Explication et réparation de l'incohérence dans les CSP : de la contrainte au tuple

Explanation and reparation of inconsistency in CSP: from constraints to tuples

Éric Grégoire Bertrand Mazure Cédric Piette

CRIL-CNRS
Université d'Artois
rue Jean Souvraz SP18
F-62307 Lens Cedex France
{gregoire,mazure,piette}@cril.fr

Résumé

Dans ce papier, nous proposons une nouvelle forme d'explication et de correction de l'incohérence d'un CSP. Alors que la plupart des approches fournissent à l'utilisateur un ensemble minimal de contraintes (MUC) à retirer pour restaurer la satisfaisabilité, nous proposons une alternative à grains plus fins. Elle permet à l'utilisateur de raisonner à la fois au niveau des contraintes et au niveau des tuples de valeurs interdites induits par ces contraintes. A cet effet, nous introduisons le concept d'ensemble minimal de tuples insatisfaisable (MUST : Minimal Set of Unsatisfiable Tuples). Des relations entre les MUST et les MUC (Minimal Unsatisfiable Core), nous dérivons le concept de tuples interdits partagés qui, lorsqu'ils sont retirés d'un MUC, lui permettent de devenir satisfiable. D'un point de vue pratique, nous proposons une approche en deux étapes. En premier lieu, nous utilisons l'approche récente proposée par Hemery et al. pour localiser un MUC à partir d'un CSP. Dans un second temps, un encodage spécifique vers SAT permet de transformer ce MUC en une formule CNF pour laquelle nous tirons avantage de la meilleure technique actuelle pour extraire un MUS (Miminal Unsatisfiable Sub-formula) à partir duquel il est dérivé un MUST du CSP initial. Il est intéressant de constater que le concept de tuples interdits partagés coïncide avec celui de clauses protégées, introduit récemment dans le cadre de SAT, et sur lequel réside en partie l'efficacité de la technique d'extraction de MUS. Pour conclure, la faisabilité et la viabilité de l'approche proposée sont illustrées à travers de nombreux résultats expérimentaux.

Mots Clef

CSP, MUC, MUST, explication de l'insatisfaisabilité

Abstract

In this paper, a new form of explanation and recovery technique for the unsatisfiability of discrete CSPs is introduced. Whereas most approaches amount to providing users with a minimal number of constraints that should be dropped in order to recover satisfiability, a finer-grained alternative technique is introduced. It allows the user to reason both at the constraints and tuples levels by exhibiting both problematic constraints and tuples of values that would allow satisfiability to be recovered if they were not forbidden. To this end, the Minimal Set of Unsatisfiable Tuples (MUST) concept is introduced. Its formal relationships with Minimal Unsatisfiable Cores (MUCs) are investigated. Interestingly, a concept of shared forbidden tuples is derived. Allowing any such tuple makes the corresponding MUC become satisfiable. From a practical point of view, a two-step approach to the explanation and recovery of unsatisfiable CSPs is proposed. First, a recent approach proposed by Hemery et al.'s is used to locate a MUC. Second, a specific SAT encoding of a MUC allows MUSTs to be computed by taking advantage of the best current technique to locate Minimally Unsatisfiable Sub-formulas (MUSes) of Boolean formulas. Interestingly enough, shared tuples coincide with protected clauses, which are one of the keys to the efficiency of this SAT-related technique. Finally, the feasibility of the approach is illustrated through extensive experimental results.

Keywords

CSP, MUC, MUST, explanation of unsatisfiability

1 Introduction

Dans cet article, nous nous intéressons aux CSP insatisfiables, c'est-à-dire aux problèmes de satisfaction de contraintes discrets finis pour lesquels il n'existe au-

cune solution. Récemment, plusieurs approches permettant d'expliquer l'insatisfiabilité d'un problème au niveau des contraintes ont été proposées. Par exemple, Hemery *et al.* [14] ont présenté une procédure nommée `DC(wcore)` pour la détection de MUC (Minimally Unsatisfiable Cores) d'un CSP, qui sont des sous-ensembles de contraintes tels que supprimer n'importe laquelle d'entre elles rend cet ensemble satisfiable. Cependant, le retrait d'une contrainte peut être très destructeur. Dans ce papier, une technique alternative permettant à l'utilisateur de réparer le système sans suppression de contraintes est décrite. Celle-ci consiste à indiquer les causes de l'incohérence du CSP non seulement au niveau de ses contraintes, mais également au niveau des tuples interdits de valeurs induits par celles-ci.
Dans cet objectif, la contribution de ce papier est double. D'une part, le concept d'*ensemble minimalement insatisfiable de tuples* (MUST) est introduit : il vise à indiquer les tuples interdits conflictuels rendant l'ensemble du problème sans solution, et fournissant une *explication* de l'insatisfiabilité. En outre, les relations formelles entre MUC et MUST sont étudiées, et le concept de tuples partagés en est dérivé. Autoriser n'importe lequel de ces tuples partagés rend le MUC correspondant satisfiable. D'autre part, une approche en deux étapes pour l'explication et la réparation de CSP insatisfiables est proposée. Un encodage spécifique vers SAT permet de tirer parti de l'une des meilleures techniques actuelles pour le calcul de formules minimalement insatisfiables (MUS) d'une formule booléenne. De plus, les tuples partagés coïncident avec les clauses protégées [10], qui sont l'un des éléments clés de l'efficacité de cette méthode.
Le papier est organisé comme suit. Dans un premier temps, les définitions basiques de CSP et de MUC sont fournies. Dans la section 3, les MUST sont introduits et leurs relations avec les MUC sont présentées en section 4. Ensuite, une approche originale en 2 étapes permettant l'explication et la réparation de CSP insatisfiables est décrite. La viabilité pratique de cette méthode est illustrée à travers des expérimentations intensives présentées dans la section 6. La section 7 compare les contributions de ce papier avec les travaux existants dans ce domaine. Enfin, nous concluons par des perspectives intéressantes de futures recherches.

