Améliorer SAT dans le cadre incrémental

Gilles Audemard[1], **Armin Biere**[2], **Jean-Marie Lagniez**[1], **Laurent Simon**[3]

1. *Univ. Lille-Nord de France*
CRIL/CNRS UMR8188, Lens, F-62307, France
{audemard,lagniez}@cril.fr

2. *Univ. Bordeaux*
Labri/CNRS UMR58000, Talence F-33405, France
lsimon@labri.fr

3. *Institute for Formal Models and Verification, JKU Linz, Autriche*
biere@ju.at

RÉSUMÉ. *Les avancées spectaculaires obtenues dans le cadre de la résolution pratique du problème SAT ont rejailli bien au delà de ses frontières. Ainsi, à l'heure actuelle, de nombreux problèmes dont la classe de complexité est supérieure à NP peuvent être traités de manière pratique en se reposant directement ou indirectement sur l'utilisation de solveurs SAT. Dans un grand nombre de cas, la résolution de ces problèmes consiste à appeler un solveur SAT sur plusieurs instances analogues. Ce type de résolution, appelé résolution incrémentale de SAT, est en passe de devenir l'état de l'art dans bien des domaines. Il devient donc important de prendre en compte ce nouveau mode opératoire lors de la conception des nouveaux démonstrateurs SAT. Dans cet article, nous proposons d'améliorer le démonstrateur SAT* `Glucose` *afin d'en faire un démonstrateur incrémental efficace dans ce cadre précis. Pour cela, nous étendons la notion de qualité de clauses apprises dans le cas de démonstrateur incrémental basé sur l'utilisation intensive d'hypothèses. Afin de valider expérimentalement notre contribution, nous avons étudié ses performances sur une utilisation importante et typique des démonstrateurs SAT incrémentaux : la recherche de noyaux minimaux inconsistants. Nous pensons que ces améliorations peuvent directement bénéficier à la plupart des autres applications basées sur les démonstrateurs SAT incrémentaux.*

ABSTRACT. *The spectacular progresses obtained in the practical solving of SAT problems had a number of important impacts beyond its own frontiers. For instance, a number of recent applications explicitly rely on a new use of SAT solvers called incremental SAT. In this new mode of operation, the same solver can be called a thousand times on almost the same formula each time (with added/removed clauses between calls). This new framework allowed a number of new scale up in some critical applications beyond NP. In this paper, we focus on optimizing the*

core of incremental SAT solvers used in this particular context. We address a number of unseen problems this new use of SAT solvers opened. By playing on clause database cleaning, assumptions managements and other classical parameters, we show that our approach immediately and significantly improves an intensive assumption-based incremental SAT solving task: finding Minimal Unsatisfiable Set. We also present a compact way of encoding the activation/deactivation of clauses in this context. We believe this work could bring immediate benefits in a number of other applications relying on incremental SAT.

MOTS-CLÉS : *Démonstrateur Automatique SAT Incrémental.*

KEYWORDS: *Theorem Prover SAT Incremental.*

DOI:10.3166/HSP.2014.1-23 © ria Lavoisier

1. Introduction

Décider de la satisfiabilité d'une formule propositionnelle fait partie des questions fondamentales en intelligence artificielle et plus généralement en informatique. Ce problème, nommé *problème de SATisfiabilité* (ou *satisfaisabilité* SAT), a grandement été étudié pratiquement et théoriquement depuis les années 70. Étant donnée sa place centrale dans la théorie de la complexité (il s'agit du premier problème a avoir été prouvé NP-complet par Cook (Cook, 1971)), il est tout naturellement devenu l'un des problèmes de référence du domaine. Il permet intuitivement de capturer la difficulté d'un grand nombre d'autres problèmes, pouvant provenir d'applications très diverses.

Depuis l'avénement des solveurs SAT dit « modernes » (Moskewicz *et al.*, 2001 ; Eén, Sörensson, 2003a), initialement introduits par Marques-Silva et Sakallah (Silva, Sakallah, 1996), des problèmes de taille de plus en plus conséquente (des millions de clauses et de variables) sont résolus quotidiennement par les approches s'appuyant sur SAT. Dans la mesure où de nombreux problèmes peuvent se réduire au problème SAT, l'amélioration de ces solveurs a eu un impact direct pour de nombreux autres domaines d'application. Ainsi, à l'heure actuelle, les méthodes les plus efficaces pour la résolution de nombreux problèmes NP-complets consistent à utiliser un démonstrateur SAT, malgré parfois des méthodes *adhoc* anciennes et sophistiquées. De plus, ce gain de performance laisse envisager la possibilité de traiter, en pratique, à l'aide de solveurs SAT, des problèmes au-delà de NP, ce qui nécessite l'appel à plusieurs oracles NP (Biere *et al.*, 1999 ; Velev, Bryant, 2003). Dans certains cas, le nombre d'appels SAT peut être tout à fait conséquent.

