

ici facilement observable par le fait qu'aucune clause dans laquelle l'une d'elles apparaît n'a déclenché de propagation unitaire. Ce sont les affectations $\neg c$ et $\neg f$ qui sont à l'origine de cet échec.

On a donc « $(\neg c) \wedge (\neg f) \Rightarrow (\Sigma$ est faux) », de laquelle on peut déduire par contraposition « $(\Sigma$ est vrai) $\Rightarrow (c \vee f)$ ». Cette nouvelle clause est alors apprise et ajoutée à la formule. Le retour-en-arrière n'est donc pas effectué sur la dernière variable de décision (i), car cette nouvelle clause ne serait toujours pas satisfaite, mais au niveau de la dernière variable affectée lui appartenant, dans notre exemple f.

L'arbre de recherche correspondant est dessiné en figure 2.2. Celle-ci permet d'observer l'élagage important permis par ces techniques complémentaires, évitant à h et i d'être affectées inutilement à *vrai* dans ce contexte infructueux.

Un problème majeur de cette technique concerne le nombre de *nogoods* appris. Dans l'absolu, l'algorithme est capable d'apprendre une clause pour chaque conflit détecté, et son nombre peut être exponentiel en fonction du nombre de variables de l'instance. Plusieurs techniques ont été proposées dans le but de ne conserver qu'un nombre polynomial de clauses apprises. On cherche bien entendu à garder les clauses qui ont les valeurs informatives les plus importantes, c'est-à-dire celles qui réduiront par la suite le plus possible l'espace de recherche. L'une de ces techniques consiste à ne conserver que les clauses apprises qui ont une taille inférieure ou égale à une longueur donnée [Marques-Silva & Sakallah 1996] ; en effet, plus une clause est courte, plus le nombre d'interprétations qui la falsifie est grand. Cependant, la technique la plus utilisée dans les solveurs modernes est basée sur un compteur pour chacunes des clauses apprises qui est incrémenté dès que la clause correspondante produit une propagation unitaire [Goldberg & Novikov 2002]. Régulièrement, la base de *nogoods* est purgée de ceux ayant un compteur à un bas niveau. Il est ici supposé que les clauses les plus efficaces pour réduire l'espace de recherche le seront également dans le futur de la recherche.

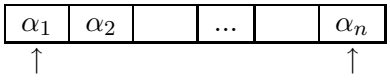
Il a été prouvé que cette technique d'apprentissage, couplée avec un nombre non limité de restarts, est **exponentiellement plus efficace que la seule procédure DLL** [Beame *et al.* 2004]. En pratique également, l'apprentissage permet d'accroître la classe des formules traitables, mais son usage est limité par la quantité de mémoire de nos ordinateurs actuels. Il faut donc, comme bien souvent, faire un compromis entre une utilisation complète de cette méthode et les ressources dont on dispose pour elle.

2.2.6 Les structures de données paresseuses

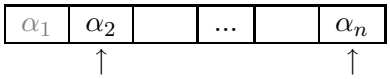
Les techniques d'apprentissage, couplées aux heuristiques modernes de choix de variables (la plupart du temps de type *fail-first*), tendent à réduire l'espace de recherche en produisant des propagations unitaires en cascade. Cette opération occupe en pratique plus de 90% du temps de calcul d'un solveur. En conséquence, accélèrer le traitement de la propagation unitaire engendre de manière globale un gain significatif pour les approches de type DPLL.

Dans ce domaine, de nombreux progrès algorithmiques ont été faits ces dernières années, mais le plus important reste incontestablement la méthode proposée dans [Zhang & Malik 2002]. Celle-ci introduit une structure de données de représentation des clauses d'une formule CNF dite « paresseuse » qui permet un traitement de la propagation unitaire d'une complexité moyenne sous-linéaire. L'idée est, pour chaque clause d'une formule CNF, de ne pas considérer tous ses littéraux, mais seulement deux d'entre eux qui ne sont pas affectés. Tant que deux littéraux de chaque clause ne sont pas affectés, aucune propagation unitaire ne peut être produite. Soit Σ une formule CNF, et $c = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ une clause n-aire telle que $c \in \Sigma$. En grisant, entourant et désignant respectivement les littéraux falsifiés, satisfaits et regardés, la représentation de c par une structure de données paresseuse se fait donc comme suit :

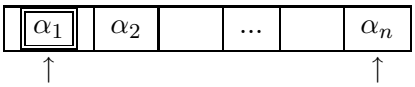
- Lorsqu'aucune variable de c n'est affectée, seuls deux littéraux de c (le premier et le dernier) sont considérés.



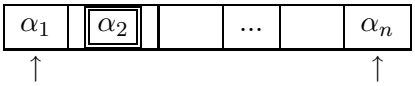
- Falsifier un littéral considéré (ici α_1) a pour conséquence de déplacer le curseur vers le prochain littéral de la clause non affecté. Lors d'un retour en arrière dans l'arbre, α_1 peut être désaffecté et de nouveau regardé à la place de α_2.



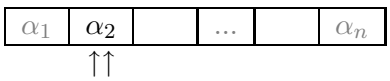
- Au contraire, si l'un des deux littéraux considérés satisfait la clause (ici α_1), celle-ci n'est plus tenue à jour jusqu'au backtrack qui rendra cette variable non assignée.



- On suppose maintenant α_2 satisfait, les autres littéraux étant non affectés. N'étant pas considérée dans c, la clause est toujours vue comme étant active. Si α_1 est falsifié par exemple, alors α_2 est « regardé », et la procédure s'« aperçoit » que la clause était satisfaite. Le cas dual (où une variable non « regardée » est affectée à *faux*) est gérée de manière similaire.



- Enfin, si les deux curseurs pointent vers le même littéral, alors il n'existe pour la clause c qu'un littéral non affecté (les autres étant nécessairement falsifiés). Une propagation unitaire est alors déclenchée.



On peut observer que, lors d'une affection de variable, la mise-à-jour de la structure ne s'effectue que sur un sous-ensemble des clauses contenant une occurrence de cette variable. Pourtant ce traitement partiel est *suffisant* pour déclencher les propagations unitaires aux moments

opportuns durant le parcours de l'espace de recherche. Ceci explique la rapidité de traitement des affectations, et en particulier de la propagation unitaire. Cependant, cette structure de données possède également quelques inconvénients : les clauses n'étant pas considérées dans leur intégralité, certains traitements, tels que l'heuristique MOMS, ne sont plus possibles, car ils requièrent une connaissance complète de l'instance après affectation(s). Seules les heuristiques sémantiques (telle que la « *conflict driven* » VSID) peuvent être appliquées avec les structures de données paresseuses, la structure du problème traité n'étant pas connue.

2.2.7 Prétraitement de la formule

Tester la satisfaisabilité d'une formule est, comme on l'a vu, un problème difficile. Cependant, certains traitements polynomiaux peuvent être appliqués à une CNF *avant* l'exécution de la procédure DPLL, dans le but de la simplifier et de rendre moins ardue la détermination de sa satisfaisabilité. D'autres transformations visent à obtenir une formule plus simple qui n'est pas équivalente, mais équisatisfaisable. Pendant longtemps, ce genre de prétraitement n'a pas été utilisé en pratique car il conduit parfois à une perte d'efficacité de la formule transformée comparée à l'originale, mais depuis peu, certains traitements purement syntaxiques de la formule se sont montrés véritablement efficaces en pratique, et ils sont maintenant utilisés pour simplifier autant que possible la formule avant d'utiliser la procédure DPLL

L'un des premiers algorithmes de prétraitement efficace, appelé `3-Resolution`, fait parti du solveur `Satz` [Li & Anbulagan 1997]. Celui-ci consiste à ajouter à la formule toutes les clauses résolvantes de taille inférieure ou égale à 3, jusqu'à saturation. De plus, toutes les clauses subsumées sont détectées et supprimées de la CNF. Un préprocesseur moins gourmand en ressources a ensuite été proposé dans [Brafman 2001]. Du nom de `2-SIM`, il a été développé afin d'améliorer l'efficacité des solveurs sur les problèmes issus du monde réel, qui contiennent souvent un grand nombre de clauses binaires. Sommairement, l'idée est donc d'utiliser ces clauses binaires pour construire un graphe d'implication, duquel on peut déduire des clauses unitaires par le calcul de la fermeture transitive. Si de telles clauses sont produites, elles sont propagées et ce procédé est itéré jusqu'à ce qu'un point fixe soit atteint. Plus tard, un nouveau préprocesseur baptisé `HyPre` a généralisé `2-SIM` en utilisant l'hyper-résolution pour déduire de nouvelles clauses binaires [Bacchus & Winter 2003]. De plus, cet algorithme détecte et substitue les littéraux équivalents de manière incrémentale. En conséquence, `HyPre` se montre plus efficace que son prédécesseur pour la plupart des formules traitées. La « classique » procédure DP, basée sur l'élimination de variables par résolution, a également été considérée comme prétraitement d'une CNF. À cause de sa complexité exponentielle en espace, un schéma plus faible a été adopté par la procédure `NiVER` [Subbarayan & Pradhan 2004]. Celle-ci élimine les variables par résolution si ce calcul ne conduit pas à augmenter le nombre de littéraux de la CNF. Appliquer `NiVER` avant de résoudre une formule permet de grandement améliorer les performances des solveurs modernes. Celle-ci a ensuite été améliorée par l'utilisation d'une nouvelle règle de substitution et de *signatures* de clauses [Eén & Biere 2005]. Le préprocesseur résultant, nommé `SatElite`, est maintenant utilisé comme étape préliminaire de la plupart des solveurs actuels et est impliqué dans les bons résultats des vainqueurs des trois dernières compétitions.

2.3 Approches incomplètes : la recherche locale

Malgré les améliorations proposées aux méthodes complètes, celles-ci peuvent se révèler très coûteuses en terme de temps dans le cas général. Parallèlement à ces techniques, on a donc développé des méthodes dites incomplètes qui ont pour objectif de trouver un modèle à une

formule s'il en existe un, en un temps polynomial. Pour cela, elles font un **parcours de l'espace d'interprétations de manière non systématique** pendant un temps donné.

Si un modèle est trouvé, la procédure stoppe son exécution et peut répondre que la formule est satisfaisable ; par contre, si au bout d'un certain nombre d'opérations ou d'un certain temps, la méthode n'a pas trouvé de modèle, elle stoppe son exécution sans pouvoir conclure quant à la satisfaisabilité de la formule. En effet, ce type de procédure ne garantit pas d'avoir parcouru l'ensemble des interprétations d'une formule (même au bout d'un temps excessivement long) : il est donc possible que la formule ait un modèle sans pour autant avoir pu le trouver : c'est la propriété d'**incomplétude** de ces méthodes.

Il existe différents types de méthodes incomplètes, les principaux étant le recuit simulé [Kirkpatrick *et al.* 1983, Spears 1996], les techniques algorithmiques « génétiques » à base de population [de Jong & Spears 1989, Young & Reel 1990, Hao & Dorne 1994], la « survey propagation » [Mézard *et al.* 2002, Mézard & Zecchina 2002] et la recherche locale. Dans ce paragraphe sont présentées différentes techniques de recherche locale, celles-ci étant certainement les plus connues de toutes les méthodes incomplètes.

2.3.1 Principes

Clairement, si les méthodes incomplètes ne parcourent qu'une portion de l'espace de recherche, les interprétations considérées ne sont pas choisies *au hasard*, mais ce sont celles qui sont susceptibles de conduire à un modèle de l'instance. Soit Σ une formule CNF, ω une interprétation de Σ, on définit alors la fonction objectif f :

$$f(\Sigma, \omega) = |\{c \in \Sigma \mid \omega \not\models c\}|$$

f représente donc le nombre clauses fausses de $\omega(\Sigma)$. Lors de la recherche locale, on cherche à minimiser cette fonction. Si pour une interprétation ω_S, $f(\Sigma, \omega_S) = 0$ alors cette interprétation est un modèle de la formule : en effet, aucune clause n'est falsifiée par cette interprétation, alors toute clause de la formule est vraie.

La fonction objectif présentée ici est donnée en guise d'exemple : d'autres peuvent bien sûr être définies, mais la plupart du temps, c'est cette fonction (ou une autre fonction équivalente, commne la maximisation du nombre de clauses satisfaites de la formule) qui est utilisée pour guider la recherche locale.

Concrètement, au lancement de la procédure, une interprétation ω_0 est générée aléatoirement. On considère $f(\omega_0)$. Si elle est égale à 0, c'est que ω_0 est un modèle de la formule, on conclut donc à sa satisfaisabilité. Dans le cas contraire, on considère $E = \{\omega \mid d_H(\omega, \omega_0) = 1\}$: les interprétations contenues dans cet ensemble sont appelées « voisines » de ω_0 car elles ne diffèrent de cette dernière que de la valeur de vérité d'un seul atome. On dit qu'on **flippe** l lorsque c'est la valeur de vérité de cette variable qui est modifiée entre une interprétation et sa voisine. L'interprétation ω_1 qui succède à ω_0 dans la recherche locale est définie ainsi :

$$\omega_1 = argmin(\omega \in E_{\omega_0}, f(\Sigma, \omega))$$

Il peut exister plusieurs interprétations dans E minimales par rapport à f. Dans ce cas, on sélectionne au hasard une de ces interprétations, ou on utilise une autre fonction pour les discriminer. Si au bout d'un certain nombre de flips aucun modèle n'est trouvé, la recherche locale stoppe son exécution sans pouvoir conclure à la satisfaisabilité de la formule.

De prime abord, la recherche locale telle qu'elle est présentée ici peut paraître une méthode efficace pour la recherche de modèles d'une instance satisfaisable. En effet, la procédure, cherchant à optimiser sa fonction objectif, améliore sans cesse l'interprétation courante dans le but de

trouver un modèle. Il peut cependant exister des cas où aucun flip ne diminue le nombre de clauses falsifiées.

Définition 2.7 (minimum)
Soient Σ une formule et ω une interprétation de Σ. On dit que ω est un minimum de Σ par rapport à la fonction objectif f si aucun flip de ω ne permet d'augmenter la valeur de f. Formellement :

$$\omega \text{ est un minimum de } \Sigma \text{ par rapport à } f$$
$$\Leftrightarrow$$
$$\forall \omega_S \text{ tel que } d_H(\omega_S, \omega) = 1,\ f(\Sigma, \omega) >= f(\Sigma, \omega_S)$$

Remarque 2.4
On distingue deux types de minima : les **minima globaux** sont des interprétations telles qu'il n'existe aucune interprétation de la formule permettant d'optimiser la fonction objectif. Pour une formule satisfaisable et la fonction f présentée précédemment, la valeur du (ou des) minimum global est de 0. D'autre part, les **minima locaux** sont des interprétations telles qu'aucun flip n'améliore la fonction, mais dont la valeur de cette dernière n'est pas optimale.

De manière générale, qu'il soit local ou global, un minimum pose un problème aux méthodes de recherche locale. En effet, dans un minimum ω_M, l'interprétation qui est choisie comme « suivante » est donc celle qui dégrade le moins la fonction objectif. Cependant, il est possible que dans cette nouvelle interprétation, la meilleure (et qui sera donc prise pour être la suivante) soit ω_M. Dans ce cas, l'algorithme étant déterministe, on entre dans un cycle et le parcours de l'espace de recherche se réduit à ces deux interprétations. Pour éviter de tomber dans des minima, ces algorithmes ont recours à des stratégies d'échappement.

2.3.2 Stratégies d'échappement

Une stratégie d'échappement a pour objectif de ne pas faire le choix optimal lors d'une recherche locale pour sortir d'un éventuel minimum. Il en existe de nombreuses, et c'est en grande partie ces stratégies qui font la distinction entre les différentes recherches locales qui ont été proposées. Nous présentons dans ce paragraphe une liste non exhaustive de stratégies parmi les plus connues :

- **random walk** [Selman *et al.* 1993] : dans le but d'échapper aux minima, il faut éviter d'effectuer le flip optimal à chaque pas. Cependant, pour trouver un modèle, le parcours de l'espace d'interprétations doit être pertinent, et la méthode de recherche locale appliquer des coups optimaux un certain nombre de fois. On introduit alors une probabilité p, et applique l'algorithme suivant :
  - avec une probabilité p, choisir aléatoirement une variable apparaissant dans une clause falsifiée et la flipper ;
  - avec une probabilité $1 - p$, effectuer le coup optimal.

  On peut également imaginer une variante à cet algorithme où on ne restreint pas le choix de la variable aux clauses falsifiées mais à l'ensemble des variables. Avec de telles stratégies, on est assuré de sortir stochastiquement des minima (si $p > 0$).

- **la méthode tabou** : cette méthode conserve dans une file appelée « tabou » la liste des n dernières variables qui ont été flippées, en interdisant le flip de toute variable se trouvant dans cette liste, même si elle représente la meilleure opportunité pour la fonction objectif. On sélectionne alors la meilleure variable qui ne soit pas dans cette liste. Cette

technique, étudiée dans de nombreux articles (par exemple [Mazure *et al.* 1997]), s'avère relativement efficace pour échapper aux minima, cependant ses performances dépendent de la taille de la liste tabou, dont la valeur optimale varie en fonction de l'instance à traiter.

- **novelty** : cette stratégie est une variante de « random walk », et classe (comme cette dernière) les variables suivant le nombre de clauses falsifiées auxquelles elles appartiennent. Soient la meilleure et la seconde meilleure variable selon cet ordre. Si la meilleure variable n'est pas la plus récemment flippée, alors elle est choisie. Sinon, avec une probabilité p, on choisit la seconde meilleure variable, et avec une probabilité $1 - p$, on sélectionne la meilleure. Comparativement à Random Walk, dans cette approche, la notion de choix aléatoire d'une variable selon une certaine probabilité a été supprimée au profit d'un choix non-optimal, mais satisfaisant puisqu'il s'agit de la variable qui lui succède dans l'ordre. De nombreuses variantes de novelty ont été proposées (telles que rnovelty ou novelty+), et incluent pour la plupart des stratégies couple-cycle. L'ensemble de ces stratégies est présenté et étudié dans [McAllester *et al.* 1997].

- **la recherche locale dynamique** : dans cette méthode initiée par [Morris 1993], l'échappement des minima locaux est effectué à travers une redéfinition de la fonction objectif. En effet, plutôt que de compter le nombre de clauses falsifiées par chaque interprétation parcourue, un compteur initialisé à 1 est associé à chaque clause de la formule. Par rapport à ce nouvel élément, la fonction à minimiser est la somme des compteurs de chaque clause [Schuurmans *et al.* 2001]. Ainsi, en début de recherche, cette fonction est équivalente à celle classiquement définie. Cependant, à chaque fois qu'une clause est falsifiée par l'interprétation courante, son compteur est incrémenté de 1. Avec cette modification dynamique de la fonction objectif, les minima locaux sont progressivement atténués jusqu'à disparaître, même en n'effectuant que des flips optimaux.
  L'une des dernières implémentations de ce type d'heuristiques est nommée **SAPS** [Hutter *et al.* 2002]. En plus d'incrémenter les poids des clauses falsifiées par chaque interprétation parcourue, des opérations supplémentaires sont effectuées sur ces compteurs. En effet, lorsque qu'un minimum local est rencontré, avec une probabilité p le poids de chaque clause est divisé par une constante, de manière à accentuer l'importance des dernières configurations rencontrées. Différentes variantes de cet algorithme, qui est l'un des plus efficaces à ce jour, ont également été proposées. En particulier, **RSAPS** est une version dite *réactive* de cette approche : la probabilité p s'auto-ajuste quand une stagnation dans la recherche est détectée, c'est-à-dire quand aucune amélioration de la fonction objectif n'a pu être obtenue lors des derniers flips.

Chapitre 3

Formules minimalement insatisfaisables (MUS)

Sommaire

Lorsque le résultat du test de satisfaisabilité d'une formule propositionnelle s'avère positif, la plupart des solveurs fournissent un modèle de la formule. En revanche, lorsqu'une méthode complète établit qu'une formule est insatisfaisable, elle ne délivre généralement aucune explication quant aux raisons de ce résultat, au-delà de la constatation qu'aucune interprétation ne satisfait la formule. Or, il est possible que seules quelques clauses soient réellement contradictoires, rendant l'ensemble de la formule insatisfaisable. De ce point de vue, la localisation au sein de la formule des raisons de son insatisfaisabilité peut se révéler utile dans nombre de domaines. Illustrons ceci par un exemple. Certaines formules issues du domaine de la conception de circuits électroniques expriment à travers une fonction booléenne des contraintes de routage, d'une manière telle que la formule est satisfaisable si et seulement si le circuit correspond à ses spécifications. À titre d'exemple, les instances dites FPGA [Nam *et al.* 1999, Nam *et al.* 2004] qui ont été utilisées lors des dernières compétitions organisées sur la résolution pratique du problème SAT encodent ce type de problèmes. Si la satisfaisabilité d'une telle formule établit la correction du circuit correspondant, son insatisfaisabilité traduit une erreur dans sa conception. Ainsi, idéalement, l'explication de l'insatisfaisabilité devrait permettre de préciser la ou les parties défectueuses du circuit. Une telle information de nature diagnostique peut être fournie par la localisation de sous-formules insatisfaisables minimales (ou `MUS` pour « *Minimally Unsatisfaisable Subformulae* ») d'une formule propositionnelle mise sous forme CNF.

Dans ce chapitre, nous nous intéressons donc aux travaux réalisés ces dernières années au sujet de la recherche de sous-formules insatisfaisables minimales. Ce domaine a fait l'objet de nombreuses recherches cette dernière décennie et des avancées significatives ont été obtenues. Ce chapitre est organisé de la manière suivante : dans la prochaine section, nous donnons la définition formelle d'un MUS, ainsi que les résultats de complexité dans le pire des cas quant à la recherche de MUS. La section 3.2 est, quant à elle, consacrée aux approches qui ont été proposées pour découvrir un MUS en temps polynomial pour certaines classes de formules. Ensuite, dans le paragraphe 3.3, les meilleurs algorithmes permettant d'approximer un MUS sont décrits et comparés. Les procédures les plus efficaces de minimisation d'une sous-formule insatisfaisable en un MUS sont détaillées dans la partie 3.4 de ce chapitre. Le paragraphe 3.5 expose quant à lui les techniques les plus compétitives permettant de calculer tous les MUS d'une formule booléenne mise sous forme CNF. Enfin, diverses autres études liées aux MUS sont regroupées dans la dernière partie de ce chapitre.

3.1 Définitions et complexité

Les sous-formules insatisfaisables minimales (ou `MUS` pour « *Minimally Unsatisfiable Subformulae* ») d'une formule CNF constituent les explications les plus fines, en termes du nombre de clauses, de l'insatisfaisabilité d'une formule, et désignent quelle(s) partie(s) de la formule doivent être « réparée(s) » pour en restaurer la satisfaisabilité. On définit un MUS comme suit :

Définition 3.1 (Formule Minimalement Insatisfaisable (MUS))
Un MUS Γ *d'une instance SAT* Σ *est un ensemble de clauses tel que*

1. $\Gamma \subseteq \Sigma$;
2. Γ *est insatisfaisable ;*
3. $\forall \Delta \subset \Gamma$, Δ *est satisfaisable.*

La découverte d'un MUS au sein d'une instance insatisfaisable peut s'avérer d'une grande utilité pour de nombreuses applications, mais il s'agit malheureusement d'un problème de lourde complexité algorithmique. En effet, le simple fait de décider si une formule CNF est un MUS est DP-complet [Papadimitriou & Wolfe 1988].

Exemple 3.1
Soit $\Sigma = \{\neg d \vee e,\ b \vee \neg c,\ \neg d,\ \neg a \vee b,\ a,\ a \vee \neg c \vee \neg e,\ \neg a \vee c \vee d,\ \neg b\}$ une formule CNF composée de 8 clauses construites sur 5 variables. Σ est insatisfaisable : en effet, aucune interprétation ne permet de satisfaire l'ensemble de ses clauses. En outre, l'ensemble $\Sigma_M = \{a,\ \neg a \vee b,\ \neg b\}$ est un MUS de Σ. Pour le prouver, il faut d'abord vérifier son insatisfaisabilité, puis la satisfaisabilité de chacune des sous-formules de Σ_M privée de l'une de ses clauses. On comprend alors son appartenance à la classe DP, qui est composée de l'intersection d'un langage de NP et d'un langage de CoNP.

De plus, une formule CNF peut posséder plusieurs MUS. Or, la restauration de la satisfaisabilité d'une formule CNF passe par le retrait d'au moins une clause de chacun de ses MUS. Ceux-ci peuvent être construits sur des ensembles disjoints de clauses, ou en partager certaines.

Exemple 3.2
Soit la formule Σ de l'exemple précédent. Σ_M n'est pas son seul MUS, puisque Σ contient l'ensemble $\Sigma_{M'} = \{b \vee \neg c,\ \neg d,\ a,\ \neg a \vee c \vee d,\ \neg b\}$, lequel consitute un second MUS. Ces 2 MUS, représentés avec le reste de la formule par un diagramme de Venn en figure 3.1 constituent

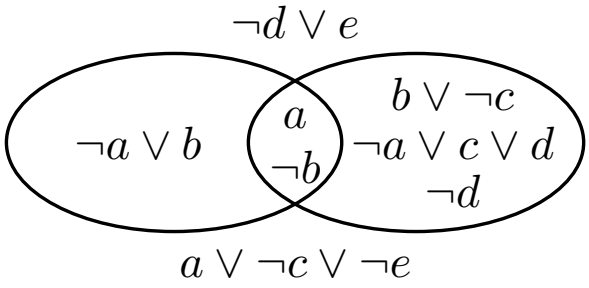


FIG. 3.1 – Ensemble des MUS d'une formule insatisfaisable

l'ensemble des MUS de Σ. Notons que les clauses « a » et « $\neg b$ » appartiennent aux 2 MUS de Σ. Le retrait de l'une d'elles restaure donc la satisfaisabilité de l'ensemble de la formule, puisqu'il « casse » toutes ses sources d'incohérence. Au contraire, la clause « $\neg d \vee e$ » n'appartient à aucun MUS et ne participe donc pas à l'insatisfaisabilité de cette formule.

Une formule CNF peut donc posséder plusieurs MUS. En fait, elle peut même en contenir un nombre qui est exponentiel par rapport à la taille de la formule, comme le montre la propriété suivante :

Propriété 3.1
Dans le pire cas, une formule CNF composée de n clauses possède en effet $C_n^{n/2}$ MUS.

Preuve
Soit une formule CNF insatisfaisable Σ composée de n clauses. *A priori*, n'importe quel sous-ensemble de Σ peut être une cause de l'incohérence de cette formule, et donc un de ces MUS. Cependant, un MUS ne peut clairement pas en contenir un autre, étant donnée sa minimalité. Le problème du nombre de MUS dans le pire cas se ramène donc au calcul du plus grand ensemble des parties de Σ, telles qu'aucune d'entre elles n'en contienne aucune autre. Ceci revient donc au dénombrement du nombre maximal de sous-ensembles incomparables d'un ensemble à n éléments, qui est connu pour être $C_n^{n/2}$. □

Ce nombre potentiellement élevé de MUS au sein d'une même formule (pour 100 clauses, il est de l'ordre de 10^{28}) rend les problèmes qui y sont corrélés d'une complexité algorithmique très élevée. Par exemple, tester si une formule appartient à l'ensemble des MUS d'une instance insatisfaisable a été prouvé Σ_2^p-difficile (une conséquence du théorème 8.2 de [Eiter & Gottlob 1992]).

3.2 Fragments polynomiaux

Nous résumons dans cette section différents travaux portant sur la recherche d'un MUS en temps polynomial, pour des classes de formules spécifiques. Dans [Davydov *et al.* 1998], un algorithme polynomial est proposé pour la recherche d'une sous-formule insatisfaisable minimale ayant une *déficience* de 1, la déficience étant définie comme la différence entre le nombre de clauses et de variables d'une formule. Il est clair que tout MUS possède une déficience positive [Aharoni & Linial 1986]. D'une manière plus générale, il a été montré que la recherche d'un MUS ayant une déficience de k (avec k entier positif) est un problème NP-difficile [Büning 2000].

Cependant, dans ce même article, un algorithme polynomial quand la déficience du MUS est de 2 est proposé. La complexité pour la génération d'une sous-formule minimale ayant une déficience de k a été ensuite établie à $n^{\mathcal{O}(k)}$ [Fleischner *et al.* 2002], où n est le nombre de variables de la formule.

Plus récemment, de nouvelles classes de formules ont été montrées polynomiales pour le problème de la recherche d'un MUS [Bruni 2005]. Cet auteur a établi qu'il est possible extraire un MUS de toute instance satisfaisant une propriété dite « *integral property* », moyennant un nombre polynomial d'opérations par rapport à la taille de la formule. Toutefois, seules quelques classes de formules respectent cette propriété (principalement les formules Horn, Horn-renommables, Horn étendues et bien équilibrées). Assez naturellement, ces instances sont également connues pour être « faciles » pour le problème de satisfaisabilité. Cependant, toutes les classes polynomiales pour SAT ne le sont pas pour la recherche d'un MUS. Citons par exemple la classe des clauses binaires.

Ces études s'avèrent essentielles, mais malheureusement l'ensemble des formules pour lesquelles elles sont applicables demeure restreint, surtout si l'on considère les formules issues de problèmes réels. Dans la section suivante, les meilleurs algorithmes connus pour l'approximation d'un MUS sont présentés, et ce pour une CNF quelconque.

3.3 Techniques algorithmiques pour la recherche d'une sous-formule insatisfaisable

Bien que le problème de recherche d'un MUS soit d'une complexité algorithmique élevée, en pratique plusieurs voies ont déjà été explorées avec succès. Toutefois, la plupart d'entre elles ne permettent pas de garantir la minimalité de la sous-formule insatisfaisable obtenue. On peut considérer ces ensembles de clauses comme des « approximations » de MUS, celles-ci pouvant être de plus ou moins bonne qualité, jusqu'à contenir plusieurs MUS dans certains cas. Nous détaillons dans les paragraphes suivants différentes techniques algorithmiques pour la recherche d'une sous-formule insatisfaisable à partir d'une formule CNF quelconque.

3.3.1 Recherche de MUS adaptative

Une première approche, dite *adaptative* [Bruni 2003], pour l'approximation d'un MUS est basée sur un *score* des clauses calculé par rapport à leur « difficulté », celle-ci étant ici une pondération donnée à une clause en fonction de l'importance présumée de la contrainte qu'elle représente dans la formule. Celle-ci est évaluée par le biais d'un historique lors d'une recherche complète sur la formule, ce processus permettant de donner des indications sur la « difficulté à satisfaire » certaines clauses de cette formule.

Concrètement, cet algorithme s'appuie sur une recherche arborescente basée sur les clauses elles-mêmes. À chaque point de choix, une clause C_s est sélectionnée. Pendant la recherche, les variables de cette clause sont affectées dans le but de satisfaire C_s, de la façon suivante. Considérons dans un premier temps le littéral $\alpha_1 \in C_s$ que l'on satisfait. Une recherche complète est ensuite opérée : si un échec survient, lors du backtrack, le signe de α_1 est inversé, et un autre littéral α_2 de C_s est choisi et affecté pour satisfaire cette clause. De la même manière si un autre conflit survient pendant la recherche, le signe de α_2 est également inversé. Si C_s est une clause binaire, on peut conclure à l'incohérence de la branche courante de l'arbre, sinon un troisième littéral α_3 de C_s est affecté pour la satisfaire. Ce procédé est itéré pour tous les littéraux de la clause sélectionnée à un point de choix donné.

Pré-traitement d branchements de variables sont effectuée dans une recherche complète en comptant le nombre de *visites* et d'*échecs* pour chaque clause. L'ensemble Σ_0 initial est vide.
Base On ajoute à Σ_0 un pourcentage c de clauses de la formule initiale, en classant les clauses par ordre décroissant de degré de difficulté. On note cet ensemble Σ_1. L'ensemble des clauses de la formule non inclues dans Σ_1 est noté C_1.
i-ème itération On effectue b branchements de variables dans une recherche complète sur l'ensemble Σ_i. Un des cas suivants se produit alors :

- Σ_i est insatisfaisable. Cet ensemble est alors un sur-ensemble d'un MUS de la formule. L'algorithme stoppe son exécution et retourne Σ_i.
- Aucune réponse après les b branchements. Une *contraction* est effectuée : on crée un ensemble Σ_{i+1} en prenant un pourcentage c des clauses les plus difficiles de Σ_i.
- Σ_i est satisfait par une interprétation ω_i. Il y a alors *expansion* : on crée un nouvel ensemble Σ_{i+1} en ajoutant à Σ_i un pourcentage c des clauses de C_k, en sélectionnant en premier lieu les clauses falsifiées par ω_i, puis les clauses les plus difficiles.

Algorithme 4 – recherche de MUS adaptative

Une clause est *visitée* pendant l'exploration de l'arbre de recherche si une affection est effectuée de manière à la satisfaire. De plus, on dit qu'on *échoue* sur une clause si une affectation dans le but de la satisfaire a produit une clause vide ou si elle-même est devenue vide suite à une autre affectation. La difficulté d'une clause est évaluée par le nombre de fois où elle a été visitée et le nombre de fois où elle s'est trouvée en échec. En effet, visiter souvent une clause montre son « importance » dans la formule, alors que son nombre d'échecs est un indice significatif quant à la difficulté à la satisfaire. Soit une clause C_j, on note v_j le nombre de visites de cette clause, f_j son nombre d'échecs et l_j la taille de cette clause. Sa difficulté φ est alors donnée par la formule $\varphi(C_j) = \frac{v_j+f_j}{l_j}$.

Les clauses sont choisies dans l'ordre suivant : les clauses unitaires sont d'abord sélectionnées, puis on poursuit l'algorithme de manière à satisfaire les clauses les plus « difficiles », celles-ci étant choisies grâce à la valeur de φ conservée pour chaque clause de la formule CNF. Grâce à ce score, on construit un ensemble Σ de clauses « difficiles », cet ensemble étant le MUS supposé. La satisfaisabilité de Σ est vérifiée de manière polynomiale (en limitant le nombre de branchements effectués). S'il est satisfaisable, on étend cet ensemble avec des clauses de moindre difficulté. Toutefois, les clauses falsifiées (par le modèle trouvé) du reste de l'instance sont ajoutées avec une priorité plus grande, car elles présentent *a priori* une difficulté à être satisfaites avec les clauses de l'ensemble considéré. Si en revanche, aucune réponse n'est fournie pour la satisfaisabilité de ce sous-ensemble de clauses au bout du nombre de branchements imparti, on suppose l'ensemble trop grand, et on le réduit. On poursuit ainsi jusqu'à ce que Σ soit insatisfaisable. Cet ensemble est alors restitué en tant qu'approximation d'un MUS de la formule originale. L'approximation adaptative d'un MUS est présentée avec plus de précisions dans l'algorithme 4.

Le pré-traitement de l'approche adaptative est essentiel, car c'est lui qui permettra de stratifier les clauses en fonction de leur difficulté φ. Cette technique d'approximation de MUS

se révèle efficace sur plusieurs types d'instances, mais, comme évoqué dans [Bruni 2003], ces performances sont grandement tributaires des paramètres b, c et d qui doivent être précisément définis pour chaque formule dans le but de trouver le plus petit sur-ensemble possible d'un de ses MUS.

3.3.2 La méthode « AMUSE »

`AMUSE : A Minimally-Unsatisfaisable Subformula Extractor` est une méthode proposée par Oh, Mneimneh, Andraus et Sakallah dans [Oh *et al.* 2004]. Contrairement à son intitulé, la technique présentée ici ne permet pas dans le cas général de détecter un MUS d'une instance, mais seulement un sur-ensemble d'un MUS. Le principe de cet algorithme consiste à ajouter à chaque clause de la formule une nouvelle variable et à modifier un algorithme complet de recherche de manière à identifier une sous-formule insatisfaisable à l'aide de ces variables auxiliaires.

Soit Σ une formule CNF insatisfaisable. On note $X = Var(\Sigma)$ et Y l'ensemble des variables ajoutées à la formule ; si Σ est composé de m clauses, alors $|Y| = m$. La formule obtenue en ajoutant les variables de Y est notée $\Sigma(X, Y)$. `AMUSE` distingue les variables de X et de Y et les traite différemment durant la recherche : précisément, les variables de X sont gérées de manière classique, c'est-à-dire affectées par le processus de décision ou de déduction, et désaffectées à la suite de conflits lors du « backtracking ». Y est vu comme un ensemble de « *méta variables* » dont le but est de sélectionner les clauses qui forment la sous-formule insatisfaisable. On dit que l'algorithme *force* une variable de Y à 1 pour indiquer que la clause correspondante est candidate pour la sous-formule insatisfaisable en construction.

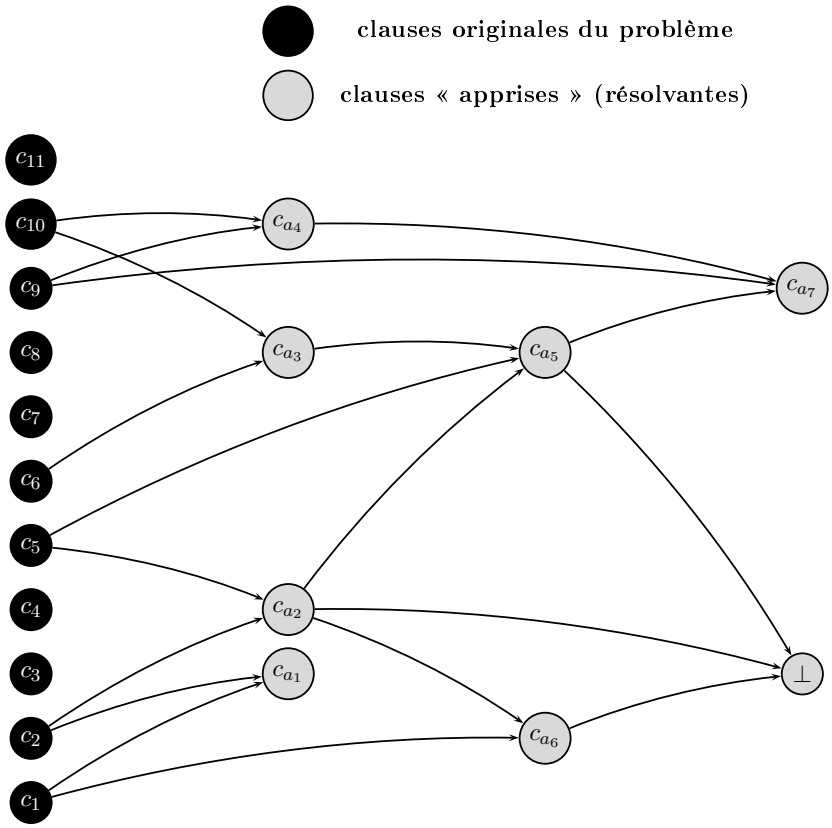
Pour comprendre le fonctionnement de cet algorithme, notons $X_{affecté}$ l'ensemble des variables de X qui ont été affectées soit par décision soit par déduction. Par ailleurs, on note $Y_{forcé}$ l'ensemble des variables Y qui ont été forcées. Initialement, les ensembles $X_{affecté}$ et $Y_{forcé}$ sont vides. Au fur et à mesure de la recherche, l'ensemble $X_{affecté}$ est étendu. Comme la formule Σ considérée est contradictoire, à un moment donné de la recherche, une certaine clause $c_i \in \Sigma$ verra tous ses littéraux appartenant à X falsifiés, obligeant par propagation unitaire l'affectation de son littéral y_i de manière à la satisfaire. Ceci indique que la clause c_i doit être « désactivée » dans le but de trouver un modèle à $\Sigma(X, Y)$. On fait alors un retour sur la variable de X qui a engendré la propagation unitaire de y_i et on affecte cette dernière à 1 (pour indiquer qu'elle participe potentiellement à une sous-formule insatisfaisable).

La recherche est poursuivie jusqu'à ce que la combinaison des variables de X et de Y provoque un conflit appris sous la forme d'une clause c_n. Si la formule donnée en entrée est insatisfaisable, nous sommes assurés d'obtenir une clause apprise ne contenant que des variables appartenant à Y : c_n est de la forme $y_{i_1} \vee y_{i_2} \vee ... \vee y_{i_k}$. Cette clause permet d'identifier une sous-formule insatisfaisable de Σ de taille k.

La méthode `AMUSE` s'avère relativement efficace et permet, sur certaines instances, d'obtenir des sous-formules insatisfaisables qui soient en réalité des MUS. Cependant, comme précisé dans [Oh *et al.* 2004], elle n'est réellement pertinente que lorsque les MUS d'une instance sont d'une taille très faible, ce qui réduit son applicabilité effective.

3.3.3 Détection par apprentissage

Différentes techniques d'apprentissage à partir des conflits ont été développées par le passé, permettant de réduire en pratique l'espace parcouru des interprétations. De telles méthodes peuvent également être utilisées pour déterminer quel sous-ensemble de clauses de l'instance a



Dans cet exemple, les clauses c_1, c_2, c_5, c_6, et c_{10} forment une sous-formule insatisfaisable. Elles sont en effet les clauses « sources » de la preuve par réfutation de la formule.

FIG. 3.2 – Exemple de graphe de résolution

été la source d'un conflit, et peuvent dès lors discriminer certaines clauses comme appartenant à un sous-ensemble insatisfaisable de la formule originale [Zhang & Malik 2003].

Informellement, le principe de ce procédé, baptisé `zCore`, est de conserver un historique sur les clauses responsables du processus d'apprentissage. Si durant la recherche, une clause vide est dérivée par apprentissage, il est possible non seulement de conclure l'instance insatisfaisable, mais également de connaître, grâce à cet historique, quelles clauses de la formule sont responsables de sa production. Or, dans le cas général, si une formule est insatisfaisable, toutes ses clauses ne sont pas nécessairement utiles à l'obtention de la clause vide par apprentissage, ce qui permet de générer une sous-formule insatisfaisable.

Afin de conserver les clauses sources de l'apprentissage d'une nouvelle clause, on utilise une structure de graphe dirigé acyclique appelé *graphe de résolution*. Chaque nœud de ce graphe représente une clause, les racines étant les clauses originales de la formule, les nœuds internes les clauses apprises durant la recherche. Les arcs, eux, symbolisent la résolution : si par exemple, un arc part d'un nœud α vers un nœud β, cela signifie que la clause représentée par le nœud α est l'une des clauses sources de l'apprentissage de la clause représentée par le nœud β. Toute clause représentée par un nœud interne est le fruit de la résolution de tous ses nœuds pères. Un exemple de graphe de résolution est présenté dans la figure 3.2.

Pour déterminer les clauses responsables de l'obtention de la clause vide, il suffit de consi-

dérer les nœuds racines qui sont ancêtres de la clause vide. En effet, les clauses racines qui ne sont pas des ancêtres de la clause vide ne sont donc pas nécessaires pour construire la preuve obtenue d'insatisfaisabilité, et les supprimer ne remet donc pas en cause cette preuve. Ainsi, cette méthode nous assure l'obtention d'une sous-formule insatisfaisable.

Par ailleurs, une fois une sous-formule obtenue, on peut de nouveau lancer une recherche complète avec historique sur cette sous-formule en entrée, dans le but d'obtenir une nouvelle preuve impliquant un nouveau sous-ensemble insatisfaisable de clauses. On peut, en itérant ainsi plusieurs fois cette procédure, produire des sous-formules de plus en plus petites (en termes du nombre de clauses) jusqu'à l'obtention d'un point fixe. Cependant, aucune garantie concernant la minimalité de la sous-formule produite ne peut être apportée.

Un inconvénient majeur de `zCore` réside dans la taille du graphe de résolution. En effet, celle-ci peut-être assez importante, même pour une formule de taille raisonnable. La solution apportée est de conserver le graphe de résolution dans un fichier. Ce choix de stockage ne ralentit pas énormément le processus, car les nœuds du graphe de résolution sont classés dans l'ordre *topologique*. En effet, quand une clause est générée par résolution, toutes ses clauses sources appartiennent nécessairement déjà au graphe. Cependant, quand la clause vide est dérivée, le parcours dans le graphe se fait dans l'ordre topologique inverse, ce qui pose un problème de performance, car le parcours en sens inverse d'un fichier placé sur disque est généralement très inefficace. Avant ce parcours, la solution utilisée dans [Zhang & Malik 2003] est d'inverser le graphe sur le disque en utilisant un espace tampon. Cette technique d'extraction de sous-formules insatisfaisables s'avère relativement efficace, même sur des instances de grande taille. Cependant, la sous-formule obtenue en tant que point fixe ne présente aucune garantie de constituer un MUS. Récemment, une méthode dérivée de `zCore` a été proposée [Gershman *et al.* 2006]. Celle-ci propose en sus une analyse du graphe de résolution dans le but d'éliminer des clauses du sur-ensemble calculé.

3.4 Procédures de minimisation d'une sous-formule insatisfaisable

La plupart des méthodes efficaces en pratique ne fournissent pas un MUS mais une sous-formule insatisfaisable qui n'en constitue qu'une approximation. Toutefois, certaines applications nécessitent l'obtention d'une formule insatisfaisable qui soit minimale.

Une telle application peut par exemple être trouvée dans [Oh *et al.* 2004], où un problème de routage codé en logique propositionnelle clausale est montré insatisfaisable. Tandis que sa seule incohérence n'offre aucune information quant à ses causes, un examen plus précis de la CNF permet la découverte de quatre ensembles minimaux inconsistants de clauses. En particulier, deux d'entre eux spécifient le routage de trois flux à travers deux canaux, un problème ressemblant à celui des « pigeons », clairement insatisfaisable. Dans cet exemple, si l'extraction de sous-ensembles inconsistants de la formule originelle peut fournir des informations importantes sur la façon dont le problème peut être réparé, seule leur minimalité assure que l'information soit bien acheminée après réparation de chacune des sous-formules extraites.

Comme les méthodes heuristiques présentées dans la section précédente ne peuvent garantir la minimalité des sous-formules extraites, plusieurs approches permettant de minimiser ces approximations ont été proposées. Celles-ci sont détaillées dans les paragraphes suivants.