2 CSP et MUC

Dans cette section, les concepts basiques de CSP et MUC sont définis.

Définition 1
Un problème de satisfaction de contraintes (CSP) est un couple $P = \langle V,C \rangle$ où :

1. *V est un ensemble fini de n variables $\{v_1, \dots, v_n\}$ tel que chaque variable $v_i \in V$ possède un domaine d'instanciation fini noté $dom(v_i)$, qui contient l'ensemble des valeurs possibles de v_i ;*
2. *C est un ensemble fini de m contraintes $\{c_1, \dots, c_m\}$ tel que chaque contrainte $c_j \in C$ porte sur un sous-ensemble de variables de V noté $Var(c_j)$, et est défini par une relation de compatibilité sur les valeurs de $Var(c_j)$, noté $R(c_j)$.*

Définition 2
Résoudre un CSP $P = \langle V,C \rangle$ consiste à vérifier si P admet au moins une solution, c'est-à-dire une affectation de valeur pour chaque variable de V telle que chaque contrainte de C est satisfaite. Si P admet au moins une solution, alors il est dit satisfiable, sinon P est dit insatisfiable.

Dans ce papier, il sera utile d'adopter une définition alternative mais équivalente des CSP, exprimée en termes des tuples de valeurs interdits.

Définition 3
Soit $\langle V,C \rangle$ un CSP et $c \in C$ tel que $Var(c) = \{v_c^1, \dots, v_c^l\}$. Un tuple de valeurs interdites est un élément de l'ensemble $dom(v_c^1) \times \cdots \times dom(v_c^l)$ tel que $R(c)$ n'est pas satisfaite. L'ensemble des tuples de valeurs interdites pour une contrainte c est noté $T(c)$.

Par rapport à cette notation, les CSP peuvent être redéfinis ainsi.

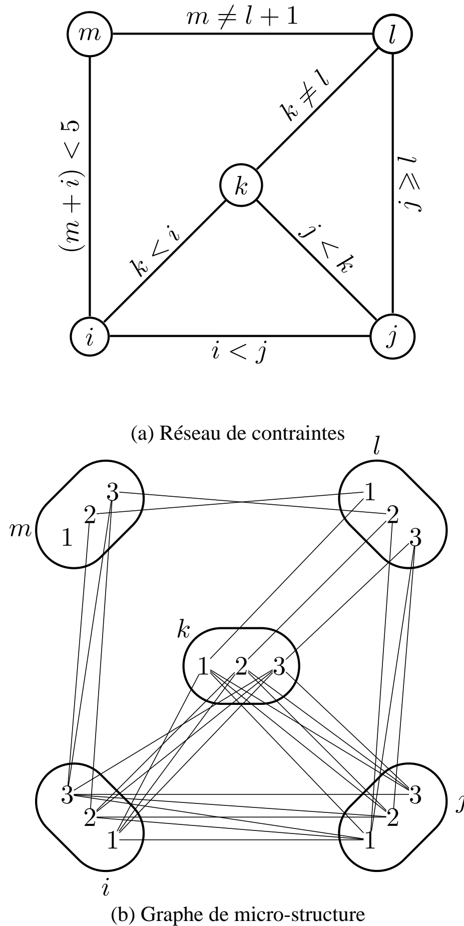
Définition 4
Un CSP est un couple $P = \langle V,C \rangle$ où :

1. *V est un ensemble fini de n variables $\{v_1, \dots, v_n\}$ tel que chaque variable $v_i \in V$ possède un domaine d'instanciation fini noté $dom(v_i)$, qui contient l'ensemble des valeurs possibles de v_i ;*
2. *C est un ensemble fini de m contraintes $\{c_1, \dots, c_m\}$ tel que chaque contrainte $c_j \in C$ porte sur un sous-ensemble de variables de V noté $Var(c_j)$, et est définie par la relation $T(c_j)$, qui contient l'ensemble des tuples de valeur de $Var(c_j)$ interdits par c_j.*

De cette manière, P peut également être noté $\langle V, \{(Var(c_1), T(c_1)), (Var(c_2), T(c_2)), \dots, (Var(c_n), T(c_n))\}\rangle$.