Ainsi, depuis quelques années une nouvelle utilisation des démonstrateurs SAT, appelé "SAT incrémental", a vu le jour afin de répondre à ce nouveau type de besoin. Ce mécanisme, initialement proposé dès les premières versions de `Minisat` (Eén, Sörensson, 2003a), est aujourd'hui une technique utilisée dans des applications très diverses : vérification formelle bornée (Eén, Sörensson, 2003b), extraction de noyaux inconsistants (Nadel, 2010), MAXSAT (Fu, Malik, 2006), ou encore en vérification inductive (Bradley, 2012). Dans ce contexte, les démonstrateurs SAT ne sont pas lancés une seule fois sur des formules potentiellement énormes (typiquement déroulant

une discrétisation du temps), mais peuvent être appelés des milliers de fois sur des instances proches les unes des autres avec des clauses ajoutées et/ou supprimées à chaque appel. Ainsi, la proximité des différents problèmes laisse entrevoir la possibilité de ré-utiliser au maximum les informations pouvant être collectées entre les appels successifs au démonstrateur. Il faut cependant faire attention, car il est clair que si des clauses sont supprimées d'un appel à l'autre, les informations ayant été dérivées (clauses apprises) à partir de ces dernières ne peuvent plus être ré-utilisées. Pour palier ce problème, il est nécessaire d'ajouter un mécanisme d'utilisation particulier au démonstrateur SAT. Ce mécanisme est l'utilisation de littéraux nouveaux, appelés *hypothèses*, et qui vont contrôler les clauses ajoutées ou supprimées tout en permettant d'identifier les dépendances logiques devant être éventuellement supprimées. Ces hypothèses sont donc des littéraux spéciaux ajoutés à chaque clause et qui sont systématiquement affectés au début de la recherche.

Dans cet article, nous prenons un cas d'utilisation typique des démonstrateurs SAT incrémentaux : la recherche de noyaux minimum inconsistants (Grégoire *et al.*, 2006 ; Nadel, 2010 ; Ryvchin, Strichman, 2011) (MUS). Nous nous focalisons uniquement sur le moteur SAT afin d'améliorer les performances globales de l'extracteur de MUS construit au dessus du démonstrateur SAT `Glucose` (Audemard, Simon, 2009). Pour effectuer cela, nous avons choisi d'étudier expérimentalement notre travail sur `Muser` (Belov, Marques-Silva, 2011), qui est un extracteur MUS open source très efficace et facilement modifiable. Nous pensons que les améliorations induites pourront directement être utilisées dans d'autres applications typiques de SAT incrémental.

2. Le problème SAT et ses extensions

2.1. Notations et Rappels

Une formule CNF Σ est une conjonction (interprétée comme un ensemble) de *clauses*, où une clause est une disjonction (interprétée comme un ensemble) de *littéraux*. Un littéral est une variable positive x ou négative $\neg x$. Les deux littéraux x et $\neg x$ sont dit *complémentaires*. Une clause unitaire est une clause possédant un unique littéral (appelé *littéral unitaire*). Une clause vide, notée $\perp$, est interprétée comme fausse, tandis que la *formule* CNF *vide*, notée $\top$, est interprétée comme vraie. Une ensemble de littéraux est *complet* s'il contient un littéral pour chaque variable apparaissant dans Σ et est *fondamental* s'il ne contient pas un littéral et son complémentaire. Une *interprétation* $\mathcal{I}$ d'une formule booléenne Σ associe une valeur $\mathcal{I}(x)$ à certaines variables x de la formule Σ. Une interprétation peut être représentée par un ensemble de littéraux complet et fondamental. Un *modèle* d'une formule Σ est une interprétation $\mathcal{I}$ qui satisfait la formule, *i.e.* toutes les clauses de la formule. Enfin, SAT est le problème de décision qui consiste à vérifier si une formule Σ sous Forme Normale Conjonctive (CNF) possède ou non un modèle.

Algorithme 2.1 : Solveur de type CDCL

```
1  𝓘 = ∅ ;                                            /* interprétation */
2  d = 0 ;                                        /* niveau de décision */
3  Γ = ∅ ;                              /* ensemble des clauses apprises */
4  while (true) do
5  |  c = propagationUnitaire(Σ ∪ Γ,𝓘);
6  |  with (c ≠null) ;                                  /* Nouveau conflit */
7  |  then
8  |  |  γ =analyseConflit(Σ ∪ Γ,𝓘,c);
9  |  |  bl = calculRetourArriere(γ,𝓘);
10 |  |  if (bl < 0) then return UNSAT Γ = Γ ∪ {γ};
11 |  |  if (redémarrage()) then bl = 0 retourArriere(Σ ∪ Γ,𝓘,bl);
   |  |  /* Mise à jour de 𝓘 */
12 |  |  d = bl;
13 |  else
14 |  |  with (toutes les variables sont affectées) then
15 |  |  |  return SAT;
16 |  |  ℓ = choixLiteralDecision(Σ ∪ Γ);
17 |  |  d = d + 1;
18 |  |  𝓘 = 𝓘 ∪ {⟨ℓ@d⟩};
19 |
20 end
```

2.2. Démonstrateurs basés sur l'analyse de conflits (CDCL)

Nous présentons ici brièvement le schéma général d'un solveur SAT de type SAT (Moskewicz *et al.*, 2001 ; Eén, Sörensson, 2003a). Ces solveurs incorporent la propagation unitaire, une heuristique de choix de variable basée sur l'activité, une heuristique de choix de polarité, l'apprentissage de clauses basé sur l'analyse du graphe d'implication, une stratégie de redémarrage et une politique de réduction de la base de clauses apprises.