Dans [Huang 2005], une première approche pour la minimisation est présentée sous le nom de `MUP` (*Minimal Unsatisfiability Prover*). Celle-ci utilise le concept de littéraux « sélecteurs de clause » (comme l'approche `AMUSE` présentée dans la section 3.3.2). Cependant, plutôt que d'ajouter n variables pour une formule contenant n clauses, il est ici suggéré de les augmenter

avec seulement $k = \lceil \log(n+1) \rceil$ variables. On ajoute ensuite à chaque clause une combinaison différente de littéraux (notés l_i) en utilisant ces nouvelles variables comme suit : $c_1 \leftarrow c_1 \vee l_1 \vee \cdots \vee l_{k-1} \vee l_k$, $c_2 \leftarrow c_2 \vee l_1 \vee \cdots \vee l_{k-1} \vee \neg l_k$, $c_3 \leftarrow c_3 \vee l_1 \vee \cdots \vee \neg l_{k-1} \vee l_k$, etc.

De cette façon, la formule est prouvée insatisfaisable minimale si et seulement si sa version « augmentée » possède exactement n modèles sur les variables ajoutées, car le retrait de chacune de ses clauses restaure dans ce cas la satisfaisabilité de la formule. Ainsi, le problème de la minimalité est réduit à celui du comptage de modèles [Huang 2005]. `MUP` utilise ensuite des diagrammes binaires de décision (BDD) pour éliminer des variables. En plus de pouvoir prouver la minimalité d'une formule, `MUP` est capable de fournir les clauses à retirer pour en obtenir une, dans le cas où la formule donnée en entrée n'est pas insatisfaisable minimale. Toutefois, l'efficacité de cette méthode dépend largement de la qualité de l'approximation fournie. Ainsi, il est préférable de lui fournir en entrée une approximation obtenue par l'une des méthodes présentées dans la section 3.3, plutôt qu'une formule CNF originale. Comme son nom l'indique, `MUP` permet d'affiner une approximation d'un MUS, mais est peu adaptée pour extraire un MUS d'une large formule insatisfaisable.

Une autre approche dédiée à la minimisation d'un MUS a été proposée conjointement à `zCore` [Zhang & Malik 2003]. L'algorithme, baptisé `zminimal`, consiste à tester la satisfaisabilité de l'approximation privée tour-à-tour de chacune de ses clauses. Si la sous-formule obtenue est insatisfaisable, alors la clause peut être ôtée définitivement de l'approximation. Celle-ci restant incohérente, elle contient encore au moins un MUS. Au contraire, si après le retrait d'une clause l'approximation devient cohérente, alors celle-ci participe nécessairement à toutes les sources d'incohérence de l'approximation, et donc à tous ses MUS. Ainsi, `zminimal` teste les clauses de la formule une-à-une et retourne en fin de procédure un MUS.

Ce mécanisme est également bien connu dans d'autres communautés de recherche. Par exemple, cette approche est dite « destructrice » dans le cadre de la résolution de problèmes de satisfaction de contraintes (CSP), car elle consiste à tester la satisfaisabilité du problème auquel on retire une contrainte, et de l'ôter définitivement si le problème résultant demeure insatisfaisable. Le problème est donc progressivement *détruit* jusqu'à devenir minimal pour l'insatisfaisabilité. D'autres approches de minimisation qui ont été introduites pour le problème de satisfaction de contraintes peuvent aisément être adaptées au cadre clausal booléen. Celles-ci sont décrites dans le paragraphe 7.2.1 page 116.

La conjonction d'une méthode d'approximation avec l'un de ces algorithmes de minimisation permet d'obtenir une formule minimalement insatisfaisable (MUS) pour de nombreuses formules, qu'elles soient aléatoires ou émanent de problèmes réels.

3.5 Approches pour le calcul de tous les MUS

Dans cette section, nous nous intéressons aux méthodes complètes, dans le sens où celles-ci fournissent l'ensemble exhaustif de tous les MUS d'une formule. Bien que dans le pire des cas, cet ensemble possède une cardinalité qui soit exponentielle par rapport à la taille de la formule, il demeure en général d'une taille raisonnable lorsque des formules encodant des problèmes réels sont considérées.

Les premières méthodes générales permettant de calculer tous les ensembles minimalement insatisfaisables d'un système de contraintes sans solution ont été apportées par [Bakker *et al.* 1993, Han & Lee 1999]. Celles-ci correspondent à différentes explorations d'un arbre appelé `CS`, dont le principe est d'énumérer les sous-problèmes inhérents à celui d'origine. Malheureusement, ce type de techniques est limité par l'explosion combinatoire du nombre de sous-formules d'une CNF, et

se montre très peu efficace en pratique.

Une autre approche pour calculer les ensembles minimaux insatisfaisables de contraintes n'a pas été développée pour le cadre clausal booléen, mais dans celui du data-mining, pour la recherche de *patterns* spéciaux [Gunopulos *et al.* 2003]. Cet algorithme, du nom de `Dualize And Advance` (ou `DAA`) a également été utilisé dans le domaine des contraintes de Herbrand, dans un objectif de diagnostic d'erreurs et de déboggage de programmes Haskell [Bailey & Stuckey 2005]. La méthode se base sur la dualité entre les concepts de MUS et de MSS (pour *Maximal satisfaisable Subformula*) ; en effet, alors qu'un MUS est un sous-ensemble insatisfaisable de contraintes tel que chacun de ses sous-ensembles soit satisfaisable, un MSS est un sous-ensemble satisfaisable de contraintes tel que l'ajout de n'importe quelle autre contrainte de la formule le rend incohérent. En fait, l'ensemble des MUS peut être déduit de celui-ci des MSS, car chacun de ces ensemble est un *ensemble intersectant minimal* (« minimal hitting set ») de l'autre.

Définition 3.2 ((minimal) hitting set)
Étant donnée une collection d'ensembles Ω construits sur un domaine D, un ensemble intersectant ou hitting set *de Ω est un ensemble d'éléments de D qui possède au moins un élément commun à chaque ensemble de Ω. Formellement :*

$$H \text{ est un hitting set de } \Omega \text{ si et seulement si } H \subseteq D \text{ et } \forall S \in \Omega, H \cap S \neq \emptyset$$

De plus, un hitting set est dit minimal *si et seulement si aucun élément ne peut lui être ôté sans perdre cette propriété.*

Cette relation est également connue sous le nom de « *hypergraph transversal problem* (*HTP*) ». Ainsi, il suffit de calculer tous les MSS de la formule pour en déduire tous ses MUS.

En pratique, `DAA` calcule les MSS d'une manière directe, incrémentale, en les « grossissant ». Étant donnée une *graine*, sous la forme d'un ensemble satisfaisable de contraintes (initialisé à l'ensemble vide), un MSS est construit en tentant d'ajouter chaque contrainte restante du problème à la graine, et en gardant uniquement celles ne créant pas de conflit. Après avoir traversé toutes les clauses possibles, l'algorithme a construit un ensemble de clauses qui est un MSS, puisque les clauses exclues rendent ce sous-ensemble insatisfaisable. Quand un MSS a été calculé de cette manière, l'ensemble intersectant est calculé de manière à produire un MUS et/ou générer une nouvelle graine. Chaque élément insatisfaisable de l'ensemble intersectant est un MUS ; si un élément est satisfaisable, alors il est utilisé comme graine pour une prochaine itération. [Gunopulos *et al.* 2003] apporte de plus la garantie que cette graine va conduire à la production d'un nouveau MSS. Une fois tous les MSS calculés, l'ensemble intersectant ne produit que des ensembles insatisfaisables, qui sont tous les MUS de la formule, et ne produit donc plus de graine.

Cette méthode a été revisitée récemment dans [Liffiton & Sakallah 2005], où une variante plus efficace est proposée. Dans cette version de l'algorithme qui construit également l'ensemble des MUS à partir de l'ensemble des MSS par *HTP*, les deux ensembles ne sont plus calculés simultanément, mais de manière indépendante, successivement. Cette approche, dont nous proposons par la suite une hybridation avec une recherche locale en vue d'améliorer son efficacité, est décrite précisément dans le paragraphe 5.1.1 page 72.

3.6 Autres travaux autour des MUS

De nombreux autres travaux ont été effectués autour des MUS, que ce soit au niveau de la complexité théorique du problème de leur génération, que de leur application pour des problèmes

théoriques (logique non monotone, fusion de croyances, etc.). Dans cette section, nous décrivons succinctement quelques autres travaux relatifs au calcul de MUS.

Le problème d'optimisation associé à la recherche d'un MUS est celui de trouver le plus petit MUS d'une formule, en termes du nombre de clauses. Plusieurs approches ont ici encore été proposées par le passé. Parmi elles, celle introduite dans [Lynce & Marques-Silva 2004] consiste en une simple énumération de formules. Sommairement, un appel à une méthode d'approximation d'un MUS (cf. Section 3.3) est effectué, dans le but de calculer une borne maximale quant à la taille du plus petit MUS. Toute sous-formule d'une taille inférieure à cette approximation est testée pour la satisfaisabilité, et on conserve la plus petite sous-formule prouvée insatisfaisable. Celle-ci est donc le plus petit MUS de la CNF. Toutefois, le nombre de sous-formules étant combinatoire par rapport à la taille de la formule, cette technique n'est applicable que pour de très petites formules. Une nouvelle approche plus efficace pour la génération du plus petit MUS a par la suite été proposée dans [Mneimneh *et al.* 2005]. Cette méthode est basée sur le principe du « *branch & bound* », puisqu'elle itère des solutions MaxSat afin de calculer une borne inférieure et supérieure sur la taille de ce plus petit MUS. En fonction de ces bornes, la méthode calcule et vérifie la satisfaisabilité de sous-formules spécifiques pour obtenir le « SMUS » (pour *Smallest Minimally Unsatisfiable Subformula*), comme défini dans [Mneimneh *et al.* 2005]. Même si cette approche semble être la meilleure actuellement, elle n'est utilisable en pratique que pour de très petites CNF, restreignant considérablement ses champs d'application.

Une étude plus théorique sur la catégorisation des clauses en fonction de leur rôle dans l'insatisfaisabilité est publiée dans [Kullmann *et al.* 2006]. En particulier, leur appartenance à un ou plusieurs MUS peut déterminer leur classification. Ainsi, on appelle *clause nécessaire* toute clause appartenant à tous les MUS de la formule. Ces clauses appartiennent à toute réfutation par résolution de la formule, et leur retrait restaure la satisfaisabilité de celle-ci. Les *clauses potentiellement nécessaires* appartiennent quant à elles à au moins un MUS de la formule (ou à plusieurs, mais pas à tous les MUS). Leur retrait ne suffit pas à restaurer la satisfaisabilité, mais elles peuvent devenir nécessaires avec le retrait de clauses appropriées dans la formule. Les clauses n'appartenant à aucune de ces 2 catégories sont dites *non nécessaires*, et n'appartiennent à aucun MUS. Le retrait de n'importe quelle combinaison de clauses au sein de cet ensemble garantit de préserver l'insatisfaisabilité de la formule, mais dans ce cas, la complexité de la preuve par réfutation de cette dernière peut devenir beaucoup plus longue. [Kullmann *et al.* 2006] fournit d'autres classifications, plus fines, permettant de mieux comprendre les différents rôles et propriétés que peuvent possèder les clauses au sein d'une instance insatisfaisable.

3.7 Conclusion

Le problème de la génération de sous-formules minimales insatisfaisables (MUS) a fait l'objet de nombreuses études ces dernières années. Des avancées significatives dans la résolution pratique de ce problème ont ainsi pu être observées, à la fois dans l'optique de l'approximation d'un MUS que dans le développement d'algorithmes permettant leur calcul exhaustif. Dans la deuxième partie de ce document, nos différentes contributions à l'extraction de tels ensembles sont dépeintes.

Deuxième partie

Extraction de MUS basée sur la recherche locale

Chapitre 4

Extraction d'un MUS

Sommaire

L'extraction d'un MUS au sein d'une formule CNF permet de fournir à l'utilisateur une explication minimale (en termes d'ensemble de clauses contenues) à l'incohérence d'une formule. Intuitivement, ce calcul permet de localiser une cause de cette inconsistance, étape pouvant s'avérer utile à la *réparation* de la formule. En outre, de nombreuses applications pratiques, en particulier dans le domaine du *model checking* [McMillan 2005] ou de vérification de processeurs [Andraus *et al.* 2006], ont récemment été mises en avant par ces communautés, prouvant la grande viabilité de ce concept.

Dans ce chapitre sont décrites de nouvelles approches algorithmiques permettant l'approximation de formules minimalement insatisfaisables, ou MUS. Ces méthodes tirent leur originalité de l'utilisation de la recherche locale, alors que la plupart des approches connues sont basées sur la procédure DPLL. Nous montrons, par l'établissement de propriétés intéressantes, pourquoi la méthode incomplète peut être d'une grande utilité pour ce problème. De plus, des expérimentations montrent l'efficacité de ces nouvelles approches sur un grand nombre de problèmes.

Les différents travaux présentés dans ce chapitre ont fait l'objet de plusieurs publications : [Grégoire *et al.* 2005, Grégoire *et al.* 2006b, Grégoire *et al.* 2006d].

4.1 La trace, une heuristique pour la détection de zones sur-contraintes

Au-delà de la simple recherche d'un modèle, la recherche locale est depuis quelques années étudiée et utilisée pour résoudre d'autres problèmes autour de SAT. En particulier, la façon flexible dont elle parcourt l'espace de recherche offre un réel attrait pour les heuristiques ; elle permet en effet de parcourir un « échantillon » d'interprétations dans l'espace de recherche. De plus, on suppose les interprétations parcourues pertinentes, puisqu'elles sont dirigées par la fonction objectif qui cherche à minimiser le nombres de clauses falsifiées par l'interprétation courante. C'est donc sa flexibilité et son efficacité dans le parcours de l'espace de recherche qui fait de la recherche locale un outil particulièrement intéressant pour « sonder » les composantes de la formule (variables ou clauses par exemple), et observer leur comportement.

Plus précisément, il a été observé que la recherche locale se montre particulièrement pertinente pour détecter des zones de l'espace de recherche rendues sur-contraintes par la formule CNF. Dans les paragraphes suivants, nous présentons quelques exemples d'applications utilisant cette propriété de la recherche locale.

4.1.1 Approcher une sous-formule contradictoire

Dans [Mazure *et al.* 1998], une heuristique basée sur la recherche locale pour l'approximation de sous-formules insatisfaisables est proposée. Celle-ci se base sur l'inconsistance des MUS qui fournit cette propriété élémentaire :

Propriété 4.1
Soient Σ une formule CNF insatisfaisable, Γ un MUS de Σ, ω une interprétation complète de Σ et c une clause. On a alors :

$$\forall \omega, \exists c \in \Gamma \text{ tel que } \omega \not\models c$$

Comme il n'existe aucune interprétation satisfaisant toutes les clauses d'un MUS, dualement, pour chaque interprétation il existe au moins une clause du MUS qui est falsifiée. À partir de ce résultat, il est raisonnable de supposer alors que, **lors d'une recherche locale, les clauses les plus souvent falsifiées sont celles qui constituent le(s) MUS de la formule**. Lors d'une recherche locale, on espère ainsi pouvoir les discriminer en comptant, pour chaque clause, le nombre de fois où elle a été falsifiée : on appelle cela **la trace**.

Cependant, un problème récurrent à l'attribution de scores est ici présent : à partir de quel niveau considère-t-on les clauses comme faisant partie de la sous-formule insatisfaisable ? Dans l'absolu, cette question ne semble pas posséder de réponse car de nombreux paramètres entrent en jeu. Il s'agit par exemple de considérer le nombre d'interprétations parcourues pendant la

recherche locale, car c'est avant tout de ce nombre que dépendent les valeurs de la trace, ou la taille des clauses, car celles-ci imposent le nombre d'interprétations qui les falsifient.

Mais si ces facteurs restent traitables, il en est un qui influence majoritairement les résultats obtenus et que l'on peut difficilement connaître : la structure intrinsèque de la formule. Celle-ci est en effet d'une importance capitale pour la recherche locale qui peut être conduite vers certains points d'attraction dans l'espace de recherche. Ces points d'attraction sont en grande partie dus aux propriétés structurelles de l'instance, qui peuvent amener certaines clauses d'un MUS à être beaucoup plus souvent falsifiées que d'autres.

On voit donc les difficultés à déterminer le niveau du score à partir duquel on peut obtenir une sous-formule insatisfaisable. Malgré cela, la recherche locale reste un outil efficace pour discriminer les clauses d'une instance participant à son inconsistance.

4.1.2 Une heuristique efficace de choix de variable

La recherche locale peut également être d'un grand intérêt dans le cadre de la recherche complète. Étudiée dans [Crawford 1993] puis [Mazure *et al.* 1998], **la recherche locale peut être utilisée comme heuristique de choix de variables** : on suppose ici que les variables contenues dans les clauses souvent falsifiées sont celles qui permettent de réduire de manière efficace l'arbre de recherche ; puisqu'elles possèdent des difficultés à être satisfaites, elles conduisent logiquement assez rapidement à des contractions à un niveau peu profond de l'arbre. C'est grâce à cette heuristique qu'a été prouvée pour la première fois l'insatisfaisabilité des formules pseudo-aléatoires `AIM-no` [Mazure *et al.* 1998].

4.1.3 Une aide pour la résolution de problèmes de complexité supérieure

On peut, de surcroît, résoudre ou approximer des problèmes situés plus haut dans la hiérarchie polynomiale à partir d'un échec de la recherche locale. Ainsi, dans [Grégoire *et al.* 2002], un algorithme permet, sous certaines conditions, de **produire un sous-ensemble maximal préféré consistant de clauses d'une formule contradictoire** ou **de calculer des modèles préférés d'une formule satisfaisable** à partir de la trace de la recherche locale. Ces traitements « simplifiés » nécessitent tout de même un nombre restreint d'appels à un solveur SAT mais peuvent être effectués de manière pratique grâce aux informations fournies par une recherche locale préalable.

Ces exemples permettent de montrer comment la recherche locale peut être mise à profit dans de nombreux problèmes autour de SAT, et notamment dans le cadre de la recherche de sous-formules insatisfaisables. Cependant, lors du parcours de la recherche locale, il peut être utile de considérer également les interprétations proches de l'interprétation courante dans le but d'approximer les MUS : c'est cette technique qui est étudiée dans le paragraphe suivant.

4.2 Voisinage et clauses critiques

Dans la suite de ce chapitre, il est admis que les formules considérées sont composées exclusivement de clauses fondamentales, c'est-à-dire qu'il n'existe aucune clause contenant un littéral et son complémentaire.

L'heuristique présentée dans [Mazure *et al.* 1998] suggère de compter le nombre de fois où chaque clause a été falsifiée durant une recherche locale, dans l'objectif d'approximer un MUS.

Cependant, un grand nombre d'interprétations falsifient des clauses n'appartenant à aucun MUS, ce qui réduit l'efficacité de l'heuristique en terme du nombre de clauses formant une sous-formule insatisfaisable. Nous proposons donc de ne pas compter systématiquement chaque clause falsifiée, mais de tenir compte du voisinage de l'interprétation pour savoir si la clause va être considérée ou non.

Définition 4.1 (clause unisatisfaite)
Soit c une clause. On dit que c est unisatisfaite par une interprétation ω si et seulement si ω satisfait un et un seul littéral de c.

Définition 4.2 (clause critique)
Soit Σ une formule CNF. Une clause $c \in \Sigma$ falsifiée par une interprétation ω est dite critique si et seulement si l'opposé de chaque littéral de c apparaît dans au moins une clause de Σ unisatisfaite par ω. Ces clauses unisatisfaites relatives à une clause critique c sont dites **liées** *à c.*

Exemple 4.1
Soient la formule insatisfaisable $\Sigma = \{a \vee \neg b,\ \neg b \vee c,\ \neg a \vee \neg b \vee \neg c,\ \neg a \vee b \vee \neg c,\ a \vee c,\ a \vee b,\ \neg a \vee b \vee c\}$ et l'interprétation $\omega = \{a,\ \neg b,\ \neg c\}$.
En considérant l'interprétation ω, la clause falsifiée $\neg a \vee b \vee c$ est critique.
Ses clauses liées sont :

- *pour son littéral* $\neg a$ *:* $\{a \vee b\ ,\ a \vee c\}$
- *pour son littéral* b *:* $\{\neg a \vee b \vee \neg c\}$
- *pour son littéral* c *:* $\{\neg b \vee c\}$

On remarque qu'une clause critique peut avoir plusieurs clauses unisatisfaites liées pour un même littéral.

Propriété 4.2
Soit c une clause critique sous une interprétation ω, alors tout flip tel que c soit satisfaite conduit à falsifier une clause satisfaite par ω.

Preuve
Immédiat : par définition, c est critique si et seulement si elle est falsifiée et l'opposé de chacun de ses littéraux apparaît dans au moins une clause unisatisfaite. Flipper une variable apparaissant dans c conduit donc nécessairement à falsifier une de ses clauses liées. □

On voit donc clairement l'utilité de ces définitions : si une formule Σ possède une clause critique c sous une interprétation ω, alors aucun flip ne permet de satisfaire Σ. Il existe dans le cas général des interprétations qui rendent critiques des clauses n'appartenant pas à l'ensemble des MUS de l'instance, mais, ces derniers étant la cause de l'insatisfaibilité de la formule, on estime que les clauses les plus souvent critiques lors d'une recherche locale appartiennent probablement à l'ensemble des MUS. L'idée est donc d'incrémenter le score des clauses critiques pendant la recherche, plutôt que de simplement incrémenter le score des clauses falsifiées. Une telle technique peut être facilement implémentée dans un algorithme de recherche locale. De surcroît, elle implémente une définition qui est une approximation d'une propriété propre aux clauses appartenant aux MUS.

Propriété 4.3
Soit ω une interprétation optimale pour MaxSat sur une instance Σ. Alors, toute clause falsifiée c par rapport à ω appartient à au moins un MUS de Σ et est critique par rapport à ω. De plus, au moins une clause unisatisfaite liée à c appartient également à (au moins) un MUS de Σ.

Preuve
Chaque clause falsifiée par rapport à ω appartient à un MUS car ω est optimale pour le nombre de clauses satisfaites et au moins une clause par MUS est nécessairement falsifiée. Le fait que toute clause falsifiée par rapport à ω est critique est prouvé par la propriété 4.5 car ω est un minimum global. De plus, si un flip permet de satisfaire une de ces clauses, alors on a nécessairement une autre clause du MUS qui doit être falsifiée. On a donc trivialement au moins une clause liée aux clauses critiques par rapport à ω qui appartient à un MUS de Σ. □

Notre approche est une approximation dans le sens où les clauses critiques et leurs liées sont considérées pendant l'ensemble de la recherche, et pas seulement à la meilleure étape de la procédure MaxSat. En effet, être une clause critique n'est un critère ni nécessaire ni suffisant pour appartenir aux MUS. Comme l'illustre l'exemple suivant, une clause critique par rapport à une interprétation non-optimale pour MaxSat peut n'appartenir à aucun MUS.

Exemple 4.2
Soit $\Sigma = \{a \vee d, \neg a \vee \neg b, \neg d \vee e, f, \neg e \vee \neg f\}$. Clairement, Σ est cohérente. $\neg e \vee \neg f$ est falsifiée par rapport à $\omega = \{a, b, d, e, f\}$ et est critique.

De plus, une clause appartenant à un MUS peut être falsifiée par une interprétation ω sans pour autant être critique par rapport à elle.

Exemple 4.3
Soit la formule minimalement incohérente $\Sigma = \{a \vee d, b, \neg a \vee \neg b, \neg d \vee e, f, \neg e \vee \neg f\}$. $\neg a \vee \neg b$ est falsifiée par $\omega = \{a, b, d, e, f\}$ mais n'est pas critique par rapport à cette interprétation.

On voit donc bien que, si considérer un voisinage partiel de l'interprétation courante à travers les clauses critiques implémente une définition propre aux clauses appartenant aux MUS d'une formule, nous n'avons aucune garantie que toutes soient concernées. Or, pour assurer une forme de *correction* à ce concept, il doit pouvoir être valide pour n'importe quelle clause participant à l'incohérence d'une formule CNF. Ceci est permis par la propriété suivante, qui montre que chaque clause d'un MUS peut être rendue critique par une interprétation particulière :

Propriété 4.4
Soient Γ un MUS, c une clause telle que $c \in \Gamma$, ω une interprétation telle que $\forall c_S (\neq c) \in \Gamma$, $\omega \vDash c_S$. c est alors critique par rapport à ω.

Preuve
Soient Γ un MUS et c une clause telle que $c \in \Gamma$. Par définition, on a $\Gamma \setminus c$ satisfaisable. Soit ω un modèle de $\Gamma \setminus c$. Prouvons que c est critique par rapport à ω :

- *c est falsifiée :* si c n'est pas falsifiée alors Γ admet comme modèle ω ; Clairement, ceci est impossible, car Γ est un MUS.
- *c est critique :* si une variable apparaissant dans c est flippée dans ω, alors au moins une clause de Γ est falsifiée à son tour puisque Γ est insatisfaisable. Cela signifie que cette nouvelle clause falsifiée était unisatisfaite et liée à c. On a donc c critique par rapport à ω. □

Cette propriété garantit, en particulier, l'existence, pour chaque clause appartenant à l'ensemble des MUS, d'une interprétation qui la rende critique et permet de comprendre pourquoi il est intéressant, pour une heuristique approximant des MUS, de considérer le voisinage des interprétations parcourues au travers des clauses critiques.

Malgré tout, la garantie d'**au moins** une interprétation rendant critique chaque clause d'un MUS est un peu faible : l'espace de recherche est en effet exponentiel par rapport au nombre de variables de la formule, et trouver cette interprétation relève *a priori* du hasard. Cependant, sous certaines conditions, on peut minorer de façon plus efficace ce nombre.

Corollaire 4.1
Soient Σ une formule insatisfaisable et $\Gamma \subset \Sigma$ un MUS construit sur un sous-ensemble de variables de Σ. Pour toute clause c appartenant à Γ, il existe au moins $2^{|Var(\Sigma)|-|Var(\Gamma)|}$ interprétations rendant critique c.

Preuve
Soit ω_P une interprétation complète sur $Var(\Gamma)$. On sait (cf. propriété précédente) que pour chaque clause c de Γ, il existe dans cet ensemble d'interprétations au moins l'une entre elles qui rend critique c, indépendamment de l'assignement des autres variables de $Var(\Sigma)$. Or, par dénombrement, on trouve assez facilement qu'il existe $2^{|Var(\Sigma)|-|Var(\Gamma)|}$ interprétations possibles si on fixe les variables de $Var(\Gamma)$. Le corollaire est donc facilement demontré. □

On voit qu'il est, en théorie, exponentiellement d'autant plus « facile » de discriminer un MUS s'il est construit sur un sous-ensemble réduit des variables de l'instance. Ce minorant s'avère intéressant, car toutes les variables de l'instance ne sont pas nécessairement impliquées dans l'un de ces MUS. Nous verrons dans le paragraphe suivant, qu'en pratique, ils sont même relativement souvent construits sur un sous-ensemble réduit de ses variables.

4.2.1 Quelle portion de l'espace de recherche parcourir ?

Si les résultats quant au nombre *minimal* d'interprétations rendant critiques les clauses du MUS sont certes encourageants, dans le pire des cas (un MUS construit sur l'ensemble des variables d'une formule), il peut n'y avoir qu'**une seule interprétation** qui rende critique une ou plusieurs clauses du MUS. Or, pour pouvoir être approximé, l'ensemble des clauses d'un MUS doit pouvoir « être critique » un certain nombre de fois. Comment garantir heuristiquement de parcourir un espace de recherche pertinent pour rendre critiques les clauses appartenant à l'ensemble des MUS ? Ces interprétations possèdent-elles des propriétés particulières permettant de guider la recherche locale ? La propriété suivante permet de mieux comprendre quelle portion de l'espace de recherche est la plus appropriée à parcourir :

Propriété 4.5
Dans un minimum (local ou global), l'ensemble des clauses falsifiées est critique.

Preuve
par l'absurde : on suppose que l'on est dans un minimum mais qu'il existe au moins une clause falsifiée c qui n'est pas critique. Soit ω l'interprétation courante. Comme c est une clause falsifiée mais non critique, alors il existe un littéral $l \in c$ tel que son opposé n'apparaît dans aucune clause unisatisfaite par ω. Soit ω_S l'interprétation ω où on flippe la variable « l ». ω_S satisfait donc la clause c sans falsifier aucune clause satisfaite par ω (puisque l'opposé du littéral l n'apparaît dans aucune clause unisatisfaite par ω). Il apparaît alors que la fonction objectif (i.e. le nombre de clauses satisfaites par l'interprétation courante) est optimisée en passant de ω à ω_S : ω ne représente donc pas un minimum puisqu'il existe un flip permettant d'améliorer

la fonction objectif. De cette contradiction, on conclut que dans un minimum, toutes les clauses falsifiées sont nécessairement critiques. □

Cette propriété suggère que la recherche locale est relativement bien adaptée pour le parcours de l'espace de recherche rendant critique le plus grand nombre possible de clauses : on connaît en effet la tendance de ce type d'algorithme à être « attiré » dans les concavités de l'espace de recherche, et rencontrer des difficultés à en sortir.

En outre, pour comprendre le véritable intérêt de cette propriété, il faut rappeler que pour chaque interprétation parcourue, au moins une clause de chaque MUS est falsifiée. On peut donc dériver assez facilement le corollaire suivant :

Corollaire 4.2
Dans un minimum (local ou global), au moins une clause de chaque MUS est critique.

4.2.2 De l'importance de la fondamentalité des clauses

En préambule de ce paragraphe, nous avons supposé que toutes les clauses des formules sont fondamentales : nous allons voir ici pourquoi cette hypothèse est importante. Considérons la formule trivialement satisfaisable : $\Sigma = \{a \vee \neg b, b \vee c, \neg a \vee \neg b \vee c\}$. On ajoute à cette formule l'ensemble de clause $\Gamma = \{\neg x \vee x \mid x \in Var(\Sigma)\}$.

Soit maintenant l'interprétation $\omega = \{a, b, \neg c\}$. Sous cette interprétation, la clause $\neg a \vee \neg b \vee c$ est non seulement falsifiée, mais devient critique, car liée aux clauses contenues dans Γ. Dans les faits, pour toute formule, l'ajout de l'ensemble Γ rend, pour chaque interprétation, toute clause falsifiée critique : on voit clairement que quelle que soit l'interprétation, les clauses de Γ sont unisatisfaites.

Considérer des clauses tautologiques invalide la propriété selon laquelle un flip sur une variable contenue dans une clause critique falsifie au moins une clause satisfaite sous l'interprétation courante, puisque par construction, les clauses tautologiques sont satisfaites pour toute interprétation. Il apparaît alors qu'avant d'approximer les MUS d'une formule CNF, il faut veiller à lui ôter l'ensemble de ses clauses non fondamentales (pré-traitement polynomial). Cette modification de la formule préserve trivialement la satisfaisabilité de la formule : il y a équivalence entre la formule avant et après pré-traitement.

4.2.3 Conclusions par l'exemple

Exemple 4.4
Soit Σ une formule SAT telle que :

$$\Sigma = \begin{cases} \underline{a} \vee \underline{b} \vee c, & \leftarrow \\ \neg b \vee e, & \\ \neg a \vee \underline{b} \vee c, & \leftarrow \\ \neg a \vee \neg b, & \leftarrow \\ \underline{a} \vee d, & \\ \underline{b} \vee \underline{\neg c}, & \leftarrow \\ \underline{\neg d} \vee e, & \\ \underline{a} \vee \neg b, & \leftarrow \\ \underline{\neg e} \vee \underline{\neg f} & \end{cases}$$

Σ est insatisfaisable et contient un unique MUS. Les clauses de Σ qui composent ce MUS ont été fléchées. Soit $\omega = \{\neg a, \neg b, c, d, e, f\}$ l'interprétation courante explorée par la recherche locale. $b \vee \neg c$ et $\neg e \vee \neg f$ sont les seules clauses de Σ falsifiées par ω. Les littéraux falsifiés par rapport à ω sont soulignés.

Si une heuristique de comptage considère l'ensemble des clauses falsifiées pour approximer ou calculer exactement un MUS, alors ces 2 clauses verront leur compteur incrémenté bien que seule $b \vee \neg c$ appartienne au MUS de Σ. Considérer un voisinage partiel de l'interprétation courante en utilisant le concept de clause critique permet de n'incrémenter que cette dernière clause. En fait, $b \vee \neg c$ est critique par rapport à ω tandis que l'autre clause falsifiée ne l'est pas. En effet, si on flippe c dans ω, alors $a \vee b \vee c$ devient falsifiée, car elle est unisatisfaite et liée par c à la clause $b \vee \neg c$. Un phénomène similaire se produit si b est flippé. Soulignons que ces 2 clauses unisatisfaites appartiennent également à l'unique MUS de Σ. Au contraire, $\neg e \vee \neg f$ peut être satisfaite en effectuant un flip sans falsifier aucune autre clause. Elle ne doit donc pas être considérée pour l'augmentation de son compteur évaluant son appartenance heuristique à un MUS.

4.3 AOMUS, un nouvel algorithme pour l'extraction d'une sous-formule insatisfaisable

Dans cette section est étudié comment, à partir de l'indication heuristique de la recherche locale couplée à la notion de clause critique, on peut dériver un algorithme pratique pour la détection d'un MUS au sein d'une formule CNF. Différents choix essentiellement pratiques et techniques sont discutés. Enfin, une comparaison expérimentale intensive avec les meilleures méthodes connues est présentée, et montre l'intérêt de l'approche.

4.3.1 « Casser » le problème par le retrait itéré de quelques clauses

Nous avons vu dans le paragraphe précédent que prendre en compte le voisinage des interprétations parcourues par une recherche locale peut permettre de faire l'approximation d'une sous-formule inconsistante dans une formule. Cependant, dans le cadre qui nous intéresse, on cherche à détecter ou, à défaut, approximer un MUS d'une formule et non une sous-formule inconsistante. Or, dans le cas général, un formule contradictoire peut posséder plusieurs MUS en son sein. **Comment discriminer ses MUS si la formule en possède plusieurs ?** En effet, chaque clause contenue dans un MUS de l'instance va heuristiquement être critique un certain nombre de fois. Il semble alors difficile de pouvoir affirmer à quel(s) MUS appartient chaque clause. Pour les instances possédant plusieurs MUS, ce problème risque de rendre difficile l'approximation d'un seul MUS dans le cas général.

Parallèlement à ce problème, une difficulté récurrente au score d'éléments apparaît. À partir de la trace, on classe les clauses par ordre décroissant de score : on a donc probablement dans la partie supérieure de ces clauses ordonnées l'ensemble des clauses participant à l'inconsistance de la formule. Cependant, **à partir de quel seuil peut-on supposer la présence de l'ensemble des clauses participant à l'inconsistance de la formule ?** En fait, cette valeur dépend de nombreux facteurs tels que le nombre de flips effectués, la taille de la formule, mais aussi la difficulté intrinsèque de l'instance, celle-ci étant *a priori* impossible à estimer numériquement.

Le choix qui a alors été fait pour pallier ces problèmes est **de ne retirer que les clauses ayant un score très faible** comparativement à l'ensemble. De cette manière, on espère être

« assuré » de ne pas toucher aux clauses participant à l'un des MUS de l'instance. Cela dit, en ne supprimant que ces clauses, si la formule obtenue est probablement inconsistante, l'approximation est grossière, car l'instance est dans le cas général un très large sur-ensemble d'un MUS et peut même toujours contenir plusieurs MUS. On décide alors d'**itérer l'heuristique** en ne retirant à chaque fois que les clauses ayant les scores les plus faibles. **On retire ces clauses à chaque itération jusqu'à obtenir une formule consistante**. On sait alors que l'algorithme a retiré des clauses participant à l'inconsistance, et on considère l'instance obtenue à l'avant-dernière itération : celle-ci est inconsistante, et le retrait de quelques-unes de ses clauses (celles dont les scores ont été les plus bas) restaure la consistance : on a donc heuristiquement approximé un des MUS de la formule initiale. Le retrait itéré des clauses ayant les scores les plus faibles possède en outre de nombreux avantages pour notre problèmatique :

- **casser progressivement certains MUS de l'instance.** Si la formule est localement inconsistante, lors des premières itérations de l'heuristique, il est probable que ce sont les clauses ne participant à aucun MUS de l'instance qui soient ôtées de l'instance (puisqu'elles sont moins souvent critiques que les autres, elles auront les scores les plus faibles). Cependant, au bout d'un certain nombre itérations, la plupart des clauses appartiennent à l'un des MUS de l'instance. Lors de l'itération suivante, bien que tous les scores soient relativement hauts, on retire encore une partie des clauses. Si l'une des clauses supprimées appartient à un ou plusieurs MUS de la formule, alors ce MUS est « cassé » : en retirant l'une de ses clauses, on restaure la satisfaisabilité en son sein, et lors de la prochaine itération, le reste de ses clauses aura logiquement un score beaucoup plus faible, puisque n'appartenant plus à aucun MUS. De plus, si la formule obtenue après l'itération k contient plusieurs MUS liés par un ensemble commun de clauses Θ, il est peu probable que l'une de ces clauses soit retirée : en effet, celles-ci sont les seules à permettre l'obtention du minimum global (quand l'une d'entre elles est falsifiée), la recherche locale essayant d'atteindre ce minimum, il est logique que les clauses de Θ soient les plus souvent critiques. Ce sont donc probablement les clauses contenues dans le plus petit nombre de MUS de la formule qui auront les scores les plus bas et seront retirées pendant les itérations successives. Itérer ainsi permet donc de casser un certain nombre de MUS et d'en approximer un seul avec une relative précision ;
- **modifier progressivement la structure de l'instance.** Intuitivement, cet attrait fourni par la répétition de l'heuristique ne semble pas essentiel ; pourtant, il s'avère être un atout intéressant. En effet, la structure d'une instance fournit implicitement des zones d'attraction pour la recherche locale, ce qui rend parfois plus ou moins rendondantes les interprétations parcourues. Supprimer certaines clauses permet donc de modifier la structure de l'instance et par là même de faire varier l'espace de recherche considéré par la méthode incomplète ;
- **supprimer le « bruit » des clauses n'appartenant à aucun MUS.** Comme nous l'avons vu dans le paragraphe précédent, certaines interprétations peuvent rendre critiques des clauses n'appartenant pas à l'ensemble des MUS (cf exemple 4.3 page 49). Ce fait, que l'on appelle informellement le bruit, est pour notre objectif une difficulté à considérer car les clauses n'appartenant pas à l'ensemble des MUS mais qui sont critiques un nombre de fois suffisant pour ne pas être éliminées lors de l'itération courante sont difficilement discriminables. Le fait de retirer peu à peu les clauses de l'instance permet de réduire ce phénomène : pour le comprendre, il suffit de considérer que chaque clause peut potentiellement être unisatisfaite par un certain nombre d'interprétations et être liée avec une clause rendue par là critique. Le fait de retirer progressivement les

clauses de l'instance permet de reduire le nombre de clauses unisatisfaites et également le nombre de clauses critiques « indésirables ».

4.3.2 Éviter le test répété d'inconsistance

Le retrait successif des clauses aux scores faibles possède de nombreux avantages mais également un inconvénient de taille. En effet, l'algorithme proposé peut se résumer ainsi : calculer un score via la recherche locale pour chaque clause d'une instance, puis retirer celles ayant obtenu le score le plus faible, et ceci jusqu'à ce que l'instance devienne satisfaisable. Ce procédé induit donc un test d'inconsistance à chaque itération, ce qui peut rendre la méthode très lourde à exécuter en pratique. Comment peut-on alors réduire le temps de calcul nécessaire à la réalisation de cet algorithme ?

La solution proposée est très simple : comme des recherches locales sont itérées sur une formule amincie au fur et à mesure, on suppose la formule insatisfaisable si la méthode incomplète ne trouve pas de modèle au bout du nombre de flips imparti, et n'est considérée satisfaisable que lorsque la recherche locale parvient à en exhiber au moins un. Le fait d'utiliser la recherche locale ne fait perdre aucune ressource en temps, puisque si aucun modèle n'est trouvé (la formule est donc vue insatisfaisable), elle calcule *à la volée* le nombre de fois où chaque clause est critique ; dans le cas contraire, son exécution s'achève (la trace n'est d'aucun d'intérêt dans ce cas). L'inconvénient majeur de ce choix est qu'une formule peut être satisfaisable sans que la recherche locale ne trouve de modèle : l'idée est donc de conserver dans une pile les différentes sous-formules produites lors des itérations successives de la recherche locale. Quand celle-ci exhibe un modèle, prouvant la cohérence de la sous-formule courante, la formule précédemment générée est dépilée, et sa cohérence testée par une approche complète de type DPLL. Si celle-ci est effectivement insatisfaisable, alors c'est l'approximation du MUS (voire le MUS exact) recherchée, puisque retirer quelques clauses de cette formule (de surcroît celles obtenant les plus faibles scores) restaure sa satisfaisabilité. Dans le cas contraire, une nouvelle sous-formule est dépilée. Ces opérations sont répétées jusqu'à trouver la plus petite sous-formule incohérente.

Cette technique s'appuyant sur une pile de formules permet de substituer des tests d'incohérence répétés à chaque itération de la RL par une vérification au terme de la recherche heuristique. Or, les intérêts d'une telle méthode sont nombreux. D'une part, il est beaucoup moins coûteux en pratique d'avoir à effectuer des tests `SAT` sur de « petites » formules (ce sont les dernières générées), que de vérifier l'incohérence de la formule dès que la recherche locale supprime quelques clauses. D'autre part, en pratique, la recherche locale se montre efficace pour la satisfaisabilité. Ainsi, la plupart du temps, la dernière étape se résume à la vérification de l'incohérence de l'avant-dernière formule.

Ainsi est gérée la nature incomplète de la recherche locale. L'algorithme proposé est donc complet, dans le sens où pour toute formule donnée en entrée, il retourne une sous-formule insatisfaisable.

4.3.3 Pondérer les valeurs ajoutées à la trace

Les clauses d'une instance ne sont pas toutes sur un pied d'égalité : de leur taille, en particulier, dépend le nombre d'interprétations qui les rendent critiques, puisque d'une part, plus elles sont longues, plus elles sont difficilement falsifiables et d'autre part, plus elles impliquent de clauses unisatisfaites. Le simple comptage du nombre de fois où chaque clause a été rendue critique n'est donc pas satisfaisant. Il convient de pondérer la trace.

Le choix qui a été fait est d'ajouter au score de chaque clause critique le nombre de clauses

Fonction AOMUS(Σ : une **CNF insatisfaisable**) : **une approximation d'un MUS de** Σ
  $pile \longleftarrow \emptyset$;
  Tant que (($RL + Score(\Sigma)$ échoue à trouver un modèle)) **faire**
    empiler(Σ) ;
    $\Sigma \longleftarrow \Sigma \setminus PlusFaiblesScores(\Sigma)$;
  Fait
  Répéter
    $\Sigma \longleftarrow$ dépiler()
  jusqu'à ce que (Σ soit incohérent)
  Retourner Σ ;
Fin

Algorithme 5 – `AOMUS` (*Approximate One MUS*)

qui lui sont liées. Cette solution permet de prendre en compte la taille de la clause puisqu'une clause de grande taille implique nécessairement plus de clauses unisatisfaites pour être critique. L'autre intérêt de cette pondération est de valoriser la « difficulté » de la clause : soient deux clauses c_1 et c_2 d'une instance ϕ telles que $taille(c_1) = taille(c_2)$. Si ces deux clauses sont rendues critiques par deux interprétations, éventuellement identiques, avec r clauses liées à c_1 et s à c_2 ($r < s$), il est normal qu'au score de la clause c_2 soit ajouté un nombre supérieur à celui de c_1 puisqu'en moyenne, le flip d'une variable de c_2 produit plus de clauses falsifiées que le flip d'une variable de c_1. Ceci peut signifier que la deuxième clause est située dans l'intersection de plusieurs MUS, et il semble naturel d'incrémenter son score en conséquence.

Ce choix a été fait dans un souci d'équité entre les clauses de la formule pour la création de la trace de la recherche locale. De nombreuses autres techniques pour le compte peuvent être imaginées : on peut, par exemple, prendre pour chaque variable, le minimum du nombre de clauses unisatisfaisables par cette variable ou essayer de regarder si, après un éventuel flip, les clauses unisatisfaites sont à leur tour critiques. Cependant, un compromis semble le bienvenu entre la précision de la trace et le temps de calcul nécessaire à sa création, aussi le choix d'ajouter le nombre de clauses liées pour chaque clause critique a-t-il été retenu.

4.3.4 Algorithme et résultats expérimentaux

Au vu des observations faites tout au long de ce chapitre et des choix d'implémentation qui ont été faits, on aboutit à la procédure `AOMUS` synthétisée dans l'algorithme 5. Les paramètres de cette procédure sont les suivants. `Walksat` [Kautz *et al.* 2004] a été choisi pour la recherche locale, l'heuristique de choix de variable étant *Rnovelty+* [McAllester *et al.* 1997]. Les autres paramètres ont été choisis à partir de nombreux tests sur divers benchmarks. À la fin de chaque itération de la RL, les clauses dont le score est inférieur à $(min\text{-}score + \frac{\#Flips}{\#Clauses})$ sont supprimées, où *min-score* est le plus petit score d'une clause ; $\#Flips$ et $\#Clauses$ sont respectivement le nombre de flips effectués par la RL, et le nombre de clauses de Σ.