Définition 5
Une contrainte $c \in C$ du CSP $P = \langle \{v_1, \dots, v_n\}, C \rangle$ est falsifiée par une affectation $A \in dom(v_1) \times \cdots \times dom(v_n)$ ssi la projection de A sur $Var(c)$ appartient à $T(c)$.

Dans la suite de ce papier, le terme *tuple (interdit)* sera utilisé comme raccourci pour *tuple interdit de valeurs* et seuls les CSP binaires seront considérés, c'est-à-dire tous CSP dont les contraintes mentionnent au plus deux variables. L'usage des CSP binaires ne restreint pas l'impact de cette étude, dans le sens où il est bien connu que tout CSP discret peut être réduit en un CSP binaire équivalent en temps polynomial.



(a) Réseau de contraintes

(b) Graphe de micro-structure

FIG. 1 – Représentations graphiques de l'exemple 1

Exemple 1
Soit V ={i,j,k,l,m}, où chaque variable possède le même domaine d'instanciation {1,2,3}. Soit C un ensemble de 7 contraintes. Dans la figure 1a, le CSP $P = \langle V,\ C\rangle$ est représenté par son *réseau de contraintes*, un graphe non-orienté où chaque variable est un nœud et chaque contrainte une arête étiquetée par la relation correspondante.

Il peut être également utile de représenter un CSP par son *graphe de micro-structure*, qui est un graphe n-parti (pour un CSP à n variables) regroupant au sein de chaque « partie » les valeurs du domaine de chaque variable, et reliant par une arête chaque couple interdit (ou autorisé) de valeurs par les contraintes du problème. Dans les graphes de micro-structure présentés dans cet article, on représentera les tuples interdits du CSP. Le graphe de micro-structure de cet exemple est donné dans la figure 1b.

Quand un CSP est insatisfiable, il possède au moins un *noyau minimalement insatisfiable*, ou MUC. Un MUC est un sous-ensemble de contraintes d'un CSP qui est insatisfiable et tel que tous ses sous-ensembles propres soient satisfiables.

Définition 6
Soit $P = \langle V, C\rangle$ et $P' = \langle V', C'\rangle$ deux CSP. P' est un noyau insatisfiable de P ssi

1. *P' est insatisfiable ;*
2. *$V' \subseteq V$ et $C' \subseteq C$.*

P' est un noyau minimalement insatisfiable (MUC) de P ssi

1. *P' est un noyau insatisfiable P ;*
2. *P' ne possède aucun noyau insatisfiable.*

Exemple 2
Dans l'exemple précédent, P est insatisfiable. En effet, P contient un MUC $P' = \langle V, \{i < j, j < k, k < i\}\rangle$: aucune affectation de valeurs pour i, j et k ne peut être trouvée telle que ces trois contraintes soient satisfaites, et supprimer l'une de ces contraintes conduit à restaurer la satisfiabilité de ce sous-problème.

Extraire un MUC d'un CSP insatisfiable est un problème NP-difficile. Plus précisément, décider si une contrainte appartient à l'ensemble des MUC d'un CSP est dans Σ_2^p [8]. De plus, le nombre de MUC au sein d'un CSP est exponentiel dans le pire cas ; il est en effet de $\mathcal{O}(C_m^{m/2})$, avec m le nombre de contraintes du CSP. Notons que l'intersection des MUC d'un CSP peut être non vide. Plusieurs techniques ont été proposées dans la littérature pour le calcul des MUC, l'approche `DC(wcore)`, introduite récemment par Hemery *et al.* [14] étant présentée par ses auteurs comme la plus efficace, dans la plupart des cas.

3 Ensemble minimalement insatisfiable de tuples (MUST)

Restaurer la satisfiabilité d'un CSP peut être effectué à travers la réparation de chacun de ses MUC. Une façon naturelle de « casser » l'insatisfiabilité d'un MUC passe par la suppression de l'une de ses contraintes. Cependant, une telle suppression peut apparaître comme un acte très destructif. Alternativement, il peut être préférable d'*affaiblir* une ou plusieurs contraintes, plutôt que de les supprimer. Une façon d'effectuer une telle opération est de fournir à l'utilisateur certains tuples interdits permettant de restaurer la satisfaisabilité du sous-problème correspondant, s'ils étaient autorisés.
Nous introduisons ainsi le concept de MUST, et étudions dans ce but pourquoi celui-ci est un premier pas intéressant dans cette voie. Un MUST (*Minimally Unsatisfiable Set of Tuples*) d'un CSP insatisfiable P est un ensemble insatisfiable de tuples interdits tel que tout sous-ensemble de tuples est satisfiable.