Les démonstrateurs SAT "*modernes*" sont basés sur le paradigme CDCL (*conflict driven clause learning*) (Moskewicz *et al.*, 2001). Bien qu'à l'origine, ils ont été proposés comme une extension de l'algorithme DPLL (Davis *et al.*, 1962) comprenant l'apprentissage de clauses (Silva, Sakallah, 1996 ; Zhang *et al.*, 2001), il est maintenant admis qu'ils doivent être considérés comme un mélange d'algorithmes de retour arrière et de moteurs de résolution. L'algorithme 2.1 donne le schéma général d'une méthode de type SAT (Moskewicz *et al.*, 2001). Une branche typique d'un tel solveur est une séquence de décisions (effectuées au moyen d'heuristiques dynamiques (Moskewicz *et al.*, 2001 ; Pipatsrisawat, Darwiche, 2007)), suivies de propagations, répétée jusqu'à ce qu'un conflit survienne. Chaque littéral de décision (lignes 18–20) est affecté à un niveau de décision (d), les littéraux qui se dé-

duisent (par propagation unitaire) de cette décision ont le même niveau ($l@d$ indique que le littéral l est affecté au niveau d). Une interprétation $\mathcal{I}$ peut donc être écrite $\mathcal{I} = \{\langle \ell_{k_{1_1}}@1, \ldots \ell_{k_{1_i}}@1\rangle, \langle \ell_{k_{2_1}}@2, \ldots \ell_{k_{2_j}}@2\rangle \ldots \langle \ell_{k_{d_1}}@d, \ldots \ell_{k_{d_t}}@d\rangle\}$. Les $\langle$ et $\rangle$ partitionnent l'interprétation $\mathcal{I}$ suivant les différents niveaux de décision et les variables $\ell_{k_{1_1}}, \ell_{k_{2_1}}, \ldots$ sont les variables de décision de chaque niveau. Si tous les littéraux sont affectés, alors $\mathcal{I}$ est un modèle de Σ (lignes 16–17). À chaque fois qu'un conflit est atteint par propagation unitaire (c est alors la clause falsifiée, lignes 6–7), un nogood γ est calculé (ligne 8) en utilisant une méthode donnée, le plus souvent le premier UIP (Unique Implication Point) (Zhang *et al.*, 2001) et un niveau de backjump *bl* est calculé (ligne 13). À ce moment, on peut avoir prouvé l'incohérence de la formule. Si ce n'est pas le cas, on fait un saut arrière (on enlève de l'interprétation $\mathcal{I}$ des séquences de littéraux) et le nouveau niveau de décision devient égal au niveau de backjump (lignes 13–14). Enfin, de temps à autre, les solveurs SAT forcent des redémarrages (Biere, 2008 ; Audemard, Simon, 2012) et dans ce cas, on remonte tout en haut de l'arbre de recherche (ligne 12). Nous détaillons par la suite l'analyse de conflits qui est importante pour la compréhension de cet article. Pour le reste, le lecteur, intéressé par les nombreuses recherches autour des solveurs SAT, pourra se référer à (Marques-Silva *et al.*, 2009).

Le but de l'analyse de conflit est de trouver un ensemble de littéraux responsables d'un conflit, i.e. d'une clause falsifiée. Au moyen d'une nouvelle clause (un « *nogood* » ou encore *clause assertive*) obtenue par résolution, cette analyse de conflit indique au solveur qu'il n'existe pas de solution dans un certain espace de recherche et propose un nouvel espace de recherche à visiter pour la suite. Ce nogood va alors être ajouté à la formule initiale. Dès lors, le solveur ne pourra plus être confronté au même échec. Le schéma d'analyse de conflits le plus couramment utilisé est le UIP «Unique Implication Point» (Zhang *et al.*, 2001).

Considérons la formule CNF suivante :

$c_1 = x_1 \vee x_2$ $c_5 = \neg x_6 \vee x_7 \vee x_8$ $c_9 = x_{12} \vee \neg x_{13}$
$c_2 = \neg x_2 \vee \neg x_3$ $c_6 = \neg x_4 \vee x_8 \vee x_9$ $c_{10} = x_{12} \vee x_{14}$
$c_3 = \neg x_2 \vee \neg x_4 \vee \neg x_5$ $c_7 = \neg x_9 \vee x_{10} \vee x_{11}$ $c_{11} = x_{12} \vee \neg x_6 \vee x_{15}$
$c_4 = x_3 \vee x_5 \vee x_6$ $c_8 = x_8 \vee \neg x_{11} \vee \neg x_{12}$ $c_{12} = x_{13} \vee \neg x_{14} \vee \neg x_{16}$
$c_{13} = \neg x_{14} \vee x_{15} \vee x_{16}$

On considère également l'interprétation partielle obtenue par les littéraux de décision $\neg x_1@1$, $x_4@2$, $\neg x_7@3$ et $x_{10}@4$. On a donc $\mathcal{I} = \{\langle \neg x_1@1, x_2@1, \neg x_3@1\rangle, \langle x_4@2, \neg x_5@2, x_6@2\rangle, \langle \neg x_7@3, \neg x_8@3, \neg x_9@3\rangle, \langle \neg x_{10}@4, x_{11}@4, \neg x_{12}@4, \neg x_{13}@4, x_{14}@4, x_{15}@4, x_{16}@4\rangle\}$, Les littéraux grisés dans la formule sont les littéraux falsifiés par l'interprétation $\mathcal{I}$. L'interprétation $\mathcal{I}$ falsifie la formule (clause c_{13}).