De plus, cette procédure a été raffinée de la manière suivante. Supposons que la sous-formule courante soit effectivement insatisfaisable. Pendant la recherche locale, si l'une des interprétations

			AMUSE		Bruni	zCore		Clauses fausses		AOMUS	
Instance	#var	#cla	#cla	Temps	#cla	#cla	Temps	#cla	Temps	#cla	Temps
fpga10_11	220	1122	561	73.9	-	561	28.5	561	18.3	561	13.1
fpga10_12	240	1344	663	104	-	672	71.2	561	30.1	561	16.9
fpga10_13	260	1586	561	74.6	-	793	167	561	51.7	561	25.9
fpga10_15	300	2130	561	80.6	-	1065	570	561	128	561	44.2
fpga11_12	264	1476	738	928	-	738	112	738	66.8	738	65.5
fpga11_13	286	1742	870	1971	-	871	505	738	180	738	56.7
fpga11_14	308	2030	856	1200	-	1015	1565	738	415	738	69.5
fpga11_15	330	2340	738	1008	-	*time out*		738	568	738	52.1
aim-100-1_6-no-1	100	160	47	0.10	47	47	0.12	47	0.31	47	0.38
aim-100-1_6-no-2	100	160	54	0.07	54	54	0.05	53	0.27	53	0.38
aim-100-1_6-no-3	100	160	57	0.10	57	57	0.09	57	0.29	57	0.40
aim-100-1_6-no-4	100	160	48	0.10	48	48	0.09	48	0.27	48	0.29
aim-100-2_0-no-1	100	200	19	0.05	19	19	0.09	19	0.22	19	0.19
aim-100-2_0-no-2	100	200	39	0.11	39	39	0.09	39	0.42	39	0.77
aim-100-2_0-no-3	100	200	27	0.09	27	27	0.12	27	0.30	27	0.42
aim-100-2_0-no-4	100	200	32	0.10	32	31	0.09	31	0.33	31	0.59
aim-200-1_6-no-1	200	320	55	0.11	55	55	0.09	55	0.48	55	0.63
aim-200-1_6-no-2	200	320	82	0.11	82	81	0.14	81	0.43	81	0.51
aim-200-1_6-no-3	200	320	83	0.06	86	83	0.07	83	0.37	83	0.44
aim-200-1_6-no-4	200	320	46	0.11	46	46	0.10	46	0.35	46	0.39
aim-200-2_0-no-1	200	400	53	0.13	54	54	0.14	53	0.67	53	1.13
aim-200-2_0-no-2	200	400	50	0.12	50	50	0.12	50	0.65	50	1.07
aim-200-2_0-no-3	200	400	37	0.08	37	37	0.23	37	0.39	37	0.49
aim-200-2_0-no-4	200	400	42	0.12	42	42	0.13	43	0.59	42	0.84
aim-50-1_6-no-4	50	80	20	0.06	20	20	0.04	20	0.16	20	0.16
aim-50-2_0-no-4	50	100	21	0.07	21	21	0.14	21	0.21	21	0.22
2bitadd_10	590	1422	929	148	-	815	343	1212	42.7	806	189
barrel2	50	159	79	0.22	-	77	0.04	100	0.35	77	0.36
jnh10	100	850	119	0.09	161	68	0.88	128	9.35	79	42.3
jnh11	100	850	188	0.85	129	121	0.18	117	42.2	111	117
jnh13	100	850	80	0.83	106	57	0.76	64	40.7	75	118
jnh14	100	850	105	0.22	124	91	0.14	76	49.9	72	130
jnh15	100	850	154	0.23	140	119	0.23	108	56.3	119	109
jnh16	100	850	404	0.31	321	298	0.95	314	29.7	283	64.8
jnh19	100	850	173	0.21	122	109	0.18	115	54.8	114	130
jnh20	100	850	141	0.11	120	102	0.23	104	21.7	87	48.9
jnh5	100	850	151	0.09	125	86	0.39	140	12.6	88	46.2
jnh8	100	850	156	0.09	91	90	0.22	162	28.9	69	90.5
jnh302	100	900	23	0.21	-	23	0.10	23	56.2	23	114
jnh303	100	900	188	0.23	-	118	0.31	156	62.1	98	149
jnh304	100	900	32	0.21	-	37	0.14	32	54.9	32	131
jnh305	100	900	104	0.24	-	89	0.17	78	74.3	73	176
jnh306	100	900	289	0.25	-	215	0.46	254	32.7	274	92.7
SGI_30_80_1	480	30464	*time out*		-	*time out*		*time out*		*time out*	
homer06	180	830	415	35.0	-	415	15.9	415	11.0	415	9.04
homer07	198	1012	506	37.7	-	506	21.6	415	12.6	415	10.7
homer08	216	1212	506	52.4	-	606	44.5	554	23.4	415	19.8
homer09	270	1920	832	217	-	960	141	415	93.2	504	60.9
homer10	360	3460	1096	196	-	940	624	1614	148	503	466
homer11	220	1122	561	324	-	561	23.4	561	41.7	561	15.6
homer12	240	1344	672	576	-	672	76.2	708	25.9	564	41.0
homer13	260	1586	793	931	-	793	152	579	67.4	561	76.7
homer14	300	2130	793	999	-	1065	714	561	347	561	28.0
homer15	400	3840	1546	3427	-	*time out*		677	247	561	1048
homer16	264	1476	*time out*		-	738	115	738	78.4	738	61.3
homer17	286	1742	*time out*		-	871	369	870	127	738	68.3
linvrinv2	24	61	51	0.07	-	51	0.05	57	0.17	53	0.11
linvrinv3	90	262	253	0.09	-	250	0.15	250	0.41	244	0.32
linvrinv4	224	689	689	4.15	-	689	22.2	689	21.3	657	19.1
marg5x5	105	512	*time out*		-	*time out*		*time out*		*time out*	

TAB. 4.1 – Extraction d'une sous-formule insatisfaisable : AOMUS vs autres approches

parcourues falsifie une seule clause, alors nous sommes sûrs que cette clause appartient à tous les MUS de la sous-formule courante (puisque son retrait restaure la cohérence de la sous-formule) ainsi qu'à tous les MUS de ses sous-formules insatisfaisables. Cette clause est donc marquée comme *protégée*, et ne peut plus être retirée de la sous-formule calculée par la suite.

À partir de cet algorithme, différentes expérimentations ont été effectuées. Celui-ci a en effet été comparée aux meilleures méthodes existantes applicables dans le cas général (c'est-à-dire non réduite à certaines classes d'instances par exemple). Parmi celles-ci, on compte `zCore` [Zhang & Malik 2003], l'extracteur de sous-formules insatisfaisables associé au solveur `zChaff`, qui est actuellement considéré comme l'un des solveurs SAT les plus efficaces. Nous avons également exécuté la procédure de [Lynce & Marques-Silva 2004], l'algorithme `AMUSE` [Oh *et al.* 2004] et repris les résultats expérimentaux de [Bruni 2003], son système n'étant pas disponible. Bien que la comparaison avec la méthode de Bruni soit difficile d'un point de vue expérimental, il apparaît que sa procédure n'a été testée que sur de petites instances. Nous avons également comparé notre approche avec une adaptation de `AOMUS` où la fonction de score est l'heuristique initiale, telle que présentée dans [Mazure *et al.* 1998] ; on compte donc simplement ici le nombre de fois où une clause est falsifiée. L'ensemble de ces expérimentations a été effectué sur des processeurs Pentium IV, 3 Ghz sous un système Linux Fedora Core 2. Chaque méthode dispose d'une heure de temps CPU pour chaque instance ; si la procédure n'a pas terminé son exécution au bout de ce temps imparti, elle est stoppée et la note « *time out* » est reportée. Dans le cas contraire, le nombre de clauses de la sous-formule retournée est donnée (#cla) avec le temps CPU nécessaire à son obtention (temps), ce dernier étant noté en secondes. Un extrait typique des résultats expérimentaux obtenus est disponible dans la table 4.1, et illustre le bon comportement d'`AOMUS` par rapport à ces différentes techniques, à la fois en terme de temps nécessaire au calcul et dans la qualité des approximations obtenues.

`zCore` se montre très compétitif quand un unique MUS est présent mais échoue à délivrer de bons résultats quand la formule possède plusieurs MUS. En effet, `zCore` ne se concentre pas sur l'extraction d'un MUS, mais cherche des preuves d'inconsistance. `AMUSE` se comporte lui aussi assez bien sur de nombreux benchmarks mais s'avère souvent moins compétitif lorsque des formules plus larges et/ou difficiles sont considérées, à la fois en terme de temps nécessaire au calcul et au niveau de la qualité des approximations obtenues. Par exemple, la série de problèmes industriels `FPGA_11_*` fournit des instances de taille variée, à la fois pour le nombre de variables comme de clauses, mais elles possèdent toutes le même MUS composé de 738 clauses. `AMUSE` et `zCore` éprouvent des difficultés à se concentrer sur ce MUS quand ces problèmes sont grands. En fait, la formule retournée est un sur-ensemble de ce MUS, et le temps d'exécution croît de manière importante avec la taille de l'instance. Au contraire, `AOMUS` fournit toujours ce même MUS dans des temps raisonnables.

Dans [Gershman *et al.* 2006], les auteurs suggèrent que la minimalité n'est guère un critère prépondérant, car de nombreuses applications ne la requièrent pas. L'un des arguments avancés met en avant la taille des explications qui peuvent être fournies ; en effet, il est parfois possible d'extraire une explication d'incohérence d'une CNF qui soit plus petite, en terme de nombre de clauses, qu'une autre pourtant minimale. Comme les approches heuristiques sont en général bien plus efficaces pour extraire de petites sous-formules que les méthodes exactes, il est proposé d'évaluer ces approches sans considérer l'optimalité des ensembles retournés. Ainsi, les auteurs instaurent la notion de *vélocité* pour mesurer l'efficacité de ces méthodes. La vélocité est déterminée à partir du temps mis à calculer l'approximation, avec la taille de cette dernière. Avec ces éléments, on peut facilement calculer le nombre de clauses « éliminées » par seconde, et donc connaître la meilleure méthode. Mais il est clair que cette méthode de comparaison n'est pas toujours satisfaisante ; déterminer la meilleure approche reste un problème en soi, qu'il est

difficile de traiter objectivement.

Malgré les affirmations de [Gershman *et al.* 2006], nombreux sont les domaines où l'on désire avoir la connaissance d'un MUS *exact* et non d'une approximation, afin de localiser de manière optimale les contraintes entrant en contradiction (cf. section 3.4 page 38). Dans cet objectif, nous proposons une nouvelle approche de minimisation qui, jointe à `AOMUS`, permet l'obtention d'un formule minimalement insatisfaisable.

4.4 De l'approximation à l'exactitude : un nouveau procédé de minimisation

Si une sous-formule incohérente permet de localiser plus précisément une source d'« erreur » au sein d'une instance, seule la minimalité fournit un ensemble conflictuel de clauses. En effet, pour « réparer » une formule, au moins une clause de chacun de ses MUS doit lui être retirée. Or, une simple sous-formule incohérente peut également contenir des contraintes qui ne participent pas à l'insatisfaisabilité de la formule. Leur retrait est donc inutile.

La nécessité d'extraire de manière exacte un MUS a conduit au développement de procédés de minimisation, qui prennent en entrée une approximation d'un MUS, pour retourner un MUS. Dans ce paragraphe est décrit un nouveau procédé de minimisation, appelé `fine-tune` et développé conjointement à `AOMUS` pour former un algorithme extrayant un MUS, nommé `OMUS`.

4.4.1 Algorithme basique

Un algorithme naturel permettant de calculer un MUS à partir d'une formule incohérente consiste simplement à retirer l'une de ces clauses et tester sa satisfaisabilité. Si celle-ci est satisfaisable, alors la clause retirée participe nécessairement à l'incohérence de la formule, et doit être conservée. Dans le cas contraire, elle est supprimée définitivement de la formule. En testant itérativement toutes les contraintes d'une CNF de cette façon, l'ensemble des clauses retenues forme un MUS.

Cette approche est similaire à la *méthode destructive*, l'une des techniques de minimisation vers un MUC présentée dans le paragraphe 7.2 page 116. Elle nécessite clairement un nombre linéaire (au nombre de clauses de l'approximation donnée en entrée) d'appels à un solveur complet.

4.4.2 Détection et utilisation des clauses protégées

Cet algorithme basique a été raffiné par l'utilisation d'une recherche locale qui permet de réduire le nombre d'appels à une méthode complète. Avant de tester la consistance de chaque sous-formule, on exécute une recherche locale sur l'approximation de MUS fournie. Pendant cette recherche incomplète, on effectue un calcul de la trace en fonction de nombre de fois où chaque clause a été critique ; en outre, comme `AOMUS`, quand une seule clause a été falsifiée, celle-ci est marquée comme *protégée*. A la fin de la RL, la détection de clauses protégées permet d'éviter certaines vérifications d'appartenance au MUS recherché.

Assez clairement, on peut déduire que **les clauses protégées appartiennent à tous les MUS** de l'approximation, si celle-ci en possède plusieurs. Puisque leur retrait restaure la satisfaisabilité de la formule, aucun MUS ne peut être construit sans elles, car elles sont essentielles à l'incohérence de la formule. Les tests correspondants (de complexité NP) peuvent donc être évités. De surcroît, dans le cas où toutes sont protégées, plus aucun test n'est nécessaire pour conclure à la minimalité :

Fonction fine-tune(Σ : Une approximation d'un MUS) : **un MUS de** Σ
  $RL + Score(\Sigma)$;
  Pour chaque clause $c \in \Sigma$ triée par rapport à son score **faire**
    Si (c n'est pas protégée **ET** ($\Sigma \setminus \{c\}$ est insatisfaisable)) **Alors**
      $\Sigma \longleftarrow \Sigma \setminus \{c\}$;
    Fin Si
  Fin Pour
  Retourner Σ ;
Fin

Algorithme 6 – `fine-tune`

Propriété 4.6
Soit Σ une formule insatisfaisable. Si toutes les clauses de Σ sont marquées comme protégées, alors il s'agit d'un MUS.

Preuve
Immédiat : d'une part, on sait que Σ est incohérente. D'autre part, pour chaque clause c de Σ, $\Sigma\setminus\{c\}$ est satisfaisable (c est protégée puisqu'un modèle a été trouvé à $\Sigma\setminus\{c\}$). C'est clairement la définition d'un MUS. □

Les clauses protégées ne couvrent pas nécessairement l'ensemble de l'approximation, mais elles contribuent en général à réduire significativement le nombre d'appels à un oracle NP et donc à diminuer le temps d'exécution global de la procédure.

En outre, l'ordre dans lequel les clauses sont testées est régi par leur score dans la trace de le recherche locale. Ainsi, on espère éliminer des clauses ne faisant pas partie de tous les MUS de l'approximation dès les premiers appels. La RL étant exécutée régulièrement, on espère ainsi pouvoir déduire de nouvelles clauses protégées et éviter de nouveaux tests NP-complets. Ce procédé est synthétisé dans l'algorithme 6.

Dans [Kullmann *et al.* 2006], une étude vise à catégoriser les clauses d'une instance insatisfaisable en fonction de leur rôle dans son incohérence. En particulier, les clauses qui appartiennent à toute réfutation par résolution sont appelées *nécessaires*, et retirer l'une d'entre elles restaure la satisfaisabilité de la formule. Ce concept de clause nécessaire est lié à celui de clause protégée, puisque ce dernier consiste à marquer, de manière incomplète, les clauses nécessaires de la formule. Celles-ci appartiennent à tous les MUS qu'elle contient, ainsi qu'à tous les MUS de ses sous-ensembles insatisfaisables. Il apparaît que ce raffinement basé sur les clauses protégées se montre très utile, et permet un gain significatif des performances de la procédure.

4.4.3 Expérimentations

L'algorithme `AOMUS` couplé avec la procédure `fine-tune` forme une approche nommée `OMUS` (cf. algorithme 7) capable d'extraire un MUS de n'importe quelle formule incohérente. Nous avons comparé cette technique à d'autres permettant d'obtenir exactement un MUS.

Cependant, à notre connaissance, un seul algorithme a été proposé dans ce but dans le cadre booléen : il s'agit de `zMinimal`, qui est basé sur la procédure `zCore`. En fait, celui-ci calcule

Fonction OMUS(Σ : Une formule insatisfaisable) : **un MUS de** Σ
  $\Sigma \longleftarrow$ `AOMUS`(Σ) ;
  $\Sigma \longleftarrow$ `fine-tune`(Σ) ;
  Retourner Σ ;
Fin

Algorithme 7 – `OMUS` (*Compute One MUS*)

une sous-formule insatisfaisable grâce à ce dernier, puis la réduit pour atteindre un MUS par des tests pas-à-pas, comme notre procédure `fine-tune`. Les 2 approches ont été comparées, et des résultats expérimentaux sont donnés dans la Table 4.2.

Une fois encore, notre approche se montre très compétitive comparée à `zMinimal`, puisqu'elle permet l'extraction d'un MUS dans des temps raisonnables pour la plupart des instances testées, là où `zMinimal` ne peut retourner de résultat en 1 heure de CPU pour les instances les plus grandes et/ou difficiles. Soulignons que les procédures de minimisation de `zMinimal` et d'`AOMUS` sont similaires. Ces résultats peuvent donc s'expliquer comme suit.

Tout d'abord, `AOMUS` retourne des approximations qui sont en général de meilleure qualité que celles fournies par `zCore`. Il s'ensuit que la minimisation d'`OMUS` nécessite un nombre réduit d'appels à des oracles NP et CoNP. De plus, ces approximations sont parfois construites sur un plus petit nombre de variables, faisant des appels plus efficaces à ces méthodes complètes. Comme expliqué plus haut, la première étape de `fine-tune` retourne également un ensemble de clauses protégées, qui n'ont pas à être testées puisque l'on sait par avance qu'elles appartiennent au MUS calculé. Par conséquent, un grand nombre de tests peuvent être évités. En fait, sur certains problèmes, toutes les clauses sont protégées à la fin de l'étape d'« approximation », et plus aucun test n'est requis pour conclure quant à la minimalité de la formule extraite.

En outre, on peut remarquer que les MUS découverts par les 2 approches ne sont pas nécessairement les mêmes (cf. la série `jnh`). En effet, `OMUS` fournit des MUS plus petits (en terme de nombre de clauses impliquées) que `zMinimal`, dans la plupart des cas. Une nouvelle fois, ceci peut s'expliquer par l'usage qui est fait de la recherche locale. Soit une formule Σ composée de 2 MUS, Σ_{M1} et Σ_{M2} avec $|\Sigma_{M1}| < |\Sigma_{M2}|$. On sait que toute interprétation falsifie au moins une clause de chaque MUS. Or, comme l'un des MUS possède moins de contraintes que l'autre, ces clauses sont en moyenne plus souvent falsifiées que l'autre, et leur score est donc plus élevé. La RL recherchant à réduire le nombre de clauses falsifiées par l'interprétation courante, des minima sont souvent rencontrés et le même raisonnement peut être tenu pour les clauses critiques.

Ainsi, de manière « naturelle », la recherche locale aura tendance à ce concentrer sur les plus petits MUS de la formule, contrairement aux autres approches, basées pour la plupart sur une recherche complète.

4.5 Une méthode constructive pour approcher un MUS

4.5.1 Idée générale

L'approche `AOMUS`, bien que basée sur des principes algorithmiques très différents des autres méthodes connues, comme `zCore` ou `AMUSE` (pour ne citer qu'elles) est à leur image une approche

Instance	#var	#cla	`zminimal`		`OMUS`	
			#cla	Temps	#cla	Temps
fpga10_11	220	1122	561	49.8	561	13.7
fpga10_12	240	1344	643	548	561	17.0
fpga10_13	260	1586	643	3406	561	31.9
fpga10_15	300	2130	*time out*		561	68.2
fpga11_12	264	1476	738	105	738	66.3
fpga11_13	286	1742	*time out*		738	84.7
fpga11_14	308	2030	*time out*		738	304
fpga11_15	330	2340	*time out*		738	85.2
aim-100-1_6-no-1	100	160	47	0.13	47	0.39
aim-100-1_6-no-2	100	160	53	0.10	53	0.38
aim-100-1_6-no-3	100	160	53	0.15	57	0.41
aim-100-1_6-no-4	100	160	48	0.14	48	0.29
aim-100-2_0-no-1	100	200	19	0.06	19	0.23
aim-100-2_0-no-2	100	200	39	0.11	39	0.71
aim-100-2_0-no-3	100	200	27	0.13	27	0.45
aim-100-2_0-no-4	100	200	31	0.12	31	0.59
aim-200-1_6-no-1	200	320	55	0.16	55	0.64
aim-200-1_6-no-2	200	320	80	0.18	80	0.53
aim-200-1_6-no-3	200	320	83	0.23	83	0.83
aim-200-1_6-no-4	200	320	46	0.15	46	0.39
aim-200-2_0-no-1	200	400	53	0.21	53	1.21
aim-200-2_0-no-2	200	400	50	0.16	50	1.12
aim-200-2_0-no-3	200	400	46	0.09	37	0.54
aim-200-2_0-no-4	200	400	42	0.16	42	0.88
aim-50-1_6-no-4	50	80	20	0.05	20	0.17
aim-50-2_0-no-4	50	100	28	0.06	21	0.27
2bitadd_10	590	1422	*time out*		716	268
barrel2	50	159	77	0.09	77	0.44
jnh10	100	850	67	0.31	79	42.9
jnh11	100	850	115	0.34	111	118
jnh13	100	850	57	0.85	69	119
jnh14	100	850	88	0.24	72	130
jnh15	100	850	108	0.48	115	111
jnh16	100	850	253	3.52	260	82.1
jnh19	100	850	104	0.30	106	133
jnh20	100	850	99	0.21	87	75.8
jnh5	100	850	86	0.17	86	46.9
jnh8	100	850	85	0.16	67	99.1
jnh302	100	900	23	0.11	23	113
jnh303	100	900	114	0.48	98	149
jnh304	100	900	35	0.17	32	132
jnh305	100	900	89	0.27	73	177
jnh306	100	900	195	1.16	207	99.2
SGI_30_80_1	480	30464	*time out*		*time out*	
homer06	180	830	415	15.6	415	9.37
homer07	198	1012	415	540	415	19.2
homer08	216	1212	552	573	415	24.7
homer09	270	1920	*time out*		415	81.2
homer10	360	3460	*time out*		415	513
homer11	220	1122	561	23.9	561	16.3
homer12	240	1344	561	1426	561	62.3
homer13	260	1586	*time out*		561	78.5
homer14	300	2130	*time out*		561	30.6
homer15	400	3840	*time out*		561	1104
homer16	264	1476	738	328	738	62.9
homer17	286	1742	*time out*		738	87.4
linvrinv2	24	61	51	0.07	51	0.17
linvrinv3	90	262	240	0.83	240	0.63
linvrinv4	224	689	651	667	651	133
marg5x5	105	512	*time out*		*time out*	

TAB. 4.2 – Extraction d'un MUS : `OMUS` vs `zMinimal`

destructive, dans le sens où elle considère l'ensemble de la formule CNF pour en retirer certaines contraintes jusqu'à obtenir une approximation de MUS, ou *core*.

Dans ce paragraphe, nous présentons au contraire une approche dite *constructive* pour approcher un MUS. En effet, celle-ci débute son calcul en considérant la formule vide et ajoute progressivement des clauses jusqu'à ce qu'une sous-formule insatisfaisable, qui est l'approximation calculée, soit générée.

La méthode constructive a été bien étudiée dans le cadre de la minimisation d'un problème inconsistant vers l'une de ces causes irréductibles. Celle-ci est présenté en détail dans la section 7.2.1 page 118, avec d'autres approches de minimisation « classiques ». L'approche constructive n'est en réalité que très peu utilisée en pratique, à cause de sa complexité théorique élevée dans le pire cas. Cependant, ceci n'est vrai que pour les méthodes systématiques (qui utilisent en général la notion de contraintes de transition) ayant pour objectif l'obtention d'un ensemble *minimal*. Ce que nous présentons dans ce paragraphe est une approche heuristique qui effectue l'approximation d'un MUS, et n'est donc pas soumis à ce mauvais résultat de complexité théorique. Tout comme `AOMUS`, celle-ci se base sur la propriété 4.1, corollaire à la définition même de MUS, qui stipule qu'au moins une clause d'un MUS est falsifiée par toute interprétation. Cette propriété implique que si au sein d'une formule incohérente, on connaît un ensemble Γ contenant toutes les clauses d'un MUS de cette formule sauf une, celle-ci peut être déduite en considérant la clause que falsifient tous les modèles de Γ. Plus précisément, plusieurs clauses peuvent être falsifiées par cet ensemble. Dans ce cas, chacune d'entre elles peut compléter Γ pour en faire un MUS. Ce même raisonnement peut assez clairement être appliqué à des sous-ensembles de Γ. En calculant l'ensemble de ses modèles et en recherchant dans l'ensemble de la formule un ensemble minimal de clauses permettant de falsifier cet ensemble de modèles ; un MUS de la formule peut ainsi être formé par l'union de cet ensemble minimal de clauses avec Γ. Malheureusement, pour des raisons d'efficacité, il n'est pas possible d'énumérer tous les modèles d'une sous-formule satisfaisable, car cet ensemble peut clairement être exponentiel. Ainsi, la voie heuristique choisie est présenté dans le paragraphe suivant.

4.5.2 À la recherche de modèles « représentatifs »

Comme l'avons vu, un MUS peut être construit via l'énumération de modèles d'une CNF. Cependant, cette opération étant très lourde d'un point de vue calculatoire, il nous faut nous tourner vers une voie heuristique afin de calculer un sous-ensemble de ces modèles qui soit représentatifs, c'est-à-dire qui falsifient uniformément les parties manquantes du MUS en construction.

Supposons que nous ayons un ensemble de clauses Γ qui soit un sous-ensemble de MUS. Une recherche locale est exécutée sur Γ, et on considère tous les modèles découverts. Les modèles sont alors étendus à la partie restante de la CNF, et ses clauses falsifiées par rapport aux modèles découverts voient leur score associé (initialisé à 0) incrémenté. Comme au moins une clause de chaque ensemble pouvant compléter Γ pour former un MUS est falsifiée, si l'échantillon de modèles de Γ fourni par la RL est pertinent, il suffit de sélectionner les clauses ayant obtenu les plus hauts scores et de les ajouter à l'approximation du MUS. Ainsi, à la fin de la procédure incomplète, Γ est augmentée de ces clauses ayant obtenu des hauts scores, et on réitère la recherche d'un ensemble de modèles à ce nouvel ensemble de clauses.

Ces opérations sont répétées jusqu'à ce qu'aucun modèle ne puisse être obtenu à la formule régulièrement grossie Γ. Cependant, à cause de la nature incomplète de la recherche locale, cet ensemble de clauses n'est pas garanti insatisfaisable. Pour assurer la complétude de l'algorithme, quand la recherche locale échoue à trouver un modèle, une procédure systématique de type DPLL est exécutée sur cette formule. Si Γ est prouvée insatisfaisable, elle est retournée comme

```
Fonction construct_core(Σ : une CNF insatisfaisable) : une approximation d'un
MUS de Σ
    appel_DPLL ← faux ;
    Γ ← ∅ ;
    Γ ← recherche_clauses_protegees() ;
    Σ ← Σ\Γ ;
    Tant que (vrai) faire
        Si (appel_DPLL) Alors
            Si (DPLL(Γ)==UNSAT) Alors
                Retourner Γ ;
            Sinon
                graine ← modele_trouvé() ;
                appel_DPLL ← faux ;
            Fin Si
        Sinon
            graine ← génération_aléatoire() ;
        Fin Si
        S_ω ← RL_recherche_modeles(Γ, graine) ;
        Φ ← choisit_p_plus_falsifiees(Σ, S_ω, p) ;
        Γ ← Γ ∪ Φ ;
        Σ ← Σ\Φ ;
        Si (S_ω == ∅) Alors
            appel_DPLL ← vrai ;
        Fin Si
    Fait
Fin
```

Algorithme 8 – Construction d'un *core*

approximation de MUS de la formule ; sinon, un modèle ω, qui représente un certificat de la consistance de Γ, est découvert. Dans ce cas, ce modèle est utilisé comme *graine* pour la RL afin d'effectuer un nouveau calcul d'échantillon de modèles tout en s'assurant d'en obtenir au moins un. De plus, le précédent lancement de la RL n'ayant pas découvert de modèles, il est très probable que cette partie de l'espace de recherche n'a pas été explorée. Comme celui-ci possède au moins un modèle, la graine fournie permet de diriger la RL vers cette zone éventuellement fertile en modèles.

Ainsi, la première itération de l'algorithme débute avec la formule vide, en considérant toutes les interprétations pour calculer les scores des clauses de la formule (puisque celles-ci sont toutes modèles de cette formule particulière). Fort heureusement, comme mentionné précédemment, il est bien connu que la recherche locale est une heuristique puissante pour localiser les parties sur-contraintes de la formule [Mazure *et al.* 1998], contenant en général les intersections éventuellement entre les MUS de la formule. Ces clauses sont alors choisies, et par la suite seules les clauses falsifiées par rapport à leurs modèles sont ensuite ajoutées. De plus, la notion de *clause protégée*, définie dans la section 4.4.2, est utilisée puisqu'elle permet de détecter les clauses appartenant à tous les MUS de la formule.

Avec ces caractéristiques, l'algorithme constructif proposé est complet pour le problème de l'extraction d'une sous-formule insatisfaisable de toute CNF. Cette approche est décrite dans l'algorithme 8.

À chaque itération de l'algorithme de nouvelles clauses sont ajoutées à Γ. La propriété suivante assure qu'aucune clause redondante avec Γ ne peut lui être ajoutée.

Propriété 4.7
Soit Σ une formule CNF, Γ une sous-formule de Σ, et c une clause. $\forall c \in (\Sigma \backslash \Gamma)$, c ne peut être ajoutée à Γ par la méthode constructive si $\Gamma \models c$.

Ainsi, il n'est pas possible pour notre politique constructive d'ajouter à l'approximation de MUS des clauses qui lui sont redondantes, c'est-à-dire des clauses qui « couvrent » le même espace de recherche. Cette caractéristique n'est pas partagée par les méthodes destructives.

Cependant, plusieurs clauses (p dans l'algorithme 8) sont ajoutées après chaque itération, et si certaines d'entre elles sont redondantes en conjonction de Γ, elles peuvent tout de même être présentes dans l'approximation construite. Néanmoins, dans notre cas, ce phénomène n'est possible que pour les clauses ajoutées simultanément et ce nombre de clauses étant un argument de l'algorithme, il peut être paramétré par l'utilisateur. S'il est petit, un plus grand nombre d'itérations peuvent être nécessaires à la construction du *core*, mais celui-ci sera probablement d'une grande qualité. Au contraire, avec un grand nombre de clauses ajoutées, seules quelques itérations suffisent pour l'obtention d'une approximation, mais celle-ci sera de piètre qualité. Encore une fois, les approches constructives semblent intéressantes, car elles fournissent un paramètre représentant un compromis entre la qualité de l'approximation et le temps passé à son calcul.

Exemple 4.5
Soit Σ une formule CNF contenant 13 clauses impliquant 4 variables telle que :

$$\Sigma = \begin{cases} c_1 : (\neg a \vee c \vee d) & c_2 : (a \vee b \vee c) \\ c_3 : (\neg a \vee \neg b \vee c) & c_4 : (c \vee \neg d) \\ c_5 : (\neg a \vee \neg c \vee d) & c_6 : (\neg b \vee d) \\ c_7 : (a \vee \neg c \vee d) & c_8 : (a \vee \neg b \vee \neg c) \\ c_9 : (b \vee \neg c) & c_{10} : (\neg b \vee \neg c \vee \neg d) \\ c_{11} : (a \vee \neg c \vee \neg d) & c_{12} : (a \vee c \vee d) \\ c_{13} : (\neg a \vee b \vee \neg d) & \end{cases}$$

L'espace de recherche induit par ces 4 variables est représenté à travers une table de Karnaugh en figure 4.1, avec les clauses falsifiées par les différentes interprétations possibles de Σ. Notons tout d'abord que Σ est clairement inconsistante, puisque toute interprétation est falsifiée par au moins une clause de la CNF. Par exemple, $\omega_1 = \{\neg a, b, c, d\}$ falsifie les clauses c_8 et c_{10}.

On peut remarquer que certaines interprétations ne falsifient qu'une unique clause. Par exemple, c_4 est seulement violée par l'interprétation $\omega_2 = \{\neg a, b, \neg c, d\}$. En conséquence, cette clause est *nécessaire* [Kullmann *et al.* 2006] puisque sans elle, ω_2 deviendrait un modèle de la formule. Cette clause participe donc à toutes les sources d'inconsistance de Σ et appartient à tous ses MUS. Σ possède 2 autres clauses nécessaires, qui sont c_1 et c_9. En pratique, grâce au concept de clause protégée, ce type de clauses est en général facilement détecté car elles représentent des *minima* attractifs pour la procédure incomplète. Supposons qu'à la première itération de notre approche, ces 3 clauses soient marquées comme protégées et utilisées dans l'approximation initiale Γ. En conséquence, seuls les modèles de cette CNF sont maintenant considérés pour calculer le score des autres clauses de Σ. L'ensemble des modèles de $\Gamma = c_1 \cup c_4 \cup c_9$ est représenté en figure 4.2 par des cellules blanches, les interprétations falsifiées par cette CNF étant en gris. On peut noter que les clauses c_3 et c_{13} ne peuvent dès lors plus être ajoutées à Γ, puisqu'elles ne sont

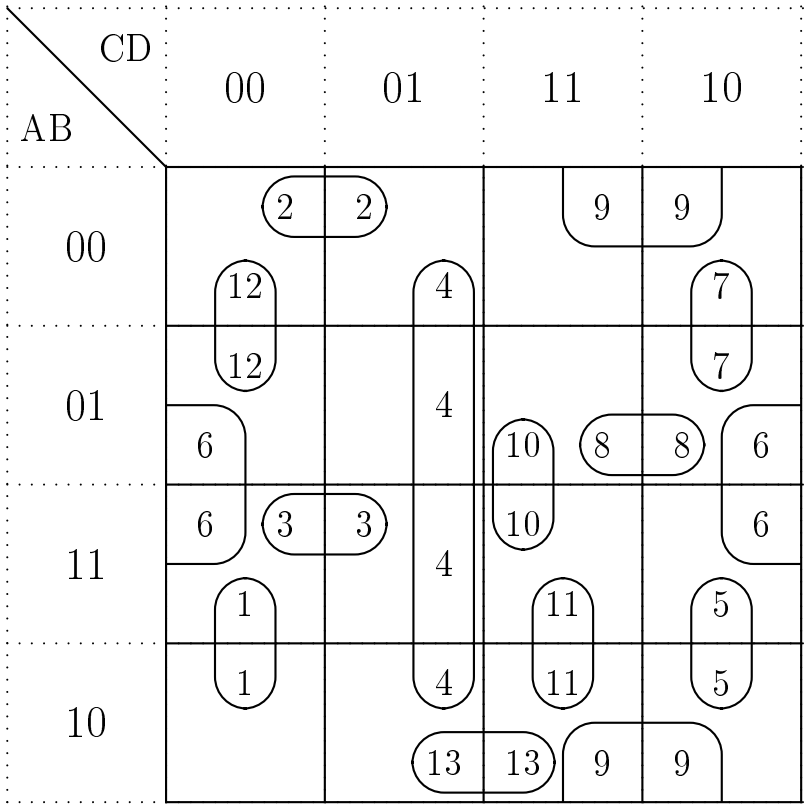


FIG. 4.1 – Table de Karnaugh de l'exemple 4.5

falsifiées que lorsque au moins l'une des clauses nécessaires inclues dans Γ l'est également. Ces 2 clauses n'appartiennent à aucun MUS de Σ. Au contraire, les preuves d'inconsistances requises par la plupart des méthodes destructives connues peuvent utiliser ces clauses pour effectuer une propoagation unitaire par exemple, et être contenue dans l'approximation retournée.

Lors de la deuxième itération qui ne considère que les clauses falsifiées de Σ par rapport à l'échantillon de modèles de Γ calculé, si c_{12} est candidat à l'ajout, cela empêchera également c_2 et c_6 d'être ajoutées par la suite, puisque ces clauses ne peuvent être falsifiées par rapport à un modèle de $\Gamma \cup \{c_{12}\}$.

Toutefois, pour des raisons pratiques, après chaque itération, plusieurs clauses sont ajoutées simultanément, et si par exemple c_6 et c_{12} obtiennent de hauts scores, les deux peuvent être ajoutées à Γ.

4.5.3 Vers la construction de MUS spécifiques

Les approches visant à extraire un MUS, ou une approximation, d'une formule CNF insatisfaisable, en localise un arbitrairement, la plupart du temps basé sur la preuve de réfutation obtenue. Cependant, une formule propositionnelle peut contenir plusieurs MUS (dans le pire cas un nombre exponentiel), et l'utilisateur pourrait préférer extraire un MUS contenant des informations, ou clauses, spécifiques. Un tel calcul peut se montrer extrêmement utile en pratique : si un circuit FPGA n'est pas routable, il permet par exemple de vérifier si une partie spécifique du circuit est impliquée dans son défaut de routage. Les approches « classiques » ne permettent

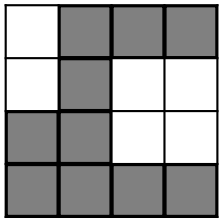


FIG. 4.2 – Modèles de la CNF formée des clauses nécessaires de Σ

pas d'effectuer un tel calcul, qui ne peut être obtenu qu'avec une approche complète, c'est-à-dire une approche calculant l'ensemble exhaustif des MUS d'une formule.

La politique constructive que nous avons présentée dans les paragraphes précédents offre un tel choix à l'utilisateur, même si on ne peut garantir que les clauses voulues apparaissent effectivement dans un des MUS de l'approximation calculée. Pour cela, il suffit simplement d'utiliser l'algorithme proposé, en commençant le calcul avec les clauses désirées plutôt qu'avec la formule vide. Notons Δ l'ensemble de clauses sélectionné par l'utilisateur pour construire un *core*. Comme nous l'avons montré, les clauses redondantes avec l'approximation en construction ne peuvent lui être ajoutées. Ainsi, en imposant le premier ensemble de clauses à Δ, aucune autre contrainte qui lui est redondante ne peut être considérée par la procédure.

Bien que la présence de ces clauses comme *graine* de la procédure constructive assure leur présence dans l'approximation obtenue, on ne peut malheureusement garantir leur participation à son inconsistance, comme le montre l'exemple suivant.

Exemple 4.6
Considérons la formule CNF de l'exemple 4.5 et supposons que l'utilisateur veuille obtenir un *core* impliquant l'ensemble de clauses $\{c_1, c_3, c_8, c_{12}\}$.

Tout d'abord, c_1 est une clause nécessaire, et appartient donc à tous les MUS de Σ. Au contraire, c_3 étant une clause qui n'est *jamais nécessaire* (en suivant la terminologie de [Kullmann *et al.* 2006]) et n'appartient donc à aucun MUS. Par conséquent, le choix de l'utilisateur implique une construction non-optimale pour l'approximation. Enfin, c_8 et c_{12} sont des *clauses potentiellement nécessaires* et appartiennent à au moins un MUS de la CNF. En ajoutant ces clauses avant la première itération et en supposant les deux clauses nécessaires détectées, Γ contient alors l'ensemble de clauses $\{c_1, c_3, c_4, c_8, c_9, c_{12}\}$. En utilisant cette approximation partielle, les seules clauses pouvant être falsifiées par rapport à ses modèles sont c_5, c_6, c_{10} et c_{11} ; certaines d'entre elles vont donc être ajoutées à Γ pour former l'approximation finale. Si c_5 et c_{11} sont choisies, le *core* résultant ne peut être minimisé qu'en supprimant c_3. Ainsi, toutes les clauses sélectionnées par l'utilisateur (excepté cette dernière) participent effectivement au MUS obtenu. Malheureusement, le choix potentiel de c_6 et c_{10} rendrait c_8 redondante (par rapport à l'approximation) et n'appartiendrait pas à tous ses MUS.

Cet exemple montre que l'approche constructive permet de diriger le calcul vers un MUS spécifique, même s'il est impossible d'assurer que les clauses choisies fassent effectivement partie de tous les MUS de l'approximation. Nous pensons que ce calcul incomplet est un bon compromis face à la complexité élevée d'un tel problème. En effet, en supposant $P \neq NP$, on ne peut s'attendre au développement de procédures complètes efficaces, puisque vérifier si une formule appartient à l'ensemble des MUS d'une formule CNF est Σ_2^p-difficile (une conséquence du théorème 8.2 de [Eiter & Gottlob 1992]).

4.5.4 Expérimentations

Dans le but de valider expérimentalement la méthode constructive, nous avons implémenté les idées décrites dans les paragraphes précédents et comparé cette implémentation avec les meilleures approches connues pour l'approximation d'un MUS : `zCore` [Zhang & Malik 2003], `AMUSE` [Oh *et al.* 2004] et `AOMUS`, présentée précédemment. La méthode de [Bruni 2003] n'étant pas disponible et ayant été expérimentée uniquement sur des problèmes simples, n'a pas été considérée dans nos travaux expérimentaux. La table 4.3 reporte un échantillon de ces expérimentations. Comme cas d'étude, nous avons utilisé la plate-forme `ubcsat` [Tompkins & Hoos 2005] comme recherche locale, avec l'heuristique `RSAPS` [Hutter *et al.* 2002]. Pour l'approche complète, le solveur `minisat` [Eén & Sörensson] a été choisi puisqu'il est reconnu comme l'une des méthodes systématiques les plus efficaces pour SAT. Toutes les expérimentations ont été conduites sur des processeurs Pentium IV, 3 Ghz avec 1 Go de mémoire vive, et pour chaque problème testé, une limite de 10 000 secondes a été respectée. Au-delà de cette limite, la note « *time out* » est reportée.

Tout d'abord, seul `AMUSE` ne parvient pas à fournir avec précision l'unique MUS des formules `{23,42}.shuffled`, qui nous viennent du domaine du *model checking*. Celles-ci encodent la vérification formelle du microprocesseur Sun PicoJava II et comme indiqué par [McMillan 2005], le MUS découvert permet d'identifier les composants du système pertinents pour la propriété vérifiée.

Nous avons également effectué des tests expérimentaux sur le problème de coloriage de graphes (`{3,4}col-*`). Sur ces CNF, l'obtention d'une sous-formule insatisfaisable permet de localiser un sous-graphe qui ne peut être colorié avec le nombre donné de couleurs, et rend le problème complet sans solution. Sur ces problèmes, l'approche constructive s'avère clairement la mieux adaptée quant à la taille de l'approximation obtenue. En outre, ce résultat est obtenu en un temps raisonnable, bien que généralement `AMUSE` retourne son approximation, plus grossière, en un temps plus court. Par exemple, sur le benchmark `4col100_9_8`, `constructMUS` extrait une approximation de 1445 clauses en un peu moins de 6 minutes alors que `zCore` en retourne une composée de 1618 clauses en plus d'une heure et demi. `AMUSE` et `AOMUS` quant à eux, permettent l'obtention de sous-formules insatisfaisables de 1510 et 1502 clauses en environ 1 et 23 minutes, respectivement. Notons que sur cette famille de benchmarks, la construction est systématiquement plus précise que l'utilisation d'une méthode destructive.

En outre, de nombreux problèmes, tels que la famille `rope-*`, sont globalement insatisfaisables ; c'est-à-dire qu'il s'agit de MUS. Sur ces formules, les approches basées sur DPLL n'effectuent qu'un seul appel au solveur complet, et toute clause participant à la preuve d'inconsistance, l'algorithme stoppe son exécution. Néanmoins, la résolution de ces problèmes est en général difficile, car leurs preuves d'inconsistance sont très grandes. L'analyse nécessaire de cette preuve par les méthodes reposant sur DPLL (et en particulier `zCore` avec son traitement du graphe de réfutation) peut alors alourdir considérablement ces procédures. Quelle que soit la méthode basée sur la RL (`AOMUS` ou `constructMUS`), ce problème est en pratique, très souvent évité. En effet, toutes les clauses de ces instances sont nécessaires, et la recherche locale réussit très souvent à les prouver protégées d'une manière peu coûteuse en ressources. Un « simple » test d'inconsistance est alors effectué, ce qui explique leurs meilleurs résultats que ceux de `zCore` et `AMUSE` (cf par exemple `Urquhart-s3-b1`).

De manière générale, nos résultats montrent que cette première implémentation de `constructMUS` fournit des résultats très satisfaisants sur de nombreuses CNF, comparée aux meilleures approches connues (cf par exemple `hwb-*, gt-*`). En revanche, `AOMUS` semble rester une méthode de choix pour certains problèmes industriels (cf. par exemple `homer`), alors que les

Instances			zCore		constructMUS		AMUSE		AOMUS	
Nom	#var	#cla	#cla	temps	#cla	temps	#cla	temps	#cla	temps
23.shuffled	198	474	221	0.05	221	4.83	230	0.04	221	2.05
42.shuffled	378	904	421	0.08	421	5.00	434	0.09	421	3.80
3col20_5_5	40	176	46	0.04	42	19.6	60	0.06	46	13.8
3col20_5_6	40	176	43	0.07	40	18.3	46	0.06	66	7.15
3col20_5_7	40	176	40	0.05	40	18.5	60	0.06	40	9.39
3col20_5_8	40	176	52	0.05	40	18.6	46	0.06	55	8.96
4col100_9_5	200	1806	*time out*		1462	335	1566	56.4	1512	1550
4col100_9_6	200	1806	1458	7030	1451	351	1505	79.9	1506	5222
4col100_9_7	200	1806	1596	3165	1461	352	1472	38.7	*time out*	
4col100_9_8	200	1806	1618	6694	1455	356	1510	71.5	1502	2638
4col100_9_9	200	1806	1556	5202	1458	377	1527	45.1	1484	2973
5cnf...30f4	30	419	316	0.65	237	127	369	0.13	340	26.7
5cnf...40f1	40	608	601	0.86	397	189	593	0.34	418	39.5
Urquhart-s3-b1	43	316	58.3	58.7	316	7.75	316	340	316	327
am_4_4	433	1458	929	6.3	944	25.6	902	3.95	929	29.2
am_5_5	1076	3677	2046	7614	2244	62.5	2046	765	2140	76.4
ca004	60	168	114	0.07	108	4.52	117	0.06	108	0.50
ca008	130	370	276	0.17	255	4.83	283	0.08	255	1.93
ca016	272	780	584	0.72	559	8.73	646	0.30	559	5.33
ca032	558	1606	1176	2.66	1230	14.4	1316	3.46	1281	52.9
ca064	1132	3264	2421	4.29	2793	20.2	2687	22.9	2613	141
ezfact16_1	193	1113	41	0.09	169	20.4	100	0.10	41	383
ezfact16_2	193	1113	47	0.07	169	20.8	55	0.09	41	382
gt-012	144	1398	1356	7.4	1124	171	1219	1.40	1205	162
gt-014	196	2289	2113	53.4	1940	301	1713	5.27	1989	581
hanoi4u	1312	16856	*time out*		7605	300	6434	9670	*time out*	
homer06	180	830	415	14.2	461	50.2	415	38.1	415	9.66
homer07	198	1012	506	19.2	531	62.3	506	38.6	415	13.9
homer08	216	1212	606	39.5	650	79.0	506	54.4	415	20.3
hwb-n20-01	134	630	624	1248	624	93.6	627	58.8	624	351
hwb-n20-02	134	630	625	1270	624	142	628	50.6	625	508
hwb-n20-03	134	630	626	272	622	58.6	626	180	625	191
linvrinv2	24	61	51	0.05	50	6.03	51	0.05	53	0.10
linvrinv3	90	262	250	0.15	239	16.9	253	0.09	244	0.30
linvrinv4	224	689	689	16.4	653	42.3	689	4.46	657	14.2
mm-2x2-5-5-s.1	324	2064	1290	136	1857	152	1791	40.4	2064	259
rope_0010	360	840	840	0.28	840	0.44	840	1.23	840	1.15
rope_0020	720	1680	1680	0.71	1680	0.43	1680	6.02	1680	3.31
rope_0030	1080	2520	2520	1.19	2520	0.45	2520	27.1	2520	7.06
rope_0050	1800	4200	4200	2.20	4200	0.55	4200	191	4200	18.3
rope_0075	2700	6300	6300	4.46	6300	0.69	6300	426	6300	43.6
rope_0100	3600	8400	8400	8.16	8400	7.42	8400	1754	8400	82.9

TAB. 4.3 – Évaluation expérimentale de `constructMUS`

approches basées sur DPLL s'avèrent plus performantes sur des familles telles que les `am_*_*`. Si la méthode développée est aisément validée en pratique à la vue de ces résultats, on note tout de même que les 4 méthodes testées fournissent des résultats réellement disparates en fonction des CNF considérées. Évidemment, le fait que, contrairement à l'évaluation de solveurs, le résultat soit ici multi-critère (taille de l'approximation et temps d'exécution), n'aide pas à la comparaison. En outre, si une instance possède 2 MUS de tailles différentes, une approche qui pourra extraire exactement le plus gros d'entre eux est-elle moins intéressante qu'une autre retournant une vulgaire approximation du plus petit (et donc, malgré tout, un ensemble de clauses réduit) ? L'évaluation expérimentale de ces algorithmes est donc un problème en soi. Toutefois, il apparaît que le comportement de notre nouvelle technique est très satisfaisant sur de nombreuses instances, à la fois en terme de temps et de taille d'ensembles extraits, en comparaison des meilleures approches actuelles.

