Extraction de sous-formules minimales inconsistantes

Eric Gregoire, Bertrand Mazure, Cédric Piette

Extraction de sous-formules minimales inconsistantes

Éric Grégoire **Bertrand Mazure** **Cédric Piette**

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

Résumé

Une sous-formule minimale inconsistante (Minimally Unsatisfiable Subformula ou MUS en anglais) représente la plus petite cause d'incohérence d'une instance SAT en terme de nombre de clauses. Extraire un ou plusieurs MUS s'avère donc très utile, car ceux-ci circonscrivent les sources d'inconsistance d'une formule CNF. Dans ce papier, une nouvelle méta-heuristique permettant l'approximation ou le calcul d'un MUS est présentée. Une comparaison avec les méthodes les plus compétitives est effectuée, et montre que le plus souvent, en pratique, cette approche surpasse les résultats obtenus par les autres méthodes existantes.

1 Introduction

SAT est le problème de décision NP-complet qui consiste à vérifier si un ensemble de clauses propositionnelles admet au moins une interprétation qui les satisfasse toutes. Ces dernières années, ont vu apparaître une grande communauté de chercheurs impliquée dans l'étude sur les plans théorique et pratique de SAT (voir par exemple http ://www.SATlive.org), ainsi que dans ses applications dans divers domaines de l'intelligence artificielle.

Plus récemment, de nombreuses personnes se sont concentrées sur la tâche ardue de l'extraction de sous-formules minimales inconsistantes (Minimally Unsatisfiable Subformula ou MUS en anglais) d'instances contradictoires. Bien que ce problème ait une complexité très élevée dans le pire des cas, puisque par exemple décider si une formule appartient à l'ensemble des MUS d'une instance incohérente est Σ_p^2-difficile, extraire un ou plusieurs MUS peut s'avérer très utile, puisque ceux-ci expliquent ce qui est *erroné* dans une formule.

De surcroît, ce problème possède de nombreux domaines d'applications tels que la validation de bases de connaissance, la correction de circuits ou les diagnostics de panne, qui nécessitent ces explications d'inconsistance. Quand, par exemple, la satisfiabilité d'une base de connaissance est testée, on préfère souvent connaître quelles clauses entrent en contradiction, plutôt que de juste savoir que l'ensemble de la base est inconsistant.

Récemment, plusieurs approches ont été proposées pour approcher ou calculer des MUS. Malheureusement, elles concernent certaines classes d'instances ou ne sont applicables qu'avec de petites formules. Parmi elles, le travail de Renato Bruni [5] montre comment extraire polynomialement un MUS par des techniques de programmation linéaire pour des formules obéissant à la propriété appelée « integral point ». Cependant, seules quelques classes de formules répondent à cette propriété (principalement les formules composées de clauses de Horn, Horn renommables, les formules équilibrées ou « matched »).

On trouve également dans [1, 6, 9] des études sur la complexité et les aspects algorithmiques de l'extraction de MUS pour des classes d'instances spécifiques.

Dans [4], Bruni a également proposé une méthode permettant l'approximation de MUS par le biais d'une recherche adaptative guidée par la difficulté supposée des clauses. Zhang et Malik décrivent dans [21] un moyen de faire l'extraire une sous-formule incohérente par l'apprentissage de *nogoods* impliqués dans la dérivation par résolution de la clause vide. Dans [15], Lynce et Marques-Silva ont proposé une méthode complète pour extraire le plus petit MUS (en terme de clauses) d'une formule. Oh et ses co-auteurs ont présenté dans [18] une approche basée sur

la procédure DPLL qui marque certaines clauses dans le but d'approcher un MUS. Liffiton et Sakallah ont montré comment les MUS d'une formule peuvent être calculées à l'aide du concept dual de base maximale consistante [14].