Définition 7
Soit $P = \langle V, \{(\mathit{Var}(c_1), T(c_1)), \ldots, (\mathit{Var}(c_n), T(c_n))\}\rangle$ *un CSP insatisfiable. Le CSP* $P' = \langle V, \{(\mathit{Var}(c_1), T'(c_1)), \ldots, (\mathit{Var}(c_n), T'(c_n))\}\}\rangle$ *est un* MUST *(pour* Minimally Unsatisfiable Set of Tuples *en anglais) de P si et seulement si :*

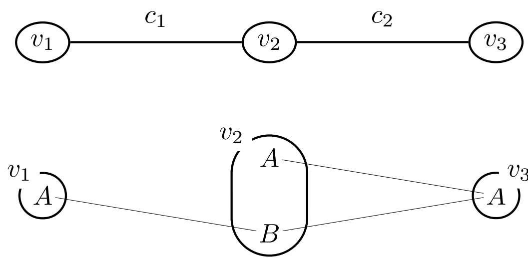


FIG. 2 – Représentations graphiques de l'exemple 3

1. *P' est insatisfiable ;*
2. *$\forall i$ t.q. $1 \leq i \leq m$, $T'(c_i) \subseteq T(c_i)$;*
3. *$\forall i$ t.q. $1 \leq i \leq m$, $\forall\, T''(c_i) \subset T'(c_i)$,
$\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_i), T''(c_i)), ..., (Var(c_m), T'(c_m))\}\rangle$ est satisfiable.*

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 seul 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 3
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 2, le réseau de contraintes ainsi que le graphe de micro-structure de ce CSP sont donnés. Clairement, P est insatisfiable et ne possède qu'un seul MUC, fait de la seule contrainte c_2, puisque celle-ci empêche toute affectation consistante entre les variables v_2 et v_3. Au contraire, à un niveau d'abstraction plus bas, P possède 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.

4 MUST au sein d'un MUC

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

Propriété 1
Au moins un MUST peut être extrait de tout CSP insatisfiable.

Preuve. *Soit $P = \langle V, C\rangle$ un CSP insatisfiable. 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 insatisfiable, et ne contient aucun MUST. Poursuivre ce raisonnement conduit à prouver par induction l'existence du CSP insatisfiable $P' = \langle V, \emptyset\rangle$, alors qu'un tel problème est clairement satisfiable.* □

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

Propriété 2
Soit P un MUC contenant m contraintes. Il existe au moins m tuples tels 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 qui satisfait $P' = \langle V, C\backslash\{c_i\}\rangle$, avec $c_i \in C$. c_i est falsifiée par A : en effet, dans le cas contraire, P serait cohérent et ne pourrait donc pas être un MUC. La projection de A sur $Var(c_i)$ appartient donc à $T(c_i)$ (cf. Définition 5 page 2). 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 si l'un d'entre eux est autorisé. 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

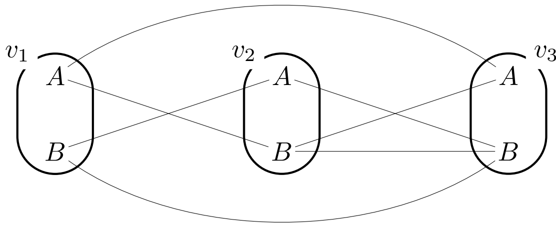


FIG. 3 – Graphe de micro-structure de l'exemple 4

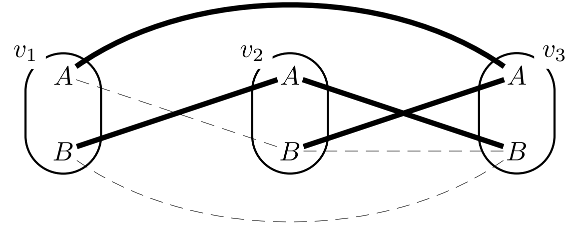


FIG. 4 – Tuples partagés de l'exemple 4

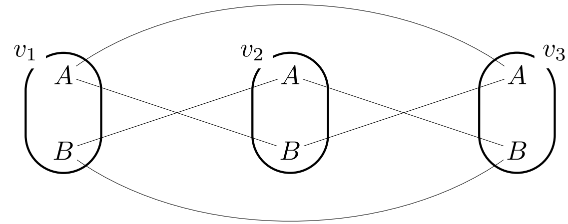


FIG. 5 – Graphe de micro-structure des deux MUST de l'exemple 4

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
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 4
Soit $P = \langle\{v_1, v_2, v_3\}, \{c_1, c_2, c_3\}\rangle$ un CSP tel que :

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 3. Notons que P est un MUC, puisqu'il est insatisfiable et lui ôter n'importe laquelle de ses contraintes conduit à un problème satisfiable. P possède deux MUST, dont les graphes de micro-structure sont représentés en figure 5. 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 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 l'obtention d'un problème satisfiable. Cet exemple montre également que le CSP composé des seuls tuples partagés n'est pas nécessairement insatisfiable. En fait, on peut facilement prouver que ceci n'est vrai que pour les MUC ne contenant qu'un seul MUST.

Ces premiers résultats plaident en faveur d'une politique en deux étapes en vue de l'explication de l'insatisfiabilité en termes de MUC, MUST et tuples partagés. En effet, la recherche de MUST dans le cas général ne semble pas prometteuse puisque les MUST peuvent ne coïncider avec aucun MUC du problème. Au contraire, une approche intéressante consiste en la recherche de MUC dans un premier temps. Ceux-ci fournissent des explications de l'incohérence exprimées par des ensembles irréductibles de contraintes. L'utilisateur peut alors supprimer l'une d'elles pour casser l'insatisfiabilité du MUC. Il peut également choisir de calculer un MUST correspondant au MUC découvert. Plus précisément, s'il décide d'extraire les tuples partagés, il peut alors choisir d'autoriser parmi cet ensemble un tuple, qui est alors *suffisant* pour permettre à l'ensemble de devenir cohérent. De cette façon, on donne à l'utilisateur la possibilité d'affaiblir un ensemble de contraintes problématiques plutôt que de les supprimer. Une telle politique peut être itérée jusqu'à ce que l'ensemble (restant) de MUC du CSP résultant ait été traité.