4.6 Conclusions

Dans ce chapitre, nous avons présenté deux nouvelles techniques permettant l'extraction d'un MUS. Ces nouvelles approches tirent leur originalité de l'utilisation de la recherche locale pour la localisation d'une zone inconsistante particulière d'une CNF. En outre, les algorithmes `AOMUS` et `OMUS` ne tiennent pas uniquement compte de l'interprétation courante pour le calcul heuristique du MUS, mais également d'un certain voisinage permettant d'acquérir une nouvelle information, que nous avons montrée théoriquement et pratiquement viable. L'approche constructive, quant à elle, assure qu'à chaque itération, aucune information redondante ne peut être ajoutée au MUS construit et offre à l'utilisateur un paramètre permettant de régler la qualité de l'approximation extraite en fonction des ressources en temps dont il dispose. Les procédures, développées suivant ces idées, se révèlent des méthodes très compétitives sur de nombreuses instances, voire les meilleures approches connues pour l'extraction d'un MUS, pour certaines familles.

Plusieurs perspectives s'ouvrent naturellement à la vue de ces résultats. Dans un premier temps, il semblerait intéressant d'envisager une hybridation entre ces nouvelles approches basées sur la recherche locale. Différentes combinaisons peuvent clairement être imaginées, et leur étude théorique couplée à des tests expérimentaux seront l'objet de recherches futures. Ensuite, une autre voie vise à l'intégration du concept de clause critique à l'approche constructive. Bien que sa nature se prête moins aisément à la prise en compte du voisinage des interprétations parcourues, celle-ci offre une piste de choix pour de futures recherches.

Chapitre 5

Calcul de tous les MUS

Sommaire

La localisation d'un MUS permet de fournir à l'utilisateur une explication minimale (en termes de cardinalité d'ensemble détenu de clauses) à l'incohérence d'une formule. Cependant, une même formule peut posséder plusieurs MUS. En fait, dans le pire cas, le nombre de MUS est même exponentiel ; une formule SAT composée de n clauses peut en effet exhiber $C_n^{n/2}$ MUS. Dans ce cas, le simple fait de les énumérer n'est pas réalisable en pratique. Fort heureusement, le nombre de MUS s'avère souvent limité dans les problèmes d'applications pratiques. Par exemple, en diagnostic de pannes [Hamscher *et al.* 1992], on suppose le plus souvent qu'une seule erreur est présente, ce qui peut induire un nombre plus limité de MUS.

Dans ce chapitre sont décrites nos différentes contributions au calcul ou à l'approximation de l'ensemble des MUS d'une formule CNF. Dans un premier temps, nous présentons une nouvelle technique algorithmique qui améliore la meilleure méthode complète connue, à savoir l'approche de [Liffiton & Sakallah 2005]. Ensuite, nous introduisons différents concepts visant à pallier l'explosion potentielle du nombre de MUS d'une formule en approximant de plusieurs

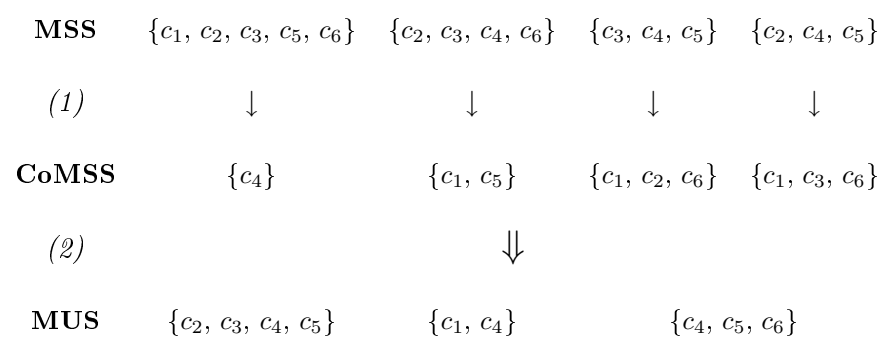


FIG. 5.1 – Transformation de l'ensemble des MSS vers celui des MUS

façons cet ensemble. Enfin, nous discutons d'une application des notions et algorithmes mis en jeu dans ce chapitre à la gestion de bases de croyances stratifiées.

Ces différentes contributions ont donné lieu à plusieurs publications [Grégoire *et al.* 2007a, Grégoire *et al.* 2007c, Grégoire *et al.* 2007e].

5.1 HYCAM, une méthode hybride pour le calcul de tous les MSS/MUS

Nous devons le meilleur algorithme connu pour dériver l'ensemble des MUS d'une formule CNF à Mark Liffiton et Karem A. Sakallah. Dans cette partie du texte, nous proposons une hybridation de cet algorithme avec une recherche locale préliminaire, permettant un gain non négligeable en ressources pour ce calcul exhaustif.

5.1.1 L'algorithme de [Liffiton & Sakallah 2005]

D'une manière similaire à l'approche `Dualize And Advance` proposée précédemment par [Bailey & Stuckey 2005], la méthode de [Liffiton & Sakallah 2005] utilise la relation entre les MUS et les Sous-formules Maximalement Satisfaisables (MSS) d'une formule.

Définition 5.1 (Formule Maximalement satisfaisable (MSS))
Une formule maximalement satisfaisable ou MSS (pour Maximal Satisfiable Subset *en anglais)* Γ *d'une instance SAT* Σ *est un ensemble de clauses tel que :*

1. $\Gamma \subseteq \Sigma$;
2. Γ *est satisfaisable ;*
3. $\forall \Delta \subseteq (\Sigma \backslash \Gamma)$ *tel que* $\Delta \neq \emptyset$, $\Gamma \cup \Delta$ *est insatisfaisable.*

L'ensemble complémentaire d'un MSS par rapport à une instance SAT est appelé CoMSS.

Définition 5.2 (CoMSS)
Le CoMSS d'un MSS Γ *d'une instance SAT* Σ *est donné par* $\Sigma \backslash \Gamma$.

Les CoMSS représentent donc les ensembles minimaux de clauses qui peuvent être ôtés à une formule pour en restaurer la consistance. On a donc la propriété suivante :

Propriété 5.1
Soient Σ une formule CNF, ω une interprétation complète, et Γ l'ensemble des clauses de Σ falsifiées par ω. Γ est un sur-ensemble de l'un des CoMSS de Σ.

Les concepts de CoMSS et MUS sont corrélés. Un CoMSS contient en effet au moins une clause de chaque MUS. Plus précisément, un CoMSS est un ensemble intersectant minimal (cf. définition 3.2 page 40) de l'ensemble des MUS.

De manière duale, chaque MUS d'une instance SAT est également un ensemble intersectant irréductible des CoMSS. De cette façon, bien que le `MINIMAL-HITTING-SET` soit un problème NP-difficile, son irréductibilité rend ce problème moins gourmand en ressources. En fait, un MUS peut même être déduit en temps polynomial à partir de l'ensemble des CoMSS.

Exemple 5.1
Soit Σ une formule CNF insatisfaisable telle que :

$$\begin{array}{rll} \Sigma = & (\ x_1\ ) & \mathbf{c_1} \\ \wedge & (\neg x_3) & \mathbf{c_2} \\ \wedge & (x_2 \vee x_3) & \mathbf{c_3} \\ \wedge & (\neg x_1) & \mathbf{c_4} \\ \wedge & (x_1 \vee \neg x_2) & \mathbf{c_5} \\ \wedge & (x_2) & \mathbf{c_6} \end{array}$$

On fournit l'ensemble des MSS de Σ dans la première ligne de la figure 5.1. Le calcul de l'ensemble de ces MUS s'effectue en 2 étapes. Premièrement, le complémentaire de chaque MSS, appelé CoMSS, est produit *(1)*. Il reste en second à calculer l'ensemble intersectant, qui s'obtient en calculant tous les ensembles contenant exactement une clause de chaque CoMSS *(2)*. Les ensembles ainsi calculés sont les MUS de Σ.

Le problème de la génération de tous les MUS est donc « relégué » au calcul exhaustif de tous les CoMSS. Dans cette optique, la méthode de [Liffiton & Sakallah 2005] (notée L&S) utilise un algorithme incrémental calculant des CoMSS de plus en plus grands (en nombre de clauses impliquées), et par conséquent des MSS de plus en plus petits.

Dans un premier temps, chaque clause c_i de la formule est enrichie d'un nouveau littéral $\neg y_i$. Ces littéraux, construits sur de nouvelles variables spécialement introduites, sont appelés *sélecteurs de clause*. Ils peuvent être vus comme des marqueur d'anormalité, au sens de [McCarthy 1986]. Ainsi, en résolvant cette nouvelle formule, l'affection de y_i à *faux* a pour effet de désactiver, ou supprimer c_i de l'instance, en la satisfaisant trivialement.

Exemple 5.2
Soit Σ la formule CNF insatisfaisable représentée ci-dessous.

$$\begin{array}{cc} \Sigma & \Sigma_y \\ \\ c_1 : \neg c \vee \neg b & c_1 : \neg c \vee \neg b \vee \neg \mathbf{y_1} \\ c_2 : \neg a \vee \neg c & c_2 : \neg a \vee \neg c \vee \neg \mathbf{y_2} \\ c_3 : c & c_3 : c \vee \neg \mathbf{y_3} \\ c_4 : a \vee b & c_4 : a \vee b \vee \neg \mathbf{y_4} \\ c_5 : \neg c \vee d & c_5 : \neg c \vee d \vee \neg \mathbf{y_5} \\ c_6 : b \vee \neg d & c_6 : b \vee \neg d \vee \neg \mathbf{y_6} \\ c_7 : \neg a \vee e & c_7 : \neg a \vee e \vee \neg \mathbf{y_7} \\ c_8 : \neg e & c_8 : \neg e \vee \neg \mathbf{y_8} \end{array} \quad \Longrightarrow$$

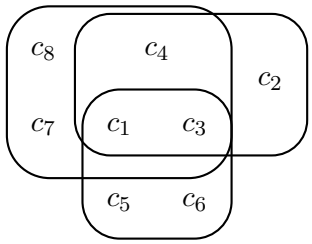


FIG. 5.2 – MUS de l'exemple 5.2

Σ est construite sur 5 variables et contient 8 clauses. Celles-ci forment 3 MUS : $MUS^1_\Sigma = \{c_1, c_2, c_3, c_4\}$, $MUS^2_\Sigma = \{c_1, c_3, c_5, c_6\}$ et $MUS^3_\Sigma = \{c_1, c_3, c_4, c_7, c_8\}$, représentés en figure 5.2. La première étape *transforme* Σ en Σ_y par l'ajout de sélecteurs de clauses.

On exécute sur Σ_y un DPLL modifié qui n'autorise qu'un certain nombre de sélecteurs de clauses à être falsifiés. Pour chaque valeur de cette borne, initialisée à 1 et incrémentée à chaque itération, une recherche exhaustive est effectuée pour déterminer tous les modèles de Σ_y. De ces modèles peuvent être déduits les CoMSS de Σ : ceux-ci sont en effet constitués des clauses satisfaites par un des littéraux ajoutés.

Exemple 5.3
Leur retrait « casse » donc toutes ses sources d'inconsistance et restaure sa satisfaisabilité.

Quand un modèle de Σ_y est trouvé, le CoMSS correspondant est enregistré, et On considère les formules Σ et Σ_y de l'exemple 5.2. Lors de la première itération de l'algorithme L&S (avec une borne à 1 sur les sélecteurs de clauses), 2 modèles à Σ_y sont exhibés : $\omega_y = \{\neg a, b, c, d, \neg e, \neg y_1\}$ et $\omega'_y = \{\neg a, b, \neg c, \neg d, \neg e, \neg y_3\}$. On peut donc en déduire que $\{c_1\}$ et $\{c_3\}$ sont des CoMSS de la formule Σ. Elles sont en effet les seules clauses à appartenir aux 3 MUS de la formule (cf. figure 5.2). une clause empêchant de recalculer cette solution (ainsi que tous ses sur-ensembles) est insérée dans Σ_y. Cette contrainte, appelée *clause bloquante* est constituée de la disjonction des variables y_i des clauses formant le CoMSS trouvé.

Exemple 5.4
Après la première itération de L&S sur la formule Σ_y de l'exemple 5.2, les CoMSS $\{c_1\}$ et $\{c_3\}$ sont générés. Les clauses bloquantes y_1 et y_3 sont donc ajoutées à la formule, qui devient $\Sigma^1_y = \Sigma_y \wedge (y_1) \wedge (y_3)$. De la même manière, après la deuxième itération (où l'on autorise cette fois 2 sélecteurs de clause à être falsifiés), on obtient les CoMSS $\{c_4, c_5\}$ et $\{c_4, c_6\}$, et la formule évolue en $\Sigma^2_y = \Sigma^1_y \wedge (y_4 \vee y_5) \wedge (y_4 \vee y_6)$.

Sans l'augmentation de la formule par ces clauses, lors de la deuxième itération de l'algorithme, l'ensemble $C = \{c_3, c_8\}$ pourrait être dérivé comme CoMSS. Or, celui-ci n'en est pas un, $\{c_3\}$ ayant déjà été extrait. C n'est donc pas minimal pour l'inclusion, et n'est en conséquence pas un CoMSS de Σ.

L'algorithme décrit permet donc d'obtenir des CoMSS de plus en plus grands, en incrémentant la borne sur les sélecteurs de clauses. Toutefois, il existe une valeur au-delà de laquelle il n'existe plus de tels ensembles de clauses. Comment la déterminer ?

La méthode proposée consiste à vérifier que la nouvelle instance augmentée des clauses bloquantes est toujours satisfaisable sans aucune borne sur les variables y_i, avant d'incrémenter

$$\Sigma_y^3 \left\{ \begin{array}{l} \Sigma_y^2 \left\{ \begin{array}{l} \Sigma_y^1 \left\{ \begin{array}{l} \Sigma_y \left\{ \begin{array}{l} \neg c \vee \neg b \vee \neg y_1 \\ \neg a \vee \neg c \vee \neg y_2 \\ c \vee \neg y_3 \\ a \vee b \vee \neg y_4 \\ \neg c \vee d \vee \neg y_5 \\ b \vee \neg d \vee \neg y_6 \\ \neg a \vee e \vee \neg y_7 \\ \neg e \vee \neg y_8 \end{array} \right. \\ y_1 \\ y_3 \end{array} \right. \\ y_4 \vee y_5 \\ y_4 \vee y_6 \end{array} \right. \\ y_2 \vee y_5 \vee y_7 \\ y_2 \vee y_5 \vee y_8 \\ y_2 \vee y_6 \vee y_7 \\ y_2 \vee y_6 \vee y_8 \end{array} \right.$$

FIG. 5.3 – Évolution de la CNF

la borne courante pour une nouvelle recherche. Si ce n'est pas le cas, l'ensemble exhaustif des CoMSS a été extrait, puisque toute interprétation falsifie un sur-ensemble des CoMSS de la formule (cf. propriété 5.1). L'algorithme termine ainsi son exécution. Dans le cas contraire, la borne est incrémentée et une nouvelle recherche vers de plus grands CoMSS est effectuée.

Exemple 5.5
Au terme de la première itération de L&S, un test de consistance sur Σ_y^1 est lancé (sans borne sur les sélecteurs de clauses). Clairement, cette CNF est satisfaisable. Après extraction des CoMSS de taille 2, une vérification similaire est effectuée sur Σ_y^2. Celle-ci est consistante : l'un de ses modèles est par exemple $\omega_y^2 = \{a, \neg b, c, \neg d, e, y_1, \neg y_2, y_3, \neg y_5, \neg y_8\}$. La troisième itération est la dernière ; elle permet la génération des CoMSS $\{c_2, c_5, c_7\}, \{c_2, c_5, c_8\}, \{c_2, c_6, c_7\}$ et $\{c_2, c_6, c_8\}$, et la formule Σ_y^3 dont l'évolution est synthétisée en figure 5.3 est insatisfaisable, ce qui garantit l'obtention de tous les CoMSS de la formule originale Σ.

D'un point de vue pratique, l'algorithme L&S tel qu'il est présenté ici ne nécessite que de légères modifications d'une méthode complète de type DPLL, puisque le problème de la génération de CoMSS est effectué à travers la recherche des modèles d'une formule. Il peut donc bénéficier des récents progrès algorithmiques sur SAT (structures « paresseuses », prétraitement de la formule, apprentissage de « *nogoods* », etc.). L'implémentation effectué par [Liffiton & Sakallah 2005] est basé sur `Minisat` [Eén & Sörensson], qui est connu pour être l'un des meilleurs solveurs modernes. L'approche L&S est résumée dans l'algorithme 9.

Dans les paragraphes suivants, nous introduisons une hydridation de l'approche L&S avec une recherche locale permettant de restreindre les ressources nécessaires à la génération des CoMSS d'une formule CNF. La seconde partie de l'algorithme, calculant l'ensemble complet des MUS à partir des CoMSS, est effectuée à travers l'*hypergraph transversal problem*, et sera considérée telle quelle.

Fonction extrait_CoMSS(Σ : une **CNF insatisfaisable**) : **l'ensemble des CoMSS de** Σ
  $\Sigma_y \longleftarrow ajout\_selecteur\_clause$;
  $borne \longleftarrow 1$;
  $CoMSS \longleftarrow \emptyset$;
  Tant que (Σ_y est satisfaisable) **faire**
    [La fonction « DPLL_avec_borne » calcule tous les CoMSS de Σ_y *de taille* borne *et ajoute les clauses bloquantes à cette formule]*
    $CoMSS \longleftarrow CoMSS \cup DPLL\_avec\_borne(\Sigma_y, borne)$;
    $borne \longleftarrow borne + 1$;
  Fait
  Retourner $CoMSS$;
Fin

Algorithme 9 – Calcul de tous les CoMSS par [Liffiton & Sakallah 2005]

5.1.2 Calcul approché des MSS d'une formule

Dans cette section, nous montrons comment la méthode présentée précédemment peut être grandement améliorée en l'hybridant avec une recherche locale préliminaire, qui jouera le rôle d'oracle pour le processus de recherche exhaustive. Cette nouvelle approche est baptisée `HYCAM` (pour *HYbridization for Computing All MUS* en anglais).

En premier lieu, présentons notre méthode de manière intuitive. Clairement, une (rapide) recherche locale initiale, exécutée à la recherche d'un modèle, peut rencontrer certains MSS (par rapport à l'ensemble d'interprétations parcourues). Quand ce phénomène se produit, il peut être utile de conserver les CoMSS correspondants, pour éviter d'avoir à les calculer par la suite de manière complète. De plus, plutôt que de vérifier si nous sommes en présence d'un MSS ou non, on préfère enregistrer le CoMSS *candidat* correspondant ; celui-ci sera ensuite vérifié durant la partie complète de l'algorithme.

Clairement, nous avons à étudier quelles interprétations parcourues par la recherche locale peuvent conduire à de bons candidats aux CoMSS et des critères doivent être définis pour conserver un nombre limité de CoMSS potentiels. Dans cette optique, le concept de clause critique peut se révèler extrêmement utile, dans le sens où il fournit une condition nécessaire à la « CoMSS-candidature » qui peut être vérifiée très rapidement. Quand un ensemble de candidats aux CoMSS est enregistré, l'approche incrémentale de [Liffiton & Sakallah 2005] peut alors exploiter cette information d'une manière efficace.

D'une manière plus détaillée, un algorithme de recherche locale est exécuté sur l'instance SAT initiale. Le but est d'enregistrer autant de CoMSS potentiels que possible, en se basant sur le fait que cette famille d'algorithmes convergent vers les minima locaux, ce qui peut se transcrire en de bonnes approximations de MSS. Une approche directe consiste à conserver pour chaque interprétation parcourue l'ensemble de clauses falsifiées. Cependant, il semble inutile d'enregistrer les sur-ensembles de CoMSS candidats ; ceux-ci ne sont clairement pas des CoMSS, puisqu'ils ne sont pas minimaux par rapport à l'inclusion ensembliste. Dans ce but, nous avons adapté une technique de gestion de la sous-sommation proposée par [Zhang 2005], basée sur des listes d'oc-

```
Fonction RL_approximation(Σ : une CNF insatisfaisable) : un ensemble de CoMSS candidats
    cand ⟵ ∅ ;
    #echec ⟵ 0 ;
    ω ⟵ genere_interpretation_aleatoire() ;
    Tant que ((#echec < #NB_ECHECS_AUTORISES)) faire
        nouveauCand ⟵ FAUX ;
        Pour j = 1 à #FLIPS faire
            Soit Δ l'ensemble des clauses falsifiées par ω ;
            Si ((∀C ∈ Δ, C est critique) ET Δ n'est pas impliqué par cand) Alors
                supprimeEnsembleImpliqué(Δ,cand) ;
                cand ⟵ Δ ∪ cand ;
                nouveauCand ⟵ VRAI ;
            Fin Si
            flip(ω) ;
            Si (non(nouveauCand)) Alors #echec ⟵ #echec + 1 ; Fin Si
        Fin Pour
    Fait
    Retourner Σ ;
Fin
```

Algorithme 10 – Approximation de l'ensemble des CoMSS par la recherche locale

currence et des opérations ensemblistes. Sommairement, l'intersection des clauses d'une formule Σ où apparaissent tous les littéraux d'une clause c permet de fournir l'ensemble des clauses de Σ que c sous-somme (*backward subsumption*). L'opération inverse (*forward subsumption*), permettant de savoir si une nouvelle clause est sous-sommée, utilise un marquage des littéraux au sein des clauses, et reprend le concept de *watched literals* afin d'accélérer son traitement. En adaptant cette technique aux ensembles de clauses plutôt qu'aux ensembles de littéraux, seuls les CoMSS candidats minimaux pour l'inclusion parmi l'ensemble de clauses falsifiées déjà calculé sont enregistrés.

Exemple 5.6
Soit une recherche locale exécutée sur une instance Σ. Si deux interprétations parcourues falsifient les ensembles de clauses $\{c_i, c_j, c_k\}$ et $\{c_j, c_k\}$, il n'est d'aucune utilité de conserver le premier ensemble de clauses. Celui-ci ne peut être un CoMSS de la formule, puisque la découverte du deuxième ensemble prouve qu'il n'est pas minimal (son complémentaire n'est donc pas « maximalement satisfaisable »).

Ainsi, la technique de [Zhang 2005] est utilisée pour ne conserver à chaque étape que les ensembles de clauses minimaux pour l'inclusion. En outre, nous avons également exploité le concept de clause critique, qui s'est révèlé efficace pour la localisation et l'isolation d'un MUS, quand il est allié à un algorithme incomplet de recherche locale (cf. Chapitre 4).

Intuitivement, une clause critique est donc une clause falsifiée qui nécessite qu'une autre clause soit à son tour falsifiée pour qu'elle soit satisfaite, en effectuant un unique *flip*. La propriété suivante montre comment ce concept permet d'éliminer facilement de mauvais candidats CoMSS.

Fonction HYCAM(Σ : une **CNF insatisfaisable**) : **l'ensemble des MSS de** Σ
  $cand \longleftarrow$ RL_approximation(Σ) ;
  $k \longleftarrow 0$;
  $\Sigma_y \longleftarrow$ AjoutSelecteurClauses_Yi(Σ) ;
  $MSS \longleftarrow \emptyset$;
  Tant que (SAT(Σ_y)) **faire**
    supprimeEnsembleImpliqué($\{\Sigma \backslash C | C \in MSS\}$,$cand$) ;
    $\Sigma_y \longleftarrow$ ClausesBloquantesTaille(k,$cand$) ; $MSS \longleftarrow MSS \cup \{\Sigma \backslash C | C \in cand$ et $|C| = k\}$;
    $MSS \longleftarrow MSS \cup$ SAT_avec_borne(k,Σ_y) ;
    $k \longleftarrow k + 1$;
  Fait
  Retourner MSS ;
Fin

Algorithme 11 – `HYCAM`

Propriété 5.2
Soient Σ une instance SAT et ω une interprétation. Soit Γ un sous-ensemble non vide de Σ tel que toutes les clauses de Γ soient falsifiées par ω. Quand au moins une clause de Γ n'est pas critique par rapport ω, alors Γ est n'est pas un CoMSS de Σ.

Preuve
Par définition, quand une clause c_f de Γ n'est pas critique par rapport à ω, il existe un littéral $c \in c_f$ dont la valeur de vérité peut être inversée (ou « *flippée* ») sans falsifier aucune autre clause de Σ. Γ n'est donc pas minimal et ne peut être un CoMSS de Σ. □

En pratique, tester si les clauses falsifiées sont critiques peut s'effectuer efficacement et évite à avoir à enregistrer de trop nombreux CoMSS candidats, ce qui peut s'avérer assez lourd en conjonction de la gestion de ces ensembles de clauses par la technique de [Zhang 2005].

En considérant ces caractéristiques, une recherche locale est exécutée sur l'instance SAT initiale, et retourne une série de CoMSS candidats. Cette information s'avère extrêmement utile, et permet d'améliorer en pratique l'étape complète de L&S. La façon dont sont utilisées les informations fournies par cette procédure est détaillée dans le paragraphe suivant.

5.1.3 Utilisation des MSS candidats pour la réduction de leur calcul exact et exhaustif

L'algorithme L&S est incrémental dans le sens où il calcule progressivement des CoMSS de plus en plus grands. Ainsi, après que n itérations aient été effectuées, tous les CoMSS dont la cardinalité est inférieure ou égale à n ont été générés. Clairement, si grâce à la recherche locale préliminaire, on a enregistré des CoMSS candidats contenant exactement $n + 1$ clauses qui ne sont pas des sur-ensembles des CoMSS déjà calculés, nous avons la garantie que ceux-ci sont de vrais CoMSS. Ainsi, nous n'avons pas à les calculer, et les clauses bloquantes correspondantes peuvent être insérées directement. Clairement, il n'est donc pas nécessaire d'effectuer le test SAT

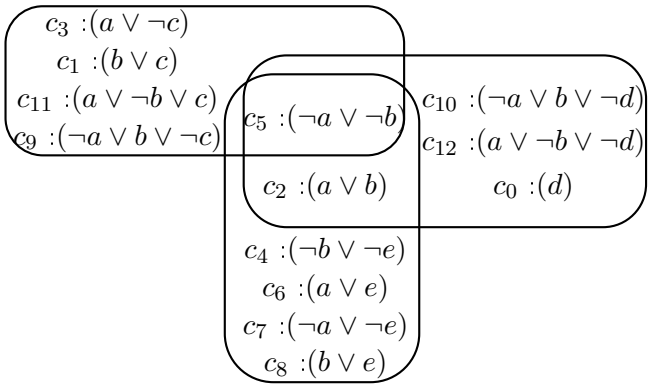


FIG. 5.4 – MUS de l'exemple 5.7

au terme de la $n^{ème}$ itération, puisque nous avons connaissance de l'existence de plus grands CoMSS. De plus, l'insertion préalable de clauses bloquantes permet d'éviter à la fois des tests NP-complets et CoNP-complets pendant la recherche. Ceci est illustré par l'exemple suivant.

Exemple 5.7
Soient Σ une instance insatisfaisable, et Σ_y l'instance SAT correspondante augmentée des littéraux $\neg y_i$, sélecteurs de clauses de L&S.

$$\Sigma = \begin{cases} c_0 : (d) & c_1 : (b \vee c) \\ c_2 : (a \vee b) & c_3 : (a \vee \neg c) \\ c_4 : (\neg b \vee \neg e) & c_5 : (\neg a \vee \neg b) \\ c_6 : (a \vee e) & c_7 : (\neg a \vee \neg e) \\ c_8 : (b \vee e) & c_9 : (\neg a \vee b \vee \neg c) \\ c_{10} : (\neg a \vee b \vee \neg d) & c_{11} : (a \vee \neg b \vee c) \\ c_{12} : (a \vee \neg b \vee \neg d) & \end{cases}$$

Σ est une formule insatisfaisable composée de 13 clauses qui sont construites sur 5 variables. Σ contient 3 MUS, représentés par la figure 5.4, et admet 19 MSS. Supposons que L&S et HYCAM soient tous deux exécutés avec cette instance en entrée. Ses clauses sont augmentées par les littéraux négatifs $\neg y_i$, jouant le rôle de sélecteurs de clauses. Supposons également que la recherche locale effectuée par HYCAM fournisse 4 CoMSS candidats : $\{c_5\}$, $\{c_3, c_2\}$, $\{c_0, c_1, c_2\}$ et $\{c_3, c_8, c_{10}\}$.

Si les variables à affecter sont choisies dans l'ordre lexicographique, alors a et b sont d'abord assignées à *vrai* et c_5 est falsifiée. L&S tente alors de prouver que cette clause forme un CoMSS, ce qui nécessite un test NP-complet (il a en effet à trouver un modèle à la formule $\Sigma \backslash \{\neg a \vee \neg b\} \cup \{a, b\}$). Au contraire, avec HYCAM, la clause bloquante y_5 est ajoutée avant la première itération de l'algorithme complet, puisque la recherche locale a auparavant retourné ce CoMSS. En conséquence, quand a et b sont affectées à *vrai*, l'algorithme DPLL fait immédiatement un retour en arrière (« *backtrack* »), puisque l'ensemble insatisfaisable $\{y_5, \neg y_5\}$ a été obtenu, sans nécessiter aucun test NP-complet.

D'une manière similaire, l'introduction de clauses bloquantes additionnelles par HYCAM réduit le nombre de tests CoNP-complets effectués. Par exemple, supposons que e soit la première variable assignée (à *faux*) et que les variables suivantes soient affectées dans l'ordre lexicogra-

Instance	**(#v,#c)**	**#MSS**	**#CoMSS cand. réels**		**L&S (sec.)**	**HYCAM (sec.)**
hole6	(42,133)	133	133	133	0.040	0.051
hole7	(56,204)	204	204	204	0.75	0.33
hole8	(72,297)	297	293	293	33	1.60
hole9	(90,415)	415	415	415	866	30
hole10	(110,561)	561	559	559	7159	255
x1_16	(46,122)	122	122	122	0.042	0.041
x1_24	(70,186)	186	186	186	7.7	0.82
x1_32	(94,250)	250	241	241	195	28
x1_40	(118,314)	314	314	314	2722	486

TAB. 5.1 – L&S vs `HYCAM` sur des instances globalement insatisfaisables

phique. Quand a et b sont affectées à *vrai*, L&S essaie de nouveau de prouver que $\{c_5\}$ est un CoMSS. Puisque $\neg e$ est une conséquence logique de $\Sigma \backslash \{\neg a \vee \neg b\} \cup \{a, b\}$, il n'existe aucun modèle à $\Sigma \backslash \{\neg a \vee \neg b\} \cup \{a, b, \neg e\}$. Clairement, un tel test est dans CoNP. Grâce aux CoMSS candidats fournis préalablement par `HYCAM`, on peut éviter de parcourir cette partie de l'espace de recherche. En effet, puisque l'on sait que $\{c_5\}$ est un CoMSS de Σ, quand a et b sont affectées à *vrai*, plus aucun test CoNP n'est nécessaire, avec cette interprétation partielle.

Comme le montre cet exemple, l'introduction de chaque clause bloquante obtenue par l'hybridation proposée avec une RL peu coûteuse en ressources permet d'éviter un test NP et un certain nombre (qui varie de 0 à un nombre exponentiel en la taille de la formule) de tests CoNP. De plus, les tests vérifiant l'existence d'autres CoMSS ne sont plus nécessaires, si au moins un candidat retourné n'est pas un sur-ensemble des CoMSS déjà obtenus.

5.1.4 Validation expérimentale

`HYCAM` a été implémenté et comparé à L&S d'un point de vue pratique. Pour ces deux algorithmes, l'étape complète est basée sur l'architecture de MiniSat [Eén & Sörensson], qui est à ce jour l'un des meilleurs solveurs SAT. Comme cas d'étude, Walksat [Kautz *et al.* 2004] a été utilisée pour le prétraitement incomplet. Les nombres de *tries* et de *flips* effectués par la recherche locale ne sont pas fixes, mais dépendent de l'efficacité de cette méthode à trouver de nouveaux CoMSS. En fait, à chaque « étape », on effectue un petit nombre de *flips*, et si aucun nouveau candidat n'est capturé, un compteur d'échecs est incrémenté. Quand ce compteur atteint un seuil (expérimentalement fixé à 30), on considère qu'aucun nouveau candidat ne peut être obtenu « facilement » (i.e. sans effectuer un grand nombre de mouvements dans l'espace de recherche) par cette méthode. Cette technique de terminaison offre un bon compromis entre le nombre de CoMSS extraits et le temps passé à leur découverte. D'ailleurs, pour toutes les expérimentations effectuées, cette étape préliminaire ne dépasse pas 5% du temps de calcul global de `HYCAM`.

Nos tests expérimentaux ont été conduits sur des processeurs Intel Xeon 3GHz sous Linux CentOS 4.1. (kernel 2.6.9) avec 2 Go de mémoire vive. Dans l'ensemble des tables reportant les expérimentations conduites, sont notés de gauche à droite : le nom de l'instance testée (*Instance*), ses nombres de variables et clauses (*#v,#c*), le nombre de MSS extraits (*#MSS*), les nombres de CoMSS candidats et réels calculés par la recherche locale de `HYCAM` (*#CoMSS cand., réel*, respectivement) et enfin le temps en secondes mis par L&S et `HYCAM` pour effectuer le calcul.

Instance	(#v,#c)	#MSS	#CoMSS cand.	#CoMSS réels	L&S (sec.)	HYCAM (sec.)
rand_net40-25-10	(2000,5921)	5123	4318	2729	893	197
rand_net40-25-5	(2000,5921)	4841	6950	598	650	174
rand_net40-30-10	(2400,7121)	5831	3458	2405	1748	386
rand_net40-30-1	(2400,7121)	7291	4380	662	1590	1325
rand_net40-30-5	(2400,7121)	5673	2611	2507	2145	402
ca032	(558,1606)	1173	1159	1159	4	1
ca064	(1132,3264)	2412	2324	2263	59	3
ca128	(2282,6586)	4899	2878	2422	691	18
ca256	(4584,13236)	9882	9553	9064	*time out*	277
2pipe	(892,6695)	3571	2094	1849	130	36
2pipe_1_ooo	(834,7026)	3679	1822	1587	52	30
2pipe_2_ooo	(925,8213)	5073	2286	1825	148	61
3pipe_1_ooo	(2223,26561)	17359	7327	3481	5153	2487
am_5_5	(1076,3677)	1959	3250	65	68	57
c432	(389,1115)	1023	1019	1019	4	1
c880	(957,2590)	2408	2141	1866	28	3
bf0432-007	(1040,3668)	10958	3332	2136	233	98
velev-sss-1.0-cl	(1453,12531)	4398	2987	2154	1205	513

TAB. 5.2 – L&S vs HYCAM sur différentes instances SAT difficiles

Les CoMSS candidats et réels désignent respectivement les ensembles de clauses retournés par la recherche locale préliminaire comme CoMSS potentiels, et les ensembles de clauses qui parmi eux se sont avérés être effectivement des CoMSS. La table 5.3 contient de plus les colonnes « #MUS » indiquant le nombre de MUS de l'instance et « MSS→MUS » le temps nécessaire à leur calcul. Pour toutes ces expérimentations, une limite de temps de 3 heures CPU a été respectée. Au delà de cette limite, la note « *time out* » est reportée.

Tout d'abord, dans la table 5.1, nous présentons des résultats expérimentaux sur le calcul de MSS sur le célèbre problème dit « des pigeons » ainsi que sur les benchmarks « *xor-chains* » [SATLIB], qui sont des instances globalement inconsistantes, dans le sens où retirer n'importe laquelle de leurs clauses restaure la consistance de la formule ; ce sont donc des MUS. Clairement, de telles instances possèdent un nombre de CoMSS égal au nombre de clauses dont elles sont composées, et chacun est de taille 1. On a donc la garantie d'un nombre polynomial (et même linéaire) de CoMSS. En pratique, un écart signicatif peut être observé en faveur de HYCAM. De surcroît, la différence de temps grandit avec la taille de l'instance. En fait, pour ces instances, la recherche locale réussit la plupart du temps à calculer l'ensemble des CoMSS, et l'étape complète est souvent réduite à un test d'insatisfaisabilité. Au contraire, L&S doit explorer de nombreux nœuds supplémentaires de l'arbre de recherche pour prouver que chacune de ces clauses est un CoMSS avant de les retourner.

Dans la table 5.2, les résultats expérimentaux sur des benchmarks plus difficiles, extraits de la compétition SAT annuelle [Compétitions SAT], sont reportés. Leur nombre de CoMSS est souvent exponentiel, et leur calcul exhaustif impossible en pratique. Nous avons donc réduit la recherche aux CoMSS de taille inférieure à 5 clauses. Comme ces résultats le montrent, HYCAM surpasse L&S. Par exemple, considérons l'instance rand_net40-30-10. Cette formule contient

Instance	**(#v,#c)**	**#MSS**	**#CoMSS cand.**	**réels**	**L&S (sec.)**	**HYCAM (sec.)**	**#MUS**	**MSS→MUS (sec.)**
mod2-3c-u-9-8	(87, 232)	232	232	232	3745	969	1	0.006
mod2-r3-u-105-3	(105, 280)	280	280	280	2113	454	1	0.008
2pipe	(892, 6695)	10221	3142	1925	298	226	$> 2.10^5$	*time out*
php-012-011	(132, 738)	738	734	734	*time out*	2597	1	0.024
hcb3	(45, 288)	288	288	288	10645	6059	1	0.006
1dlx_mc_ex_f	(776, 3725)	1763	946	665	10.4	6.8	$> 3.10^5$	*time out*
hwb-n20-02	(134, 630)	622	588	583	951	462	1	0.01
hwb-n22-02	(144, 688)	680	627	626	2183	811	1	0.025
ssa2670-141	(986, 2315)	1413	1374	1341	2.83	1.08	16	0.15
clqcolor-08-05-06	(116, 1114)	1114	1114	1114	107	62	1	0.007
dlx2_aa	(490, 2804)	1124	1020	970	3.12	0.94	32	0.023
addsub.boehm	(492, 1065)	1324	20256	347	35	29	$> 6.10^5$	*time out*

TAB. 5.3 – L&S vs `HYCAM` : calcul de tous les MUS

5831 MSS (dont le complémentaire n'excède pas la limite fixée). L&S et `HYCAM` fournissent ce résultat en 1748 et 386 secondes, respectivement. Pour l'instance `ca256`, `HYCAM` extrait 9882 MSS en moins de 5 minutes tandis que L&S ne peut les calculer en 3 heures de temps CPU. Notons également que `HYCAM` peut également retourner des CoMSS contenant 5 clauses après que le calcul soit effectué, puisque tous les ensembles de cette taille qui ne sont pas des sur-ensembles de CoMSS calculés sont également des CoMSS.

Dans la table 5.3, des résultats sur des benchmarks difficiles pour le calcul de l'ensemble des MSS et des MUS sont donnés. Comme expliqué plus haut, les deux approches, que ce soit L&S ou `HYCAM`, nécessitent l'obtention préalable de tous les MSS pour calculer l'ensemble des MUS. Comme le calcul des MSS est permis en un moindre coût par `HYCAM`, celui-ci permet d'obtenir l'ensemble complet des MUS pour de nombreuses formules, et ceci plus rapidement qu'avec L&S. Clairement, quand le nombre de MSS ou de MUS est exponentiel, les calculer et les énumérer exhaustivement se révèle impossible en pratique.

Par exemple, L&S n'a pas été capable de calculer tous les MSS de l'instance `php-012-011` en 3 heures de temps CPU, et ne peut donc découvrir son unique MUS. `HYCAM` l'extrait quant à lui en 2597 secondes. Sur tous les benchmarks ayant un nombre limité de MUS, `HYCAM` est clairement plus efficace que L&S. Ainsi, L&S et `HYCAM` découvrent les 32 MUS de `dlx2_aa` en 3,12 et 0,94 secondes, respectivement. On remarque en outre que le temps additionnel passé à calculer tous les MUS à partir des MSS est souvent très court quand leur nombre rend ce problème traitable.

5.2 Approcher l'ensemble des MUS par le calcul d'une couverture inconsistante

Comme nous l'avons vu, les techniques calculant de manière exhaustive tous les MUS d'une formule peuvent se montrer inutilisables en pratique, car leur nombre est exponentiel en la taille de l'instance, dans le pire cas. Si pour de nombreux problèmes issus d'applications réelles, le nombre de MUS est souvent limité, rien ne peut cependant le garantir. Il semble donc naturel

de chercher à approximer l'ensemble des MUS d'une instance le plus précisément possible, afin de pouvoir également réparer ce type de formule.

Nous proposons deux types d'approximation à l'ensemble des MUS : dans un premier temps, nous étudions l'intérêt des couvertures inconsistantes strictes, puis une méthode heuristique permettant d'extraire un grand nombre (polynomial) de MUS.

5.2.1 Extraire une couverture inconsistante stricte

L'incohérence d'une formule CNF peut avoir différentes causes, mais celles-ci sont parfois *connectées*. Nous introduisons ici une nouvelle méthode pour calculer des MUS *indépendants*, c'est-à-dire ne partageant aucune clause. Cette idée est principalement motivée par le fait que des MUS indépendants expriment des causes indépendantes, ou non corrélées, d'inconsistance au sein d'une même formule.

Une telle approximation doit fournir un nombre suffisant de cause d'incohérence tout en garantissant un nombre polynomial de MUS. Ceci conduit au concept de *couverture inconsistante stricte* d'une formule contradictoire.

Définition 5.3 (couverture inconsistante (stricte))
Deux ensembles de clauses sont dits indépendants *si et seulement si leur intersection est vide. Une couverture inconsistante (stricte) IC d'une formule SAT insatisfaisable* Σ *est l'union ensembliste de MUS (indépendants) de* Σ *telle que* $\Sigma \backslash IC$ *est satisfaisable.*

Assez clairement, une même instance SAT peut exhiber plusieurs couvertures inconsistantes strictes. En outre, comme le montre l'exemple suivant, leur « taille » (c'est-à-dire le nombre de MUS qu'elles contiennent) peut également être variable.

Exemple 5.8
Soit $\Sigma = \{a, b, \neg a \vee \neg b, \neg a, c, \neg c \vee a\}$ une formule CNF. Les MUS de Σ, représentés par un diagramme de Venn en figure 5.5, sont au nombre de 3 : $MUS_1 = \{a, \neg a\}$, $MUS_2 = \{a, b, \neg b \vee \neg a\}$ et $MUS_3 = \{\neg a, c, \neg c \vee a\}$. Σ admet 2 couvertures inconsistantes strictes, qui sont $IC_1 = MUS_1$ et $IC_2 = MUS_2 \cup MUS_3$. En effet, le retrait de l'un de ces ensembles de MUS restaure clairement la satisfaisabilité de Σ.

Un résultat direct du concept de couverture inconsistante stricte permet d'obtenir une borne minimale sur le nombre de clauses qui sont falsifiées par une solution MaxSat d'une instance insatisfaisable.