Enfin, dans [17], une heuristique d'approximation de MUS est suggérée. Celle-ci est basée sur le fait empirique que les clauses les plus souvent falsifiées pendant une recherche locale appartiennent le plus souvent aux MUS de la formule. Cette heuristique a également été utilisée pour améliorer les performances de la procédure DPLL. Dans ce papier, une variante ainsi que de nouvelles extensions de cette heuristique sont étudiées. Pendant la recherche locale, une partie du voisinage de l'interprétation courante est explorée dans le but de savoir si oui ou non les clauses falsifiées doivent être comptées pour le calcul de la trace. Cette heuristique est étendue pour le calcul d'un ensemble de MUS. Cette nouvelle approche dépasse souvent les résultats obtenus par les méthodes les plus compétitives déjà proposées, d'un point de vue expérimental.

Le papier est organisé comme suit. Dans le paragraphe suivant, le concept de MUS est présenté formellement. Dans le paragraphe 3, la notion essentielle de clause critique est présentée et analysée. Dans le paragraphe 4, une nouvelle approche permettant l'approximation voire le calcul d'un MUS est présentée. De nombreux résultats expérimentaux sont fournis dans le paragraphe 5. Avant de conclure, le paragraphe 6 montre comment cette approche peut être étendue pour calculer des ensembles de MUS.

2 Formules minimales inconsistantes (MUS)

Soit $\mathcal{L}$ la logique propositionnelle standard, construite sur un ensemble fini de variables booléennes notées a, b, etc. Les connecteurs logiques sont notés de manière usuelle, comme suit : la disjonction $\vee$, la conjonction $\wedge$, l'implication $\Rightarrow$, l'équivalence $\Leftrightarrow$.

Les formules sont notées par des lettres majuscules telles que C. Les ensembles de formules sont représentés par des lettres grecques, comme Γ ou Σ.

Une interprétation est une fonction qui assigne une des valeurs $\{vrai, faux\}$ à chacune des variables booléennes. Une formule est dite consistante, ou cohérente, s'il existe au moins une interprétation qui la satisfasse, c'est-à-dire telle qu'elle soit $vrai$. Une interprétation est notée par une lettre majuscule telle que I et est représentée par l'ensemble des littéraux qu'elle satisfait.

Toute formule de $\mathcal{L}$ peut être représentée (en préservant sa nature logique) en utilisant un ensemble (interprété comme une conjonction) de clauses, où une clause est une disjonction finie de littéraux. Enfin, un littéral est une variable booléenne ou sa négation.

SAT est le problème de décision NP-complet de référence, et consiste à tester si un ensemble de clauses propositionnelles est consistant ou non, c'est-à-dire s'il existe une interprétation qui satisfasse toutes les clauses de l'ensemble ou non.

Si une formule est contradictoire, alors elle possède au moins une sous-formule minimale inconsistante.

Définition 1
Une sous-formule minimale inconsistante Γ (en anglais Minimally Unsatisfiable Subformula ou `MUS`*) d'une instance Σ est un ensemble de clauses tel que :*

1. $\Gamma \subseteq \Sigma$
2. Γ *est insatisfiable*
3. *Tout sous-ensemble strict de* Γ *est satisfiable*

Le calcul de MUS est une lourde tâche dans le pire des cas. En effet, décider si un ensemble est un MUS est un problème DP-complet [19], et décider si une formule appartient à l'ensemble des MUS d'une instance contradictoire est Σ_p^2-difficile [8]. Notons que, bien que si dans le pire des cas, le nombre de MUS que peut contenir une instance construite sur n clauses est de $C_n^{n/2}$, ce nombre est souvent très réduit pour les instances industrielles ou issues de problèmes réels. Par exemple, en diagnostic de pannes [12], on effectue souvent l'hypothèse d'une panne unique, ce qui se traduit par un unique MUS.

3 Une nouvelle méta-heuristique pour détecter les MUS

Dans [17], il est montré comment une recherche locale peut s'avérer utile pour approcher les MUS. L'idée de base est fondée sur le fait que les clauses les plus souvent falsifiées lors d'une recherche locale infructueuse pour la satisfiabilité appartiennent le plus souvent aux MUS de la formule, quand celle-ci est effectivement contradictoire. En appelant *score d'une clause* le nombre de fois où celle-ci a été falsifiée durant une recherche locale (RL), discriminer les clauses ayant un score élevé permet d'obtenir une bonne approximation de l'ensemble de MUS de la formule.