Le nogood responsable du conflit va être généré en effectuant des résolvantes entre les clauses responsables du conflit (utilisé durant la propagation unitaire) en remontant de celui-ci vers la variable de décision du dernier niveau jusqu'à obtenir une clause

contenant un seul littéral du dernier niveau de décision (c'est le rôle de la fonction `analyseConflit`). Sur notre exemple, on a :

$$
\begin{aligned}
c_1^{\otimes} &= c_{13} \otimes_{x_{16}} c_{12} &&= x_{13}@4 \vee \neg x_{14}@4 \vee x_{15}@4 \\
c_2^{\otimes} &= c_1^{\otimes} \otimes_{x_{15}} c_{11} &&= \neg x_6@2 \vee x_{12}@4 \vee x_{13}@4 \vee \neg x_{14}@4 \\
c_3^{\otimes} &= c_2^{\otimes} \otimes_{x_{14}} c_{10} &&= \neg x_6@2 \vee x_{12}@4 \vee x_{13}@4 \\
c_4^{\otimes} &= c_3^{\otimes} \otimes_{x_{13}} c_9 &&= \neg x_6@2 \vee x_{12}@4
\end{aligned}
$$

La dernière clause générée est la première résolvante possédant un et un seul littéral du dernier niveau de décision (x_{12}) qui est également le premier UIP. C'est cette clause, dite *assertive*, qui va être apprise par le solveur. Non seulement cet apprentissage permettra d'éviter d'atteindre à nouveau le même échec, mais il va également permettre de faire un retour arrière. En effet, la connaissance de cette clause falsifiée au niveau de décision 4, nous permet d'affirmer que x_{12} aurait du être affecté à *vrai* par propagation unitaire dès l'affectation de x_6, donc au niveau 2 (c'est le rôle de la fonction `calculRetourArriere`). Le solveur va donc défaire les affectations des niveaux de décisions 4 et 3 et affecter x_{12} à vrai au niveau 2 en lui donnant comme raison la nouvelle clause $c_4^{\otimes}$. Ainsi, et contrairement à un solveur DPLL classique, le solveur SAT peut effectuer des sauts arrière (c'est le rôle de la fonction `retourArriere`).

A chaque conflit, les solveurs SAT apprennent donc une nouvelle Clause dont la taille peut être relativement importante. Dès lors, ils se heurtent à deux problèmes majeurs : l'utilisation de la mémoire et le temps utilisé pour la propagation unitaire. Pour lutter contre cela, la base de clauses apprises est régulièrement nettoyée (Goldberg, Novikov, 2002 ; Audemard, Simon, 2009 ; Audemard *et al.*, 2011). Différents scores sont utilisées pour différencier les clauses que l'on considère importantes pour la recherche (et que l'on va donc conserver) sont utilisés (Eén, Sörensson, 2003a). Notre démonstrateur `Glucose` utilise la notion de LBD comme score (Audemard, Simon, 2009). Ce score, statique, compte le nombre de niveaux de décisions différents qui apparaissent dans la clause apprise (la clause apprise $c_4^{\otimes}$ a un LBD égal à 2).

2.3. Noyaux inconsistants minimaux

Lorsqu'une formule propositionnelle est inconsistante, il peut être intéressant d'exhiber les raisons de cette inconsistance, *i.e.* un sous-ensemble de clauses de la formule initiale qui suffit à rendre cette formule inconsistante. Ce sous-ensemble peut permettre de localiser l'origine d'un défaut de conception, comme c'est le cas, par exemple, avec les instances FPGA, issues des circuits (Nam *et al.*, 2004). Ainsi, un noyau minimalement inconsistant (MUS, pour Minimum Unsatisfiable Core) est un sous-ensemble de la formule initiale qui est inconsistant et pour lequel la suppression d'une clause restaure la satisfiabilité. Plus formellement, on a :

DÉFINITION 1. — *Un MUS Ψ d'une formule CNF Γ est un ensemble de clauses tel que*

– $\Psi \subseteq \Gamma$
– Ψ *est insatisfiable*
– $\forall \Delta \subsetneq \Gamma$, Δ *est satisfiable*

EXEMPLE 2. — On considère la formule suivante :

$$\begin{array}{lll} x_1 \vee x_2 \vee x_3 & x_1 \vee \neg x_2 & x_1 \vee \neg x_3 \\ \neg x_1 \vee x_2 \vee x_3 & x_1 \vee x_4 & \neg x_2 \vee x_3 \vee x_4 \\ \neg x_1 \vee \neg x_2 & \neg x_1 \vee \neg x_3 & \neg x_1 \vee \neg x_3 \vee x_4 \end{array}$$

Cette formule est insatisfiable et l'ensemble des clauses non grisé est un MUS de cette formule.