Propriété 5.3
Soient Σ *une instance insatisfaisable et IC une couverture inconsistante stricte de* Σ*. Soit* $|IC|$ *le nombre de MUS indépendants contenus dans IC. Pour toute interprétation* ω *de* Σ*, au moins* $|IC|$ *clauses de* Σ *sont falsifiées par* ω*.*

Preuve
Par définition, une couverture inconsistante stricte contient des MUS disjoints par rapport aux clauses qui les composent. Puisque pour toute interprétation ω, au moins une clause par MUS est falsifiée, au moins $|IC|$ clauses sont donc falsifiées par ω. □

Exemple 5.9
Considérons la formule Σ de l'exemple 5.8. Si l'extraction de IC_2 fournit le résultat exact à MaxSat, il n'en est pas de même pour IC_1 : la suppression de cet unique MUS « casse » 2 autres

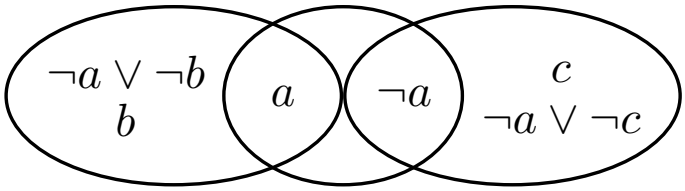


FIG. 5.5 – MUS de l'exemple 5.8

Fonction ICMUS(Σ : **une formule insatisfaisable**) : **une couverture inconsistante stricte de** Σ
    $IC \longleftarrow \emptyset$;
    Tant que (Σ est insatisfaisable) **faire**
        $MUS \longleftarrow$ `OMUS`(Σ) ;
        $IC \longleftarrow IC \cup MUS$;
        $\Sigma \longleftarrow \Sigma \setminus MUS$;
    Fait
    Retourner IC ;
Fin

Algorithme 12 – `ICMUS` (find a strict Inconsistent Cover of MUS)

MUS disjoints, et il est assez facile de comprendre ici que la couverture inconsistante stricte ne peut fournir qu'une borne inférieure pour une solution MaxSat, la plupart du temps.

D'un point de vue diagnostic, bien qu'une couverture inconsistante stricte ne contienne pas nécessairement l'ensemble exhaustif des MUS présents au sein d'une formule, elle fournit une série d'explications minimales qui est suffisante pour expliquer et potentiellement réparer assez de sources d'inconsistance pour restaurer la satisfaisabilité de la formule entière si elles sont toutes réparées. De surcroît, certaines formules possèdent des MUS très petits comparés à la taille de l'instance ; supprimer ou réparer ces MUS est souvent suffisant pour restaurer leur cohérence.

Comme les couvertures inconsistantes strictes sont composées de MUS indépendants, une façon de les obtenir est d'extraire un MUS, de le retirer de la formule, et de répéter ces opérations jusqu'à ce que la sous-formule courante soit consistante. Cette méthode est appelée `ICMUS` (en anglais : find a strict Inconsistent Cover of MUSes) et est décrite dans l'algorithme 12. D'une manière similaire à `(A)OMUS`, le test d'incohérence CoNP de la boucle principale est substitué en pratique par un échec de la recherche locale à trouver un modèle de la formule. Un gain conséquent peut alors être observé en pratique, tout en conservant la complétude de l'algorithme, puisque si la formule est satisfaisable, aucun MUS ne pourra en être extrait, et la procédure stoppera son exécution en temps fini.

Pour tester la viabilité de ce concept, une couverture inconsistante stricte a été extraite de nombreux benchmarks pour SAT, provenant pour la plupart de la compétition annuelle

Instance	#var	#cla	Temps	#**MUS** de la couverture (#var,#cla) pour chaque MUS
dp02u01	213	376	1.19	**1** (47,51)
dp03u02	478	1007	362	**1** (327,760)
23.cnf	198	474	2.68	**1** (165,221)
42.cnf	378	904	9	**1** (315,421)
fpga10_11_uns_rcr	220	1122	56	**2** (110,561) (110,561)
fpga11_12_uns_rcr	264	1476	128	**2** (132,738) (132,738)
ca002	26	70	0.61	**1** (20,39)
ca004	60	168	1.11	**1** (49,108)
ca008	130	370	5.26	**1** (110,255)
term1_gr_rcs_w3	606	2518	6180	**11** (12,22) (21,33) (30,58) (12,22) (12,22) (12,22) (12,22) (12,22) (12,22) (24,39) (21,33)
C220_FV_RZ_14	1728	4508	28	**1** (10,14)
C220_FV_RZ_13	1728	4508	46	**1** (9,13)
C170_FR_SZ_96	1659	4955	18	**1** (81,233)
C208_FA_SZ_121	1608	5278	21	**1** (18,32)
C168_FW_UT_851	1909	7491	83	**1** (7,9)
C202_FW_UT_2814	2038	11352	304	**1** (15,18)
jnh208	100	800	14	**1** (76,119)
jnh302	100	900	63	**2** (27,28) (98,208)
jnh310	100	900	184	**2** (12,13) (90,188)
ezfact16_1	193	1113	203	**1** (37,54)
ezfact16_2	193	1113	104	**1** (32,41)
ezfact16_3	193	1113	207	**1** (63,128)
3col40_5_3	80	346	4.64	**1** (64,136)
fphp-012-010	120	1212	57	**1** (120,670)

Tab. 5.4 – Couvertures inconsistantes strictes de différentes classes de formules

[Compétitions SAT]. Dans la table 5.4, quelques résultats expérimentaux sont donnés. Sans surprise, de nombreux problèmes insatisfaisables possèdent de très petites couvertures inconsistantes en terme de nombre de clauses. Considérons par exemple la formule `ezfact16_2`. Cette instance contient 1113 clauses, en particulier un MUS de 41 clauses. Ce MUS est en fait une couverture inconsistante stricte car son retrait restaure la cohérence de la formule. Le travail d'analyse et de réparation de l'insatisfaisabilité en est donc d'autant réduit, et permet de concentrer notre attention sur cette zone d'incohérence.

L'instance industrielle `term1_gr_rcs_w3` possède quant à elle une couverture inconsistante stricte constituée de 11 petits MUS. Grâce à ce résultat, on peut déduire que toute interprétation falsifie au moins 11 clauses. En effet, comme montré par la propriété 5.3, ce type de couverture induit une borne minimale sur le nombre de clauses falsifiées par une interprétation MaxSat. Notons également que les couvertures inconsistantes calculées sur les formules `FPGA` suggèrent la présence d'une forme de symétrie sur ce type d'instances.

Bien que les ensembles de MUS retournés soient suffisants pour la *réparation* d'instances, celle-ci n'est pas optimale dans le cas général. En effet, le retrait d'autres combinaisons de clauses

peut par exemple être préféré, mais celui-ci n'est possible que par la connaissance d'autres sources d'incohérence. Ainsi, dans le paragraphe suivant, une approche permettant de calculer un plus grand nombre de MUS qu'avec une couverture inconsistante est présentée.

5.2.2 Approcher l'ensemble exhaustif des MUS

Comme nous l'avons vu précédemment, l'extraction d'une couverture inconsistante stricte est suffisante pour l'analyse et la réparation de formules CNF. Cependant, celles-ci peuvent être raffinées ou accélérées par la détection d'un plus grand nombre de MUS au sein d'une formule. Dans le contexte de la vérification d'équivalence entre circuits, il a par exemple été montré que l'extraction de plusieurs MUS conduit à une vérification moins coûteuse, et montre l'intérêt de telles extractions [Andraus *et al.* 2006]. En conséquence, nous nous posons le problème d'extraire un grand nombre de MUS, tout en restant polynomial quant à leur nombre. Ce travail peut être vu comme un compromis entre la couverture inconsistante stricte et le calcul exhaustif des MUS (également appelé « *clutter* » par [Bruni 2005]), pas toujours réalisable en pratique.

Comme un MUS peut être « cassé » par le retrait d'une de ses clauses, une approche naïve consiste à détecter itérativement un MUS puis à retirer l'une de ses clauses de l'instance (afin de s'assurer de ne pas le détecter une nouvelle fois), jusqu'à ce que la formule devienne cohérente. Une telle approche permet d'obtenir l'ensemble des MUS quand tout couple de MUS possède une intersection vide. Cependant, les MUS peuvent présenter une intersection non vide. En conséquence, si l'on retire une clause d'un MUS, tout MUS contenant également cette clause est « cassé » et ne peut plus être détecté. Pour éviter ce problème dans la mesure du possible, on préfère retirer les clauses appartenant au moins grand nombre de MUS possible. L'heuristique suivante a été conçue dans cet objectif.

On sait qu'au sein d'une formule CNF, toute interprétation falsifie nécessairement au moins une clause de chacun de ses MUS. Or, certaines interprétations falsifient moins de clauses que d'autres. On peut donc supposer que lorsque ce cas se produit, les clauses falsifiées appartiennent aux intersections des MUS de la formule, et donc à grand nombre d'entre eux. En se basant sur la procédure `OMUS`, on choisit donc, quand une clause de la formule est falsifiée, de stocker le nombre de clauses falsifiées avec elle. Pendant le parcours de la recherche locale, quand une clause est à nouveau falsifiée, on conserve le minimum entre le nombre précédemment stocké et le nombre courant de clauses falsifiées de la formule. Après qu'un MUS ait été détecté, la clause de cet ensemble ayant le plus haut score est retirée de la formule. On espère ainsi détecter la clause appartenant à un faible nombre de MUS.

Exemple 5.10
Soit la formule CNF $\Sigma = \{\neg d \vee e,\ b \vee \neg c,\ \neg d,\ \neg a \vee b,\ a,\ a \vee \neg c \vee \neg e,\ \neg a \vee c \vee d,\ \neg b\}$ présentée dans l'exemple 3.1 et dont les deux MUS sont représentés en figure 3.1 page 33. Toute interprétation falsifiant par exemple la clause $\neg a \vee c \vee d$ falsifie également au moins une autre clause de Σ, puisque cette formule contient un MUS ne contenant pas cette clause. En supposant le parcours de la recherche locale pertinent, le score de cette clause est donc de 2, puisqu'*au minimum* quand cette clause est falsifiée, deux clauses le sont en fait dans la formule. Le même phénomène se produit pour toute clause appartement à la différence symétrique des 2 MUS de Σ. Au contraire, la clause $\neg b$ peut être seule falsifiée, puisqu'elle appartient à l'intersection des MUS de la formule. Si une telle interprétation est parcourue par la RL, alors son score est de 1.

Quand l'un des MUS de Σ est extrait, l'une de ses clauses est choisie pour être retirée. Notre heuristique sélectionne donc l'une des clause n'étant pas dans l'intersection des MUS de le formule, celle-ci ayant le score le plus élevé, ce qui assure de pouvoir extraire le second MUS

Fonction ASMUS(Σ : **une formule insatisfaisable**) : **une approximation de l'ensemble des MUS de** Σ
  $S \longleftarrow \emptyset$;
  Tant que ((Σ est insatisfaisable)) **faire**
    $MUS \longleftarrow$ `OMUS`(Σ) ;
    $S \longleftarrow S \cup \{MUS\}$;
    $c \longleftarrow$ une clause de MUS t.q. c ait le plus haut score (en terme de nombre clauses falsifiées minimum) ;
    $\Sigma \longleftarrow \Sigma \setminus \{c\}$;
  Fait
  Retourner S ;
Fin

Algorithme 13 – `ASMUS` (Approximating the Set of MUS)

de la formule. Clairement retirer une clause appartenant au contraire à l'intersection restaure la cohérence de Σ et plus aucun MUS ne pouvant donc être localisé par la suite.

Cette approche, baptisée `ASMUS` (pour Approximate Set of MUS), est décrite dans l'algorithme 13. Clairement, cette approche est incomplète. Cependant, elle fournit de très bons résultats relativement aux méthodes déjà proposées, comme illustré par les résultats expérimentaux de la table 5.5. Pour ces expérimentations, la limite de temps a été fixée à 20 000 secondes. Les instances AleatX_Y_Z sont le fruit du modèle standard de génération aléatoire [Chvátal & Szemerédi 1988], où X est le nombre de variables, Y le nombre de clauses. Les formules XAIMY_Z sont la concaténation de X formules AIMα_β avec, $\alpha = \frac{Y}{X}$ et $\beta = \frac{Z}{Y}$.

Nous avons comparé `ASMUS` avec l'algorithme complet proposé par [Liffiton & Sakallah 2005] d'un point de vue expérimental. La table 5.5 montre que les deux approches fournissent le nombre exact de MUS présents dans les instances AIM, avec des temps d'exécution similaires. Sur des instances plus difficiles, telles que les Aleat30_75_*, `ASMUS` extrait presque l'ensemble des MUS, et son temps d'exécution est souvent bien meilleur que celui de la méthode complète.

De plus, l'algorithme de [Liffiton & Sakallah 2005] peut avoir un temps exponentiellement long, puisque une formule CNF peut posséder un nombre exponentiel de MUS ; comme cette approche ne calcule individuellement les MUS qu'après avoir calculé l'ensemble des bases maximales consistantes de la formule (qui peuvent également être exponentielles), la procédure devient souvent intraitable en pratique. Au contraire, notre technique d'approximation ne souffre pas d'un tel revers et est *anytime*, puisque les MUS sont calculés les uns après les autres. Par exemple, considérons la formule aléatoire `Aleat20_70_2`. Celle-ci n'est composée que de 70 clauses, mais ces contraintes forment plus de 114 000 MUS. Cette instance étant d'une taille très petite, [Liffiton & Sakallah 2005] est capable de calculer l'ensemble de ses MUS. Toutefois, sur des formules légèrement plus grandes, comme `dp02u01` (213 variables, 376 clauses), l'ensemble des MUS ne peut être retourné en 20 000 secondes, alors que notre approche en extrait 14 en 26 secondes.

Au sein de ce genre de formules, toute clause appartient *a priori* à plusieurs MUS, et notre méthode ne peut tous les extraire. Toutefois, c'est justement dans de telles situations que les méthodes complètes se révèlent impuissantes, et où un compromis semble le bienvenu.

Instance	#var	#cla	L.&S.		ASMUS	
			#MUS	Temps	#MUS	Temps
aim100-1_6-no-1	100	160	1	0.18	1	0.31
aim200-1_6-no-1	200	320	1	0.14	1	0.68
aim200-1_6-no-2	200	320	2	0.22	2	0.76
aim200-2_0-no-3	200	400	1	0.12	1	0.56
aim200-2_0-no-4	200	400	2	0.26	2	0.88
Aleat20_70_1	20	70	127510	6.9	6	4.9
Aleat20_70_2	20	70	114948	10.8	13	8.7
Aleat30_75_1	30	75	11	59.82	7	2.2
Aleat30_75_2	30	75	9	26.84	8	2.9
Aleat30_75_3	30	75	10	12.84	10	3.7
Aleat50_218_1000	50	218	Time out		67	173
Aleat50_218_100	50	218	Time out		39	126
2AIM100_160	100	160	2	0.21	2	0.69
2AIM400_640	400	640	2	14.9	2	3.1
3AIM150_240	150	240	3	73.84	3	1.46
4AIM200_320	200	320	Time out		4	2.82
dp02u01	213	376	Time out		14	26.12
Homer06	180	830	Time out		2	17.47

TAB. 5.5 – Approcher l'ensemble de MUS

5.3 Applications à la gestion de bases de connaissances propositionnelles stratifiées

Les algorithmes développés pour l'approximation de MUS ont déjà trouvé de nombreuses applications dans des domaines variés tels que la vérification formelle, et en particulier dans celui du *model checking* [Clarke *et al.* 2003, McMillan 2005]. Toutefois, le concept de MUS, et plus généralement, d'ensembles de MUS tel que présentés dans ce chapitre, peuvent également s'avérer utiles dans la gestion de bases de connaissances. Dans cette section, nous montrons comment les algorithmes proposés pour l'extraction de ces ensembles peuvent être dérivés en approches efficaces pour la gestion de bases de connaissances propositionnelles stratifiées.

5.3.1 Notions préliminaires

Quand une base de connaissances (ou de croyances, dans ce contexte, nous utilisons les deux termes indifféremment) est incohérente, n'importe quelle information peut y être dérivée : la base est alors dite *triviale*, puisque *ex falso quodlibet*[1]. Il convient alors de ne pas considérer l'ensemble des croyances de la base, mais uniquement un sous-ensemble de celles-ci. Cependant, on préfère clairement une sous-base contenant un maximum d'informations de la base originale ; ainsi, la sous-base considérée sera une sous-formule maximale satisfaisable, ou MSS (cf. définition 5.1 page 72). Notons que de manière générale, on considère toujours l'une des MSS d'une base pour en utiliser ses croyances : en effet, si la base est consistante, elle est alors son propre et unique MSS. Dans le cas contraire, dériver de l'information nécessite le calcul d'au moins un de ces

[1] *Du faux, tout peut être dérivé.*

ensembles.

Si la valeur de chacune des connaissances de la base est identique, alors le simple calcul d'une solution MaxSat permet d'obtenir l'une des sous-bases non triviales contenant un maximum de connaissances. Toutefois, la plupart du temps, toutes les informations de la base ne sont pas sur un pied d'égalité : certaines d'entre elles présentent une importance ou un degré de certitude plus accru, et l'on préfère conserver ces connaissances plutôt que d'autres. On définit alors le concept de base de connaissances stratifiées.

Soit KB une formule propositionnelle sous forme normale conjonctive (CNF). On considère KB comme un ensemble de n clauses, cet ensemble étant interprété comme une conjonction. Supposons que les n clauses de KB soient partitionnées en t strates préférentielles telles que $KB = S_1 \cup S_2 \cup \cdots \cup S_t$.

Soient $c_i \in S_a$, $c_j \in S_b$ avec $a < b$. Dans ce cas, on a c_i préférée à c_j, c'est-à-dire que si la restauration de la satisfaisabilité de KB passe par la suppression de l'une de ces deux clauses, alors c_j sera supprimée. Ainsi définies, les strates représentent une relation de pré-ordre total sur les clauses de KB.

On trouve dans la littérature de nombreux formalismes permettant la caractérisation de sous-bases consistantes préférées d'une base inconsistante de croyance KB (cf. par exemple [Lehmann 1995]). Dans cette étude, nous avons choisi d'utiliser à titre d'exemple celui présenté dans [Benferhat *et al.* 1993]. Soit KB une base de croyances, $A = A_1 \cup A_2 \cup \cdots \cup A_t$ et $B = B_1 \cup B_2 \cup \cdots \cup B_t$ deux sous-ensembles consistants de KB, avec $A_i = A \cap S_i$ et $B_i = B \cap S_i$, avec i un entier positif tel que $i \leq t$.

Définition 5.4 (sous-base maximale consistante préférée)
On note $A <<_{S_1 \cup \cdots \cup S_t} B$ *si et seulement si* $\exists i$ *tel que* $|A_i| < |B_i|$ *et* $\forall j < i$, $|A_j| = |B_j|$. *Une sous-base consistante de KB qui est maximale par rapport à* $<<_{S_1 \cup \cdots \cup S_t}$ *est appelée* sous-base maximale consistante préférée de KB.

Dans la suite de ce chapitre, nous présentons plusieurs pistes basées sur les algorithmes proposés dans le cadre d'extraction de MUS permettant le calcul de sous-bases maximales consistantes préférées selon cette préférence qualifiée de « lexicographique » dans la littérature.

5.3.2 Algorithmes pour le calcul de bases maximales préférées

Classiquement, pour une base de croyances KB, 2 types d'inférence d'une formule Γ sont possibles :

- l'inférence crédule, qui est permise si Γ peut être inféré de manière classique à partir de l'une des bases maximales préférées de KB ;
- l'inférence sceptique, qui est permise si Γ peut être inféré de manière classique à partir de toutes les bases maximales préférées de KB .

Clairement, si une base de connaissances KB est consistante, alors elle est sa propre sous-base maximale consistante préférée. Cependant, l'ajout ou la révision d'informations dans une base de croyances peut générer une inconsistance, qu'il nous incombe de gérer. Dans ce cas, la base peut posséder une ou plusieurs sous-bases préférées. Plusieurs approches ont, par le passé, été proposées pour l'extraction de telles bases (cf. par exemple [Grégoire 1999]), mais la plupart se concentrent sur l'approximation de la base de croyances préférée, ce problème étant placé très haut dans la hiérarchie polynomiale. En effet, décider si une formule est conséquence pour l'inférence sceptique de KB selon la préférence « lexicographique » définie précédemment est Δ_2^p-complet, tandis que le problème de décision de la conséquence pour l'inférence crédule a quant à elle été prouvée Σ_2^p-complet [Cayrol *et al.* 1998].

L'une des solutions pour inférer dans ce contexte est de parcourir les modèles préférées de KB, c'est-à-dire les modèles falsifiant des contraintes représentant les connaissances les moins préférées de la base. Cependant, cette technique induit le parcours de ces modèles optimaux pour chaque calcul d'inférence. Une autre solution est la compilation de l'information, à travers le calcul effectif des sous-bases préférées de KB. Une fois calculées, celles-ci sont stockées et utilisées pour les différentes requêtes d'inférence. C'est cette voix qui est explorée ici. L'objectif de la suite de ce paragraphe est donc la dérivation d'un algorithme pour le calcul de bases maximales préférées à partir d'algorithmes mis au point pour l'extraction de MUS.

Soit une base de croyances S composée de n strates $S = S_1 \cup S_2 \cup \cdots \cup S_n$ avec $c_a \in S_i$ une croyance strictement préférée à $c_b \in S_j$ si et seulement si $i < j$. Une première approche pour calculer ces bases maximales préférées est d'associer à chaque croyance c_i d'une strate S_a un poids p_i tel que :

$$p_i > \sum_{\{c_j \in s_u \mid s_t \text{ est préférée à } s_u\}} p_j$$

Ainsi, toutes les croyances d'une strate doivent posséder le même poids, et ce poids doit excéder la somme des poids de toutes les croyances moins préférées. Une fois cette pondération établie, il « suffit » pour calculer une base maximale préférée d'utiliser la variante pondérée du problème MaxSat, à savoir *weighted MaxSat*. Par rapport à cette formule propositionnelle pondérée, toute solution optimale pour *weighted MaxSat* est une base de croyances préférée. Ainsi, l'obtention d'une seule de ses solutions optimales permet d'établir partiellement l'inférence crédule tandis que l'inférence sceptique nécessite le calcul de toutes ses solutions optimales.

Nous proposons un nouvel algorithme basé sur `HYCAM` permettant de calculer exhaustivement ou non les sous-bases maximales préférées d'une base de croyances inconsistante. Celui-ci se décompose en plusieurs étapes, qui sont détaillées ici.

5.3.3 Calcul du seuil d'inconsistance

Plutôt que de calculer les solutions optimales grâce à un appel à *weighted MaxSat*, nous pensons qu'il est préférable de connaître *a priori* les croyances invariantes, c'est-à-dire celles qui seront constantes pour toutes les bases préférées. Cela conduit au concept de *partionnement de variabilité* d'une base de croyances.

Définition 5.5 (partionnement de variabilité)
Toute base de croyances S peut être partionnée en $S = S_{Invar^+} \cup S_{Var} \cup S_{Invar^-}$ avec :

- S_{Invar^+} *:* partie invariante positive. *Contient les croyances appartenant à toutes les bases maximales préférées.*
- S_{Var} *:* partie variable. *Contient les croyances contenues dans certaines bases préférées, mais pas toutes.*
- S_{Invar^-} *:* partie invariante négative. *Contient les croyances n'appartenant à aucune base maximale préférée.*

On appelle ces 3 ensembles partionnement de variabilité *d'une base de croyances.*

Toute technique permettant de catégoriser certaines croyances de la base dans un de ces 3 sous-ensembles conduit *a priori* à un gain en pratique pour le calcul des bases maximales préférées, puisque cela induit une connaissance sur les informations appartenant nécessairement aux sous-bases desquelles on pourra inférer. Or, il est possible de construire partiellement la partie invariante positive, moyennant un coût relativement bas en pratique. Pour cela, le concept de *seuil d'inconsistance* se révèle utile :

Définition 5.6 (seuil d'inconsistance)
Soit $S = S_1 \cup \cdots \cup S_n$ une base de croyances inconsistante partionnée en n strates. L'entier t (tq $1 \leq t < n$) est appelé seuil d'inconsistance *si et seulement si $S_1 \cup \cdots \cup S_t$ est satisfaisable et $S_1 \cup \cdots \cup S_t \cup S_{t+1}$ est insatisfaisable.*

Clairement, le calcul du seuil d'inconsistance permet d'obtenir une partie de l'ensemble S_{Invar^+}.

Propriété 5.4
Soit $S = \{S_1, \ldots, S_n\}$ une base de croyances et t son seuil d'inconsistance. Nous avons :

$$\forall i \in [1..t], \forall c \in S_i, c \in S_{Invar^+}$$

Ainsi, le seuil d'inconsistance, calculé incrémentalement à partir des croyances les plus préférées aux moins préférées, permet d'obtenir partiellement les éléments contenus dans la partie invariante positive. En pratique, il suffit de lancer une recherche locale, peu coûteuse en ressources, sur la première strate de la base. Si un modèle est exhibé, alors on ajoute la deuxième strate, et ainsi de suite, jusqu'à un échec de la recherche locale pour la satisfaisabilité. On obtient ainsi un premier seuil t_{RL} de croyances appartenant à la partie invariante positive. Une recherche exacte du seuil d'inconsistance peut alors être envisagée grâce des algorithmes complets de type DPLL. Ceci peut-être effectué de manière incrémentale (en ajoutant les strates une à une). On n'a donc à faire que des tests de satisfaisabilité, peu coûteux en pratique, et un seul test d'insatisfaisabilité pour prouver que le seuil a été dépassé. Une autre option envisageable est le découpage dichotomique de la base, qui garantit un nombre logarithmique d'appels à un solveur, mais dans ce cas, des tests CoNP plus nombreux sont à prévoir.

Ces connaissances préliminaires, obtenues grâce à l'une de ces techniques, permettent *a priori* un calcul plus efficace de la/des base(s) préférée(s) par la suite, grâce à l'algorithme décrit dans le paragraphe suivant.

5.3.4 Calcul exact des sous-bases préférées

Une fois le seuil d'inconsistance calculé, on sait que certaines connaissances appartiennent à toutes les bases préférées, et il est donc inutile de considérer toutes les bases excluant une de ces croyances. De même, si une autre technique algorithmique permet de déduire d'autres croyances appartenant à la partie invariable positive, celles-ci permettraient de réduire l'ensemble des sous-bases candidates à la préférence générée par la stratification.

De manière duale, toute croyance dont on connaît l'appartenance à la partie négative du partionnement de variabilité pourrait dès lors être « oubliée » lors du calcul de bases maximales préférées, ce qui réduirait le nombre des sous-bases à tester. Malheureusement, aucune méthode efficace n'a pour l'instant été proposée pour la détection de telles strates.

Un appel à *weighted MaxSat* comme exprimé dans le début de cette section n'a donc plus lieu d'être ; le gain potentiel apporté par le seuil d'inconsistance ne serait pas exploité.

Dans [Cha *et al.* 1997], une nouvelle variante au problème SAT est introduite. Celle-ci se nomme *partial MaxSat* et peut être exprimée comme suit : étant données deux formules Γ_A et Γ_B, trouver une interprétation qui satisfasse toutes les clauses de Γ_A et qui maximise le nombre de clauses satisfaites de Γ_B. On voit assez clairement comment il est possible d'utiliser un tel problème (pour lequel différents algorithmes, aussi bien complets qu'incomplets ont été développés) pour notre calcul de base préférée. En imposant $\Gamma_A = S_{Invar^+}$ et $\Gamma_B = S \backslash S_{Invar^+}$, un appel à ce type d'algorithme calculera la sous-base satisfaisable contenant toutes les croyances

Fonction compute_Max_Pref_KB($KB = S_1 \cup \cdots \cup S_n$: une **base de connaissances insatisfaisable**) : **la base maximale consistante préférée de** KB
  $t \longleftarrow$ cherche_Seuil_Inconsistance(KB) ;
  Pour chaque clause $c \in S_{t+1} \cup \cdots \cup S_n$ **faire**
    $c \longleftarrow c \cup \{y_c\}$;
  Fin Pour
  $MSS \longleftarrow$ RL_approximation(KB) ;
  $current\_opt\_MSS \longleftarrow$ select_optimale(MSS,$S_1 \cup \cdots \cup S_n$) ;
  $borne \longleftarrow 1$;
  Tant que (Non(prouvée_optimale($current\_opt\_MSS$)) **ET** Σ_y est satisfaisable) **faire**
    $current\_opt\_MSS \longleftarrow current\_opt\_MSS \cup$ DPLL_avec_borne($\Sigma_y, borne$)
    $borne \longleftarrow borne + 1$;
    $current\_opt\_MSS \longleftarrow$ select_optimale($current\_opt\_MSS$,$S_1 \cup \cdots \cup S_n$) ;
  Fait
  Retourner $current\_opt\_MSS$;
Fin

Algorithme 14 – Extraction de bases maximales préférées via `HYCAM`

de la partie invariante positive ainsi qu'un maximum de croyances du reste de la base, mais en faisant fi de la relation préférentielle.

Pour obtenir la ou les bases préférées, nous allons utiliser une variante à *partialMaxSat*, qui est définie ici.

Définition 5.7 (partial (#)weighted MaxSat)
Étant données deux formules Γ_A et Γ_B (avec un poids associé à chaque clause de Γ_B), le problème partial (#)weighted MaxSat *consiste à trouver une (toutes les) interprétation(s) qui satisfait(font) toutes les clauses de Γ_A et qui maximise(nt) la somme des poids des clauses satisfaites de Γ_B.*

Avec une connaissance idéale du partitionnement de variabilité de la base, un calcul plus efficace (en terme de temps nécessaire) pourrait être effectué, en imposant $\Gamma_A = S_{Invar^+}$ et $\Gamma_B = S_{var}$. En tenant compte de cette étape préliminaire, un algorithme permettant le calcul de la/des base(s) maximale(s) préférée(s) d'une base de connaissances, fondé sur `HYCAM`, peut être développé. Celui-ci peut bénéficier de l'approximation fournie par la recherche locale comme étape préliminaire, ainsi que des informations fournies par le seuil d'inconsistance.

L'approche proposée, nommée `compute_Max_Pref_KB`, est dépeinte dans l'algorithme 14. Celle-ci se décompose en plusieurs étapes. Tout d'abord, comme décrit précédemment, une recherche du seuil d'inconsistance de la base est effectuée, soit par recherche locale puis méthode complète sur un nombre de strates grossissant, soit par dichotomie. Ensuite, conformément aux principes d'`HYCAM`, des sélecteurs de clauses sont ajoutés à chaque élément de la base, sauf aux clauses détectées comme faisant partie des strates invariantes positives. En effet, ces littéraux supplémentaires permettent de « désactiver » certaines clauses, quand celles-ci sont en réalité falsifiées. Comme les clauses contenues dans cette partie de la base appartiennent à toutes ses

sous-bases préférées, elles ne peuvent être falsifiées durant la recherche, ce qui réduit exponentiellement le nombre de bases à considérer (sauf si $seuil = 1$).

Comme dans l'algorithme original, une recherche locale est ensuite exécutée afin d'obtenir un ensemble MSS candidats. Parmi cet ensemble, la ou les bases préférées suivant la stratification fournie sont sélectionnées : il s'agit donc des bases courantes préférées ($current\_opt\_MSS$). Cette sélection des optimaux, effectuée dans l'algorithme 14 par la fonction $select\_optimale$, est clairement polynomiale dans le nombre de MSS extraits.

La première itération de la partie complète d'`HYCAM` est ensuite exécutée, retournant l'ensemble des bases maximales satisfaisables de taille $n - 1$. Cette recherche exhaustive peut bien entendu tirer parti de l'information fournie par la recherche locale préliminaire, même si ces sous-formules ne sont pas optimales par rapport aux préférences conférées par les strates. À la fin de cette itération, les nouvelles bases courantes préférées sont calculées. Si celles-ci peuvent être prouvées optimales, alors la procédure retourne cet ensemble avant de stopper son exécution. Malheureusement, en l'absence de toute information *a priori* sur la base, seule l'appartenance des connaissances rejetées par les MSS à la strate la moins préférée permet une telle conclusion. En outre, si la formule obtenue Σ_y n'est pas satisfaisable, alors il n'existe pas de MSS de taille inférieure, et les MSS courants sont donc optimaux. Si aucun de ces deux cas n'est rencontré, une nouvelle itération est effectuée, à la recherche de nouveaux MSS. Ces opérations sont répétées jusqu'à ce que l'une des conditions, preuve de l'optimalité des sous-bases obtenues, soit remplie. Ainsi défini, l'algorithme présenté est correct et complet pour le problème d'extraction des sous-bases maximales consistantes préférées.

5.3.5 Tests expérimentaux

Afin d'évaluer empiriquement l'approche proposée, nous avons modifié `HYCAM` suivant les principes de l'algorithme 14. Il existe certaines bases de croyances stratifiées provenant de problèmes réels, mais celles-ci sont toutes, à notre connaissance, d'une taille très réduite ; elles peuvent ainsi être aisément traitées par n'importe quel algorithme bénéficiant des dernières avancées des approches de type DPLL. Ces bases « triviales » ne permettraient donc pas d'évaluer l'efficacité de notre nouvel algorithme. En outre, il ne semble malheureusement pas exister de bases de taille conséquente permettant une telle évaluation.

Nous avons donc choisi de stratifier un problème académique de manière à obtenir une base représentant un exemple concret non trivial. Notre attention s'est retenue sur le problème des 8 reines et des 5 cavaliers, qui nous vient d'une démonstration de localisation de l'incohérence par l'heuristique de choix de variables `dom/wdeg` [Boussemart *et al.* 2004], décrite en section 6.2.2 page 109. Ce problème consiste à placer 8 reines (notées r_1 à r_8) sur un échiquier de 8 cases de côté, de manière à ce qu'aucune d'entre elles ne se situent sur la même ligne, colonne ou diagonale. De surcroît, le problème consiste à placer sur ce même échiquier 5 cavaliers (notées c_1 à c_5) de façon à ce que chacun puisse prendre 2 de ces voisins, selon les règles des échecs. Ce problème est connu pour être insatisfaisable, et l'on choisit donc de stratifier les contraintes qui découlent de sa modélisation booléenne afin d'obtenir une solution approchée suivant les préférences suivantes, exprimées de la forte à la plus faible :

- *Strate 1* : clause « at-least » et « at-most ». Ces clauses sont garantes de l'existence et de l'unicité de chaque pièce sur l'échiquier. Grâce à cet ensemble de clauses clairement cohérent, on s'assure que dans chaque solution relaxée du problème, toute pièce (reines et cavaliers) occupe une et une seule case ;

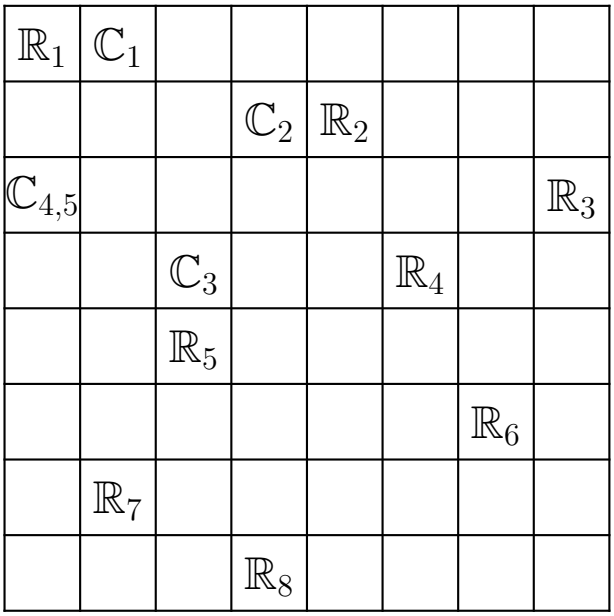


FIG. 5.6 – Une solution au problème des 8 reines et des 4 cavaliers

- *Strate 2* : problème des reines. On décide de donner une « priorité » préférentielle très forte à la résolution du problème des reines, par rapport aux autres contraintes du problème ;
- *Strate 3* : problème des cavaliers. Cette strate inclut les contraintes qui stipulent que tout cavalier doit être en position de prendre 2 de ses voisins (ou éventuellement être sur la même case) ;
- *Strate 4* : exclusions mutuelles des cavaliers c_1, c_2 et c_3. Si une solution n'est possible qu'en relaxant le problème avec 3 cavaliers au lieu de 5, alors seules des contraintes de cette strate seront violées dans la ou les solution(s) préférée(s) apportée(s) ;
- *Strate 5* : exclusion des cavaliers c_4 et c_5. Si une so lution n'est possible qu'en relaxant le problème avec 4 cavaliers seulement au lieu de 5, alors seules des contraintes de cette strate seront violées dans la ou les solutions préférée(s) apportée(s). Si de plus la ou les solution(s) préférée(s) exclu(en)t des connaissances des strates 4 et 5, alors le problème ne possède des solutions qu'en ne considérant que 2 cavaliers ;
- *Strate 6* : exclusion des reines/cavaliers. Dans cette strate est codée l'information interdisant à une reine et un cavalier d'être placés sur la même case. Ainsi, si les problèmes des 8 reines et des 5 cavaliers sont consistants indépendamment, c'est-à-dire s'ils sont placés sur 2 échiquiers distincts, l'algorithme pourra déterminer une solution excluant uniquement des connaissances de cette strate. C'est alors leur cumul sur le même échiquier qui serait la cause de l'incohérence de ce problème.

Le problème ainsi modélisé implique 376 variables et possède 30 732 clauses partitionnées suivant la schéma de stratification précédent. On peut noter qu'aucune information n'est donnée à proprement parler sur l'exclusion mutuelle des reines. Toutefois, cette connaissance est induite

#Variable	#Clause	#time out	#temps moyen (hors time out)
75	450	0	0,05
80	480	0	0,06
85	510	0	0,05
90	540	0	0,06
95	570	0	0,07
100	600	2	22,1
105	630	4	28,6
110	660	5	75,8
115	690	45	314

TAB. 5.6 – Quelques résultats de calcul de bases préférées sur des problèmes générés aléatoirement

de manière implicite par le problème des reines, et ces clauses seraient par conséquent redondantes. L'algorithme développé a été exécuté afin d'obtenir une solution maximale cohérente préférée suivant la stratification, et a retourné un ensemble de solutions qui n'excluent que des connaissances faisant partie de la strate 5 en 28 secondes, et dont l'optimalité a été prouvée au bout de 87 secondes. L'une de ces solutions est dépeinte dans la figure 5.6, de nombreuses autres lui étant symétriques.

Clairement, nous pouvons constater que ce problème n'admet de solutions que si on autorise deux cavaliers à être sur le même emplacement, ce qui revient à n'en considérer que 4. Son optimalité prouve également que même en prenant les 2 problèmes originaux de manière indépendante, aucune solution ne peut être trouvée. Leur hybridation n'est donc pas la cause de son incohérence, mais c'est bien le problème des 5 cavaliers qui est en cause. En le relaxant, l'ensemble admet donc des solutions qui sont mises au jour par ce nouvel algorithme.

Afin d'évaluer les limites de cette nouvelle méthode, nous avons également effectué différents tests expérimentaux sur des bases de connaissances générées aléatoirement. Pour cela, nous avons produit des formules 3-SAT de différentes tailles suivant le modèle standard de génération aléatoire, en choisissant de construire des problèmes possédant un nombre six fois supérieur de clauses que de variables. Neuf séries de 50 bases de connaissances, allant de 75 à 115 variables, ont ainsi été créées. Chaque base a été partitionnée en 6 strates de taille égale (en nombre de clauses), fournissant un jeu de tests pour notre approche. Chaque base a été soumise à notre approche, avec une limite de 1 heure de temps CPU. Dans la table 5.6 sont reportés les résultats obtenus, en détaillant le nombre de *time out* obtenus (c'est-à-dire le nombre de fois où les bases optimales n'ont pas été retournées) et la moyenne de temps nécessaire, en omettant les cas de dépassement de temps limite.

Comme on peut le constater, malgré le rapport élevé du nombre de variables sur le nombre de clauses, générant de nombreuses sources d'inconsistance et donc un nombre élevé de bases maximales cohérentes (rappelons que le rapport C/V d'équi-probabilité SAT/UNSAT pour les formules 3-SAT est estimé à 4,25), notre algorithme parvient à extraire toutes les solutions optimales de chaque base testée comportant moins de 100 variables. Toutefois, pour les bases de tailles plus importantes, il arrive que notre méthode ne puisse fournir un ensemble de solutions en les prouvant optimales en 1 heure de temps CPU, et des dépassements de limite de temps se produisent (*#time out* non nul). Ce nombre, tout comme le temps moyen, grimpe même assez logiquement de façon très rapide avec la taille des bases testées.

On peut cependant noter certaines caractéristiques intéressantes de notre algorithme ren-

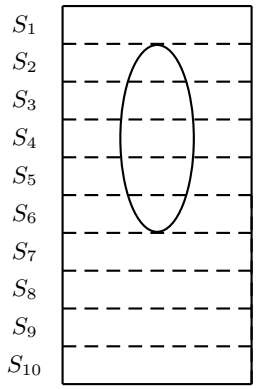


FIG. 5.7 – Exemple de répartition d'un MUS au sein d'une base de connaissances stratifiée

dant son utilisation intéressante, même dans ces cas de dépassement de temps. Premièrement, il est clair que celui-ci est incrémental, dans le sens où il maintient à chaque instant un ensemble de solutions courantes optimales, qui ne peut qu'être de meilleure qualité avec le temps passé à son calcul. Ceci procure à l'approche un aspect *anytime*. Ainsi, si l'optimalité n'est pas requise, différents degrés de qualité peuvent être obtenus, suivant les ressources dont dispose l'utilisateur. En outre, comme dans la résolution exacte de la plupart des problèmes d'optimisation, la majeure partie du temps de calcul n'est pas consacrée au calcul de la solution optimale, mais à la preuve de son optimalité. Ainsi, il est probable que dans de nombreuses situations où un dépassement de temps a été observé, la ou les base(s) maximale(s) préférée(s) ai(en)t été découverte(s), mais l'inexistence de meilleures solutions n'a pas été prouvée.

Différentes pistes peuvent être envisagées afin de réduire cette preuve d'optimalité en pratique. L'une d'elle, qui semble prometteuse, pourrait utiliser la notion de MUS. En effet, si on imagine une base de connaissances stratifiée, on pourrait réduire le nombre de bases à tester pour prouver l'optimalité par l'extraction d'un des MUS de la base. Considérons par exemple une base de connaissances contenant 10 strates (numérotées S_1 à S_{10} de la plus préférée à la moins préférée), et possédant un MUS réparti suivant la figure 5.7.

Si une technique permet d'extraire ce MUS efficacement, alors on sait qu'au moins une connaissance de la strate 6 (ou plus préférée) est incluse dans toute solution optimale. Si par exemple, lors de la première itération, une base maximale excluant uniquement une information de la $6^{ème}$ strate est obtenue, alors on pourrait grâce à ce MUS conclure immédiatement à son optimalité sans avoir à calculer de plus grandes MSS, dans l'espoir d'en trouver certains se situant exclusivement dans les strates inférieures. En effet, il n'en existe aucune, puisque qu'au moins un élément du MUS découvert est falsifié par toute interprétation. Ce type d'information pourrait être utilisé dans la fonction *prouvée_optimale* qui ne se contenterait plus de vérifier l'appartenance des bases préférées courantes à la strate la moins préférée. La connaissance d'un MUS de la base pourrait en effet permettre d'affaiblir la condition de vérité de ce prédicat, rendant inutile l'exploration de nouvelles sous-bases cohérentes afin de prouver l'optimalité des bases déjà calculées.

Le nombre de sous-bases à tester pourrait encore être réduit par l'obtention d'un ensemble de MUS de la base, tel que les couvertures inconsistantes présentées précédemment. Ainsi, une utilisation astucieuse de la dualité entre les MSS et les MUS pourrait s'avérer efficace afin d'éviter un parcours exhaustif des sous-bases à tester, augmentant potentiellement la classe des bases de

connaissances traitables de manière exacte en pratique. Il semble clair que la notion de MUS est promise à un grand avenir dans la résolution de ce genre de problème. Son rôle fera l'objet de futures investigations, à la fois théoriques et pratiques.

En outre, les algorithmes présentés dans ce chapitre peuvent être dérivés dans d'autres cadres proposés pour la logique non monotone, comme les ATMS introduits par [de Kleer 1986]. Ce concept dont le sigle a pour signification « *Assumption-Based Truth Maintenance System* » est un outil de maintient de la cohérence permettant entre autre d'effectuer des raisonnements révisables sur des bases de connaissances. Pour cela, dans le cadre d'une base de connaissance donnée Σ, les ATMS utilisent un ensemble d'hypothèses noté H et peuvent déterminer quel(s) sous-ensemble(s) $E \subset H$ (appelé *label*) permet(tent) d'expliquer (au sens de déduire) un littéral donné l. En particulier, l'utilisation des ATMS pour le raisonnement non-monotone suppose de pouvoir maintenir, dans le contexte de Σ, les environnements incohérents, c'est-à-dire les ensembles minimaux d'hypothèses ne pouvant cohabiter *(nogood)*. Ces ensembles d'hypothèses ne sont pas des MUS, puisqu'ils ne sont minimaux que pour les hypothèses contenues, et non pour les connaissances de la base Σ. Toutefois, les approches proposées dans ce chapitre pourraient être adaptés pour effectuer efficacement ce calcul.

5.4 Conclusions

Dans ce chapitre ont été dépeintes nos contributions au problème d'extraction de tous les MUS d'une formule CNF insatisfaisable. Dans un premier temps, nous avons amélioré la meilleure approche exacte connue par l'utilisation d'une recherche locale préliminaire permettant de lui faire gagner un ordre de grandeur en pratique. L'algorithme ainsi obtenu est nommé `HYCAM`. Ensuite, pour pallier l'éventuelle exponentialité du nombre de MUS au sein d'une formule, nous avons défini différents ensembles de MUS permettant par leur réparation de restaurer la satisfaisabilité de l'ensemble de la formule. Ces ensembles, ou couvertures inconsistantes, forment une approximation de tous les MUS d'une formule. Enfin, nous avons appliqué un des algorithmes développés à la gestion de base de connaissances stratifiées, et avons discuté de la possible utilisation des MUS et des ensembles de MUS dans cet objectif.

Troisième partie

Extensions au problème de satisfaction de contraintes (CSP)

Chapitre 6

Une (courte) introduction aux CSP

Sommaire

Le problème de satisfaction de contraintes (ou CSP), est, à l'instar de SAT, l'un des thèmes de recherche les plus actifs en Intelligence Artificielle. Cela n'est pas leur seul point commun, puisque tous deux consistent en la détermination de la cohérence d'un système de contraintes.

Dans ce chapitre, une introduction au problème CSP est effectuée, à travers sa définition formelle ainsi que de la description des algorithmes actuellement utilisés en pratique pour sa résolution. Enfin, le concept d'ensemble minimalement inconsistant de contraintes (ou MUC) d'un CSP est présenté.