Une telle heuristique a été étudiée dans [16, 17] de manière intensive. Celle-ci a d'ailleurs été étendue pour résoudre divers problèmes de décision et d'optimisation calculatoirement plus difficiles [10, 3, 11, 2]. Dans la suite de ce papier, nous supposerons que les formules dont nous parlerons sont incohérentes.

L'heuristique présentée précédemment nécessite d'incrémenter le score des clauses qui sont falsifiées à chaque interprétation, même si celles-ci n'appartiennent à aucun MUS. Bien que de cette manière, on puisse dans certains cas résoudre efficacement le problème de l'extraction d'un

MUS, on ne peut compter que sur des indications heuristiques qui ne sont pas nécessairement fiables. On sait en effet que lors d'une RL certaines zones de l'espace de recherche sont amenées à être largement plus explorées que d'autres ; cela peut être dû à la structure particulière du problème, ou plus simplement à l'interprétation initialement générée de manière aléatoire, et qui peut être d'une grande importance pour l'espace parcouru par la suite. Il semble alors possible que certaines clauses puissent être falsifiées relativement souvent, bien qu'elles n'appartiennent à aucun MUS de l'instance. Dans le but de décider si une clause falsifiée doit être effectivement comptée ou non lors du calcul de la trace, nous pensons que certaines parties du voisinage de l'interprétation courante peuvent fournir des informations intéressantes, et doivent par conséquent être prises en compte. L'idée générale est de tenir compte de la structure de C, et de n'incrémenter son score que si elle ne peut être satisfaite qu'en conduisant d'autres clauses à être falsifiées à leur tour. Nous allons voir que cette technique implémente des définitions qui permettent de dériver une propriété intrinsèque aux clauses appartenant aux MUS.

Pour illustrer ces concepts, utilisons l'exemple suivant : soit $\Delta = \{a \vee b \vee c, \neg a \vee b, \neg b \vee c, \neg c \vee a, \neg a \vee \neg b \vee \neg c\}$. Δ est incohérente et est son propre MUS. Soit $I = \{a, b, c\}$ une interprétation. Relativement à cette interprétation, seule la clause $\neg a \vee \neg b \vee \neg c$ est falsifiée. Dans la suite, la notion de clause unisatisfaite va être utile.

Définition 2
Une clause C est unisatisfaite par une interprétation I ssi exactement un littéral de C est satisfait par I.

Dans notre exemple, les clauses $\neg a \vee b$, $\neg b \vee c$ et $\neg c \vee a$ sont unisatisfaites par $I = \{a, b, c\}$.

Définition 3
Une clause C falsifiée par une interprétation I est critique relativement à I ssi l'opposé de chacun des littéraux de C appartient à une clause unisatisfaite par I. Ces clauses unisatisfaites (qu'on suppose non tautologiques) sont dites liées à C.

Dans l'exemple, la clause falsifiée $\neg a \vee \neg b \vee \neg c$ par I est critique par rapport à I, et ses clauses liées sont les unisatisfaites $\neg a \vee b$, $\neg b \vee c$ et $\neg c \vee a$.

Le rôle de ces définitions est facilement compréhensible par la propriété suivante.

Propriété 1
Soit C une clause critique par rapport à une interprétation I. Tout flip sur I vers I' tel que C soit satisfaite par I' conduit I' à falsifier une clause qui était satisfaite par I.

Preuve
Si C est critique, pour chaque littéral l de C, $\exists C'$ t.q. C' est unisatisfaite par rapport à I et $\bar{l}$ appartient à C'. C est falsifiée par I, donc l est $faux$ par rapport à I et $\bar{l}$ est $vrai$ par rapport à I. $\bar{l}$ est le seul littéral $vrai$ de C', donc si l'interprétation de l est flippée, C' devient falsifiée. □