□

La recherche d'un MUS (une formule peut contenir plusieurs MUS de taille différente) est un problème difficile, prouvé DP-complet (Papadimitriou, Wolfe, 1988). Il faut en effet effectuer plusieurs appels à un démonstrateur SAT pour exhiber un MUS. Malgré tout, différentes techniques existent pour extraire efficacement un MUS. On peut les classer en trois grandes catégories. Toutes d'abord les approches destructives, qui partent de la formule de départ et suppriment des clauses au fur et à mesure des appels à l'oracle SAT (Bakker *et al.*, 1993). On trouve par ailleurs les approches dites constructives, qui partent de l'ensemble vide et ajoutent des clauses au fur et à mesure jusqu'à obtenir un MUS (Siqueira N., Puget, 1988). Enfin, les approches dichotomiques alternent entre ajouts et suppressions de clauses (Hemery *et al.*, 2006). Plusieurs techniques permettent d'optimiser le nombre d 'appels à l'oracle SAT nécessaire (rotation de modèle (Belov, Marques-Silva, 2011), clause critique (Grégoire *et al.*, 2006), pré-traitement (Belov *et al.*, 2013)). Nous ne rentrerons pas dans les détails de toutes ses approches, le lecteur intéressé pourra se référer à (Belov *et al.*, 2012). Il est par contre très important de noter que toutes ses approches nécessitent des appels à un démonstrateur SAT sur des sous-formules de la formule initiale afin de déterminer l'apartenance d'une clause au MUS en cours de construction.

3. SAT incremental

Comme évoqué précédemment, dans de nombreux domaines où SAT est utilisé, il y a en réalité plusieurs appels à l'oracle SAT sur des formules relativement proches les unes des autres. C'est le cas en vérification formelle bornée (BMC) (Biere *et al.*, 1999) ou encore en planification (Kautz, Selman, 1992). Dans ce dernier cas, un ensemble de clauses Σ_I code l'état initial, un ensemble de clauses Σ_b code le but et l'encodage traduit la tentative de construire un plan de taille k. Un ensemble de clauses Σ_k code alors les pré-conditions, post-conditions... Si le problème $\Sigma = \Sigma_i \wedge \Sigma_b \wedge \Sigma_k$ est satisfiable, on a alors déterminé un plan de taille k. Dans le cas contraire, on doit incrémenter la borne k et tester la satisfiabilité de $\Sigma' = \Sigma_i \wedge \Sigma_b \wedge \Sigma_{k+1}$. En règle

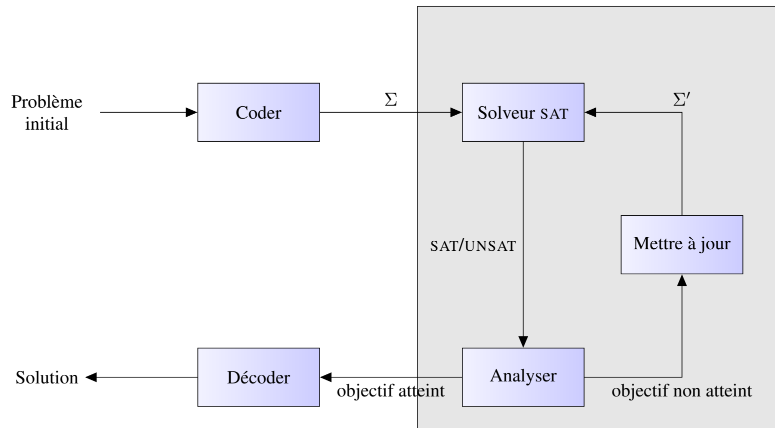


FIGURE *1. Fonctionnement général d'un solveur SAT incrémental*

générale, et comparativement à la formule Σ, la formule Σ' va avoir un ensemble de clauses supplémentaire Σ^+ et un ensemble de clauses en moins Σ^-, et on a donc $\Sigma' = \Sigma \backslash \Sigma^- \cup \Sigma^+$.

De plus, si on note que les démonstrateurs CDCL se basent essentiellement sur le passé pour avancer dans la recherche au moyen des heuristiques (VSIDS), des redémarrages dynamiques et surtout de l'apprentissage de clauses, il parait assez clair que, pour viser les meilleures performances possibles, il faut laisser le démonstrateur SAT *en vie* entre deux appels successifs sur ces formules relativement proches et lui donner la possibilité de :

– Ajouter des variables
– Ajouter des clauses
– Supprimer des clauses

C'est cela qui est habituellement appelé SAT incrémental et qui a initialement été proposé par les auteurs de `Minisat` (Eén, Sörensson, 2003a). La figure 1 montre le fonctionnement général d'un démonstrateur SAT incrémental.

L'ajout de nouvelles clauses et de nouvelles variables sont des opérations extrêmement faciles à réaliser et sont nativement incluses dans tous les démonstrateurs CDCL. La suppression de clauses initiales est plus complexe, mais peut être basée sur l'utilisation astucieuse d'hypothèses. Expliquons comment.