Plusieurs approches ont par le passé été proposées pour l'extraction d'un MUC. Le problème qui nous est posé est donc le calcul pratique d'un MUST et des tuples partagés au sein d'un MUC. Dans la section suivante, il est montré que, modulo un encodage spécifique vers SAT, les tuples partagés coïncident avec le concept de clauses protégées [10], qui jouent un rôle essentiel dans l'efficacité de l'une des techniques d'extraction d'ensembles minimaux de contraintes dans le cadre booléen, qui sont appelés MUS (pour Minimally Unsatisfiable Sub-formulas). De cette façon, une traduction booléenne est effectuée et permet de bénéficier de cette technique jugée comme l'une des meilleures existantes.

5 Utilisation de OMUS pour le calcul de MUST et de tuples partagés

Calculer un MUST peut être effectué à travers plusieurs techniques bien connues dans le cadre des CSP, telles que les méthodes destructive, constructive et dichotomique de minimisation [4]. Cependant, ces approches ne délivreraient que des MUST, uniquement. Le calcul des tuples partagés nécessiterait soit un nombre linéaire supplémentaire de tests de satisfiabilité, ou de considérer toutes les solutions de chaque relaxation obtenues par le retrait d'une contrainte du MUC calculé.

Au contraire, une approche permettant l'obtention d'un MUST et des tuples partagés dans le même temps est introduite dans cette section.

Quand un MUC a été obtenu par la technique `DC(wcore)` de Hemery *et al.* [14], celui-ci est converti dans le cadre booléen, de façon à ce que le calcul des tuples du MUST soit effectué à travers celui d'un MUS, qui est un ensemble minimal de clauses d'une formule CNF. Le schéma de transformation choisi est une forme de *direct encoding* [6] qui consiste à encoder chaque valeur potentielle de chaque variable par une variable booléenne distincte C_{v_i}. Ainsi, le nombre de variables de la formule booléenne est donné par la somme des tailles des domaines de chaque variable mentionnée dans le MUC. Soit $P = \langle V, C\rangle$ un MUC. Les clauses suivantes sont créés.

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
   $\neg C_{v_a} \vee \neg C_{v_b}\ \forall v \in V\ \forall (v_a, v_b) \in dom(v) \times dom(v)$ tel que $v_a \neq v_b$
3. *Conflict clauses* Ces clauses codent les tuples interdits du problème
   $\neg C_{v_i} \vee \neg C_{v_j}\ \forall c \in C\ \forall (v_i, v_j) \in T(c)$

Cette forme de codage a été choisie car il 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 prouvées facultatives [27] ; 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 les clauses *at-Least-one* en fassent partie.

Plus précisément, nous utilisons la technique `OMUS` de Grégoire *et al.* [10], qui est à ce jour l'une des techniques les plus efficaces pour la découverte d'un MUS au sein d'une instance SAT insatisfiable. L'un des éléments clés de son efficacité est dû au concept de *clauses protégées*. Ces clauses protégées s'avèrent coder les tuples partagés du formalisme CSP.

L'approche `OMUS` est également basée sur le concept de clause critique, qui représente une clause falsifiée par une interprétation particulière et telle que tout changement minimal visant à la satisfaire conduit une autre clause à être falsifiée à son tour. Sommairement, la technique `OMUS` est une approche en deux temps. Tout d'abord, un surensemble d'un MUS est calculé en itérant des recherches locales qui comptent le nombre de fois où chaque clause est critique, et en retirant périodiquement celles ayant obtenu les plus faibles scores. Ensuite, un processus de minimisation permet de retourner un MUS. De plus, à chaque fois qu'une unique clause est falsifiée pendant la recherche locale, celle-ci est marquée et n'est jamais retirée de la formule. Ces clauses sont dites *protégées*, et appartiennent à tous les MUS de la formule booléenne. Par rapport à ce codage, les clauses protégées coïncident avec les tuples partagés du MUC transcrit. Ces tuples sont alors fournis en même temps que le MUST, sans aucun surcoût calculatoire. De plus, la première étape de l'algorithme basée sur la recherche locale retourne parfois un ensemble insatisfiable de clauses protégées, qui forme par conséquent un MUS, et la deuxième étape de l'approche peut être évitée.