Dans le but de discriminer les clauses appartenant aux MUS, l'idée est d'incrémenter le score des clauses critiques ainsi que celui des clauses liées à ces clauses 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 2
Soit I une interprétation optimale pour MaxSAT sur une instance Σ. Alors, toute clause falsifiée C par rapport à I appartient à au moins un MUS de Σ et est critique par rapport à I. 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 I appartient à un MUS car I est optimale pour le nombre de clauses satisfaites et au moins une clause par MUS est nécessairement falsifiée. Le fait que toute clause fausse par rapport à I est critique est prouvé par la propriété 4 car I 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 I 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. Soit $\Delta = \{a \vee d, \neg a \vee \neg b, \neg d \vee e, f, \neg e \vee \neg f\}$. Clairement, Δ est cohérente. $\neg e \vee \neg f$ est falsifiée par rapport à $I = \{a, b, d, e, f\}$ et est critique. De plus, une clause appartenant à un MUS peut être falsifiée par une interprétation I sans pour autant être critique par rapport à elle. Par exemple, soit $\Delta = \{a \vee d, b, \neg a \vee \neg b, \neg d \vee e, f, \neg e \vee \neg f\}$. Clairement, Δ est une formule minimale inconsistante. $\neg a \vee \neg b$ est falsifiée par $I = \{a, b, d, e, f\}$ mais n'est pas critique par rapport à cette interprétation. Toutefois, la propriété suivante nous assure que toute clause appartenant à un MUS peut être détectée par notre heuristique.

Propriété 3
Soit Γ un MUS. Pour toute clause $C \in \Gamma$, il existe une interprétation I telle que C est critique par rapport à I.

Preuve
Soit Γ un MUS et C une clause t.q. $C \in \Gamma$. Par définition, on a $\Gamma \setminus C$ satisfiable. Soit M un modèle de $\Gamma \setminus C$. Prouvons que C est critique par rapport à M.

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

Cette propriété nous assure que toute clause appartenant à un MUS peut être critique par rapport à au moins une interprétation. Toutefois, si cette propriété garantit l'existance d'une telle interprétation, elle ne peut nous assurer que notre heuristique basée sur la recherche locale sera capable de parcourir l'ensemble des interprétations rendant critiques les clauses appartenant aux MUS. En effet, détecter un MUS nécessite la détection de toutes les clauses le composant, et la recherche locale, par nature, ne visite pas exhaustivement toutes les interprétations possibles. Cependant, la propriété suivante ainsi que son corollaire nous indiquent que la recherche locale parcourera probablement des interprétations où les clauses appartenant aux MUS sont critiques. Il est en effet bien connu que la RL est en général attirée par les minima. La propriété 4 nous donne l'assurance que l'ensemble des clauses falsifiées par une interprétation sont critiques dans les minima locaux et globaux.

Définition 4
Un minimum local est une interprétation telle qu'aucun flip ne peut accroître le nombre de clauses satisfaites. Un minimum global (ou solution) est une interprétation optimale par rapport au nombre de clauses satisfaites.

Propriété 4
Dans un minimum (local ou global), toutes les clauses falsifiées sont critiques.

Preuve
Si une variable apparaîssant dans une clause fausse est flippée, alors cette clause est satisfaite et au moins une clause anciennement satisfaite est falsifiée (puisqu'un flip ne peut accroître le nombre de clauses satisfaites dans un minimum). Ceci signifie que cette clause nouvellement falsifiée était unisatisfaite. On peut donc conclure que la clause falsifiée dans le minimum était donc nécessairement critique. □

De surcroît, un corollaire nous assure que dans les minima, au moins une clause de chaque MUS est critique.

Corollaire 1
Dans un minimum (local ou global), au moins une clause par MUS est critique.

4 Approcher et calculer un MUS

Dans ce paragraphe, il est démontré qu'une métaheuristique basée sur le score des clauses critiques est viable dans l'objectif d'approcher ou de calculer un MUS. En pratique, pour des raisons d'efficacité, on n'augmente que le score des clauses critiques. Calculer le score des clauses en tenant compte des clauses unisatisfaites liées ne conduit pas à des gains importants en terme de performance, du moins avec l'implémentation que nous avons effectuée et les instances testées.