Lorsqu'une clause est supprimée de la formule courante, il n'est alors pas possible de ré-utiliser directement les clauses apprises générées à l'aide de ces dernières. Pour pallier ce problème, il est possible d'ajouter des hypothèses aux démonstrateurs. Celles-ci sont caractérisées par un ensemble $\mathcal{A}$ de littéraux qui sont choisis comme

variables de décisions et affectés à vrai avant toute autre variable de décision (une autre possibilité est donnée dans (Nadel, Ryvchin, 2012)). Il existe donc un niveau de décision par hypothèse, la recherche ne commençant réellement que lorsque toutes ses hypothèses ont été affectées et on a alors comme premier niveau de décision, $l_c = |\mathcal{A}| + 1$.

Quand une hypothèse est utilisée pour activer/désactiver une clause, celle-ci doit alors être associée à une nouvelle variable (s_i) du problème, qui est appelée *sélecteur* pour cette clause. La variable s_i est alors ajoutée à la clause. Si le littéral associé est faux (resp. vrai) suivant les hypothèses courantes, alors la clause est activée (resp. désactivée). Les sélecteurs n'apparaissant que positivement dans la formule, les clauses apprises obtenues en utilisant le processus d'analyse de conflits introduit dans la section précédente gardent donc une empreinte (le sélecteur s_i associé) de toutes les clauses initiales de la formule utilisées pour les produire. Ainsi, en affectant à vrai le sélecteur s_i, on désactive non seulement la clause initiale associée mais également toutes les clauses apprises dérivées à partir de celle-ci. Ceci est illustré ci-dessous :

EXEMPLE 3. — Reprenons la formule de la section 2.2. Nous ajoutons un sélecteur s_i à chaque clause c_i. Nous choisissons d'activer toutes les clauses, comme nous sommes au début de la recherche, la propagation des sélecteurs n'entraine aucune autre propagation et on a donc $\mathcal{I} = \{\langle \neg s_1@1 \rangle \ldots \langle \neg s_{13}@13 \rangle$ et l'on obtient alors exactement la formule initiale. En reprenant la même interprétation que dans la section 2.2, seulement les niveaux des variables de décision changent et, ainsi, $\neg x_1$ est affecté au niveau 14. On obtient donc le même conflit que l'on peut analyser, pour génèrer la clause assertive :

$$c_1^{\otimes} = c_{13} \otimes_{x_{16}} c_{12} = x_{13}@17 \vee \neg x_{14}@17 \vee x_{15}@17 \vee s_{13}@13 \vee s_{12}@12$$

$$c_2^{\otimes} = c_1^{\otimes} \otimes_{x_{15}} c_{11} = \neg x_6@15 \vee x_{12}@17 \vee x_{13}@17 \vee \neg x_{14}@17 \vee s_{13}@13 \vee s_{12}@12 \vee s_{11}@11$$

$$c_3^{\otimes} = c_2^{\otimes} \otimes_{x_{14}} c_{10} = \neg x_6@15 \vee x_{12}@17 \vee x_{13}@17 \vee s_{13}@13 \vee s_{12}@12 \vee s_{11}@11 \vee s_{10}@10$$

$$c_4^{\otimes} = c_3^{\otimes} \otimes_{x_{13}} c_9 = \neg x_6@15 \vee x_{12}@17 \vee s_{13}@13 \vee s_{12}@12 \vee s_{11}@11 \vee s_{10}@10 \vee s_9@9$$

Les clauses apprises contiennent donc tous les sélecteurs des clauses initiales utilisées pour les générer. Sur ce simple exemple la clause $c_4^{\otimes}$ contient donc 5 littéraux sélecteurs.

Si, lors d'un nouvel appel à l'oracle SAT, on décide de désactiver la clause c_{13}, on choisira d'affecter au niveau 13 le litéral s_{13}. On désactivera par là même la clause c_{13} mais également la clause $c_4^{\otimes}$, car elles sont toutes les deux satisfaites par ce litéral. □

Si, pendant la recherche, un nogood nécessite d'affecter une hypothèse dans la phase inverse de celle choisie initialement, le problème est alors insatisfiable par rapport à ces hypothèses. On peut, en conséquence et en raisonnant à partir de ce dernier nogood, extraire le sous-ensemble des hypothèses réellement responsables du conflit.

De plus amples détails sur SAT incrémental sont disponibles dans (Soh, 2011 ; Wieringa, 2014 ; Nadel, Ryvchin, 2012).

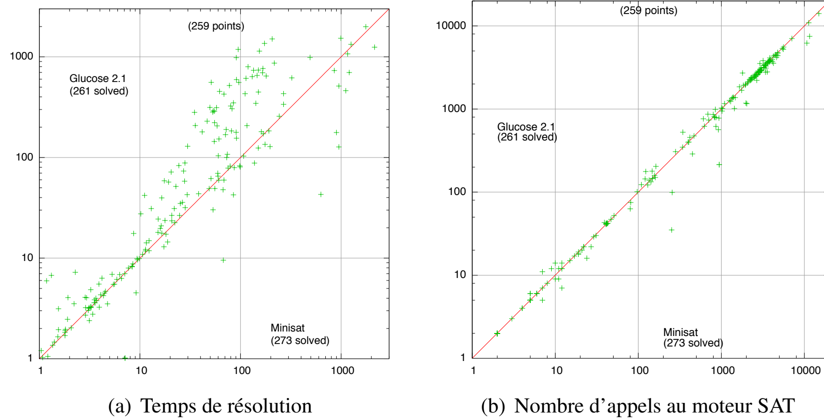
3.1. Application aux MUS

L'application de SAT incrémental à l'extraction des MUS est naturelle pour tous les types d'approches (destructive, constructive ou dichotomique) définies dans la section précédente. En effet, ces dernières doivent activer et désactiver des clauses à chaque appel à l'oracle SAT.