6.1 Définitions formelles et représentations

6.1.1 Définitions basiques

Définition 6.1 (CSP)
Un problème de satisfaction de contraintes (CSP) est défini par un couple $\langle V, C \rangle$ *où :*

- *V est un ensemble fini de variables $\{v_1, \ldots, v_n\}$ prenant chacune leur valeur dans un ensemble discret fini appelé* domaine *et noté $dom(v_i)$ ($i \in [1..n]$) ;*
- *C est un ensemble fini de contraintes $\{c_1, \ldots, c_m\}$ où une contrainte c_j ($j \in [1..m]$) porte sur un sous-ensemble de V, noté $Var(c_j)$, et définit une relation de compatibilité sur les valeurs de $Var(c_j)$, noté $R(c_j)$.*

La relation de compatibilité de chaque contrainte induit un ensemble de tuples interdits sur les valeurs des variables impliquées par cette contrainte.

Définition 6.2 (Tuples interdits)
Soient un CSP $\langle V, C\rangle$ et une contrainte $c \in C$ telle que $Var(c) = \{v_{c_1}, \ldots, v_{c_l}\}$. Un tuple interdit est un élément de $dom(v_{c_1}) \times \cdots \times dom(v_{c_l})$ tel que la relation de compatibilité $R(c)$ n'est pas satisfaite. L'ensemble des tuples interdits pour une contrainte c est noté $T(c)$.

Il est ainsi possible de définir indifféremment un CSP soit par le couple $\langle V, C = \{c_1, c_2, \ldots, c_n\}\rangle$, soit par le couple $\langle V, \{(Var(c_1), T(c_1)), (Var(c_2), T(c_2)), \ldots, (Var(c_n), T(c_n))\}\rangle$. Dans un souci de concision, le terme « tuple » sera utilisé à la place de « tuple interdit » dans la suite de ce document.

Définition 6.3 (instanciation partielle (localement consistante))
Soit $P = \langle V, C\rangle$ un CSP. Une instanciation partielle *A d'un ensemble de variables $S = \{v_1, \ldots, v_k\}$ tel que $S \subset V$ affecte à chaque variable de S une valeur de son domaine. Il s'agit donc d'un k-uplet de $dom(v_1) \times \cdots \times dom(v_k)$.*
De plus, on dit qu'une instanciation partielle A d'un ensemble de variable $S \subset V$ est localement consistante *si et seulement si elle satisfait toutes les contraintes portant exclusivement sur des variables de S.*

Définition 6.4 (Contrainte falsifiée)
Une contrainte c d'un CSP $\langle V = \{v_1, \ldots, v_n\}, C\rangle$ est falsifiée par une affectation $A \in dom(v_1) \times \cdots \times dom(v_n)$ des variables si et seulement si la projection de A sur $Var(c)$ est incluse dans $T(c)$.

Un CSP $P = \langle V, C\rangle$ est dit *satisfaisable* s'il existe une affectation A telle qu'aucune contrainte $c \in C$ ne soit falsifiée. A est alors appelée *modèle* de P. Si une telle affectation n'existe pas, le CSP est dit *insatisfaisable*.

Exemple 6.1
Soit $P = \langle V, C\rangle$, avec :
- $V = \{a, b, c\}$ et $dom(a) = dom(b) = dom(c) = \{0, 1, 2\}$;
- $C = \{a \neq c,\ (a + b + c) < 5\}$

On a :
$P = \langle V, C\rangle$
$\langle \{a, b, c\}, \{c_1,\ c_2\}\rangle$
$\langle \{a, b, c\}, \{a \neq c,\ (a + b + c) < 5\}\rangle$
$\langle \{a, b, c\}, \{(Var(c_1),\ T(c_1)),\ (Var(c_2),\ T(c_2))\}\rangle$
$\langle \{a, b, c\}, \{(\{a, c\},\ \{(0, 0), (1, 1), (2, 2)\}),\ (\{a, b, c\},\ \{(1, 2, 2), (2, 1, 2), (2, 2, 1), (2, 2, 2)\})\}\rangle$

L'affectation $A = (a = 1,\ b = 0,\ c = 1)$ n'est pas un modèle de P : il falsifie c_1, puisque sa projection sur $Var(c_1)$ est incluse dans $T(c_1)$:

$$(a = 1,\ b = 0,\ c = 1) \overset{Proj(Var(c_1))}{\Longrightarrow} (1, 1) \in T(c_1)$$

L'affectation $A_M = (a = 0, b = 2, c = 1)$ est un modèle de P, puisqu'il ne falsifie aucune contrainte de C. P est donc satisfaisable.

Propriété 6.1
Décider si un CSP est satisfaisable est un problème NP-complet.

6.1.2 Arité d'un CSP

Au sein d'un CSP, une contrainte peut impliquer un certain nombre de variables du problème. On peut toutefois définir une forme particulière de CSP sous condition d'un nombre limité de variables impliquées par chaque contrainte.

Définition 6.5 (CSP binaire)
Soit $P = \langle V, C \rangle$ *un CSP.* P *est dit* binaire *si et seulement pour* $\forall c \in C,\ |Var(c)| \leq 2$.

Définition 6.6 (CSP n-aire)
Tout CSP non binaire est dit n-aire.

Exemple 6.2
Soit P le CSP de l'exemple 6.1. P est n-aire, puisqu'il contient la contrainte « $(a + b + c) < 5$ » qui porte sur plus de 2 variables du problème.

Propriété 6.2
Tout CSP peut être transformé en temps polynomial en un CSP binaire équivalent.

À ce jour, plusieurs transformations ont été proposées : l'une des premières, appelée méthode *duale* [Dechter *et al.* 1989], nous vient du domaine des bases de données, et consiste à générer un CSP binaire en transformant chaque contrainte du problème original en variables de la nouvelle représentation. Quelques modifications syntaxiques permettent ensuite d'obtenir un CSP binaire équivalent. Une autre technique connue est la transformation « avec variables cachées » [Dechter 1990], qui permet cette fois de conserver les variables originales du problème, même si de nouvelles sont introduites pour *simuler* les contraintes n-aires à travers des contraintes binaires.

Ainsi, dans l'absolu, tout CSP peut se ramener à un CSP binaire équivalent, moyennant une opération de transformation polynomiale. On peut donc ne considérer que les CSP binaires, puisque ceux-ci sont suffisamment expressifs pour inclure tout autre problème mettant en relation plus de deux variables au sein d'une contrainte.

6.1.3 Représentations graphiques

Plusieurs procédés ayant pour objectif la visualisation d'un CSP ont été mis au point. L'un des plus utiles est certainement le *réseau de contraintes*.

Propriété 6.3
Tout CSP peut être représenté par un graphe appelé réseau de contraintes. *Dans ce graphe, chaque nœud représente une variable et chaque arête une contraintes entre les variables.*

Cette représentation sous forme de graphe n'est valide que pour les CSP binaires. Dans le cas n-aire, il s'agit d'un *hyper-graphe*, où les contraintes sont représentées par des *hyper-arêtes*, ce qui n'est pas une représentation visuelle très satisfaisante. Toutefois, comme tout CSP peut être ramené à un CSP binaire, nous avons la garantie de pouvoir visualiser par ce type de graphes n'importe quel problème.

Exemple 6.3
Soit un CSP $P = \langle V, C \rangle$, avec $V = \{a, b, c, d\}$ et $C = \{a > b,\ (b+c)\ est\ pair,\ c \neq d,\ (b+d) < 4\}$. Les variables de P prennent leur valeur dans $\{1, 2, 3\}$ sauf b dont le domaine est $dom(b) = \{1, 2\}$. P est clairement satisfaisable, l'affectation $A = \{a = 3,\ b = 1,\ c = 1,\ d = 2\}$ est l'un de ses modèles. Le réseau de contraintes associé à P est représenté en figure 6.1.

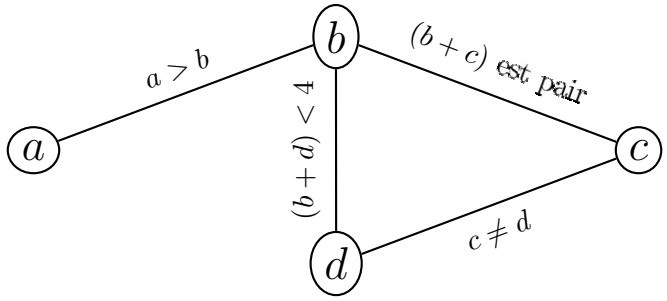


FIG. 6.1 – Réseau de contraintes associé au CSP de l'exemple 6.3

Le réseau de contraintes est un outil intéressant de modélisation d'un CSP, mais il ne permet pas de visualiser les tuples de valeurs autorisés ou interdits entre variables. Pour cela, on définit le graphe de micro-structure d'un CSP.

Propriété 6.4
Tout CSP composé de n variables peut être représenté par un graphe n-parti appelé graphe de micro-structure. *On regroupe au sein de chaque « partie » de ce graphe des nœuds représentant les valeurs du domaine de chaque variable, et une arête relie chaque couple interdit (ou autorisé) de valeurs par les contraintes du problème.*

Remarque 6.1
Dans la suite de ce document, nous prendrons comme convention de représenter les tuples interdits du problème comme arêtes du graphe de micro-structure bien qu'usuellement, ce sont plutôt les tuples autorisés qui sont représentés dans ce type de graphe. L'ensemble des tuples autorisés étant clairement le complémentaire de celui des tuples interdits, pour représenter les tuples autorisés, il suffit de supprimer toutes les arêtes du graphe et d'ajouter des arêtes entre chaque couple de valeurs entre lesquelles il n'existe pas d'arête dans le graphe originel.

Exemple 6.4
Soit $P = \langle V, C \rangle$ le CSP de l'exemple 6.3. Le graphe de micro-structure de P, représenté dans la figure 6.2, permet de vérifier facilement que P est satisfaisable, car aucune arête ne relie les valeurs de l'affectation A, données précédemment.

6.2 Méthodes de résolution

Tout comme SAT, les principales méthodes de résolution de CSP consistent en une énumération des solutions potentielles à travers le parcours d'un arbre de recherche, associée à une opération de filtrage éliminant les affectations trivialement non solutions au problème. Dans ce paragraphe, nous effectuons une courte présentation des techniques utilisées pour la résolution de CSP. Ainsi, dans un premier temps, les principaux types de filtrages proposés sont décrits, puis les heuristiques de choix de variables reconnues efficaces sont brièvement présentées.

6.2.1 Filtrages

Au sein d'un problème de satisfaction de contraintes, des incohérences locales peuvent exister. Celles-ci peuvent être exploitées et propagées de manière à réduire le domaine des va-

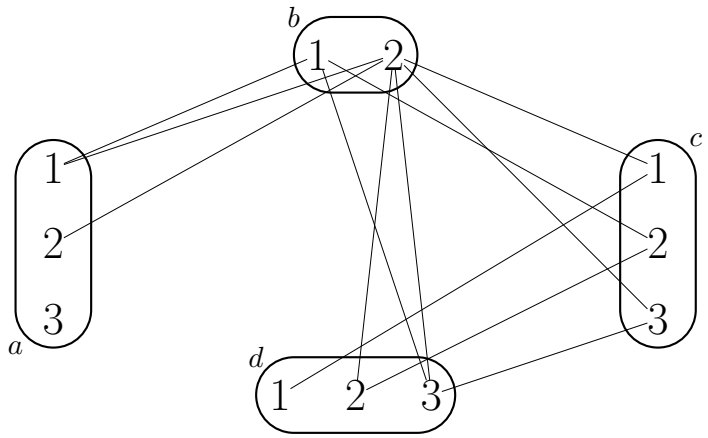


FIG. 6.2 – Graphe de micro-structures associé au CSP de l'exemple 6.3

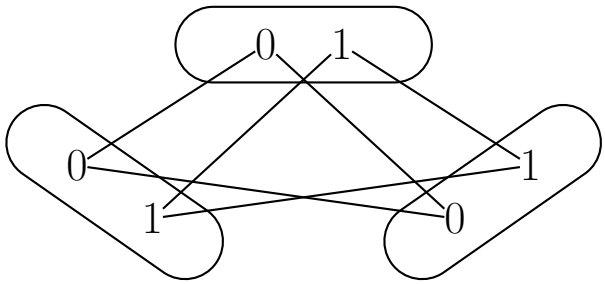


FIG. 6.3 – Graphe de micro-structure de l'exemple 6.5

riables, et donc l'espace de recherche pour une solution potentielle. Les opérations de filtrage des domaimes sont en effet l'un des piliers de la résolution de problèmes réels : ils consistent en des algorithmes polynomiaux générant un problème équivalent, mais plus simple à résoudre, puisque toute valeur d'une variable localement inconsistante peut être trivialement supprimée sans modifier l'ensemble des solutions du problème.

Une première méthode de filtrage consiste à vérifier pour chaque contrainte c impliquant 2 variables V_i et V_j, s'il existe pour chaque valeur $v_i \in dom(V_i)$ une valeur $v_j \in dom(V_J)$ compatible par rapport à c. Cette opération, appelée *consistance d'arc* ou *arc-consistance*, se définit formellement comme suit :

Définition 6.7 (consistance d'arc)
Étant donnés un CSP $\langle V, C\rangle$ et une contrainte $c \in C$ avec $Var(c) = \{v_i, v_j\}$:

- *une valeur $x_i \in dom(v_i)$ est consistante avec c si et seulement s'il existe une valeur $x_j \in dom(v_j)$ telle que le tuple (x_i, x_j) soit autorisé par c, c'est-à-dire tel que (x_i, x_j) n'appartienne pas à $T(c)$.*
- *le variable v_i est arc-consistante avec c si toutes les valeurs de $dom(v_i)$ sont consistantes avec c dans $dom(v_i)$;*
- *le problème $\langle V, C\rangle$ est arc-consistant si et seulement si toutes les variables de V sont arc-consistants avec toutes les contraintes de C.*

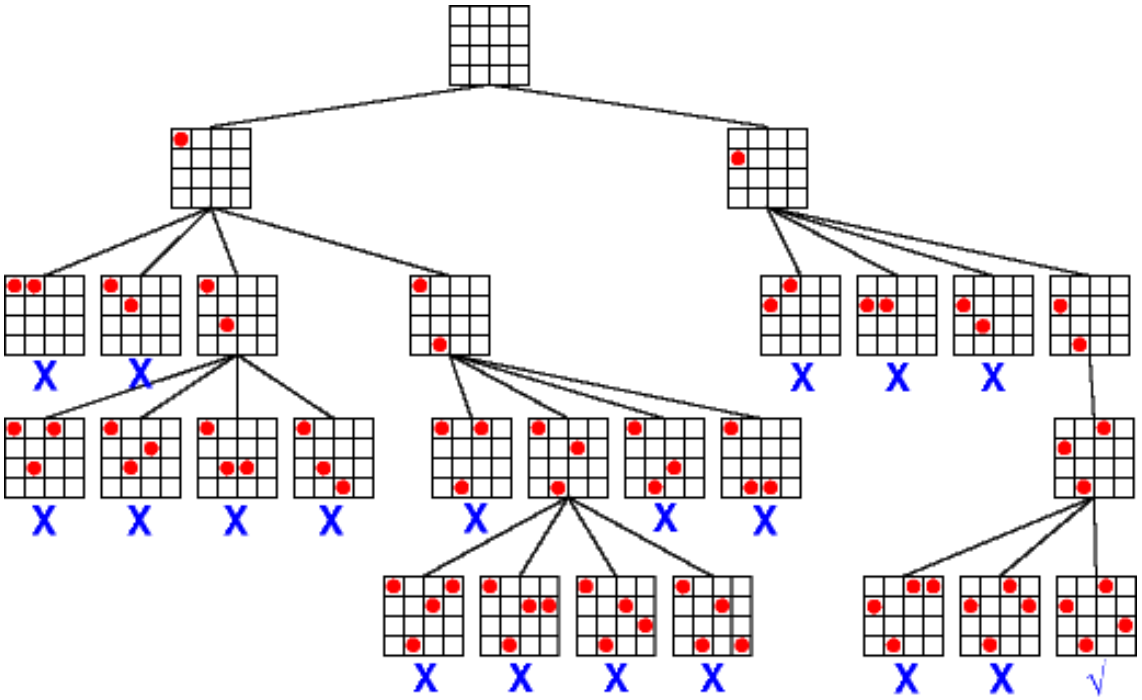


FIG. 6.4 – Arbre de recherche pour le problème des reines sans consistance d'arc

La consistance d'arc est une consistance locale, et par conséquent partielle. Un CSP peut donc être arc consistant et insatisfaisable, comme le montre l'exemple suivant :

Exemple 6.5
Soit $V = \{v_1, v_2, v_3\}$ (avec pour $i \in [1..3]$, $dom(v_i = \{0, 1\})$) et $C = \{C_{12} = (v_1 \neq v_2), C_{23} = (v_2 \neq v_3), C_{31} = (v_3 \neq v_1)\}$. Le CSP $\langle V, C \rangle$, dont le graphe de micro-structure est représenté en figure 6.3, est arc consistant et insatisfaisable. En effet, chaque valeur de chaque variable possède un support de chacune de ses voisines. Pourtant, il est évident qu'on ne peut trouver 3 valeurs différentes à 3 variables prenant leur valeur dans un domaine de taille 2.

Si la seule consistance d'arc n'est pas suffisante pour déterminer la cohérence d'un CSP, elle est en général *maintenue* à chaque point de choix de l'arbre de recherche pour éliminer les valeurs clairement conflictuelles de chaque variable et ainsi réduire l'espace de recherche.

Exemple 6.6
L'un des problèmes classiques modélisés par un CSP est celui des reines. Celui-ci consiste à répondre à la question : sur un échiquier de n cases sur n, peut-on placer n reines de façon à ce qu'aucun couple de reines ne soit sur une même ligne horizontale, verticale ou diagonale ?

En prenant comme exemple $n = 4$, ce problème admet une solution. Toutefois, sans arc-consistance, un grand nombre d'affectations non solutions doivent être parcourues avant obtention d'un modèle. L'arbre de recherche correspondant, tiré du *Guide électronique à la Programmation par Contraintes* [Barták], est représenté en figure 6.4.

Au contraire, le fait de maintenir l'arc-consistance à chaque point de choix permet de réduire considérablement le nombre d'affectations à visiter. On voit en effet dans la figure 6.5, que

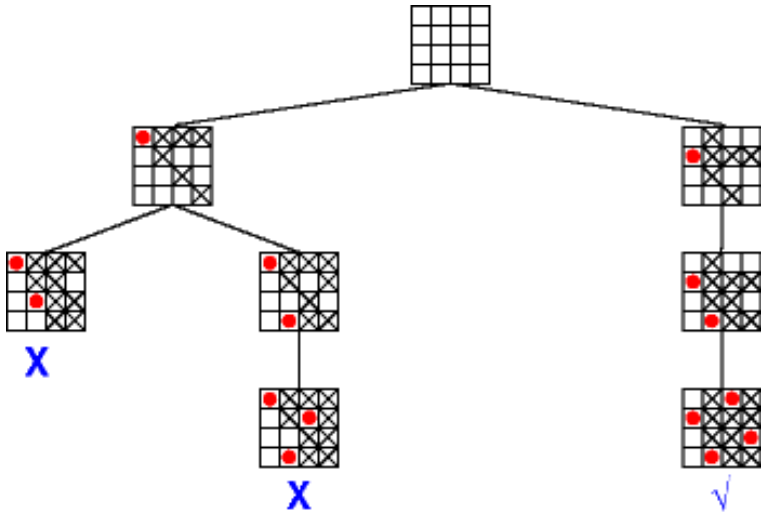


FIG. 6.5 – Arbre de recherche pour le problème des reines avec consistance d'arc

toutes les positions clairement conflictuelles sont *éliminées* dès qu'une affectation est effectuée, ce qui permet l'obtention du même modèle, avec une réduction drastique de l'espace de recherche.

L'exemple précédent montre clairement l'intérêt du maintien de l'arc-consistance à chaque point de choix de la recherche. Les algorithmes visant à maintenir cette consistance ont été intensément étudiés ces dernières années. La première méthode optimale en temps a été nommée `AC-4` [Mohr & Henderson 1986], mais celle-ci est peu efficace en pratique à cause de lourds besoins en espace pour son calcul, et d'une étape d'initialisation relativement longue. Des améliorations ont vu le jour avec `AC-5` [Van Hentenryck *et al.* 1992], un algorithme générique qui peut toutefois tirer parti des propriétés de certaines contraintes. Enfin, les « classiques » `AC-6` [Bessière 1994] et `AC-7`, qui sont capables exploiter la bidirectionnalité et de nouvelles propriétés exhibées de certaines contraintes [Bessière *et al.* 1995].

Au vue de l'efficacité de ce filtrage pour la réduction de l'espace de recherche (cf. exemple 6.6), il semble intéressant de s'intéresser à des formes de filtrages plus fortes. Ainsi la chemin-consistance fût-elle établie.

Définition 6.8 (chemin-consistance)
Soient $P = \langle V, C \rangle$ un CSP et $v_i, v_j \in V$ deux variables distinctes de ce CSP.

- *Une paire de valeur $\{(x_i, v_i), (x_j, v_j)\}$ localement consistante est* chemin consistante *si et seulement si $\forall v_k \in V$ avec $v_k \neq v_i$ et $v_k \neq v_j$, il existe $x_k \in dom(v_k)$ telle que $\{(x_i, v_i), (x_k, v_k)\}$ et $\{(x_j, v_j), (x_k, v_k)\}$ soient des instanciations localement consistantes.*
- *Une paire de variables (v_i, v_j) est chemin consistante si et seulement si toute instanciation partielle localement consistante de ces variables est chemin consistante.*
- *Un CSP $P = \langle V, C \rangle$ est chemin consistant si et seulement si $\forall v_i, v_j \in V$, la paire (v_i, v_j) est chemin consistante.*

La chemin-consistance est donc une forme de filtrage plus puissant, dans le sens où on doit pouvoir choisir tout couple de valeurs consistantes du problème et pouvoir étendre cette instanciation partielle à une troisième variable, là où avec l'arc-consistance, on choisit une unique valeur au sein d'une variable pour l'étendre à une deuxième.

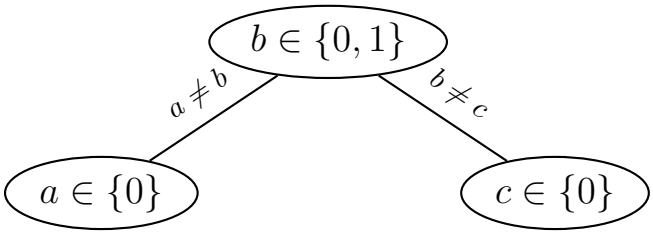


FIG. 6.6 – Réseau de contraintes de l'exemple 6.7

Ces formes de filtrages peuvent être généralisées à toute instanciation de $k-1$ variables pour l'étendre à la $k^{ème}$. On définit alors la k-consistance :

Définition 6.9 (k-consistance)
Un CSP P est k-consistant si et seulement si toute instanciation consistante de $k-1$ variables peut être étendue à n'importe quelle autre variable du problème.

Définition 6.10 (k-consistance forte)
Un CSP P est fortement k-consistant si et seulement si $\forall i \in [1..k]$ P est i-consistant.

Ces formes générales de consistance sont basées sur le principe que si une valeur ne peut participer à la construction d'une solution d'un sous-réseau de k variables alors cette valeur ne peut pas, non plus, rentrer dans la construction d'une solution complète. Pourtant, seule une forme de consistance forte peut fournir l'assurance d'une solution : en effet, un CSP peut être k-consistant mais pas i-consistant, avec $i < k$.

Exemple 6.7
Soit $P = \langle V, C\rangle$ avec $V = \{a,\ b,\ c\}$ ($dom(b) = \{0,\ 1\}$, $dom(a) = dom(c) = \{0\}$) et $C = \{a \neq b,\ b \neq c\}$. P, dont le réseau de contrainte est représenté en figure 6.6, est 3-consistant mais pas 2-consistant ; il n'est donc pas fortement 3-consistant.

En effet, tout couple d'affectation localement consistante de 2 variables ($\{a = 0,\ b = 1\}$, $\{b = 1,\ c = 0\}$ ou $\{a = 0,\ c = 0\}$) peut être étendu à la troisième ($\{a = 0,\ b = 1,\ c = 0\}$), pourtant la seule affectation $\{b = 0\}$ ne peut être étendu à aucune des deux autres variables. Cette valeur n'est donc pas arc consistante.

Avec la généralisation des différentes formes de filtrage, on obtient :

Corollaire 6.1
Un CSP est arc consistant si et seulement s'il est 2-consistant.

Corollaire 6.2
Un CSP est chemin consistant si et seulement s'il est 3-consistant.

On peut donc construire des formes de plus en plus fortes de filtrage, jusqu'à celle permettant de déterminer la cohérence d'un problème.

Propriété 6.5
Soit $P = \langle V, C\rangle$ un CSP. P est satisfaisable si et seulement si P est fortement $|V|$-consistant.

Ces techniques de filtrage permettent de réduire l'espace de recherche de manière efficace, mais sont en fait très peu utilisées en pratique. En fait, on considère que maintenir de fortes formes de consistance à chaque point de la recherche est trop coûteux par rapport à la quantité de valeurs éliminées, ce qui conduit le plus souvent à une perte en temps de calcul. Par exemple, le meilleur algorithme connu pour la chemin-consistance est cubique dans le nombre de variables du problème. La maintenir à chaque point de choix peut donc s'avérer lourd, d'un point de vue calculatoire.

En général, les solveurs modernes maintiennent uniquement l'arc-consistance, qui semble le meilleur compromis entre le nombre de valeurs qu'elle permet d'éliminer et le temps nécessaire à son calcul.

6.2.2 Heuristiques de choix variables

Tout comme pour le problème SAT, l'ordre dans lequel sont affectées les variables d'un CSP joue un rôle prépondérant dans la taille de l'arbre de recherche, celle-ci pouvant varier d'un facteur exponentiel en fonction du choix effectué. Malheureusement, décider de la variable optimale à brancher à un certain moment de la recherche est un problème NP-difficile, et l'on a recours une fois encore à des heuristiques pour le choix de la variable à affecter.

Sommairement, on distingue deux types d'heuristiques. Tout d'abord, la première catégorie, dite *statique*, conserve un ordre fixe dans les affectations effectuées tout au long de la recherche et n'exploite que l'information syntaxique relative au problème originel, c'est-à-dire avant le début de la recherche. Parmi elles, on trouve les heuristiques `lexico` qui choisit simplement les variables suivant l'ordre lexicographique, `deg` qui fait le choix de la variable ayant le plus faible degré initial [Dechter & Meiri 1989], et `width` [Freuder 1982] où les contraintes sont ordonnées pour minimiser la largeur de l'arbre de recherche.

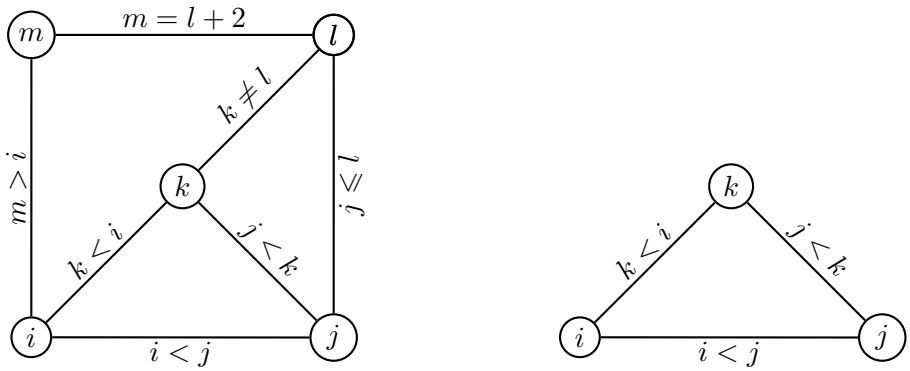
Ensuite, on trouve les heuristiques *dynamiques*, qui prennent en compte certaines informations acquises au cours de la recherche. Tout d'abord, l'heuristique `dom` [Haralick & Elliott 1980] choisit la variable ayant le domaine le plus petit, ceux-ci étant mis à jour pendant la recherche. L'idée de tenir également compte du degré de chaque variable a ensuite été émise avec les heuristiques `dom/deg` [Bessière & Régin 1996] et `dom/ddeg` [Smith & Grant 1998], permettant des gains conséquents en temps, dans la plupart des cas.

Enfin, `dom/wdeg` est une heuristique dynamique présentée dans [Boussemart *et al.* 2004] qui nécessite d'associer un compteur à chaque variable. Ceux-ci sont initialisés à 1 et incrémentés dès que la variable correspondante est impliquée dans un conflit. On choisit ensuite de brancher la variable dont le quotient de la taille de son domaine par son compteur est le plus faible. Cette technique permet de considérer la *difficulté* à satisfaire les contraintes liées à chaque variable, dans le but de choisir celle qui conduit *a priori* le plus rapidement à un échec, l'objectif de ces compteurs étant clairement de guider la recherche vers des parties inconsistantes ou fortement contraintes du problème CSP.

6.3 Ensemble Minimalement Inconsistant de Contraintes

6.3.1 Définitions, complexité

De la même manière que pour le problème SAT, alors que la démonstration de la cohérence d'un problème CSP est en général accompagnée d'un modèle qui en est un certificat, la preuve de son incohérence n'apporte que peu d'informations sur le problème.



(a) Un CSP insatisfiable (b) L'un de ses MUC

FIG. 6.7 – Réseaux de contraintes de l'exemple 6.8 et de l'un de ses MUC

Or, tout CSP incohérent admet au moins un ensemble insatisfaisable de contraintes qui est minimal, dans le sens où chacun de ses sous-ensembles est satisfaisable. Le calcul d'un tel ensemble permet de localiser des éléments qui sont en conflit, afin de diagnostiquer une cause de l'incohérence du problème et éventuellement le réparer. On considère généralement les contraintes du problème comme étant ces « éléments » conflictuels, et on définit le concept de *MUC*.

Définition 6.11 (MUC)
Soit $\langle V, C\rangle$ un CSP insatisfaisable. Le CSP $\langle V, C'\rangle$ est un MUC (Minimally Unsatisfaisable Core) *de $\langle V, C\rangle$ si et seulement si :*

1. $C' \subseteq C$;
2. $\langle V, C'\rangle$ *est insatisfaisable ;*
3. $\forall C'' \subset C'$, $\langle V, C_m\rangle$ *est satisfaisable.*

Exemple 6.8
Soient $V = \{i,\ j,\ k,\ l,\ m\}$ un ensemble de variables prenant leur valeur dans $\{0, 1, 2, 3, 4\}$ et $C = \{m > i,\ m = l + 2,\ k < i,\ k \neq l,\ j < k,\ i < j,\ j \leq l\}$ un ensemble de 7 contraintes. Le réseau de contraintes du CSP $P = \langle V,\ C\rangle$ est représenté en figure 6.7a. P est clairement insatisfaisable, et $P_{MUC} = \langle V, \{i < j,\ j < k,\ k < i\}\rangle$, l'un de ses MUC, est représenté en figure 6.7b : en effet, aucune valeur ne peut être trouvée pour les variables i, j et k telle que chacune soit plus petite que ses voisines. Une contrainte de cet ensemble doit donc être ôtée si l'on veut réparer cette partie du problème.

L'identification d'une source d'incohérence d'un problème par un MUC permet d'en restaurer localement la cohérence par la suppression d'une contrainte au sein de ce MUC. Dans l'exemple précédent, retirer n'importe quelle contrainte du MUC détecté au problème original est suffisant pour obtenir un sous-problème satisfaisable. Ce n'est malheureusement pas toujours le cas.

En effet, un CSP peut posséder un nombre de MUC exponentiel en son nombre de contraintes. De plus, la restauration de la cohérence d'un problème passe par la « réparation » de chacun de ses MUC. Même si une même contrainte peut appartenir à plusieurs MUC, cela reste un problème dont le traitement reste difficile en pratique. Toutefois, le concept de couverture inconsistante (éventuellement stricte) présenté en paragraphe 5.2.1 du chapitre précédent peut être clairement transposé au problème CSP, et un tel ensemble de MUC est suffisant pour expliquer et

potentiellement réparer l'ensemble du problème. Celui-ci implique le développement d'une technique visant à extraire un MUC d'un CSP insatisfaisable. Dans cet objectif, plusieurs approches ont déjà été proposées. Celles-ci sont évoquées et brièvement décrites dans le paragraphe suivant.

6.3.2 Un tour d'horizon des approches existantes

Plusieurs approches ont par le passé été proposées dans le but de détecter des MUC. Tout d'abord, les approches générales de [Han & Lee 1999] et [de la Banda *et al.* 2003], proposées dans d'autres contextes et présentées précédemment (cf. Chapitre 3), permettent l'obtention de tous les MUC par l'exploration d'un arbre « `CS` ». Celui-ci consiste à énumérer et tester la cohérence des sous-problèmes du CSP traité pour en extraire l'ensemble de ses MUC. Malheureusement, l'explosion combinatoire du nombre de sous-problèmes d'un CSP ne permet à cette méthode que de pouvoir traiter de très petits problèmes en pratique.

En outre, dans le cadre CSP, plusieurs travaux visent à identifier des ensembles (minimaux) conflictuels de contraintes afin d'effectuer des retours en arrière intelligents, tels que le « *dynamic backtracking* » [Ginsberg 1993, Jussien *et al.* 2000] ou le « *conflict-based backjumping* » [Prosser 1993]. Toutefois, seules quelques méthodes spécifiques à l'extraction d'un MUC ont été proposées. Notons par exemple les approches de [Mauss & Tatar 2002] et [Junker 2004] qui permettent d'obtenir une explication de l'incohérence basée sur les préférences de l'utilisateur. Mentionnons également la bibliothèque `PaLM` [Jussien & Barichard 2000] de la plate-forme de programmation par contraintes `Choco` [Laburthe 2000], qui est un outil permettant par exemple d'expliquer pourquoi il n'existe aucune solution contenant une valeur particulière à un CSP donné. De plus, en cas d'inconsistance du problème, `PaLM` est capable d'en extraire un sous-problème insatisfaisable, mais la minimalité de celui-ci n'est pas garantie.

Enfin, une nouvelle approche a récemment été proposée par [Hemery *et al.* 2006]. Cette dernière, baptisée `DC(wcore)`, améliore en fait une précédente méthode [Bakker *et al.* 1993] introduite dans le contexte du diagnostic de pannes. De surcroît, elle a été prouvée plus efficace que le système `QuickXplain` [Junker 2001], et semble être à ce jour la plus capable à extraire en pratique un MUC, pour la plupart des classes de problèmes CSP. Dans le chapitre suivant, l'approche `DC(wcore)` est présentée en détail et de nouvelles techniques permettant d'accroître plus encore son efficacité sont introduites.

Chapitre 7

Extraction d'un MUC : `wcore` et méthodes de minimisation revisités

Sommaire

La meilleure approche connue pour l'extraction d'un MUC au sein d'un CSP insatisfaisable est due à [Hemery *et al.* 2006]. Celle-ci est composée de deux étapes successives, baptisées respectivement `wcore` et `DC`. Tout d'abord un sous-problème insatisfaisable (pas nécessairement minimal) est extrait par la procédure `wcore`. Celui-ci est ensuite minimisé vers un MUC. Dans ce chapitre, nous présentons cette approche en détail et la revisitons en présentant quelques pistes permettant d'améliorer le comportement pratique de chacune de ses 2 étapes. La plupart des résultats présentés dans ce chapitre ont fait l'objet de publications [Grégoire *et al.* 2006e, Grégoire *et al.*].

7.1 De `wcore` à `full-wcore`

7.1.1 Principe de l'algorithme

La procédure `wcore` est basée sur le fait suivant : il est bien connu que quand l'incohérence d'un CSP est prouvée grâce à un algorithme de type « *branch* & *bound* », celui-ci est capable de fournir un sous-problème insatisfaisable de ce CSP [Bakker *et al.* 1993]. Ceci est permis grâce au concept de contraintes *actives*.

Définition 7.1 (contrainte active)
Soit P un CSP et α une affectation partielle des variables de P. On appelle contrainte active *(par rapport à α) toute contrainte provoquant par arc-consistance le retrait d'au moins une valeur d'une variable de P. Ces contraintes sont notées c_α.*

Fonction wcore($P = \langle V, C\rangle$: un **CSP insatisfaisable**) : **un sous-problème insatisfaisable** P
  Pour tout $c \in C$ **faire** $weight[c] = 1$; **Fin Pour**
  $C' \longleftarrow C$;
  Répéter
    $\langle V, C\rangle \longleftarrow \langle V, C'\rangle$;
    [La recherche est modifiée de manière à marquer les contraintes actives de $\langle V, C\rangle$ *et retourner le poids de chaque variables (weight) en fin de recherche]*
    $weight \longleftarrow$ MAC($\langle V, C\rangle$, $weight$) ;
    $C' \longleftarrow \{c \in C \mid c$ est active $\}$;
  jusqu'à ce que ($|C'| \leq |C|$)
Fin

Algorithme 15 – `wcore`

Remarque 7.1
Clairement, ce concept peut également être défini pour des formes de filtrages différentes de l'arc-consistance (cf. section 6.2.1). Toutefois, cette dernière étant la consistance utilisée par la grande majorité des solveurs modernes, le concept de contrainte active est classiquement défini par rapport à celle-ci.

Ainsi, les contraintes actives sont exactement celles qui, par rapport à une affectation partielle, contribuent à réduire l'espace des solutions par le retrait de certaines valeurs au sein des variables du problème. En considérant l'union des contraintes actives par rapport aux affectations partielles considérées lors d'une recherche complète, il est possible de déduire un sous-problème insatisfaisable du problème prouvé incohérent.

Propriété 7.1
Soit P un CSP insatisfaisable dont l'insatisfaisabilité est prouvée par une recherche complète Δ de type branch-and-bound. *Soit α_Δ l'ensemble des affectations partielles considérées pendant cette recherche. L'ensemble $\cup_{\alpha\in\alpha_\Delta} c_\alpha$ forme un sous-réseau insatisfaisable de P.*

Ainsi, une recherche complète prouvant l'incohérence d'un problème est suffisante pour en déduire un sous-problème insatisfaisable. Celui-ci est formé des contraintes dont la satisfaction a nécessité le retrait d'au moins une valeur du domaine d'une variable au cours de la recherche d'une solution. Ainsi, il est suffisant de marquer les contraintes actives durant une recherche complète pour obtenir un sous-problème insatisfaisable du problème traité. En effet, les autres contraintes du problème n'ayant pas participé à cette preuve d'inconsistance (en ne réduisant pas l'ensemble des solutions du problème), elles peuvent être ôtées, tout en conservant le problème insatisfaisable.

Toutefois, il semble évident que le sous-problème extrait par cette approche peut différer en fonction des affectations partielles traversées, qui sont dirigées par l'heuristique de choix de variables. En pratique, `wcore` jouit de l'efficacité de l'heuristique dynamique *dom/wdeg* [Boussemart *et al.* 2004]. Comme présentée dans le paragraphe 6.2.2 du chapitre précédent, celle-ci est basée sur des compteurs estimant les contraintes exercées sur les variables du problème, et

permettant de concentrer la recherche sur ses zones sur-contraintes.

Ainsi, une recherche complète possédant ces caractéristiques est exécutée dans le but d'obtenir un sous-problème insatisfaisable du problème original (également appelé *core* dans [Hemery *et al.* 2006]). De plus, cette opération peut être répétée sur le sous-problème obtenu afin d'en extraire un éventuel nouveau sous-problème. De cette façon, `wcore` consiste en une boucle où sont itérés des appels à un algorithme de recherche complète tant que le nombre de contraintes active décroît. De plus, les compteurs de l'heuristique *dom/wdeg* sont conservés d'un appel à l'autre, de manière à se concentrer progressivement sur une petite zone de l'espace de recherche, et ainsi réduire le nombre de contraintes actives. En pratique, la conservation de ces compteurs se montre extrêmement efficace pour la découverte de petits sous-problèmes insatisfaisables. La procédure `wcore` est résumée dans l'algorithme 15.

Avec ces caractéristiques, `wcore` est capable de fournir une approximation d'un MUC en considérant à la fin de la recherche, le plus petit sous-problème en terme de nombre de contraintes impliquées.

7.1.2 Collecter plus d'informations pour `dom/wdeg` en retardant le *backtrack*

L'efficacité de la procédure `wcore` est clairement liée à celle de l'espace de recherche parcouru par le solveur complet auquel elle est greffée. Or, comme nous l'avons vu, celui-ci incrémente les compteurs des contraintes violées pendant la recherche de manière à focaliser la recherche sur les contraintes jugées « difficiles » d'abord, grâce à l'usage de *dom/wdeg*. Il pourrait se montrer utile de modifier le solveur quand son but n'est pas de trouver le plus rapidement possible un modèle ou de prouver qu'il en existe aucun, mais de calculer le plus petit sous-problème incohérent possible. Plus précisément, quand ce type de solveur prouve qu'une contrainte est falsifiée par propagation des valeurs des dernières variables affectées, celui-ci effectue immédiatement un *backtrack*. Pourtant, ce retour-en-arrière arrive peut-être trop tôt. D'autres contraintes peuvent également être violées dans les même circonstances, et il peut s'avérer utile de prendre celles-ci en considération, comme cela a déjà pu être souligné dans d'autres contextes (cf. par exemple [Schiex & Verfaillie 1994]). Ceci peut ici être effectué grâce aux compteurs de l'heuristique de choix de variables. Bien sûr, une telle politique peut consommer plus de temps durant la recherche. Toutefois, les résultats expérimentaux présentés dans la section 7.3 montrent que collecter cette information stratégique se révèle utile et rend la procédure plus efficace, la plupart du temps.

Exemple 7.1
Soit le CSP $P = (\{a, b, c, d\}, \{a \neq b, b + c = 2, a + c = 2, c \leq d, b + d \neq 2\})$, avec $\{0, 1, 2\}$ comme domaine d'instanciation de chacune des variables. Ce problème est représenté par un réseau de contraintes en figure 7.1 *(1)*, dans lequel les nœuds sont étiquetés à la fois par le nom des variables et leur domaine respectif.

Supposons qu'une recherche pour la satisfaisabilité soit lancée, en commençant par affecter b à 0. Le domaine des variables voisines à b est tout d'abord filtré par rapport à l'arc-consistance. Plus précisément, les valeurs des variables ne satisfaisant clairement pas les contraintes du problèmes par rapport à l'instanciation partielle courante sont supprimées de leur domaine. Par exemple, la valeur *2* est supprimée du domaine de d, puisqu'elle falsifie la contrainte $b + d \neq 2$. Les domaines d'instanciation résultants sont représentées dans la figure 7.1 *(2)*. Après cette première étape de filtrage, la seule valeur de la variable c candidate à un modèle est 2. La variable est alors affectée par propagation. Ensuite, cette nouvelle information est propagée par arc-consistance. Supposons que le domaine de la variable a soit d'abord filtré ; clairement, ce dernier

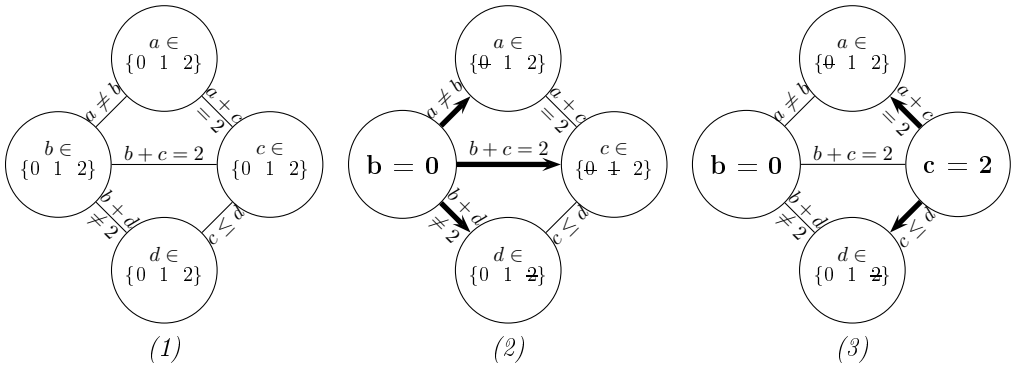


FIG. 7.1 – Filtrage des domaines par rapport à l'arc-consistance

devient vide, puisqu'aucune valeur restante au domaine de a ne satisfait la contrainte $a + c = 2$. Un backtrack est donc déclenché et l'heuristique de choix de variable incrémente le poids de cette contrainte. Cependant, celle-ci n'est pas la seule à être violée par l'affectation partielle courante. En effet, la contrainte $c \leq d$ est également falsifiée. Il semble donc naturel d'incrémenter le poids de toutes les contraintes contredites, plutôt qu'uniquement celui de la première découverte.

Ainsi, nous avons modifié un solveur basé sur MAC (pour *Maintening Arc-Consistency* en anglais) de façon à ce qu'il n'effectue pas de retour-en-arrière dès qu'une contrainte est falsifiée. Au contraire, toutes les contraintes sont vérifiées à chaque nouvelle affectation de variable. Par rapport à ce procédé, un solveur classique récolte donc une information moins systématique, puisqu'il n'incrémente qu'une contrainte choisie arbitrairement. Toutefois, dès que le domaine d'une variable est vide, appliquer l'arc-consistante jusqu'à obtenir un point fixe conduit à vider également le domaine de toutes les variables non affectées. De cette manière, toute contrainte n'impliquant pas exclusivement des variables affectées avant ce filtrage devient falsifiée. Pour éviter ce phénomène qui ne fournirait pas une information heuristique pertinente, nous avons donc choisi de limiter le filtrage de la manière suivante : supposons que l'algorithme de l'arc-consistance filtre le domaine des variables connexes (c'est-à-dire liées par une contrainte) à une variable v. Si l'un des domaines filtrés se vide, on poursuit l'arc-consistance sur les variables connexes à v vis-à-vis des contraintes impliquant v, puis l'opération est stoppée. Cette nouvelle procédure est appelée `full-wcore` (pour « *weighting* all *falsified constraints* »), en référence à `wcore` (*weight core*).

Comme les résultats expérimentaux du paragraphe 7.3 le montrent, prendre toutes les contraintes déclenchant un backtrack en considération, plutôt qu'une seule permet d'améliorer les performances à la fois des procédures `wcore` et `DC(wcore)`.