L'idée générale est la suivante. Soit Σ une formule incohérente. Tant qu'il n'est pas possible de trouver un modèle par la recherche locale à Σ, celle-ci est affaiblie des clauses ayant obtenu les plus faibles scores. À chaque étape, avant l'affaiblissement, l'ensemble de clauses testé est sauvegardé. Le dernier ensemble de clauses pour lequel la recherche locale a échoué, est vérifié inconsistant. Si tel est le cas, alors un sur-ensemble de l'un des MUS de Σ a été extrait. Sinon, ce test d'incohérence est répété sur le sur-ensemble de clauses précédent, jusqu'à ce que celui-ci soit prouvé incohérent. Cet algorithme est résumé dans la procédue `AOMUS`.

```
Procedure AOMUS(Σ) // Approximate One MUS
début
  pile := ∅ ;
  tant que ( RL+Score(Σ) échoue à
             trouver un modèle  )
  faire
    empiler(Σ) ;
    Σ := Σ \ PlusFaiblesScores(Σ) ;
  fait
  répéter
    Σ := dépiler() ;
  jusqu'à ce que ( Σ soit inconsistant )
fin
```

De plus, un MUS peut être exactement obtenu par une minimisation pas à pas du sur-ensemble détecté. Cette

procédure est appelée `fine-tune`. L'ordre dans lequel chaque clause est testée peut être guidé par leur score.

```
Procedure fine-tune(Σ)
début
   pour chaque clause c ∈ Σ // triées par score
    Si ( Σ \ c ) est inconsistant
    Alors Σ := Σ \ c ;
   fait
fin
```

L'efficacité de cette procédure dépend directement de la qualité du sur-ensemble obtenu. Dans le paragraphe suivant, les résultats expérimentaux montrent qu'en général, l'approximation calculée par `AOMUS` est de bonne qualité, car seul un petit nombre de clauses est retiré par la procédure `fine-tune` et en conséquence, seul un petit nombre de tests d'inconsistance sont effectués (quand une clause appartient à un MUS, le test se résume à la vérification de la consistance de la formule).

De plus, cette procédure a été raffinée de la manière suivante. Pendant la recherche locale, quand une seule clause a été falsifiée, alors, nous sommes sûrs que cette clause appartient à tous les MUS de la sous-formule courante (puisque son retrait restaure la consistance de la sous-formule). Cette clause est donc marquée comme protégée, et ne peut plus être retirée de Σ par la suite. Si la sous-formule générée est incohérente et contient uniquement des clauses marquées, alors nous sommes sûrs d'être en présence d'un MUS et l'étape de `fine-tune` peut être évitée. Il apparaît que ce raffinement se montre très utile, et permet un gain signicatif des performances de la procédure. L'algorithme `OMUS` comprend la procédure `AOMUS` ainsi que l'étape `fine-tune` effectuée avec ce raffinement.

Les paramètres de cette procédure sont les suivants. `Wsat` [13] a été choisi pour la recherche locale, l'heuristique de choix de variable étant $Rnovelty$+. Les autres paramètres ont été choisis à partir de de nombreux tests sur divers benchmarks. Après chaque flip, le score des clauses critiques est incrémenté par le nombre des clauses unisatisfaites qui lui sont liées. Ce choix permet en particulier de prendre en compte la taille des clauses, puisque de la taille d'une clause critique dépend le nombre de clauses qui lui sont liées. A la fin de la RL, les clauses dont le score est inférieur à $(min\text{-}score + \frac{\#Flips}{\#Clauses})$ sont supprimées, où $min\text{-}score$ est le plus petit score d'une clause ; $\#Flips$ et $\#Clauses$ sont le nombre flips effectués par la RL, et le nombre de clauses de Σ, respectivement.

Cet algorithme a été testé sur diverses instances issues de DIMACS [7] ainsi que de la compétition annuelle SAT [20], et comparé avec les différentes approches existantes non réduites à une classe spécifique d'instances, comme décrit dans le paragraphe suivant.