Un sélecteur est ajouté à chaque clause initiale et de nombreux appels SAT sont effectués en activant/désactivant les clauses initiales afin de détecter les MUS. Cette application est donc typique (et difficile) pour les démonstrateurs SAT incrémentaux : le nombre d'hypothèses est très important (égal aux nombre de clauses initiales) et de nombreux appels au solveur SAT sont effectués. Cette application rentre donc bien dans notre objectif principal, à savoir l'amélioration des moteurs incrémentaux. Nous avons basé notre travail sur `Muser` (Belov, Marques-Silva, 2011), un extracteur de MUS open source et efficace. Rappelons que nous ne travaillons pas du tout du coté des algorithmes de recherche de MUS, mais uniquement du coté du moteur SAT. Nous espérons qu'en améliorant les performances des moteurs SAT (vu comme des boites noires) nous pouvons améliorer les performances de l'extracteur de MUS. Nous avons choisi d'utiliser `Muser` avec ces options par défaut, i.e. l'algorithme hybride est utilisé (essentiellement basé sur l'oubli de clauses) avec l'affinage des ensembles de clauses et la rotation de modèles (voir (Belov, Marques-Silva, 2011) pour plus de détails). Nous utilisons les 300 instances issues de la catégorie MUS de la compétition SAT 2011. Toutes les expérimentations ont été réalisées sur un cluster Intel XEON X5550 quatre coeurs 2.66 GHz avec 32Go de RAM avec un temps CPU limite de 2400 secondes.

3.2. Utiliser Glucose dans le problème d'extraction de noyau minimum inconsistant

Si, entre chaque appel au démonstrateur SAT, nous voulons ré-utiliser des clauses apprises, il est semble naturel de chercher à déterminer quelles clauses seront utiles pour les futurs appels. Dans ce contexte, nous avons choisi d'utiliser le démonstrateur `Glucose` (Audemard, Simon, 2009), une variante de `Minisat` (Eén, Sörensson, 2003a), et utilisant le score LBD pour scorer les clauses apprises utiles à la recherche. Nous avons donc démarré notre travail en comparant `Glucose` contre `Minisat` comme moteur SAT de `Muser`. Les résultats, étonnamment mauvais, sont résumés figure 3.1 (Chaque point des figures de ce type correspond à une instance ; moins il y a de points dans la région d'une méthode, meilleure est cette dernière comparée à l'autre. De plus, pour chaque figure de ce type, nous donnons le nombre d'instances résolues pour chaque méthode et le nombre d'instances résolues par les deux (259 pour la figure 2(a)). Malgré des résultats très mauvais pour `Glucose`, les deux parties de la figure 3.1 sont intéressantes à observer. Le temps CPU est clairement en



(a) Temps de résolution (b) Nombre d'appels au moteur SAT

FIGURE 2. *Comparaison de* `Glucose` *2.1 et de* `Minisat` *comme moteur de* `Muser`*. La figure de gauche est relative au temps CPU (en secondes), la figure de droite est relative au nombre d'appels au moteur SAT*

faveur de `Minisat`. Mais la figure 2(b), comparant les deux approches du point de vue du nombre d'appels au moteur SAT, montre que la plupart du temps, les deux approches nécessitent approximativement le même nombre d'appels SAT, avec dans certains cas un nombre beaucoup plus faible lorsque `Glucose` est utilisé. Si nous regardons plus en détails, sur les 259 instances résolues par `Minisat` et `Glucose`, 103 nécessitent moins d'appels au démonstrateur si `Glucose` est utilisé, 68 plus d'appels si `Glucose` est utilisé, et dans 88 cas on obtient le même nombre d'appels. Si nous arrivons à réduire le temps nécessaire pour chaque appel au moteur `Glucose`, nous pouvons donc espérer améliorer `Muser`.

Cependant, comment interpréter des résultats aussi décevants ? `Glucose` met à jour les scores des clauses durant le processus de propagation unitaire. Avec des clauses apprises pouvant avoir des milliers de littéraux (les sélecteurs apparaissent dans ces clauses), cela peut avoir un surcout prohibitif. Néanmoins, il est aussi important de noter que `Minisat` a du mal à résoudre des problèmes de recherche de MUS difficiles (observation non reportée), en partie à cause de problèmes de mémoire. Sur de tels problèmes, le moteur SAT sera lancé un grand nombre de fois avec de nombreuses clauses apprises et de nombreux sélecteurs par clause. La capacité de `Glucose` à maintenir efficacement les *bonnes* clauses apprises peut alors s'avérer cruciale. Nous montrons dans la partie suivante comment adapter les mécanismes de `Glucose` dans le cadre incrémental, avec, au final, des améliorations sensibles par rapport à `Minisat`.