7.2 Un nouveau procédé de minimisation

7.2.1 Principes et complexité des procédés de minimisation

`wcore` et `full-wcore` retournent un sous-problème P composé de e contraintes qui est une approximation d'un MUC. La deuxième étape consiste donc à minimiser ce sous-problème, de manière à obtenir un MUC. Plusieurs techniques ont déjà été proposées à ce jour, mais toutes se basent sur le concept de *contrainte de transition*.

Fonction CS($P = \langle V, C\rangle$: un CSP) : **Un MUC de** P
  $k \longleftarrow 0$;
  Répéter
    $i \longleftarrow 1$;
    Tant que ((MAC($\langle V, \{c_1, \ldots, c_i\}\rangle$) est prouvé cohérent) **faire** $i \longleftarrow i+1$; **Fait**
    contrainte_de_transition $\longleftarrow c_i$;
    Pour j **de** $(i-1)$ **à** 1 **faire**
      $c_{j+1} \longleftarrow c_j$;
    Fin Pour
    $c_1 \longleftarrow$ *contrainte_de_transition* ;
    $C \longleftarrow C \setminus \{c_{i+1}, \ldots, c_{|C|}\}$;
    $k \longleftarrow k+1$;
  jusqu'à ce que $(k = |C|)$
  Retourner $\langle V, C\rangle$;
Fin

Algorithme 16 – CS – minimisation ConStructive

Définition 7.2 (contrainte de transition)
Soit $\langle V, C\rangle$ un CSP insatisfaisable. Soit un ordre $(c_1, \ldots, c_e)$ sur les e contraintes contenues dans C. Il existe toujours une unique contrainte c_i appelée contrainte de transition *telle que $(c_1, \ldots, c_{i-1})$ soit satisfaisable, et $(c_1, \ldots, c_i)$ soit insatisfaisable.*

Propriété 7.2
Soit c_i la contrainte de transition d'un CSP insatisfaisable $P = \langle V, \{c_1, \ldots, c_e\}\rangle$. c_i appartient à tous les MUC du problème $\langle V, \{c_1, \ldots, c_i\}\rangle$.

Preuve
Par définition, si c_i est la contrainte de transition de P, alors
- $P' = \langle V, \{c_1, \ldots, c_{i-1}\}\rangle$ est satisfaisable ;
- $P'' = \langle V, \{c_1, \ldots, c_i\}\rangle$ est insatisfaisable

Par conséquent, la contrainte c_i est « nécessaire » à l'incohérence de P'' puisque son seul retrait en restaure la satisfaisabilité. Cette contrainte appartient donc à tous les sources d'incohérence de P'', et donc à tous ses MUC. □

Le calcul d'une contrainte de transition permet donc de localiser une contrainte d'un MUC du problème et d'en éliminer d'autres (celles ayant un indice supérieur à i dans l'ordre considéré).

Le calcul d'un MUC peut être effectué en détectant la contrainte de transition c_i du problème et en réorganisant l'ordre $(c_1, \ldots, c_i)$ en $(c_i, c_1, \ldots, c_{i-1})$. Une deuxième contrainte de transition c_j est alors calculée, et l'ordre devient $(c_i, c_j, c_1, \ldots, c_{j-1})$. Le processus est itéré et stoppe quand l'ensemble des contraintes de transition est prouvé insatisfaisable. Cet ensemble est alors un MUC.

Les différentes méthodes permettant la minimisation d'un MUC ne varient que par leur technique pour la découverte de la contrainte de transition, celle-ci étant centrale à leur efficacité.

Fonction DS($P = \langle V, C\rangle$: un CSP) : **Un MUC de** P
  $k \longleftarrow 0$;
  Répéter
    $i \longleftarrow |C|$;
    Tant que ((`MAC`($\langle V, \{c_1, \ldots, c_{i-1}\}\rangle$) est prouvé incohérent) **faire** $i \longleftarrow i - 1$;
    Fait
    contrainte_de_transition $\longleftarrow c_i$;
    Pour j **de** $(i-1)$ **à** 1 **faire**
      $c_{j+1} \longleftarrow c_j$;
    Fin Pour
    $c_1 \longleftarrow$ *contrainte_de_transition* ;
    $C \longleftarrow C \setminus \{c_{i+1}, \ldots, c_{|C|}\}$;
    $k \longleftarrow k + 1$;
  jusqu'à ce que $(k = |C|)$
  Retourner $\langle V, C\rangle$;
Fin

Algorithme 17 – `DS` – minimisation DeStructive

Dans [Hemery *et al.* 2006], trois familles d'approches permettant de déterminer la contrainte de transition sont abordées. Dans la suite de ce paragraphe, celles-ci sont présentées et commentées.

La méthode constructive

La première méthode, qualifiée de constructive, considère en premier lieu l'ensemble des variables du problème et l'ensemble vide comme étant celui des contraintes. Elle consiste ensuite en l'ajout une à une des contraintes du problème alterné à un test de satisfaisabilité. Tant que le problème courant est cohérent, une nouvelle contrainte est ajoutée, jusqu'à obtenir un CSP insatisfaisable : la dernière contrainte ajoutée par la procédure est la contrainte de transition. Clairement, suivant l'ordre dans lequel les contraintes sont ajoutées, un MUC plutôt qu'un autre pourra être retourné, si le problème en possède plusieurs.

Propriété 7.3
La méthode constructive retourne le MUC dont la dernière contrainte est ajoutée en premier.

Preuve
Le principe de cette approche est d'ajouter des contraintes à un problème CSP jusqu'à ce que celui-ci devienne incohérent. Or, il ne peut le devenir qu'en contenant l'intégralité d'au moins un MUC du problème original. Ainsi, dès que la dernière contrainte de l'un MUC est ajoutée, le CSP construit devient incohérent, d'où la découverte de la contrainte de transition et l'arrêt de la procédure. □

Cette approche, introduite dans [de Siqueira & Puget 1988], n'est malheureusement pas compétitive d'un point de vue pratique. Celle-ci est décrite dans l'algorithme 16.

La complexité dans le pire cas des approches de minimisation peut être caractérisée par le nombre d'appels à une méthode complète qu'elles impliquent. En notant e le nombre de

```
Fonction DC(P = ⟨V, C⟩ : un CSP) : Un MUC de P
    k ⟵ 0 ;
    Répéter
        min ⟵ k + 1 ;
        max ⟵ |C| ;
        i ⟵ |C| ;
        Tant que (min ≠ max) faire
            med ⟵ (min + max)/2 ;
            Si ((MAC(⟨V, {c_1, ..., c_med}⟩) est prouvé incohérent)) Alors
                max ⟵ med ;
            Sinon
                min ⟵ med + 1 ;
            Fin Si
        Fait
        contrainte_de_transition ⟵ c_min ;
        Pour j de (min − 1) à 1 faire
            c_{j+1} ⟵ c_j ;
        Fin Pour
        c_1 ⟵ contrainte_de_transition ;
        C ⟵ C \ {c_{min+1}, ..., c_{|C|}} ;
        k ⟵ k + 1 ;
    jusqu'à ce que (k = |C|)
    Retourner ⟨V, C⟩ ;
Fin
```

Algorithme 18 – DC – minimisation DiChotomique

contraintes du problème P, et k le nombre de contraintes du MUC extrait, la complexité de la méthode constructive est de $\mathcal{O}(e.k)$.

La méthode destructive

La deuxième famille est dite destructive, dans le sens où son but est au contraire de retirer des contraintes du problème incohérent jusqu'à ce que celui-ci devienne satisfaisable. La dernière contrainte supprimée est alors la contrainte de transition. Assez clairement, comme avec l'approche constructive, l'ordre dans lequel sont testées les contraintes joue un rôle capital dans le MUC retourné, comme le montre la propriété suivante :

Propriété 7.4
La méthode destructive retourne le MUC dont la première contrainte est supprimée en dernier.

Preuve
La méthode destructive consiste en l'élimination successive de contraintes, jusqu'à ce que le sous-problème résultant soit satisfaisable. Or, tant que ce dernier contient au moins un MUC, il reste incohérent et toutes les contraintes testées sont éliminées. Le MUC calculé est donc

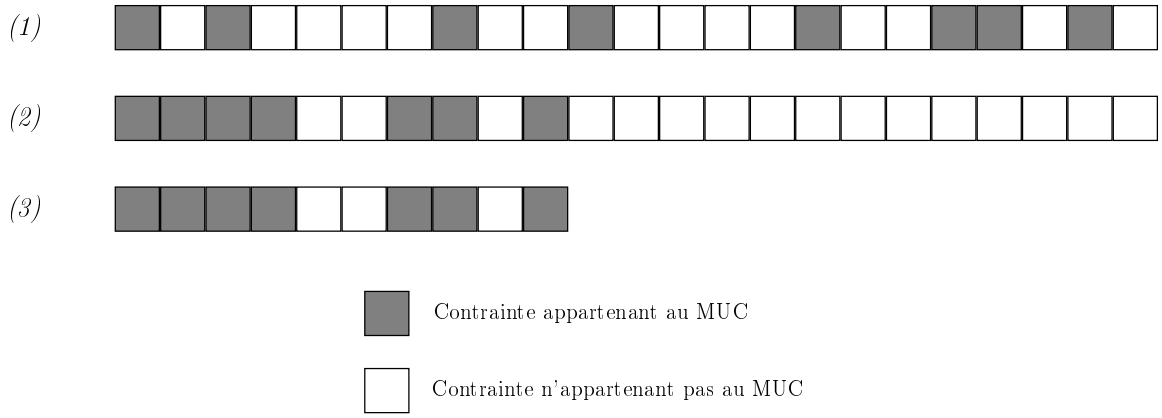


FIG. 7.2 – Visualisation du comportement de `CB`

nécessairement le dernier contenu dans l'approximation dont l'une de ces contraintes est testée. □

La méthode destructive, dont l'approche générale est synthétisée dans l'algorithme 17, a été présentée pour la première fois par [Bakker *et al.* 1993]. En reprenant les notations précédentes, la complexité de cette approche dans le pire des cas est $\mathcal{O}(e)$. Celle-ci est donc linéaire en la taille de l'entrée (i.e. le nombre de contraintes du sous-problème insatisfaisable à minimiser). En pratique, elle s'avère relativement efficace, en particulier quand l'approximation du MUC fournie est de bonne qualité.

La méthode dichotomique

Enfin, [Hemery *et al.* 2006] introduisent une recherche dichotomique sur les contraintes du problème. En notant e le nombre de contraintes du problème P, et k le nombre de contraintes du MUC extrait, la complexité (en nombre d'appels à un solveur complet) de cette méthode est de $\mathcal{O}(log(e).k)$. L'algorithme 18 dépeint cette nouvelle méthode, qui se montre très efficace, tout spécialement quand le problème donné en entrée n'est qu'une approximation grossière d'un MUC.

En se basant sur l'analyse de ces pires cas, [Hemery *et al.* 2006] recommandent l'utilisation de l'approche dichotomique, appelée `DC`. Plus spécifiquement, il est montré que sa complexité dans le pire cas est meilleure que celle de `QuickXplain` [Junker 2001]. Ces même auteurs recommandent également d'ordonner les contraintes de P en fonction de leur *difficulté* estimée par $dom/wdeg$.

7.2.2 Combiner les minimisations dichotomiques et destructives

Bien que l'analyse des pires cas de ces différentes approches penche en faveur de la méthode dichotomique, notre intuition est que d'importantes informations heuristiques fournies par `wcore` ou `full-wcore` n'ont pas été exploitées. En effet, les contraintes ayant un haut score ont une probabilité beaucoup forte d'appartenir aux MUCs du problème que celles ayant obtenues un score faible. Ainsi, les contraintes appartenant au MUC ne sont pas nécessairement dispersées

```
Fonction CB(P = ⟨V, C⟩ : un CSP) : Un MUC de P
    C ⟵ trie_contraintes(C) ;
    min ⟵ 1 ;
    max ⟵ n ;
    Tant que (min ≠ max) faire
        med ⟵ (min + max)/2 ;
        Si ((MAC(⟨V, {c_1, ..., c_med}⟩) est prouvé incohérent)) Alors
            min ⟵ med + 1 ;
        Sinon
            max ⟵ med ;
        Fin Si
    Fait
    C ⟵ C \ {c_{min+1}, ..., c_{|C|}} ;
    Pour toute c ∈ C faire
        Si ((MAC((V, C \ {c})) est prouvé incohérent)) Alors
            C ⟵ C \ {c}
        Fin Si
    Fin Pour
    Retourner ⟨V, C⟩ ;
Fin
```

Algorithme 19 – CB – minimisation ComBinant la destructive et la dichotomique

uniformément parmi les contraintes de P, mais tendent *a priori* à se grouper parmi les contraintes de plus haut score. Or, l'approche dichotomique n'exploite pas une telle information heuristique.

En outre, supposons que le sous-problème fourni par la première étape de l'algorithme soit effectivement un MUC. Dans ce cas, l'approche destructive nécessite $\mathcal{O}(e)$ appels à une méthode complète, alors que celle dichotomique va pour sa part effectuer $\mathcal{O}(log(e).e)$ de ces appels. Notons également que `full-wcore` est censé retourner une meilleure approximation que celle de `wcore`. D'un autre côté, il est naturel de s'attendre à ce que l'approche dichotomique soit plus efficace quand les contraintes du MUC sont dispersées aléatoirement dans P. Nous proposons donc de substituer des appels systématiques à une procédure dichotomique par une politique hybride expérimentalement plus efficace. Celle-ci peut être vue comme un compromis entre les approches destructives et dichotomiques.

Considérons une approximation d'un MUC P_M. Les différentes étapes de cette approche sur P_M sont représentées dans la figure 7.2. Avant de minimiser le problème donné en paramètre *(1)*, les contraintes sont donc triées en fonction de poids (*weight*) pour l'heuristique *dom/wdeg (2)*, qui est ici utilisée pour valuer sémantiquement la force de chaque contrainte sur le problème, et donc déterminer de manière heuristique leur appartenance au MUC. Ensuite, la première contrainte de transition est obtenue en utilisant l'approche dichotomique, permettant d'exploiter son efficacité en partitionnant les contraintes en deux. Cela permet en particulier d'éliminer de nombreuses contraintes de la région basse (par rapport à leur score) qui n'appartiennent certainement pas au MUC *(3)*. Ensuite, les contraintes de transition suivantes sont déterminées par l'approche destructive, plus efficace que `DC` pour des approximations de qualité. Cette nouvelle

méthode, appelée `CB` (puisqu'elle *ComBine* les autres approches), est présentée formellement dans l'algorithme 19.

7.3 Comparaisons expérimentales

Afin de valider nos hypothèses, des expérimentations intensives ont été effectuées sur de nombreux benchmarks CSP, provenant pour la plupart des dernières compétitions organisées [Compétitions CSP]. `full-wcore`, `DC(full-wcore)`, `DS(full-wcore)` et `CB(full-wcore)` ont été implémentés en `C`. De plus, `DC(wcore)` ayant été originellement codé en Java, celui-ci a été réimplémenté en C, dans le but d'effectuer une comparaison équitable. Toutes les expérimentations ont été effectuées sur des processeurs Pentium IV 3GHz, sous Linux Fedora Core 5.

Dans la table 7.1, `wcore` et `full-wcore` sont comparées. Comme aucune de ces deux procédures ne garantit la minimalité du problème retourné, nous avons ensuite appliqué les 3 approches de minimisation destructive, dichotomique et combinée à chacune d'entre elles. La méthode constructive, peu compétitive en pratique, n'a pas été considérée dans ces expérimentations. Pour chaque problème, nous listons son nombre de variables (#V), de contraintes (#C), et le nombre de contraintes du sous-problème extrait ($|UC|$) avec le temps en secondes nécessaire à son calcul. Ensuite, ces sous-problèmes ont été minimisés avec les procédures `DS`, `DC` et `CB`. Pour chacune d'entre elles, le nombre d'appels à un solveur complet est reporté, en distinguant les appels « satisfaisables » et « insatisfaisables » (respectivement #S and #U), ainsi que leur temps de calcul. Une limite de 3600 secondes a été respectée pour chacune de ses expérimentations.

Comme le montrent ces résultats, exploiter toutes les contraintes après filtrage, même quand l'une d'entre elles a été violée, permet d'obtenir un sous-problème insatisfaisable réduit. En effet, la plupart du temps, la taille du problème retourné par `full-wcore` est plus faible que celle du problème fourni par `wcore`. Par exemple, `full-wcore` permet l'obtention d'un sous-problème composé de 358 contraintes en 6,86 secondes pour le problème d'allocation de fréquences `scen1_f9`, alors que `wcore` retourne un problème insatisfaisable de 1421 contraintes en 3,67 secondes.

De plus, le fait d'explorer toutes les contraintes, même si l'une d'entre elles est déjà falsifiée, ne ralentit pas nécessairement ce calcul. Bien que plus de temps est parfois nécessaire pour faire cette approximation, il apparaît en pratique que le temps de calcul de `full-wcore` est en général du même ordre, voire inférieur à `wcore`. Ceci est dû principalement aux choix plus appropriés qui sont effectués par l'heuristique *dom/wdeg*, qui semble guidée plus directement vers les contraintes problématiques. Par exemple, le même sous-ensemble incohérent de 793 contraintes est extrait de `qcp-o15-h120-268-15` ; cependant, `full-wcore` n'a besoin que de 100 secondes pour effectuer ce calcul, alors que `wcore` demande plus de deux fois ce temps.

D'autre part, notre tentative pour améliorer les approches de minimisation semble également prometteuse. En effet, bien que `CB` ne retourne pas le meilleur résultat pour *chaque* benchmark, son comportement moyen est très satisfaisant. Par exemple, quand l'approximation fournie est mauvaise (par exemple avec `scen11_f12`), l'approche destructive se montre très inefficace, alors que celle dite dichotomique est appropriée. Notre schéma hybride permet d'éliminer de nombreuses contraintes grâce à la première étape dichotomique, et un MUC peut être obtenu en un temps raisonnable. Au contraire, `full-wcore` extrait un ensemble de contraintes de l'instance `qcp-o20-h187-9-20`, qui se révèle être un MUC. Pour ce genre d'« approximations », la procédure dichotomique atteint son pire cas, alors que l'approche destructive prouve la minimalité de ce sous-problème en un nombre linéaire d'appels « satisfaisables » à une méthode complète, qui sont en pratique rapides par rapport aux appels « insatisfaisables ». Une fois encore, `CB` se

Instance	#C	#V	`wcore`		`DC(wcore)`			`DS(wcore)`			`CB(wcore)`		
			$\|UC\|$	temps	(#S,#U)	$\|MUC\|$	temps	(#S,#U)	$\|MUC\|$	temps	(#S,#U)	$\|MUC\|$	temps
scen11_f10	4103	680	711	11.5	(96,46)	16	17.4	(16,695)	16	588	(22,513)	16	153
scen11_f12	4103	680	610	9.3	(96 ,40)	16	14.4	(-,-)	-	*t.o.*	(21,427)	16	129
scen1_f9	5548	916	1421	3.67	(-,-)	-	*t.o.*	(25,1396)	25	323	(31,1260)	25	564
composed-75-1-2-2	624	33	529	0.18	(112,23)	14	1292	(13, 516)	13	27.1	(20 ,511)	14	33.2
composed-25-1-40-2	262	33	236	0.48	(79,22)	13	13.5	(13,223)	13	13.6	(18,181)	13	6.6
composed-25-1-40-4	262	33	239	0.23	(-,-)	-	*t.o.*	(14,225)	14	5.26	(17,184)	13	5.68
composed-25-1-80-0	302	33	240	0.09	(64,22)	11	12.9	(13,227)	13	4.78	(18,146)	14	4.42
dual_ehi-85-297-1	4112	297	209	0.26	(159,36)	34	2.43	(42,167)	42	58.5	(39,108)	36	2.08
dual_ehi-85-297-24	4105	297	206	0.22	(165,29)	34	2.37	(40,166)	40	2.72	(38,105)	34	1.79
dual_ehi-85-297-26	4102	297	179	0.25	(139,32)	30	2.13	(40,139)	40	2.38	(46,113)	41	1.99
dual_ehi-85-297-44	4130	297	178	0.2	(135,29)	29	1.99	(26,152)	26	2.34	(33,85)	29	1.43
dual_ehi-85-297-49	4124	297	192	0.21	(207,28)	41	2.82	(42,150)	42	2.52	(42,64)	40	1.24
dual_ehi-85-297-65	4116	297	156	0.2	(1099,0)	156	13.6	(156,0)	156	1.66	(163,0)	156	1.74
dual_ehi-85-297-7	4111	297	160	0.22	(151,27)	30	2.34	(33,127)	33	2.17	(40,107)	34	1.95
dual_ehi-90-315-6	4365	297	200	0.27	(261,36)	50	3.81	(51,149)	51	3.1	(52,80)	47	1.67
dual_ehi-90-315-94	4380	297	174	0.23	(158,44)	33	2.61	(43,131)	43	2.4	(47,111)	42	2.08
geo50.20.d4.75.70	451	50	424	140	(-,-)	-	*t.o.*	(-,-)	-	*t.o.*	(-,-)	-	*t.o.*
qcp-o15-h120-b-268-15	3150	225	793	237	(7146,0)	793	237	(793,0)	793	26.4	(802,0)	793	26.4
qcp-o20-h187-b-27-20	7600	400	389	2.6	(3120,0)	389	116	(389,0)	389	14.4	(397,0)	389	14.7
qcp-o20-h187-b-29-20	7600	400	958	6.3	(8631,0)	958	551	(958,0)	958	63.8	(967,0)	958	64.5

Instance	#C	#V	`full-wcore`		`DC(full-wcore)`			`DS(full-wcore)`			`CB(full-wcore)`		
			$\|UC\|$	temps	(#S,#U)	$\|MUC\|$	temps	(#S,#U)	$\|MUC\|$	temps	(#S,#U)	$\|MUC\|$	temps
scen11_f10	4103	680	707	15.3	(97,45)	16	17.3	(16,691)	16	565	(22,512)	16	150
scen11_f12	4103	680	606	13.0	(97,38)	16	14.2	(-,-)	-	*t.o.*	(22,425)	16	130
scen1_f9	5548	916	358	6.86	(-,-)	-	*t.o.*	(25,333)	25	49.3	(28,242)	25	188
composed-75-1-2-2	624	33	529	3.04	(112,23)	14	1293	(13,516)	13	27.3	(20,511)	14	32.9
composed-25-1-40-2	262	33	227	0.59	(78,23)	13	25.6	(13,214)	13	4.63	(17,174)	13	4.78
composed-25-1-40-4	262	33	226	0.63	(-,-)	-	*t.o.*	(14,212)	14	4.64	(18,171)	13	5.14
composed-25-1-80-0	302	33	232	0.7	(63,22)	11	689	(13,219)	13	4.65	(17,139)	14	10.4
dual_ehi-85-297-1	4112	297	162	0.42	(169,30)	34	2.42	(40,122)	40	2.06	(37,59)	35	1.13
dual_ehi-85-297-24	4105	297	187	0.38	(181,38)	38	2.66	(42,145)	42	8.55	(41,98)	38	1.7
dual_ehi-85-297-26	4102	297	148	0.45	(151,37)	32	2.34	(36,112)	36	1.88	(42,91)	37	1.64
dual_ehi-85-297-44	4130	297	103	0.35	(624,0)	103	7.28	(103,0)	103	0.98	(109,0)	103	1.06
dual_ehi-85-297-49	4124	297	166	0.37	(224,39)	44	3.19	(39,127)	39	2.12	(43,77)	39	1.44
dual_ehi-85-297-65	4116	297	156	0.37	(1099,0)	156	13.7	(156,0)	156	1.67	(163,0)	156	1.76
dual_ehi-85-297-7	4111	297	109	0.37	(152,20)	32	2.23	(29,80)	29	1.41	(34,37)	31	0.85
dual_ehi-90-315-6	4365	297	153	0.38	(236,24)	47	3.31	(46,107)	46	2	(51,52)	46	1.25
dual_ehi-90-315-94	4380	297	146	0.36	(158,40)	32	2.56	(31,115)	31	2.13	(32,87)	30	1.54
geo50.20.d4.75.70	451	50	417	171	(-,-)	-	*t.o.*	(-,-)	-	*t.o.*	(-,-)	-	*t.o.*
qcp-o15-h120-b-268-15	3150	225	793	100	(7146,0)	793	238	(793,0)	793	26.6	(802,0)	793	26.6
qcp-o20-h187-b-27-20	7600	400	389	3.29	(3120,0)	389	116	(389,0)	389	14.3	(397,0)	389	14.7
qcp-o20-h187-b-29-20	7600	400	853	7.9	(7686,0)	853	453	(853,0)	853	52.0	(862,0)	853	52.8

TAB. 7.1 – `wcore`, `full-wcore` et méthodes de minimisation : résultats expérimentaux

comporte relativement bien, puisqu'il retourne ce MUC en moins d'une minute (comme `DS`), alors que `DC` ne peut assurer cette minimalité qu'au bout de plus de 7 minutes.

De plus, sur de nombreux benchmarks, comme `dual_ehi-85-297-24`, `CB` se montre l'approche la plus efficace pour le calcul d'un MUC. En fait, cette nouvelle approche utilise les principales caractéristiques des deux approches sur lesquelles elle se base, alors qu'en même temps, elle évite leurs revers, autant que possible. Par le retrait heuristique d'un grand nombre de contraintes par voie dichotomique, puis par des tests pas-à-pas, la procédure de minimisation s'en trouve souvent améliorée, et semble plus robuste que celles précédemment proposées.

Notons également que différents MUC peuvent être calculés par ces différentes méthodes.

7.4 Conclusions

Dans ce chapitre, nous avons proposé diverses variantes aux meilleurs algorithmes connus pour l'extraction d'un MUC au sein d'un CSP. Dans un premier temps, nous avons montré comment une exploration systématique des contraintes violées lors d'un conflit peut fournir une information heuristique plus efficace que le mise en évidence d'une seule de ces contraintes, choisie arbitrairement. Ensuite, nous nous concentrons sur une combinaison spécifique de plusieurs méthodes de minimisation, car certaines d'entre elles présentent des propriétés intéressantes, mais semblent adaptées à certains types d'approximation (grossières pour la méthode dichotomique, précises pour la destructive). La combinaison effectuée vise donc à améliorer en robustesse de l'approche de minimisation et lui offrir une efficacité « générique », pour tous types d'approximation. Il semble évident que d'autres combinaisons de ces algorithmes peuvent être envisagées, et feront l'objet de futures recherches.

Chapitre 8

Explication de l'incohérence en CSP : de la contrainte au tuple

Sommaire

Le concept d'ensemble minimalement insatisfaisable de contraintes (MUC) d'un problème CSP permet de localiser voire de réparer une instance incohérente en supprimant une ou plusieurs de ces contraintes. Dans ce chapitre, nous proposons une technique permettant d'affiner ce concept, et de restaurer la consistance d'un problème sans suppression de contraintes. Ces travaux ont fait l'objet d'une publication [Grégoire *et al.* 2007d].

8.1 Ensembles minimaux incohérents de tuples

8.1.1 Définitions, propriétés

Dans le chapitre précédent, nous avons vu que la « réparation » d'un CSP censé être satisfaisable peut être effectuée à travers celle de chacun de ses MUC. Une manière naturelle passe par la suppression d'une contrainte de ce MUC. Cependant, retirer intégralement une contrainte peut apparaître comme un acte très destructif pour la sémantique du problème modélisé. Il serait préférable, quand cela est possible, d'*affaiblir* une ou plusieurs contraintes, plutôt que les supprimer intégralement. Une telle opération nécessite clairement une étude analytique de l'incohérence au niveau des tuples interdits du problème.

Dans cet objectif, nous introduisons le concept d'ensemble minimalement incohérent de tuples (MUST pour *Minimally Unsatisfaisable Set of Tuples* en anglais), qui raffine celui MUC, défini classiquement en CSP. Celui-ci consiste en un ensemble incohérent de tuples interdits tel que si l'on autorise n'importe lequel de ces tuples, l'ensemble retrouve sa cohérence.

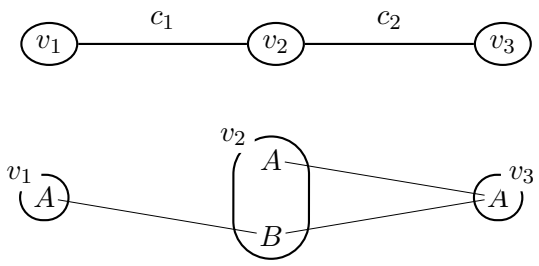


FIG. 8.1 – Représentations graphiques de l'exemple 8.1

Définition 8.1 (Ensemble minimalement incohérent de tuples)
Soit $P = \langle V, \{(Var(c_1), T(c_1)), \ldots, (Var(c_n), T(c_n))\}\rangle$ *un CSP insatisfaisable. Le CSP* $P' = \langle V, \{(Var(c_1), T'(c_1)), \ldots, (Var(c_n), T'(c_n))\}\}\rangle$ *est un* MUST *(pour* Minimally Unsatisfaisable Set of Tuples *en anglais) de* P *si et seulement si* $\forall i$ *tel que* $1 \leq i \leq n$ *:*

1. P' *est insatisfaisable*
2. $T'(c_i) \subseteq T(c_i)$
3. $\forall\, T''(c_i) \subset T'(c_i)$, $\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_i), T''(c_i)), ..., (Var(c_n), T'(c_n))\}\rangle$ *est satisfaisable.*

Ainsi défini, un MUST peut être vu comme un moyen d'expliquer l'incohérence à un niveau d'abstraction plus bas que celui permis par les contraintes. Cependant, on retrouve avec les MUST les problèmes inhérents aux ensembles minimalement incohérents. Il ne suffit donc pas de supprimer un tuple d'un MUST d'un problème pour restaurer la cohérence d'un CSP. En effet, on peut prouver facilement qu'il existe dans le pire cas un nombre exponentiel de MUST au sein d'un CSP incohérent, et que ceux-ci peuvent avoir une intersection vide. De ce fait, le retrait d'un unique tuple peut être insuffisant pour la réparation d'un problème de satisfaction de contraintes. De plus, comme le montre l'exemple suivant, les tuples contenus dans un MUST peuvent même ne pas prendre *réellement* part aux causes de l'incohérence d'un CSP.

Exemple 8.1
Soit $P = \langle V, C\rangle$ tel que $V = \{v_1, v_2, v_3\}$, avec $dom(v_1) = dom(v_3) = \{A\}$, $dom(v_2) = \{A, B\}$, et $C = \{c_1 = \{(\{v_1, v_2\}, \{(A, B)\}), c_2 = \{(\{v_2, v_3\}, \{(A, A),\ (B, A)\})\}\}$. Dans la figure 8.1, le réseau de contraintes ainsi que le graphe de micro-structure de ce CSP sont donnés. Clairement, P est insatisfaisable et ne possède qu'un seul MUC, fait de la seule contrainte c_2, puisque celle-ci empêche toute affectation valide entre les variables v_2 et v_3. Au contraire, à un niveau d'abstraction plus bas, P contient 2 MUST, qui sont :

- $P_{M_1} = \langle V, \{(\{v_1, v_2\}, \{(A, B)\}), \{(\{v_2, v_3\}, \{(A, A)\})\}\}\rangle$
- $P_{M_2} = \langle V, \{(\{v_2, v_3\}, \{(A, A),\ (B, A)\})\}\rangle$

P_{M_1} est un MUST qui contient des tuples contenus dans chacune des deux contraintes. Il ne correspond donc à aucun MUC de P. Par contre, P_{M_2} est un MUST qui est également un MUC de P. Notons que P_{M_1} contient le seul tuple interdit liant v_1 et v_2, qui ne participe pas à proprement parler à l'insatisfaisabilité de P.

Bien que ces premiers résultats puissent paraître négatifs, il est montré dans le paragraphe suivant que les MUST forment bien un concept viable pour exprimer les causes de l'incohérence d'un CSP, s'ils sont calculés au sein d'un MUC.

8.1.2 MUST au sein d'un MUC

Le fait que tout CSP incohérent contienne au moins un MUC (qui peut être le CSP lui-même) est un résultat bien connu. De manière similaire, tout CSP incohérent exhibe au moins un MUST. Un MUC étant un CSP insatisfaisable, on peut également en extraire au moins un MUST.

Propriété 8.1
Au moins un MUST peut être extrait de tout CSP insatisfaisable.

Preuve
Soit $P = \langle V, C\rangle$ un CSP insatisfaisable. Supposons que P ne contienne aucun MUST. P lui-même n'est donc pas un MUST, et on peut trouver un tuple interdit par les contraintes du problème tel que l'autoriser conserve l'incohérence du problème. Formellement, $\exists c \in C, \exists t \in T(c)$ tel que $P^1 = \langle V, (C \backslash c) \cup (Var(c), T(c) \backslash t)\rangle$ est également insatisfaisable, et ne contient aucun MUST. Poursuivre ce raisonnement conduit à prouver par induction l'existence du CSP insatisfaisable $P' = \langle V, \emptyset\rangle$, alors qu'un tel problème est clairement satisfaisable. □

De surcroît, des relations plus fortes entre MUC et MUST sont démontrées par la propriété suivante :

Propriété 8.2
Soit P un MUC contenant m contraintes. Il existe au moins m tuples tel qu'autoriser l'un d'entre eux restaure la satisfaisabilité de P. Ces tuples appartiennent à tous les MUST de P.

Preuve
Puisque $P = \langle V, C\rangle$ est un MUC, dès que l'une de ses m contraintes est retirée, le CSP résultant est cohérent. Soit A une affectation satisfaisant $P' = \langle V, C \backslash c_i\rangle$, avec $c_i \in C$. c_i est falsifiée par A : en effet, dans le cas contraire, P serait cohérent et donc pas un MUC. La projection de A sur $Var(c_i)$ est donc inclue dans $T(c_i)$ (cf. Définition 6.4 page 102). Ainsi, retirer ce tuple interdit est suffisant pour rendre P cohérent. Le même argument peut être utilisé pour chacune des m contraintes de P. On peut donc trouver au moins m tuples qui permettent de restaurer la cohérence de ce MUC s'ils sont autorisés. De manière triviale, ces tuples appartiennent nécessairement à toutes ses sources d'incohérence, et par conséquent à tous ses MUST. Ainsi, l'intersection ensembliste de tous les MUST de P contient au moins m éléments. □

Cette propriété prouve donc l'existence de tuples permettant à un MUC d'être « cassé », simplement en autorisant l'un d'eux. Ceux-ci appartiennent nécessairement à toutes les sources d'incohérence de ce MUC, et donc à tous ses MUST. De ce fait, il n'est pas possible de découvrir deux MUST dont l'intersection est vide, au sein d'un MUC. Les tuples appartenant à tous les MUST d'un MUC P sont appelés par la suite *tuples partagés* de P.

Définition 8.2 (tuples partagés)
Soit P un MUC. Les tuples partagés de P sont les tuples interdits appartenant à tous les MUST de P.

Autoriser n'importe quel tuple partagé permet de casser le MUC correspondant. De plus, calculer un MUST au sein d'un MUC permet d'obtenir un sur-ensemble des tuples partagés.

Exemple 8.2
Soit $P = \langle\{v_1, v_2, v_3\}, \{c_1, c_2, c_3\}\rangle$ un CSP tel que :

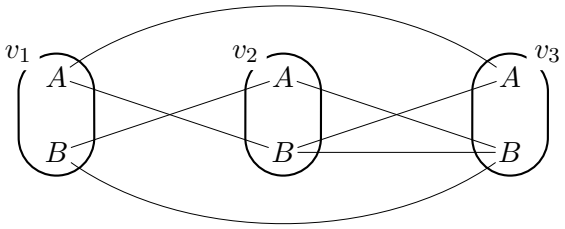


FIG. 8.2 – Graphe de micro-structure de l'exemple 8.2

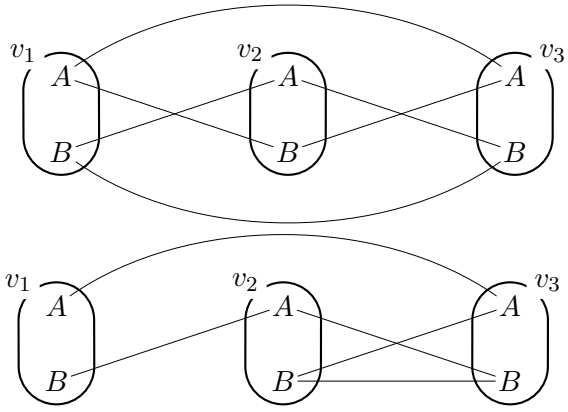


FIG. 8.3 – Graphe de micro-structure des deux MUST de l'exemple 8.2

1. $\forall i \in \{1,2,3\}, dom(v_i) = \{A, B\}$
2. $c_1 = (\{v_1, v_2\}, \{(A,B),(B,A)\})$
3. $c_2 = (\{v_2, v_3\}, \{(A,B),(B,A),(B,B)\})$
4. $c_3 = (\{v_1, v_3\}, \{(A,A),(B,B)\})$

Le graphe de micro-structure de P est donné en figure 8.2. Notons que P est un MUC, puisqu'il est insatisfaisable et lui ôter n'importe laquelle de ses contraintes conduit à un problème satisfaisable. P possède deux MUST, dont les graphes de micro-structure sont représentés en figure 8.3. En effectuant l'intersection de ces deux MUST, l'ensemble des tuples partagés de P est obtenu. Ceux-ci sont reportés en gras en figure 8.4, tandis que les tuples non-partagés sont, eux, en pointillés. Clairement, autoriser un seul tuple partagé restaure la cohérence de ces deux MUST, et par conséquent du MUC P. Cependant, autoriser un autre tuple de l'un des MUST ne garantit pas d'obtenir un problème satisfaisable. Cet exemple montre également que le CSP composé des seuls tuples partagés n'est pas nécessairement insatisfaisable. En fait, on peut facilement prouver que ceci n'est vrai que pour les MUC ne contenant qu'un seul MUST.

Ces derniers résultats plaident pour une approche en deux temps pour l'explication de l'incohérence en termes de MUC, MUST et tuples partagés. En effet, la recherche de MUST dans le cas général ne semble pas une approche prometteuse, puisqu'un MUST peut ne coïncider avec aucun MUC. Au contraire, une approche intéressante consiste à rechercher un MUC dans un premier temps, qui fournit une explication en terme de contraintes. L'expert peut alors décider de supprimer l'une de ses contraintes pour casser le MUC. Il possède cependant une alternative qui consiste à effectuer le calcul d'un MUST au sein du MUC découvert. Plus précisément, s'il

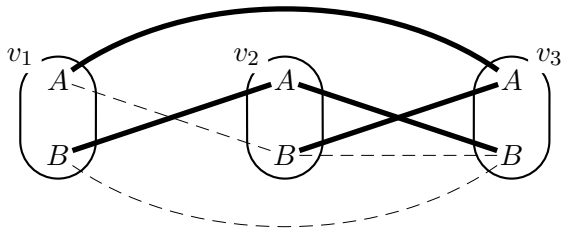


FIG. 8.4 – Tuples partagés de l'exemple 8.2

se charge de calculer les tuples partagés, l'expert pourra choisir un tuple parmi cet ensemble à autoriser. Celui-ci est *suffisant* pour casser l'incohérence du MUC. De cette façon, on a la possibilité d'affaiblir les contraintes problématiques, plutôt que les supprimer complètement. Une telle politique peut être itérée jusqu'à ce que le problème global devienne consistant.

Plusieurs algorithmes ont été proposés pour l'obtention d'un MUC, comme l'approche `CB(full-wcore)` proposée dans le chapitre précédent. Ainsi, le problème qui nous est adressé est maintenant le calcul d'un MUST, ainsi que des tuples partagés. Ces deux calculs, effectués à travers la logique propositionnelle, sont décrits dans le paragraphe suivant.

8.2 Calcul d'un MUST

8.2.1 Passage par la CNF

Le calcul d'un MUST peut être effectué à l'aide de plusieurs techniques traditionnelles venues des domaines du CSP et de la recherche opérationnelle, telle que les procédures destructives, additives ou dichotomiques [de Siqueira & Puget 1988, Bakker *et al.* 1993, Chinneck 1997, Hemery *et al.* 2006]. Cependant, ces approches ne délivreraient qu'un MUST, uniquement. L'obtention de l'ensemble des tuples partagés nécessite en sus un nombre linéaire (en fonction du nombre de tuples interdits induits par le MUC) de tests de satisfaisabilité. Au contraire, nous proposons ici une méthode permettant d'effectuer le calcul de ces deux ensembles de tuples avec un même algorithme.

Quand un MUC a été obtenu grâce à `CB(full-wcore)` (cf. chapitre précédent), celui-ci est converti dans le cadre booléen, de façon à ce que le calcul d'un MUST et des tuples partagés soit effectué à travers celui d'un MUS. Le schéma de conversion choisi est une forme de *direct encoding* [de Kleer 1989], qui consiste à coder chaque valeur de chaque variable du CSP par une nouvelle variable booléenne C_{v_i}. De cette façon, le nombre de variables de la formule CNF correspondante est donné par la somme de la taille du domaine de chaque variable du problème original. Soit $P = \langle V, C \rangle$ un MUC. On crée alors les clauses suivantes :

1. *at-Least-one clauses.* Ces clauses permettent de s'assurer qu'au moins une valeur de chaque variable est affectée dans une solution
   $C_{v_1} \vee C_{v_2} \vee ... \vee C_{v_m} \ \forall v \in V$ avec $dom(v) = \{v_1, v_2, ..., v_m\}$
2. *at-Most-one clauses.* Au contraire, ces clauses permettent de garantir qu'au plus une valeur par variable est sélectionnée dans une solution
   $\neg C_{v_a} \vee \neg C_{v_b} \ \forall v \in V \ \forall (v_a, v_b) \in dom(v_a) \times dom(v_b)$ (avec $v_a \neq v_b$)
3. *Conflict clauses.* Ces clauses codent les tuples interdits du problème
   $\neg C_{v_i} \vee \neg C_{v_j} \ \forall c \in C \ \forall (v_i, v_j) \in T(c)$

Fonction direct_encode($\langle V, C\rangle$: un **CSP**) : **une formule CNF**
  $\Sigma \longleftarrow \emptyset$;
  [Création des clauses « At-least-one *»]*
  Pour chaque $v_i \in V$ **faire**
    $\Sigma \longleftarrow \Sigma \bigcup \{\bigvee_{j \in dom(v_i)} x_{ij}\}$
  Fin Pour
  [Codage de chaque tuple interdit par une clause conflit*]*
  Pour chaque $c \in C$ **faire**
    Pour chaque $t \in T(c)$ **faire**
      $\Sigma = \Sigma \bigcup \{\bigvee_{\substack{\forall (v_i \in Var(c) \text{ et } j \in dom(v_i)) \\ \text{t.q. } j \text{ est la valeur interdite de } v_i \text{ dans } t}} \neg x_{ij}\}$
    Fin Pour
  Fin Pour
  Retourner Σ ;
Fin

Algorithme 20 – `direct_encode`

Cette forme de codage a été choisie car elle transforme chaque tuple interdit en une unique clause. En outre, les « at-Most-one » sont pas ajoutées en pratique à la formule générée. Ces clauses ont en effet été prouvée facultatives [Walsh 2000] ; de plus, les omettre permet d'obtenir une équivalence de minimalité entre les deux cadres : chaque MUS de la formule CNF correspond à un MUST du CSP transformé. Le calcul d'un MUST revient donc à celui d'un MUS dans la logique propositionnelle, pourvu que toutes les clauses *at-Least-one* en fassent partie.

Plus précisément, il est fait usage de la procédure `OMUS`, décrite dans le chapitre 4 page 45. Comme décrit, l'efficacité de cette technique est due en partie au concept de *clauses protégées*, qui codent justement les tuples partagés du problème. En pratique, quand une seule clause est falsifiée par une interprétation complète parcourue par la recherche locale de `OMUS`, celle-ci est marquée et ne sera jamais retirée de la formule, dans laquelle on recherche un MUS. En effet, il est prouvé que ces clauses appartiennent à tous les MUS de la formule, et avec le codage choisi pour la transformation du MUC, celles-ci coïncident exactement avec les tuples partagés. Ces tuples sont alors retournés à la fin de la procédure, avec le MUST extrait, sans nécessiter de ressources supplémentaires pour leur calcul. Assez clairement, cette technique de localisation de ces tuples est incomplète, dans le sens où on ne peut garantir leur découverte exhaustive. Cependant, comme nos résultats expérimentaux le montrent, en pratique de nombreux tuples partagés sont détectés via le calcul du MUST et le concept de clauses protégées.

8.2.2 Localisation de l'inconsistance au niveau tuple : un exemple

Exemple 8.3
Soit le CSP insatisfaisable $P = \langle V, C\rangle$ avec $V = \{a, b, c, d\}$ ($\forall v \in V,\ dom(v) = \{1, 2, 3\}$) et $C = \{(a + b)$ *est un multiple de* $4,\ b > d,\ c < d,\ (c = b)$ *OU* $(b = 1, c = 3)\}$. Supposons que l'on désire restaurer la cohérence de P. Dans un premier temps, une source d'inconsistance au niveau des contraintes est calculée au travers un MUC.