6 Étude expérimentale

Afin de vérifier la capacité pratique à extraire un MUST des approches proposées dans ce document, nous avons implémenté un algorithme nommé `MUSTER` (pour *MUST-ExtRaction* en anglais) et exécuté celui-ci sur de nombreux benchmarks issus de la dernière compétition CSP [5]. Son principe est donc d'extraire en premier lieu un MUC grâce à un appel à `DC(wcore)`, puis de le transformer en une formule clausale booléenne, grâce à la technique de *direct-encoding* décrite dans l'algorithme 1. Un MUS est alors calculé via la procédure `OMUS` [10]. 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 de codage décrite dans l'algorithme 2. `MUSTER` est résumé dans l'algorithme 3.

Toutes les expérimentations présentées ici ont été conduites sur un Pentium IV à 3Ghz, sous une distribution Linux Fedora Core 5. Un extrait significatif des résultats obtenus est fourni dans la Table 1. Celle-ci contient 3 colonnes principales : *Instance*, *MUC extrait* et *MUST extrait*. La 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 induisent. 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 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

Input : un CSP : $\langle V, C\rangle$
Output : un ensemble de clauses Σ (vu comme une formule CNF)

```
1  begin
2      Σ ⟵ ∅ ;
3      /* « At-least-one » clauses */
4      foreach v_i ∈ V do
5          Σ ⟵ Σ ∪ { ⋁_{∀j∈dom(v_i)} x_ij }
6      /* « Conflict » clauses */
7      foreach c ∈ C do
8          foreach t ∈ T(c) do
9              Σ = Σ ∪ { ⋁_{∀(v_i∈Var(c) et j∈dom(v_i)) t.q. j est la valeur interdite de v_i dans t} ¬x_ij }
10     return Σ ;
11 end
```

Algorithme 1 – `direct_encode`

Input : un MUS : Σ et un MUC : $\langle V, \{(Var(c_1), T(c_1)), ..., (Var(c_m), T(c_m))\}\rangle$
Output : un MUST : $\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_m), T'(c_m))\}\rangle$

```
1  begin
2      foreach c ∈ C t.q. C = {(Var(c_1), T(c_1)), ..., (Var(c_m), T(c_m))} do
3          T'(c) ⟵ ∅ ;
4          foreach t ∈ T(c) do
5              if (( ⋁_{∀(v_i∈Var(c) et j∈dom(v_i)) t.q. j est la valeur interdite de v_i dans t} ¬x_ij ) ∈ Σ) then
6                  T'(c) ⟵ T'(c) ∪ {t};
7      return ⟨V, {(Var(c_1), T'(c_1)), ..., (Var(c_m), T'(c_m))}⟩ ;
8  end
```

Algorithme 2 – `mus2must`

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 induit 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 extrait, avant d'être réduit en un MUST de 3077 tuples. En outre, au sein de ce MUST, un tuple parmi 2728 peut être supprimé pour restaurer la cohérence du MUC correspondant.

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 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

```
Input : un CSP : ⟨V, C⟩
Output : un MUST : ⟨V, {(Var(c'_1), T''(c'_1)), ..., (Var(c'_m), T''(c'_m))}⟩
1  begin
2    ⟨V, C'⟩ ⟵ DC(wcore)(⟨V, C⟩) ;
3    /* ⟨V, C'⟩ est un MUC t.q. C' ⊆ C et C' = {(Var(c'_1), T(c'_1)), ..., (Var(c'_m), T(c'_m))} */
4    Σ_CNF ⟵ direct_encode(⟨V, C'⟩) ;
5    Σ_MUS ⟵ OMUS(Σ_CNF) ;
6    /* Σ_MUS ⊆ Σ_CNF */
7    ⟨V, {(Var(c'_1), T''(c'_1)), ..., (Var(c'_m), T''(c'_m))}⟩ ⟵ mus2must(Σ_MUS , ⟨V, C'⟩) ;
8    /* ∀ 1 ≤ i ≤ m  T''(c'_i) ⊆ T(c'_i) */
9    return ⟨V, {(Var(c'_1), T''(c'_1)), ..., (Var(c'_m), T''(c'_m))}⟩ ;
10 end
```

Algorithme 3 – MUSTER

combinaison que l'on fait de ce problème un autre, qu'il soit satisfiable ou non.

Enfin, attardons-nous sur le nombre de tuples partagés. La propriété 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 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.

7 Travaux connexes

L'approche présentée dans ce papier peut être interprétée comme un raffinement des techniques d'explication qui fournissent à l'utilisateur un MUC en cas d'insatisfiabilité. Seules quelques études ont pour l'heure été publiées au sujet de l'extraction de MUC pour les CSP, ou de MUS dans le cadre booléen. La première étape de notre approche profite de l'efficacité des meilleures techniques de calcul d'un MUC [14] et d'un MUS [10], pour effectuer celui d'un MUST et des tuples partagés.