5 Résultats expérimentaux

Les différentes expérimentations ont été conduites sur des Pentium IV, 3Ghz sous Linux Fedora Core 4. Comme les résultats le montrent, l'approximation nous donne des résultats exacts la plupart du temps. De plus, la procédure `fine-tune` nous garantit qu'un noyau a effectivement été détecté. Comme la plupart des algorithmes ne garantissent pas la minimalité des sous-formules insatisfiables extraites, nous présentons les résultats avec et sans la procédure `fine-tune`. Sans cette étape, l'approche (baptisée `AOMUS`) délivre dans le cas général un sur-ensemble de MUS. Cependant, sur de nombreuses instances, ces sous-formules sont en réalité exactement des MUS. Par ailleurs, nous avons constaté que dans la majorité des cas, le dernier ensemble de clauses sur lequel la recherche locale a échoué à trouver un modèle était prouvé incohérent, ce qui implique qu'un seul test CoNP (une seule recherche complète via un DPLL) est effectué en pratique par notre approche.

Nous avons comparé notre approche avec une adaptation de `AOMUS` où la fonction de score $Scoring$ est l'heuristique initiale, telle que présentée dans [17], où l'on compte simplement le nombre de fois où une clause est falsifiée. L'approche est également comparée à `zCore`, l'extracteur de sous-formules insatisfiables de `zChaff` [21]. `zChaff` est actuellement l'un des solveurs SAT les plus efficaces. Nous avons également lancé la procédure de Lynce et Marques-Silva [15], et repris les résultats expérimentaux de Bruni [4], son système n'étant pas disponible. Bien que la comparaison avec la méthode de Bruni est difficile d'un point de vue expérimental, il apparaît que sa procédure n'a été testée que sur de petites instances.

`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. Sans surprise, notre approche se montre plus efficace que l'approche similaire où la fonction de score est basée sur l'heuristique [17]. Le plus souvent, l'approche est plus compétitive que toutes les autres sur de grandes instances contenant plusieurs MUS. De plus, seule notre méthode se montre efficace sur l'ensemble des benchmarks. Signalons que la procédure de Lynce-Silva calcule le plus petit MUS de manière exacte et complète, là où `zCore` retourne une approximation d'un MUS. Tandis qu'`OMUS` et `AOMUS` retournent respectivement un MUS et une approximation d'un MUS. Notons que les MUS (ou les approximations) découvertes par les différentes approches ne sont pas nécessairement les mêmes. Dans la table 1, quelques résultats expérimentaux sont donnés[1]. A l'exception des résultats de Bruni qui ont été extraits

[1]Des résultats expérimentaux plus complets sont disponibles sur http ://www.cril.univ-artois.fr/˜piette/extractingMUS_comparison.pdf