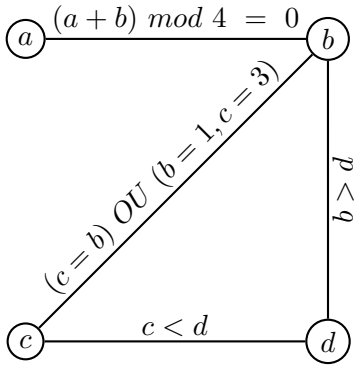


FIG. 8.5 – Réseau de contraintes de l'exemple 8.3

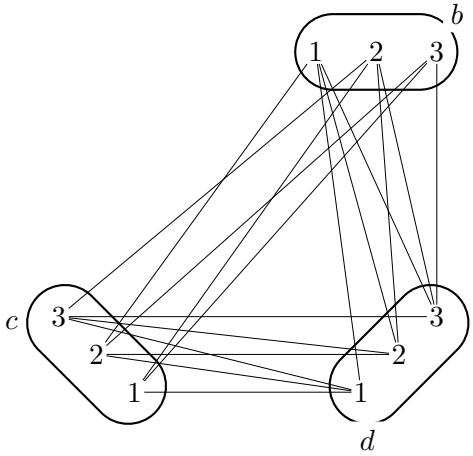


FIG. 8.6 – Graphe de micro-structure du MUC de l'exemple 8.3

En fait, P ne possède qu'un MUC, qui est $P_{MUC} = \langle V, C \backslash (a+b)$ *est un multiple de* $4 \rangle$. On peut donc choisir de supprimer qu'une des contraintes de P_{MUC}, afin de réparer P. Cependant, la suppression d'une de ses contraintes peut être un acte très destructif pour le problème modélisé. De plus, en observant le graphe de micro-structure de P_{MUC} en figure 8.6, il semble évident que l'ensemble de ces tuples n'est pas en cause de l'insatisfaisabilité de P. Un MUST de P_{MUC} est donc calculé, et représenté en figure 8.7, les tuples partagés étant en lignes pleines, et les autres tuples en pointillés. On voit donc que seuls quelques tuples de P_{MUC} entrent en contradiction. De plus, le retrait d'un tuple partagé permet d'affaiblir une contrainte, plutôt que d'en supprimer une, avec le même effet quant à la restauration de la cohérence.

Par exemple, le retrait du tuple interdisant les valeurs 1 et 2 pour les variables b et d, respectivement, est suffisant pour réparer l'ensemble du CSP. L'expert possède dès lors le choix d'ajouter cette « exception » après modélisation du problème afin de le rendre satisfaisable.

8.3 Une première étude expérimentale

Afin de vérifier la capacité pratique des approches proposées dans ce document à extraire un MUST, nous avons implémenté un algorithme nommé `MUSTER` (pour *MUST-ExtRactor* en anglais) et exécuté celui-ci sur de nombreux benchmarks issus de la dernière compétition CSP

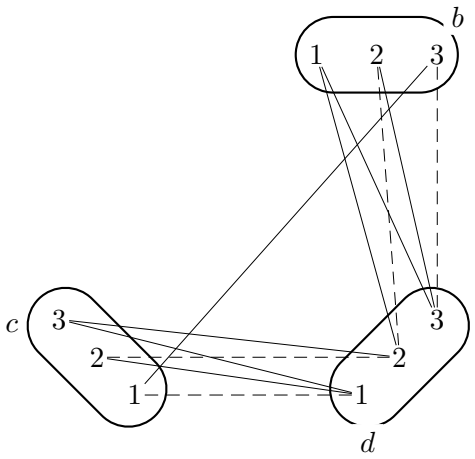


FIG. 8.7 – MUST et tuples partagés de l'exemple 8.3

Fonction mus2must(Σ : un **MUS** et $\langle V, C\rangle$: un **MUC**) : **un MUST**
  Pour chaque $c \in C$ t.q. $C = \{(Var(c_1), T(c_1)), ..., (Var(c_n), T(c_n))\}$ **faire**
    Pour chaque $t \in T(c)$ **faire**
      Si $((\bigvee_{\substack{\forall (v_i \in Var(c) \text{ et } j \in dom(v_i)) \\ \text{t.q. } j \text{ est la valeur interdite de } v_i \text{ dans } t}} \neg x_{ij}) \in \Sigma)$ **Alors**
        $T'(c) \longleftarrow T'(c) \cup \{t\}$;
      Fin Si
    Fin Pour
  Fin Pour
  Retourner $\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_n), T'(c_n))\}\rangle$;
Fin

Algorithme 21 – `mus2must`

[Compétitions CSP]. Son principe est donc d'extraire en premier lieu un MUC grâce à un appel à `CB(full-wcore)`, puis de le transformer en une formule clausale booléenne, grâce à la technique de *direct-encoding* décrite dans l'algorithme 20. Un MUS est alors calculé via la procédure `OMUS` (cf. chapitre 4 page 45). Comme nous l'avons vu, il existe une correspondance parfaite entre le MUS extrait et un MUST du CSP transformé. Celui-ci est retourné, avec les tuples partagés détectés, en utilisant une simple fonction d'encodage décrite dans l'algorithme 21. `MUSTER` est résumé dans l'algorithme 22.

Toutes les expérimentations présentées ici ont été conduites sur un Pentium IV à 3Ghz, sous une distribution Fedora Core 5 de Linux. Un extrait significatif des résultats obtenus est fourni dans la table 8.1. Celle-ci contient 3 colonnes principales : *Instance*, *MUC extrait* et *MUST extrait*. Le première de ces colonnes fournit différentes informations à propos du problème traité, à savoir son nom, le nombre de contraintes qu'il comporte, et le nombre de tuples interdits que celles-ci impliquent. La deuxième colonne est consacrée au MUC extrait de ce problème. Il y est indiqué le nombre de contraintes qu'il contient, avec celui des tuples interdits qu'elles

Fonction MUSTER($P = \langle V, C\rangle$: un **CSP insatisfaisable**) : **un MUST de** P
  [Extraction d'un MUC de P]
  $\langle V, C'\rangle \longleftarrow$ `CS(full-wcore)`$(\langle V, C\rangle)$;
  [Transformation du MUC en formule CNF]
  $\Sigma_{CNF} \longleftarrow$ direct_encode$(\langle V, C'\rangle)$;
  [Localisation d'un MUS dans la CNF générée]
  $\Sigma_{MUS} \longleftarrow$ `OMUS`(Σ_{CNF}) ;
  [Conversion du MUS en MUST de P]
  $\langle V, \{(Var(c'_1), T''(c'_1)), ..., (Var(c'_n), T''(c'_n))\}\rangle \longleftarrow$ mus2must$(\Sigma_{MUS}, \langle V, C'\rangle)$;
  Retourner $\langle V, \{(Var(c'_1), T''(c'_1)), ..., (Var(c'_n), T''(c'_n))\}\rangle$;
Fin

Algorithme 22 – `MUSTER` (pour *a MUST-ExtRactor* an anglais)

représentent. Le temps nécessaire à l'extraction de ce MUC est également reporté. Enfin, nous avons dédié la troisième colonne au MUST extrait de ce MUC. Celle-ci contient le nombre de tuples appartenant au MUST, le nombre de tuples partagés découverts (colonne *tp*) et le temps requis à l'obtention de ces résultats par `OMUS`. Une limite de temps de 3 heures de temps CPU a été respectée. Au delà de cette limite, la mention «*time out*» est reportée.

Tout d'abord, on peut remarquer qu'un MUST a pu être extrait en un temps raisonnable de quasiment tous les benchmarks testés. Par exemple, un MUC composé de 13 contraintes est d'abord découvert au sein de l'instance `composed-75-1-2-1` qui comporte, elle, 624 contraintes. Les contraintes de ce MUC rendent incompatibles 845 tuples de valeurs entre variables, mais cet ensemble peut être réduit à seulement 344 tuples, qui forment un MUST de ce problème. De surcroît, la procédure fournit à l'expert un ensemble de 104 tuples, tel que si l'un d'eux est autorisé, les contraintes du MUC deviennent cohérentes. Un résultat similaire peut être observé sur un CSP issu du problème d'allocation de fréquences de liaisons radios (RLFAP). Ce problème inconsistant implique près de 800 000 tuples interdits. Toutefois, seuls quelques uns d'entre eux participent réellement à son insatisfaisabilité. En effet, un MUC contenant moins de 5 000 tuples est d'abord exhibé, avant d'être réduit en un MUST de 3077 tuples. Dans ce MUST, un tuple parmi 2728 peut être supprimé pour restaurer la cohérence de ce MUC.

Assez clairement, à cause de la haute complexité de ce problème, nous ne pouvons toutefois pas nous attendre à pouvoir résoudre tous les problèmes dans un temps raisonnable. Par exemple, bien qu'un MUST soit extrait du problème des cavaliers et des reines `qk_8_8_5_add`, `MUSTER` n'a pas été capable de trouver aucun de ses tuples partagés. Malgré cela, nous sommes assurés que ceux-ci sont contenus parmi les quelques 10 000 tuples qui forment ce MUST. Notons également que les mêmes MUC et MUST ont été détectés au sein des CSP `qk_8_8_5_add` et `qk_8_8_5_mul`. Ceci est facilement expliquable : ces CSP résultent de différentes combinaisons entre les problèmes des 8 reines et celui des 5 cavaliers. Or, comme le seul problème des cavaliers ne possède pas de solution, une même explication pour son incohérence peut être donnée, quelle que soit la combinaison dont on fait de ce problème avec un autre, qu'il soit satisfaisable ou non.

Enfin, attardons-nous sur le nombre de tuples partagés. La propriété 8.2 nous assure qu'un MUST contenant m contraintes contient également au moins m tuples partagés. Pourtant, en pratique, ce nombre est souvent beaucoup plus grand. Du MUC de 43 contraintes extrait de

`graph2_f25`, on s'attend à trouver au moins 43 tuples partagés, mais notre algorithme nous en retourne pas moins de 1516. En les supposant équitablement répartis entre les contraintes, cela permet un choix parmi 35 tuples par contrainte, en moyenne.

8.4 Conclusions

Dans ce chapitre, nous avons introduit de nouveaux procédés permettant une gestion plus précise de l'inconsistance pour les CSP, à travers le concept de MUST. Nous avons établi une série de propriétés établissant des liens forts entre le niveau d'abstraction de ces derniers (les tuples interdits du problème) et celui des MUC (les contraintes). Nous avons ensuite utilisé ces liens pour établir une méthode permettant de calculer un MUST, ainsi que de nombreux tuples interdits duquel ils sont extraits, si l'un d'entre eux est autorisé. La procédure, nommée `MUSTER`, utilise la technique `OMUS`, présentée dans le chapitre 4 page 45. Les premiers tests expérimentaux montrent que le calcul de ces ensembles de tuples est souvent réalisable en pratique, pour de nombreux problèmes issus de la dernière compétition de solveurs CSP. Ainsi, il est possible de réparer en pratique un problème CSP en affaiblissant une ou plusieurs de ses contraintes, plutôt que de les supprimer.

[2] Tuples partagés

Instance			**MUC extrait**			**MUST extrait**		
nom	#con	#tuples	#con	#tuples	temps	#tuples	#tp^2	temps
composed-25-1-2-0	224	4440	14	910	10.7	354	119	13.1
composed-25-1-2-1	224	4440	15	975	9.09	339	59	17.5
composed-25-1-2-2	224	4440	13	845	8,82	393	86	15,3
composed-25-1-2-3	224	4440	14	910	9,13	317	84	17,5
composed-25-1-2-4	224	4440	13	845	9,66	301	60	8,74
composed-25-1-25-8	247	4555	9	585	8.85	259	116	6.25
composed-75-1-25-9	647	10555	12	780	64,8	306	89	8,87
composed-75-1-2-5	624	10440	15	975	63,5	402	109	15,9
composed-75-1-2-1	624	10440	13	845	66.4	344	104	10.9
composed-75-1-2-2	624	10440	14	910	66.7	376	48	14.4
composed-75-1-25-6	647	10555	11	715	58,2	278	55	8,15
composed-75-1-25-7	647	10555	16	1040	64,4	420	75	17,2
composed-75-1-25-8	647	10555	16	1040	59.1	461	51	23.7
composed-75-1-80-6	702	10830	11	715	61.5	278	55	8.17
composed-75-1-80-7	702	10830	16	1040	379	420	75	17.2
composed-75-1-80-9	702	10830	12	780	86.2	306	89	9.01
qk_8_8_5_add	38	19624	5	18800	3.33	10149	0	544
qk_8_8_5_mul	78	19944	5	18800	0.66	10149	0	531
qk_10_10_5_add	55	48640	5	47120	19.7	24855	0	3081
qk_10_10_5_mul	105	49140	5	47120	1.3	24855	0	2813
qk_15_15_5_add	115	250575	5	245845	830	-	-	*time out*
qk_15_15_5_mul	190	251700	5	245845	23,5	-	-	*time out*
graph2_f25	2245	145205	43	4498	427	2470	1516	426
qa_3	40	800	15	583	0.32	203	152	8.32
dual_ehi-85-297-14	4111	102234	40	1145	3.35	311	142	40.3
dual_ehi-85-297-15	4133	102433	35	1083	4.03	310	172	25.1
dual_ehi-85-297-16	4105	102156	36	1032	4.68	301	159	29.1
dual_ehi-85-297-17	4102	102112	43	1239	4.83	348	172	42.2
dual_ehi-85-297-18	4120	102324	33	972	2,74	271	124	24,8
dual_ehi-85-297-19	4118	102304	38	1057	23,63	346	171	28,5
dual_ehi-90-315-21	4388	108890	37	1120	3.16	354	129	35
dual_ehi-90-315-22	4368	108633	41	1218	4.57	410	187	43.7
dual_ehi-90-315-23	4375	108766	29	835	2.86	251	131	12.2
dual_ehi-90-315-24	4378	108793	31	974	4.57	315	167	25.4
dual_ehi-90-315-25	4398	108974	38	1106	3.89	375	179	30.4
dual_ehi-90-315-26	4370	108696	37	1084	3,9	304	160	36,0
dual_ehi-90-315-27	4374	108702	32	1001	3,16	286	130	27,3
dual_ehi-90-315-28	4369	108612	40	1217	4,88	362	186	37,7
scen6_w2	648	513100	7	8020	53.8	4872	2953	1107
scen6_w1_f2	319	274860	21	21146	489	-	-	*time out*
scen11_f10	4103	738719	16	4588	164	3077	2728	438
scen11_f12	4103	707375	16	4588	122	3053	2728	42

TAB. 8.1 – Extraction d'un MUST par `MUSTER`

Conclusion générale

Ce manuscrit rapporte nos différentes contributions aux techniques algorithmiques permettant l'extraction d'ensemble minimalement insatisfiables de contraintes.

La première étape de notre travail était destinée à la conception de méthodes de calcul d'une formule minimalement inconsistante (MUS) en logique propositionnelle clausale. Celle-ci a tout d'abord conduit au développement d'une technique originale basée sur la recherche locale, qui ne considère pas seulement l'interprétation courante, mais également un voisinage partiel pertinent à travers le concept original de clause critique. Ensuite, notre intérêt s'est porté sur la conception d'une approche constructive pour l'approximation d'un MUS. Nous avons établi une propriété intéressante sur cette technique de construction qui permet de s'assurer que celle-ci ne peut ajouter de contraintes redondantes avec l'approximation de MUS en construction. Notons qu'elle est la seule approche complète à ce jour à ne pas nécessiter une preuve de l'insatisfiabilité de la formule complète pour en extraire un tel ensemble. En pratique, `AOMUS` et `construct_core` se montrent extrêmement compétitifs et se classent parmi les meilleures approches proposées à ce jour.

Bien que l'extraction d'un MUS soit un problème essentiel pour la compréhension de l'insatisfiabilité d'une formule CNF, celle-ci peut être insuffisante dans certains cas. En effet, une réparation peut requérir la connaissance de toutes les sources d'erreur du système modélisé. Dans cet objectif, plusieurs approches ont par le passé été proposées, la meilleure d'entre elles ayant été introduite par Mark Liffiton et Karem Sakallah. En hybridant leur algorithme avec une approche incomplète de recherche locale, nous sommes parvenus à accélérer son traitement pratique d'un ordre de grandeur. En effet, on peut déduire de la trace de cette dernière certaines sous-formules maximalement satisfiables (MSS), nécessaires à l'obtention de l'ensemble exhaustif des MUS d'une formule. Ainsi, cette information obtenue par des opérations peu coûteuses en ressources permet à l'algorithme original d'éviter des tests NP-complets et CoNP-complets. Malheureusement, le nombre potentiellement exponentiel de MUS d'une formule peut rendre leur calcul impossible en pratique, même avec cette amélioration. Pour pallier cette éventuelle explosion combinatoire, le concept de couverture inconsistante à été proposé dans ce document. Celui-ci permet d'énumérer un sous-ensemble des sources d'inconsistance du problème prouvé suffisant pour expliquer et potentiellement réparer l'ensemble du problème.

Nos investigations se sont ensuite portées sur le problème de satisfaction de contraintes (CSP). En premier lieu, nous avons étudié et amélioré la meilleure approche connue permettant la localisation d'un ensemble minimal de contraintes, appelé MUC dans ce cadre. De plus, nous avons proposé par la suite un nouveau concept permettant de raffiner celui du MUC, en tenant compte des tuples de valeurs interdits entre variables, induits par les contraintes du problème. Le concept de MUST ainsi défini, nous avons montré ses liaisons fortes avec les MUC, et prouvé que tout MUST contenu dans un MUC comprend des tuples tel que le retrait, ou autorisation, de l'un d'entre eux permet de réparer localement le problème. Ainsi, ses tuples, qualifiés de *partagés*, permettent une forme de restauration de la cohérence sans retrait de contraintes. D'un point

de vue technique, le problème est transposé en logique propositionnelle afin que le calcul d'un MUST soit effectué à travers celui d'un MUS. Sa faisabilité pratique est exposée à travers des résultats expérimentaux concluants.

Ces différentes contributions ouvrent de nombreuses perspectives. Une grande partie des techniques mises au point dans cette thèse utilisent de nouvelles heuristiques dédiées à l'extraction d'ensembles de contraintes minimalement inconsistants. Si les notions mises en jeu ont été étudiées précisément, certaines d'entre elles peuvent sans doute être étendues ou généralisées. Par exemple, le concept de clause critique permet de prendre en considération un voisinage partiel des interprétations parcourues par une recherche locale. Il peut sembler intéressant d'étudier l'utilité d'un voisinage plus grand pour la capture de MUS. L'intégration de ce concept à la méthode constructive développée, moins évidente de part la nature intrinsèque de l'approche, est clairement une piste à explorer. Il semble également naturel de tenter d'élaborer une hybridation entre ces deux nouvelles approches basées sur la recherche locale. Cette combinaison fera l'objet de futures recherches.

Une autre voie de recherche offerte par ces travaux concerne l'adaptation de ces techniques algorithmiques à d'autres champs de recherche, tels que la recherche opérationnelle (RO). Il semble en effet tout-à-fait réalisable d'exporter les algorithmes présentés vers d'autres domaines, qui pourraient accueillir favorablement ces méthodes efficaces dans leur formalisme.

Bien évidemment, la mise en application de l'ensemble des techniques algorithmiques présentées dans ce document est un élément à ne pas négliger. Étant donnés, d'une part les résultats extrêmement positifs qui ont été obtenus, et d'autre part l'étendue des possibilités offertes par la localisation et la réparation de problèmes inconsistants, dériver les algorithmes obtenus ne paraît pas une gageure. Si des applications particulièrement intéressantes au concept de MUS ont déjà été étudiées, notamment dans le domaine du *model-checking*, ses perspectives applicatives apparaissent encore très vastes. Par exemple, la gestion de l'incohérence d'une base de connaissance propositionnelle est un problème très étudié par les chercheurs en intelligence artificielle ; il s'agit en effet d'un domaine essentiel, mais dont la complexité théorique dans le pire cas est extrêmement élevée, même lorsque des ingrédients épistémologiques tels que la stratification sont ajoutés au cadre logique. Nous avons proposé dans ce document une première approche, dérivée de l'un de nos algorithmes, afin de traiter ce problème en pratique. Le concept de MUS peut, dans ce contexte, s'avérer extrêmement utile, et grâce aux nouveaux algorithmes permettant leur extraction de manière souvent efficace, des solutions originales à ces problèmes pourraient être imaginées. Par exemple, la recherche d'une couverture inconsistante peut permettre de déterminer des ensembles minimaux de connaissances de la base ne pouvant cohabiter. Cette information donne donc à l'expert des choix de réparation *a minima* de la base de connaissance. De tels ensembles de clauses peuvent évidemment s'avérer utiles pour diverses opérations définies dans cette logique, telles que la fusion ou la révision d'informations. Des travaux futurs seront conduits dans ce sens.

Bibliographie

[Aharoni & Linial 1986] Ron Aharoni et Nathan Linial. « Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas ». *Journal of Combinatorial Theory Series A*, 43(2) :196–204, 1986.

[Akers 1978] Sheldon B. Akers. « Binary Decision Diagrams ». Dans *IEEE Transactions on Computers*, pages 509–516, 1978.

[Andraus *et al.* 2006] Zaher S. Andraus, Mark H. Liffiton, et Karem A. Sakallah. « Refinement strategies for verification methods based on datapath abstraction ». Dans *Proceedings of Asia South Pacific Design Automation Conference (ASP-DAC'06)*, pages 19–24, Yokohama (Japon), 2006. ACM Press.

[Bacchus & Winter 2003] Fahiem Bacchus et Jonathan Winter. « Effective preprocessing with hyper-resolution and equality reduction ». Dans *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, Portofino (Italie), 2003.

[Bailey & Stuckey 2005] James Bailey et Peter J. Stuckey. « Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization ». Dans *International Symposium on Practical Aspects of Declarative Languages (PADL'05)*, pages 174–186, Long Beach (États-Unis), 2005.

[Bakker *et al.* 1993] R. R. Bakker, F. Dikker, F. Tempelman, et P. M. Wognum. « Diagnosing and Solving Over-Determined Constraint Satisfaction Problems ». Dans *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, volume 1, pages 276–281, Chambéry (France), September 1993. Morgan Kaufmann, 1993.

[Baptista & Marques-Silva 2000] Luís Baptista et João P. Marques-Silva. « Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability ». Dans *Proceedings of Principles and Practice of Constraint Programming (CP'00)*, pages 489–494, 2000.

[Barták] Roman Barták. « On-line Guide to Constraint Programming ». http ://ktiml.mff.cuni.cz/ bartak/constraints/index.html.

[Beame *et al.* 2004] Paul Beame, Henry A. Kautz, et Ashish Sabharwal. « Towards Understanding and Harnessing the Potential of Clause Learning ». *Journal of Artificial Intelligence Research (JAIR)*, 22 :319–351, 2004.

[Benferhat *et al.* 1993] Salem Benferhat, Claudette Cayrol, Didier Dubois, Jérôme Lang, et Henri Prade. « Inconsistency management and prioritized syntax-based entailment ». Dans *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 640–645, Chambéry (France), août 1993.

[Bessière *et al.* 1995] Christian Bessière, Eugene C. Freuder, et Jean-Charles Régin. « Using Inference to Reduce Arc Consistency Computation ». Dans *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 592–598, Montréal (Canada), 1995.

[Bessière & Régin 1996] Christian Bessière et Jean-Charles Régin. « MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems ». Dans *Principles and Practice of Constraint Programming (CP'96)*, pages 61–75, 1996.

[Bessière 1994] Christian Bessière. « Arc-consistency and arc-consistency again ». *Artificial Intelligence*, 65(1) :179–190, 1994.

[Büning 2000] H.K. Büning. « On subclasses of minimal unsatisfiable formulas ». *Discrete Applied Mathematics*, 107(1–3) :83–98, 2000.

[Boussemart *et al.* 2004] Frédéric Boussemart, Frédéric Hemery, Christophe Lecoutre, et Lakhdar Sais. « Boosting systematic search by weighting constraints ». Dans *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 482–486, Valence (Espagne), Août 2004.

[Brafman 2001] Ronen I. Brafman. « A Simplifier for Propositional Formulas with Many Binary Clauses ». Dans *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 515–522, Seattle (États-Unis), 2001.

[Bruni 2003] Renato Bruni. « Approximating minimal unsatisfiable subformulae by means of adaptive core search ». *Discrete Applied Mathematics*, 130(2) :85–100, 2003.

[Bruni 2005] Renato Bruni. « On exact selection of minimally unsatisfiable subformulae. ». *Annals of mathematics and artificial intelligence*, 43(1) :35–50, 2005.

[Cayrol *et al.* 1998] Claudette Cayrol, Marie-Christine Lagasquie-Schiex, et Thomas Schiex. « Nonmonotonic reasoning : from complexity to algorithms ». *Annals of Mathematics and Artificial Intelligence*, 22(3–4) :207–236, 1998.

[Cha *et al.* 1997] Byungkiand Cha, Kazuo Iwama, Yahiko Kambayashi, et Shuichi Miyazaki. « Local Search Algorithms for Partial maxSat ». Dans *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 263–268, 1997.

[Chinneck 1997] John W. Chinneck. « *Feasibility and Viability, In : Advances in Sensitivity Analysis and Parametric Programming* », volume 6, Chapitre 14, pages 14.2–14.41. Kluwer Academic Publishers, Boston (États-Unis), 1997.

[Chvátal & Szemerédi 1988] Vašek Chvátal et Endre Szemerédi. « Many hard examples for resolution ». *Journal of the Association for Computing Machinery (JACM)*, 35(4) :759–768, 1988.

[Clarke *et al.* 2003] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Joel Ouaknine, Olaf Stursberg, et Michael Theobald. « Abstraction and counterexample-guided refinement in model checking of hybrid systems ». *International Journal of Foundations of Computer Science*, 14(4), 2003.

[Compétitions CSP] CSP competitions, http ://cpai.ucc.ie/06/Competition.html.

[Compétitions SAT] SAT competitions, http ://www.satcompetition.org.

[Cook 1971] Stephen A. Cook. « The complexity of theorem-proving procedures ». Dans *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York (États-Unis), 1971. Association for Computing Machinery.

[Crawford 1993] James M. Crawford. « Solving satisfiability problems using a combination of systematic and local search ». Dans *Working notes of the DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*, 1993.

[Davis & Putnam 1960] Martin Davis et Hilary Putnam. « A computing procedure for quantification theory ». *Journal of the ACM*, 7(3) :201–215, Juillet 1960.

[Davis *et al.* 1962] Martin Davis, George Logemann, et Donald Loveland. « A machine program for theorem proving ». *Communications of the ACM*, 5(7) :394–397, Juillet 1962.

[Davydov *et al.* 1998] G. Davydov, I. Davydova, et H.K. Büning. « An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF ». *Annals of Mathematics and Artificial Intelligence*, 23(3–4) :229–245, 1998.

[de Jong & Spears 1989] Kenneth A. de Jong et William M. Spears. « Using genetic algorithms to solve NP-complete problems ». Dans *Proceedings of the third international conference on Genetic algorithms*, pages 124–132, San Francisco (États-Unis), 1989. Morgan Kaufmann Publishers Inc.

[de Kleer & Williams 1987] Johan de Kleer et Brian C. Williams. « Diagnosing Multiple Faults ». *Artificial Intelligence*, 32(1) :97–130, 1987.

[de Kleer 1986] Johan de Kleer. « An Assumption-based Truth Maintenance System ». *Artificial Intelligence*, 28 :127–162, 1986.

[de Kleer 1989] Johan de Kleer. « A Comparison of ATMS and CSP Techniques. ». Dans *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 290–296, Detroit (États-Unis), 1989.

[de la Banda *et al.* 2003] Maria Garcia de la Banda, Peter J. Stuckey, et Jeremy Wazny. « Finding all Minimal Unsatisfiable Subsets ». Dans *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles*

and Practice of Declarative Programming (PPDL'03), pages 32–43, 2003.

[de Siqueira & Puget 1988] N. de Siqueira et Jean-François Puget. « Explanation-Based Generalisation of Failures ». Dans *Proceedings of the Eighth European Conference on Artificial Intelligence (ECAI'88)*, pages 339–344, 1988.

[Dechter & Meiri 1989] Rina Dechter et Itay Meiri. « Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems ». Dans *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 271–277, 1989.

[Dechter 1990] Rina Dechter. « On the Expressiveness of Networks with Hidden Variables. ». Dans *Proceedings of the Eighth National Conference on Artifial Intelligence (AAAI'90)*, pages 556–562, 1990.

[Dechter *et al.* 1989] Rina Dechter, , et Judea Pearl. « Tree-clustering Schemes for Constraint Networks ». *Artificial Intelligence*, 38 :353–366, 1989.

[Dequen & Dubois 2001] Gilles Dequen et Olivier Dubois. « A backbone search heuristic for efficient solving of hard 3-SAT formulae ». Dans *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 248–253, Washington (États-Unis), 2001.

[Eén & Biere 2005] Niklas Eén et Armin Biere. « Effective Preprocessing in SAT Through Variable and Clause Elimination ». Dans *Proceedings of The Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 61–75, St. Andrews (Écosse), 2005.

[Eiter & Gottlob 1992] Thomas Eiter et Georg Gottlob. « On the complexity of propositional knowledge base revision, updates and counterfactual ». *Artificial Intelligence*, 57 :227–270, 1992.

[Eén & Sörensson] Niklas Eén et Niklas Sörensson. « MiniSat home page ». http ://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat.

[Fleischner *et al.* 2002] Herbert Fleischner, Oliver Kullman, et Stefan Szeider. « Polynomial-Time Recognition of Minimal Unsatisfiable Formulas with Fixed Clause-Variable Difference ». *Theoretical Computer Science*, 289(1) :503–516, 2002.

[Freeman 1995] Jon W. Freeman. « *Improvements to Propositional Satisfiability Search Algorithms* ». Thèse d'université, Université de Pennsylvanie, 1995.

[Freuder 1982] Eugene C. Freuder. « A Sufficient Condition for Backtrack-Free Search ». *Journal of the ACM*, 29(1) :24–32, 1982.

[Gershman *et al.* 2006] Roman Gershman, Maya Koifman, et Ofer Strichman. « Deriving Small Unsatisfiable Cores with Dominators ». Dans *Proceedings of Computer-Aided Verification*, pages 109–122, Seatle (États-Unis), 2006.

[Ginsberg 1993] Matthew L. Ginsberg. « Dynamic Backtracking ». *Journal of Artificial Intelligence Research (JAIR)*, 1 :25–46, 1993.

[Goldberg & Novikov 2002] Eugene P. Goldberg et Yakov Novikov. « BerkMin : a Fast and Robust SAT-Solver ». Dans *In Proceedings of Design Automation and Test in Europe (DATE'02)*, pages 142–149, Paris, 2002.

[Goldberg 1979] A. Goldberg. « On the Complexity of the Satisfiability Problem ». Rapport Technique, New York University, 1979.

[Gomes *et al.* 1998] Carla P. Gomes, Bart Selman, et Henry Kautz. « Boosting Combinatorial Search through Randomization ». Dans *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98)*, pages 431–437, Menlo Park, jul 1998. AAAI Press.

[Grégoire 1999] Éric Grégoire. « Handling Inconsistency Efficiently in the Incremental Construction of Statified Belief Bases. ». Dans *Proceedings of the Fifth European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU-99)*, pages 168–178, 1999.

[Grégoire *et al.* 2002] Éric Grégoire, Bertrand Mazure, et Lakhdar Saïs. « Using Failed Local Search for SAT as an Oracle for Tackling Harder A.I. Problems More Efficiently. ». Dans *Proceedings of the Tenth International Conference on Artificial Intelligence : Methodology, Systems, Applications (AIMSA'2002)*, pages 51–60, Varna (Bulgarie), Septembre 2002.

[Grégoire *et al.* 2005] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « A new local search method to compute inconsistent kernel ». Dans *Proceedings of the Sixth Metaheuristics International Conference (MIC'2005)*, Vienne (Autriche), Août 2005. Actes électroniques.

[Grégoire *et al.* 2006a] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Extracting MUSes ». Dans *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 387–391, Trento (Italie), Août 2006.

[Grégoire *et al.* 2006b] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Extraction de sous-formules minimales inconsistantes ». Dans *Deuxièmes Journées Francophones de Programmation par Contraintes (JFPC'06)*, pages 201–208, Nîmes (France), Juin 2006.

[Grégoire *et al.* 2006c] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Tracking MUSes and Strict Inconsistent Covers ». Dans *Proceedings of the 6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 39–46, San Jose (États-Unis), Novembre 2006.

[Grégoire *et al.* 2006d] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Une méta-heuristique basée sur le comptage de contraintes falsifiées ». Dans *First Workshop on Metaheuristics (META'06)*, Hammamet (Tunisie), Novembre 2006. Actes électroniques.

[Grégoire *et al.* 2006e] Éric Grégoire, Bertrand Mazure, Cédric Piette, et Lakdhar Saïs. « A New Heuristic-based albeit Complete Method to Extract MUCs from Unsatisfiable CSPs ». Dans *Proceedings of the*

IEEE International Conference on Information Reuse and Integration (IEEE-IRI'06), pages 325–329, Waikoloa (États-Unis), Septembre 2006.

[Grégoire *et al.* 2007a] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Boosting a Complete Technique to Find MSS and MUS thanks to a Local Search Oracle ». Dans *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 2, pages 2300–2305, Hyderabad (Inde), Janvier 2007. AAAI Press.

[Grégoire *et al.* 2007b] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Extraction d'ensembles minimaux conflictuels basée sur la recherche locale ». Dans *Revue d'Intelligence Artificielle (RSTI-RIA)*, 2007. (accepté).

[Grégoire *et al.* 2007c] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Local-Search Extraction of MUSes ». *Constraints Journal, Special issue on Local Search in Constraint Satisfaction*, 12(3) :325–344, 2007. Yehuda Naveh & Andrea Roli (éditeurs invités).

[Grégoire *et al.* 2007d] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « MUST : Provide a Finer-Grained Explanation of Unsatisfiability ». Dans *Proceedings of the the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 317–331, Providence (États-Unis), Septembre 2007. LNCS.

[Grégoire *et al.* 2007e] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Une nouvelle methode hybride pour calculer tous les MSS et tous les MUS ». Dans *Troisièmes Journées Francophones de Programmation par Contraintes (JFPC'07)*, Rocquencourt (France), Juin 2007.

[Grégoire *et al.* 2007f] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Using Local Search to find MSSes and MUSes ». Dans *European Journal of Operational Research (EJOR)*, 2007. (accepté).

[Grégoire *et al.* 2008a] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Chapitre 8 : Sous-formules minimales insatisfaisables ». Dans Lakhdar Saïs, éditeur, *Problème SAT – progrès et défis*, 2008. (À paraître).

[Grégoire *et al.* 2008b] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Explication et réparation de l'incohérence dans les CSP : de la contrainte au tuple ». Dans *16e congrès francophone AFRIF-AFIA Reconnaissance des Formes et Intelligence Artificielle (RFIA'08)*, 2008. (accepté).

[Grégoire *et al.*] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « On Finding Minimally Unsatisfiable Cores of CSPs ». Soumis à *International Journal on Artificial Intelligence Tools (IJAIT)*.

[Gunopulos *et al.* 2003] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, et Ram Sharma. « Discovering all most specific sentences ». *ACM Transactions on Database Systems*, 28(2) :140–174, 2003.

[Hamscher *et al.* 1992] Walter Hamscher, Luca Console, et Johan de Kleer, éditeurs. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[Han & Lee 1999] Benjamin Han et Shie-Jue Lee. « Deriving minimal conflict sets by CS-Trees with mark set in diagnosis from first principles ». Dans *IEEE Transactions on Systems, Man, and Cybernetics*, volume 29, pages 281–286, 1999.

[Hao & Dorne 1994] Jin-Kao Hao et Raphaël Dorne. « An Empirical Comparison of Two Evolutionary Methods for Satisfiability Problems ». Dans *International Conference on Evolutionary Computation (ICEC'94)*, pages 451–455, 1994.

[Haralick & Elliott 1980] R.M. Haralick et G.L. Elliott. « Increasing Tree Search Efficiency for Constraint Satisfaction Problems ». *Artificial Intelligence*, 14 :263–313, 1980.

[Hemery *et al.* 2006] Frédéric Hemery, Christophe Lecoutre, Lakhdar Saïs, et Frédéric Boussemart. « Extracting MUCs from Constraint Networks ». Dans *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 113–117, Trento (Italie), 2006.

[Huang 2005] J. Huang. « MUP : A Minimal Unsatisfiability Prover ». Dans *Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, pages 432–437, Shanghai (Chine), 2005.

[Hutter *et al.* 2002] Frank Hutter, Dave A. D. Tompkins, et Holger H. Hoos. « Scaling and Probabilistic Smoothing : Efficient Dynamic Local Search for SAT ». Dans *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 233–248, 2002.

[Jeroslow & Wang 1990] Robert G. Jeroslow et Jinchang Wang. « Solving Propositional Satisfiability Problems ». *Annals of mathematics and artificial intelligence*, 1 :167–187, 1990.

[Junker 2001] Ulrich Junker. « QuickXplain : Conflict Detection for Arbitrary Constraint Propagation Algorithms ». Dans *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle (États-Unis), Août 2001.

[Junker 2004] Ulrich Junker. « QuickXplain : Preferred Explanations and Relaxations for Over-Constrained Problems ». Dans *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, 2004.

[Jussien & Barichard 2000] Narendra Jussien et Vincent Barichard. « The PaLM system : explanation-based constraint programming ». Dans *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, pages 118–133, Singapour, 2000.

[Jussien *et al.* 2000] Narendra Jussien, Romuald Debruyne, et Patrice Boizumault. « Maintaining Arc-Consistency within Dynamic Backtracking ». Dans *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 249–261, 2000.

[Kautz *et al.* 2004] Henry KAUTZ, Bart SELMAN, et David MCALLESTER. « Walksat in the 2004 SAT Competition ». Dans *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver (Canada), 2004.

[Kirkpatrick *et al.* 1983] Scott KIRKPATRICK, Charles D. GELATT, et Mario P. VECCHI. « Optimization by Simulated Annealing ». *Science*, 220(4598) :671–680, 1983.

[Kullmann *et al.* 2006] Oliver KULLMANN, Ines LYNCE, et Joao P. MARQUES SILVA. « Categorisation of Clauses in Conjunctive Normal Forms : Minimally Unsatisfiable Sub-clause-sets and the Lean Kernel. ». Dans *International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 22–35, Seatle (États-Unis), 2006.

[Laburthe 2000] François LABURTHE et l'équipe du projet OCRE. « CHOCO : implementing a CP kernel ». Dans *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, Singapour, 2000. http ://www.choco-constraints.net.

[Lehmann 1995] Daniel J. LEHMANN. « Another Perspective on Default Reasoning ». *Annals of Mathematics and Artificial Intelligence*, 15(1) :61–82, 1995.

[Li & Anbulagan 1997] Chu Min LI et ANBULAGAN. « Heuristics Based on Unit Propagation for Satisfiability Problems ». Dans *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Nagoya (Japon), 1997.

[Liffiton & Sakallah 2005] Mark H. LIFFITON et Karem A. SAKALLAH. « On Finding All Minimally Unsatisfiable Subformulas ». Dans *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 173–186, St. Andrews (Écosse), jun 2005.

[Lynce & Marques-Silva 2004] Ines LYNCE et João MARQUES-SILVA. « On computing minimum unsatisfiable cores ». Dans *International conference on theory and applications of satisfiability testing (SAT04)*, Vancouver (Canada), 2004.

[Marques-Silva & Sakallah 1996] Joao P. MARQUES-SILVA et Karem A. SAKALLAH. « GRASP : A New Search Algorithm for Satisfiability ». Dans *In Proceedings of International Conference on Computer-Aided Design (CAD'96)*, pages 220–227, Santa Clara (États-Unis), 1996.

[Mauss & Tatar 2002] Jakob MAUSS et Mugur TATAR. « Computing Minimal Conflicts for Rich Constraint Languages ». Dans *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02)*, pages 151–155, Lyon (France), 2002.

[Mazure *et al.* 1997] Bertrand MAZURE, Lakhdar SAÏS, et Éric GRÉGOIRE. « Tabu search for SAT ». Dans *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 281–285, Providence (États-Unis), 1997.

[Mazure *et al.* 1998] Bertrand Mazure, Lakhdar Saïs, et Éric Grégoire. « Boosting complete techniques thanks to local search methods ». *Annals of mathematics and artificial intelligence*, 22 :319–331, 1998.

[McAllester *et al.* 1997] David McAllester, Bart Selman, et Henry Kautz. « Evidence for Invariants in Local Search ». Dans *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence (États-Unis), 1997.

[McCarthy 1986] John McCarthy. « Applications of Circumscription to Formalizing Common Sense Knowledge ». *Artificial Intelligence*, 28 :89–116, 1986.

[McMillan 2005] Kenneth L. McMillan. « Applications of Craig Interpolants in Model Checking ». Dans *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 1–12, 2005.

[Mneimneh *et al.* 2005] Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João P. Marques Silva, et Karem A. Sakallah. « A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. ». Dans *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 467–474, St. Andrews (Écosse), jun 2005.

[Mohr & Henderson 1986] Roger Mohr et Thomas C. Henderson. « Arc and Path Consistency Revisited ». *Artificial Intelligence*, 28 :225–233, 1986.

[Morris 1993] Paul Morris. « The Breakout method for escaping local minima ». Dans *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 40–45, Washington D.C. (États-Unis), 1993.

[Moskewicz *et al.* 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, et Sharad Malik. « Chaff : Engineering an Efficient SAT Solver ». Dans *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, Las Vegas (États-Unis), juin 2001.

[Mézard & Zecchina 2002] Marc Mézard et Riccardo Zecchina. « The random K-satisfiability problem : from an analytic solution to an efficient algorithm ». *Physical Review*, 66 :56–126, Novembre 2002.

[Mézard *et al.* 2002] Marc Mézard, Parisi, et Riccardo Zecchina. « Analytic and Algorithmic Solution of Random Satisfiability Problems ». *Science*, 297 :812–815, 2002.

[Nam *et al.* 1999] Gi-Joon Nam, Karem A. Sakallah, et Rob A. Rutenbar. « Satisfiability-Based Layout Revisited : Detailed Routing of Complex FPGAs vis Search-Based Boolean SAT. ». Dans *Proceedings of International Symposium on FPGAs*, pages 167–175, 1999.

[Nam *et al.* 2004] Gi-Joon Nam, Fadi A. Aloul, Karem A. Sakallah, et Rob A. Rutenbar. « A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints. ». *IEEE Trans. Computers*, 53(6) :688–696, 2004.

[Oh *et al.* 2004] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, et Igor L. Markov. « AMUSE : a minimally-unsatisfiable subformula extractor ». Dans *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 518–523, New York (États-Unis), 2004. ACM Press.

[Papadimitriou & Wolfe 1988] Christos H. Papadimitriou et David Wolfe. « The complexity of facets resolved ». *Journal of Computer and System Sciences*, 37(1) :2–13, 1988.

[Prosser 1993] Patrick Prosser. « Hybrid algorithms for the constraint satisfaction problems ». Dans *Computational Intelligence*, volume 9(3), pages 268–299, 1993.

[SATLIB] Benchmarks on SAT, http ://www.satlib.org.

[Schiex & Verfaillie 1994] Thomas Schiex et Gérard Verfaillie. « Stubbornness : an enhancement scheme for Backjumping and Nogood Recording ». Dans *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'04)*, pages 165–169, Amsterdam (Pays-Bas), 1994.

[Schiex *et al.* 1996] Thomas Schiex, Jean-Charles Regin, Christine Gaspin, et Gérard Verfaillie. « Lazy arc consistency ». Dans *Proceedings of 13th National Conference on Artificial Intelligence AAAI'96*, pages 216–221. AAAI Press / The MIT Press, 1996.

[Schuurmans *et al.* 2001] Dale Schuurmans, Finnegan Southey, et Robert C. Holte. « The Exponentiated Subgradient Algorithm for Heuristic Boolean Programming ». Dans *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 334–341, 2001.

[Selman *et al.* 1993] Bart Selman, Henry A. Kautz, et Bram Cohen. « Local search strategies for satisfiability testing ». Dans *Proceedings of DIMACS Workshop on Maximum clique, Graph coloring, and Satisfiability*, pages 521–532, 1993.

[Smith & Grant 1998] Barbara M. Smith et Stuart A. Grant. « Trying Harder to Fail First ». Dans *Proceedings of Thirteenth European Conference on Artificial Intelligence (ECAI'98)*, pages 249–253, Brighton (Royaume-uni), 1998.

[Smullyan 1968] Raymond M. Smullyan. *First Order Logic*. Springer, Berlin (Allemagne), 1968.

[Spears 1996] William M. Spears. « Simulated Annealing for hard satisfiability problems ». Dans *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 533–558, 1996.

[Subbarayan & Pradhan 2004] S. Subbarayan et Dhiraj Pradhan. « NiVER : Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances ». Dans *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing Conference (SAT'04)*, pages 276–291, Mai 2004.

[Tompkins & Hoos 2005] Dave A. D. Tompkins et Holger H. Hoos. « UBCSAT : An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT ». Dans Holger H. Hoos et David G. Mitchell, éditeurs, *Theory and Applications of Satisfiability Testing : Revised Selected Papers of the Seventh International Conference (SAT'04)*, volume 3542 de *Lecture Notes in Computer Science*, pages 306–320, Berlin (Allemagne), 2005. Springer Verlag.

[Turing 1937] Alan Turing. « On computable numbers, with an application to the Entscheidungsproblem ». *Proceedings of the London Mathematical Society*, 42 :230–265, 1937.

[Van Hentenryck *et al.* 1992] Pascal Van Hentenryck, Yves Deville, et Choh-Man Teng. « A Generic Arc-Consistency Algorithm and its Specializations ». *Artificial Intelligence*, 57(2–3) :291–321, 1992.

[Verfaillie & Lobjois 1999] Gérard Verfaillie et Lionel Lobjois. « Problèmes incohérents : Expliquer l'incohrence, restaurer la cohérence ». Dans *Actes des 5èmes Journées Nationales de la Résolution Pratique des Problèmes NP-Complets (JNPC'99)*, pages 111–120, Lyon (France), 1999.

[Walsh 2000] Toby Walsh. « SAT v CSP ». Dans *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 441–456, Londres (Royaume-Uni), 2000.

[Young & Reel 1990] R. A. Young et A. Reel. « A Hybrid Genetic Algorithm for a Logic Problem ». Dans *9th European Conference on Artificial Intelligence (ECAI'90)*, pages 744–746, 1990.

[Zhang & Malik 2002] Lintao Zhang et Sharad Malik. « The Quest for Efficient Boolean Satisfiability Solvers. ». Dans *Proceedings of the Eighteenth International Conference on Automated Deduction (CADE'02)*, pages 295–313, 2002.

[Zhang & Malik 2003] Lintao Zhang et Sharad Malik. « Extracting small unsatisfiable cores from unsatisfiable boolean formula ». Dans *Sixth international conference on theory and applications of satisfiability testing (SAT'03)*, Portofino (Italie), 2003.

[Zhang 2005] Lintao Zhang. « On subsumption removal and on-the-fly CNF simplification ». Dans *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 482–489, 2005.