Dans le domaine des CSP, plusieurs autres travaux visaient à identifier des ensembles (minimaux) de contraintes conflictuelles (cf. par exemple [25]) enregistrés pendant la recherche, dans le but d'effectuer différents types de retour-en-arrière intelligents, tel que le *dynamic backtracking* [9] [18] ou le *conflict-based backjumping* [26]. Dans [15], une méthode non-intrusive est proposée pour ce type de détection. Cependant, on ne compte que peu de travaux de recherche concernant le problème d'extraction de MUC lui-même. Une méthode pour trouver tous les MUC d'un ensemble donné de contraintes a été présentée dans [13] et [7]. Celle-ci correspond à l'exploration d'un arbre « CS », mais est limitée par l'explosion combinatoire du nombre de sous-problèmes possibles d'un CSP. D'autres approches sont données dans [22] et [16], où une explication basée sur les préférences de l'utilisateur est extraite. On trouve également le système PaLM [17], implémenté dans la plateforme Choco [19], qui est un outil d'explication pouvant répondre à des questions du type : pourquoi n'existe-il pas de solution contenant la valeur v_i pour une certaine variable A ? De plus, en cas d'insatisfiabilité, PaLM est capable d'extraire un sous-problème insatisfiable, mais sans garantie que celui-ci soit minimal. L'approche DC(wcore) utilisée dans ce papier apparaît comme améliorant une précédente méthode introduite dans [2] pour extraire un MUC, qui avait été proposée dans le contexte spécifique du diagnostic de pannes. Elle est également prouvée plus compétitive que l'utilisation de QuickXPlain [15] pour le calcul de MUC.

Dans le cadre clausal booléen, le problème d'extraction d'un MUS d'une CNF insatisfiable a également reçu une certaine attention. Dans [3], Bruni a proposé une approche qui effectue l'approximation de MUS par le biais d'une recherche adaptative guidée par la difficulté supposée des clauses. Zhang et Malik décrivent dans [28] un moyen de faire cette détection grâce aux *nogoods* appris impliqués dans la production de la clause vide par résolution. Dans [21], Lynce et Marques-Silva présentent une technique complète qui calcule le plus petit des MUS d'une formule. Avec Mneimneh, Andraus et Sakallah [23], ces même auteurs proposent également un algorithme qui itère des solutions au problème max-SAT pour identifier un tel ensemble optimal de clauses. Oh et ses coauteurs présentent dans [24] une approche qui consiste à marquer les clauses pendant une réfutation de type DPLL pour calculer ou approximer un MUS. Notons également l'approche complète de Liffiton et Sakallah [20], récemment améliorée par une utilisation non standard de la recherche locale [11], qui vise à calculer l'ensemble exhaustif des MUS d'une formule propositionnelle.

Enfin, précisons que le problème d'extraction d'un IIR (*Irreductible Infeasible Subsystem*) a également été le sujet de recherche en programmation mathématique [4][1].

8 Conclusions et perspectives

Les résultats présentés ici ouvrent de nombreuses pistes de recherche.

Instance			MUC extrait			MUST extrait		
nom	#con	#tuples	#con	#tuples	temps (sec.)	#tuples	#tp[1]	temps (sec.)
composed-25-1-2-0	224	4 440	14	910	10,7	354	119	13,1
composed-25-1-2-1	224	4 440	15	975	9,09	339	59	17,5
composed-25-1-25-8	247	4 555	9	585	8,85	259	116	6,25
composed-75-1-2-1	624	10 440	13	845	66,4	344	104	10,9
composed-75-1-2-2	624	10 440	14	910	66,7	376	48	14,4
composed-75-1-25-8	647	10 555	16	1 040	59,1	461	51	23,7
composed-75-1-80-6	702	10 830	11	715	61,5	278	55	8,17
composed-75-1-80-7	702	10 830	16	1 040	380	420	75	17,2
composed-75-1-80-9	702	10 830	12	780	86,2	306	89	9,01
qk_10_10_5_add	55	48 640	5	47 120	19,7	24 855	0	3081
qk_10_10_5_mul	105	49 140	5	47 120	1,29	24 855	0	2813
qk_8_8_5_add	38	19 624	5	18 800	3,33	10 149	0	545
qk_8_8_5_mul	78	19 944	5	18 800	0,66	10 149	0	531
graph2_f25	2 245	145 205	43	4 498	427	2470	1 516	426
qa_3	40	800	15	583	0,32	203	152	8,32
dual_ehi-85-297-14	4 111	102 234	40	1 145	3,35	311	142	40,3
dual_ehi-85-297-15	4 133	102 433	35	1 083	4,03	310	172	25,9
dual_ehi-85-297-16	4 105	102 156	36	1 032	4,68	301	159	29,1
dual_ehi-85-297-17	4 102	102 112	43	1 239	4,83	348	172	42,2
dual_ehi-85-297-18	4 120	102 324	33	972	3,48	271	141	30,4
dual_ehi-90-315-21	4 388	108 890	37	1 120	3,16	354	129	35
dual_ehi-90-315-22	4 368	108 633	41	1 218	4,57	410	187	43,7
dual_ehi-90-315-23	4 375	108 766	29	835	2,86	251	131	12,2
dual_ehi-90-315-24	4 378	108 793	31	974	4,57	315	167	25,4
dual_ehi-90-315-25	4 398	108 974	38	1 106	3,89	375	179	30,4
dual_ehi-90-315-26	4 370	108 696	37	1 084	4,8	304	159	39,6
scen6_w2	648	513 100	7	8 020	53,8	4 872	2 953	1107
scen6_w1_f2	319	274 860	21	21 146	489	-	-	*time out*
scen11_f10	4 103	738 719	16	4 588	164	3 077	2 728	438
scen11_f12	4 103	707 375	16	4 588	122	3 053	2 728	420