TAB. 1 – Résultats expérimentaux : Approcher (AOMUS) et calculer (OMUS) un MUS

Instance	#var	#cla	Lynce&Silva [15]		Bruni [5]	Zcore [21]		Scoring like [17]		AOMUS		OMUS	
			#cla	Temps	#cla	#cla	Temps	#cla	Temps	#cla	Temps	#cla	Temps
fpga10_11	220	1122	Time out		-	561	28.51	561	18.26	561	13.06	561	13.75
fpga10_12	240	1344	Time out		-	672	71.27	561	30.11	561	16.9	561	17.03
fpga10_13	260	1586	Time out		-	793	166.99	561	51.67	561	25.95	561	31.89
fpga10_15	300	2130	Time out		-	1065	570.3	561	128.05	561	44.18	561	68.17
fpga11_12	264	1476	Time out		-	738	112.53	738	66.8	738	65.49	738	66.3
fpga11_13	286	1742	Time out		-	871	504.97	738	180.66	738	56.71	738	84.74
fpga11_14	308	2030	Time out		-	1015	1565.6	738	415.32	738	69.55	738	304.4
fpga11_15	330	2340	Time out		-	Time out		738	568.79	738	52.14	738	85.2
aim100-1_6-no-2	100	160	53	224	54	54	0.05	53	0.268	53	0.38	53	0.38
aim100-2_0-no-1	100	200	Time out		19	19	0.09	19	0.216	19	0.19	19	0.23
aim200-1_6-no-3	200	320	Time out		86	83	0.07	83	0.37	83	0.44	83	0.83
aim200-2_0-no-3	200	400	Time out		37	37	0.23	37	0.39	37	0.49	37	0.54
aim50-1_6-no-4	50	80	20	1.18	20	20	0.04	20	0.163	20	0.16	20	0.17
aim50-2_0-no-4	50	100	21	3.49	21	21	0.14	21	0.208	21	0.22	21	0.27
2bitadd_10	590	1422	Time out		-	815	343.48	1212	42.752	806	189.47	716	268.5
barrel2	50	159	Time out		-	77	0.04	100	0.35	77	0.36	77	0.44
jnh10	100	850	Time out		161	68	0.88	128	9.35	79	42.25	79	42.9
jnh20	100	850	Time out		120	102	0.23	104	21.68	87	48.93	87	75.76
jnh5	100	850	Time out		125	86	0.39	140	12.653	88	46.2	86	46.87
jnh8	100	850	Time out		91	90	0.22	162	28.964	69	90.53	67	99.07
homer06	180	830	Time out		-	415	15.96	415	10.97	415	9.03	415	9.04
homer07	198	1012	Time out		-	506	21.6	415	12.59	415	10.67	415	19.19
homer08	216	1212	Time out		-	606	44.46	554	23.43	415	19.79	415	24.65
homer09	270	1920	Time out		-	960	141.48	415	93.19	504	60.9	415	81.23
homer10	360	3460	Time out		-	940	624.11	1614	148.27	503	466.94	415	513.11
homer11	220	1122	Time out		-	561	23.44	561	41.68	561	15.6	561	16.32
homer12	240	1344	Time out		-	672	76.19	708	25.92	564	41.03	561	62.34
homer13	260	1586	Time out		-	793	152.13	579	67.38	561	76.66	561	78.51
homer14	300	2130	Time out		-	1065	714.03	561	347.19	561	28.03	561	30.64
homer15	400	3840	Time out		-	Time out		677	247.84	561	1048.28	561	1104.13
homer16	264	1476	Time out		-	738	115.49	738	78.44	738	61.31	738	62.91
homer17	286	1742	Time out		-	871	369.11	870	127.43	738	68.28	738	87.4

de [4], nous fournissons la taille (en nombre de clauses) du sous-ensemble inconsistant détecté, ainsi que le temps (en secondes) nécessaire à la terminaison de l'algorithme. Le mention *Time out* indique qu'aucun résultat n'a pu être obtenu en 1 heure de temps CPU. Par exemple, pour l'instance homer14, AOMUS extrait une approximation de MUS composée de 561 clauses en 28.03 s., et qui s'avère être exactement un MUS, puisque celui-ci est extrait par OMUS en 30.64 s. Il est intéressant de noter que ce même MUS est détecté par la version basée sur [17] en 347 s. zCore, quant à lui, extrait une approximation de MUS composée de 1065 clauses en 714 s. Cette approximation est en fait un sur-ensemble du MUS détecté par AOMUS et OMUS. De plus, on peut remarquer qu'en général, sur les instances industrielles FPGA, AOMUS (c'est-à-dire l'approche sans la procédure fine-tune) fournit le plus petit ensemble de clauses inconsistant. Soulignons que, même sur de petites instances, telles que les AIM, OMUS se montre également très compétitif.

6 Approcher l'ensemble des MUS

En nous basant sur la procédure OMUS, nous nous posons maintenant le problème d'extraire l'ensemble de MUS d'une formule, également appelé « *clutter* » par Bruni [5]. 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 ». Pour éviter ce problème dans la mesure du possible, on préfère retirer les clauses appartenant au moins de MUS possible. L'heuristique suivante a été conçue dans cet objectif.

Comme MaxSAT fournit le nombre maximal de clauses pouvant être satisfaites, le reste des clauses falsifiées par une telle interprétation appartient à un grand nombre de

TAB. 2 – Approcher l'ensemble de MUS

Instance	#var	#cla	L.&S. [14]		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

MUS. De ce point de vue, on choisit de sauvegarder pour chaque clause le minimum de clauses pouvant être falsifiées par une même interprétation. Après qu'un MUS ait été détectée, la clause de cet ensemble ayant le plus faible score est retirée de la formule.

Clairement, une telle approche (notée `ASMUS`) est incomplète. Cependant, elle fournit de très bons résultats relativement aux méthodes déjà proposées, comme présentés par les résultats expérimentaux dans la table 2. 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, 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 dans [14] d'un point de vue expérimental. La table 2 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 et Sakallah peut avoir un temps exponentiellement long, puisque une formule CNF peut possèder 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 calculées les uns après les autres. Par exemple, considérons la formule `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, [14] est capable de calculer l'ensemble de ses MUS. Toutefois, sur des formules quelques peu plus grandes, comme `dp02u01` (213 variables, 376 clauses), l'ensemble des MUS ne peut être délivré en 20 000 secondes, alors que notre approche en extrait 14 en 26 secondes.

7 Conclusion

Dans ce papier, grâce au concept original de clauses critiques, une nouvelle méta-heuristique permettant d'approcher ou calculer un MUS a été présentée. Comme l'illustrent nos résultats expérimentaux sur différentes formules, cette approche se montre viable et obtient souvent de meilleurs résultats que les approches déjà proposées. La méta-heuristique est basée sur l'idée que les contraintes les plus souvent falsifiées pendant une recherche locale participent le plus souvent à l'inconsistance de la formule. Cette idée a été raffinée pour prendre en compte le voisinage partiel des interprétations parcourues. Nous pensons qu'une telle heuristique peut être appliquée à différents problèmes de décision et d'optimisation. Ceci sera l'objet de recherches futures.

Références

[1] H.K. Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107(1–3) :83–98, 2000.

[2] F. Boussemart, F. Hémery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, Valencia (Spain), August 2004.

[3] L. Brisoux, É. Grégoire, and L. Saïs. Checking depth-limited consistency and inconsistency in knowledge-based systems. *International Journal of Intelligent Systems*, 16(3) :333–360, 2001.

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

[5] R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Annals of Mathematics and Artificial Intelligence*, 43(1) :35–50, 2005.

[6] G. Daydov, I. Davydova, and 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.

[7] DIMACS. Benchmarks on sat. ftp ://dimacs.rutgers.edu/pub/challenge/satisfiability/.

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

[9] H. Fleischner, O. Kullman, and S. Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1) :503–516, 2002.

[10] É. Grégoire and D. Ansart. Overcoming the christmas tree syndrome. *International Journal on Artificial Intelligence Tools (IJAIT'00)*, 9(2) :97–111, 2000.

[11] É. Grégoire, B. Mazure, and L. Saïs. Using failed local search for sat as an oracle for tackling harder a.i. problems more efficiently. In *Proceedings of the Tenth International Conference on Artificial Intelligence : Methodology, Systems, Applications (AIMSA'02)*, number 2443 in LNCS, pages 51–60, Varna (Bulgaria), 2002. Springer.

[12] Console L. Hamscher W. and de Kleer J., editors. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[13] H. Kautz, B. Selman, and D. McAllester. Walksat in the sat 2004 competition. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, 2004.

[14] M.H. Liffiton and K.A. Sakallah. On finding all minimally unsatisfiable subformulas. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 173–186, 2005.

[15] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, 2004.

[16] B. Mazure, L. Saïs, and É. Grégoire. A powerful heuristic to locate inconsistent kernels in knowledge-based systems. In *Proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96)*, pages 1265–1269, Granda (Spain), 1996.

[17] B. Mazure, L. Saïs, and É. Grégoire. Boosting complete techniques thanks to local search. *Annals of Mathematics and Artificial Intelligence*, 22 :319–322, 1998.

[18] 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 Confrence (DAC 2004)*, pages 518–523, 2004.

[19] C.H. Papadimitriou and Wolfe D. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37(1) :2–13, 1988.

[20] SATLIB. Benchmarks on sat. http ://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.

[21] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, Portofino (Italy), 2003.