TAB. 1 – Extraction d'un MUST

Un CSP insatisfiable peut possèder plusieurs MUC n'ayant aucune contrainte en commun. Restaurer la satisfiabilité en autorisant des tuples partagés nécessite des itérations de la procédure `MUSTER` jusqu'à ce que tous les MUC soient traités. Clairement, l'ordre dans lequel les MUC sont considérés peut avoir une influence sur les tuples problématiques fournis à l'utilisateur. Ainsi, il peut s'avérer extrêmement intéressant de développer des techniques indépendantes de l'ordre retournant directement à l'utilisateur les ensembles de tuples qui rendraient le CSP satisfiable s'ils étaient autorisés. Dans cet objectif, des résultats récents de Grégoire *et al.* [12] permettant le calcul de couvertures de MUS, c'est-à-dire des ensembles de MUS couvrant suffisamment de causes d'insatisfiabilité pour réparer la globalité de la formule, pourraient être exploités. Le concept de tuples partagés pourrait être modifié en conséquence, et des techniques calculatoires spécifiques développées dans ce but.

Bien qu'elle se montre très compétitive en pratique, notre méthode de calcul des tuples partagés reste incomplète, puisqu'elle ne garantit pas leur détection exhaustive. Une autre piste très intéressante consiste à imaginer une technique complète qui assure, modulo une potentielle explosion combinatoire, le calcul de *tous* les tuples partagés.
Ce papier fournit les premiers résultats formels à propos des MUST. Clairement, de nouvelles études visant à fournir des variantes à ces définitions pourraient se montrer intéressantes, en particulier des variantes n'étant pas liées au concept de MUC pour expliquer et réparer l'insatisfiabilité. Plusieurs MUST peuvent également être corrélés au sein d'un CSP. Etudier les propriétés formelles de leur relation est également une piste prometteuse de recherche.

Références

[1] M.K. Atlihan and L. Schrage. Generalized filtrering algorithms for infeasibility analysis. *Computers and Operations Research*, 2007. À paraître.

[2] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *Proceedings of the 13th International Joint Conference on Artificial*

[1]#tp : #tuples partagés

Intelligence (IJCAI'93), volume 1, pages 276–281. Morgan Kaufmann, 1993.

[3] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2) :85–100, 2003.

[4] J.W. Chinneck. *Feasibility and Viability,* In *Advances in Sensitivity Analysis and Parametric Programming*, volume 6, chapter 14, pages 14.2–14.41. Boston (États-Unis), 1997.

[5] Compétitions CSP, http ://cpai.ucc.ie/06/Competition.html.

[6] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 290–296, 1989.

[7] M. de la Banda, P. J. Stuckey, and J. Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDL'03)*, pages 32–43, 2003.

[8] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactual. *Artificial Intelligence*, 57 :227–270, 1992.

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

[10] É. Grégoire, B. Mazure, and C. Piette. Extracting MUSes. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 387–391, 2006.

[11] É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSSes and MUSes thanks to a local search oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 2, pages 2300–2305, 2007.

[12] É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of MUSes. *Constraints Journal : Special issue on Local Search in Constraint Satisfaction*, 12(3) :325–344, 2007.

[13] B. Han and S. Lee. Deriving minimal conflict sets by CS-Trees with mark set in diagnosis from first principles. In *IEEE Transactions on Systems, Man, and Cybernetics*, volume 29, pages 281–286, 1999.

[14] F. Hemery, C. Lecoutre, L. Saïs, and F. Boussemart. Extracting MUCs from constraint networks. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, 2006. 113–117.

[15] U. Junker. QuickXplain : Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, 2001.

[16] U. Junker. QuickXplain : Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, 2004.

[17] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, pages 118–133, 2000.

[18] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming*, pages 249–261, 2000.

[19] F. Laburthe and The OCRE Project Team. Choco: implementing a CP kernel. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, 2000. http ://www.choco-constraints.net.

[20] M.H. Liffiton and K.A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of SAT'05*, pages 173–186, 2005.

[21] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[22] J. Mauss and M. M. Tatar. Computing minimal conflicts for rich constraint languages. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02)*, pages 151–155, 2002.

[23] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques Silva, and K. A. Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 467–474, 2005.

[24] Y. Oh, M.N. Mneimneh, Z.S. Andraus, K.A. Sakallah, and I.L. Markov. AMUSE : a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41th Design Automation Conference (DAC 2004)*, pages 518–523, 2004.

[25] T. Petit, C. Bessière, and J.C. Régin. A general conflict-set based framework for partial constraint satisfaction. In *Proceedings of SOFT'03: Workshop on Soft Constraints held with CP'03*, 2003.

[26] P. Prosser. Hybrid algorithms for the constraint satisfaction problems. In *Computational Intelligence*, volume 9(3), pages 268–299, 1993.

[27] T. Walsh. SAT v CSP. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 441–456, 2000.

[28] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Sixth international conference on theory and applications of satisfiability testing (SAT'03)*, 2003.