4. Améliorer les moteurs SAT incrémentaux

LBD et sélecteurs

Rappelons tout d'abord que `Muser` introduit une variable sélecteur par clause initiale et que chaque clause apprise va posséder tous les sélecteurs associés à toutes les clauses initiales utilisées dans le processus de résolution pour les générer. Ainsi, le nombre de sélecteurs dans une clause apprise peut être extrêmement important. Cela est particulièrement visible sur la Figure 3. Pour chaque instance résolue, le nombre moyen de variables initiales et le nombre moyen de sélecteurs des clauses apprises est comparé. Comme nous l'avons rappelé précédemment, dans `Minisat` chaque hypothèse possède son propre niveau de décision (excepté lorsque elle est propagée par une précédente hypothèse). Ainsi, dans de nombreux cas, le score LBD des clauses va être dominé par le nombre de sélecteurs que possède cette clause.

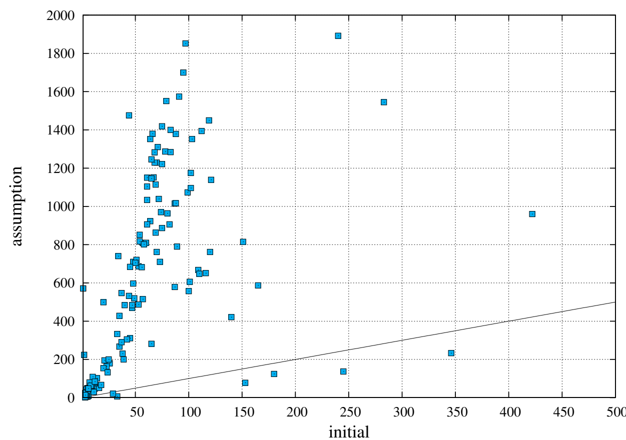


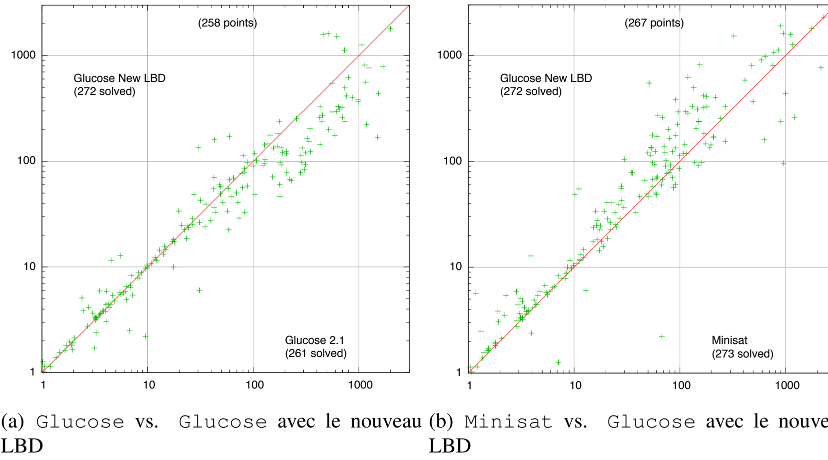
FIGURE *3. Pour chaque instance, nombre moyen de variables initiales et de sélecteurs dans les clauses apprises.*

Par conséquent, le LBD va très souvent valoir approximativement la taille de la clause. Si on ajoute à cela, le fait que le score LBD est très discriminant (les clauses de LBD égal à $n + 1$ sont significativement moins importantes que celles de LBD égal à n), le LBD joue un rôle très important et doit être calculé au plus juste. Nous proposons d'adapter simplement ce score en ne prenant pas en compte les sélecteurs dans son calcul. Ce nouveau score LBD est simplement nommé "Nouveau LBD".

La table 1 montre quelques statistiques relatives aux remarques précédentes. Sur 4 instances représentatives, nous détaillons les score LBD initiaux et les scores des nouveaux LBD. Comme nous pouvons le constater, il est clair que l'utilisation du LBD initial n'est pas du tout adapté dans le cadre de moteurs SAT utilisant les sélecteurs. La valeur des LBD se rapproche significativement de la taille des clauses apprises.

		LBD					Nouveau LBD				
			Taille		LBD			Taille		LBD	
Instance	#C	T	avg	max	avg	max	T	avg	max	avg	max
fdmus_b21_96	8541	29	1145	5980	1095	5945	11	972	6391	8	71
longmult6	8853	46	694	3104	672	3013	14	627	2997	11	61
dump_vc950	360419	110	522	36309	498	35873	67	1048	36491	8	307
g7n	15110	190	1098	16338	1049	16268	75	1729	17840	27	160

TABLE *1. Pour quelques instances significatives, nous reportons le nombre de clauses (#C), et pour chaque définition de LBD (initiale et nouvelle), nous donnons aussi le temps nécessaire pour calculer le MUS, la taille moyenne et max des clauses apprises et la valeur moyenne et max des LBD.*



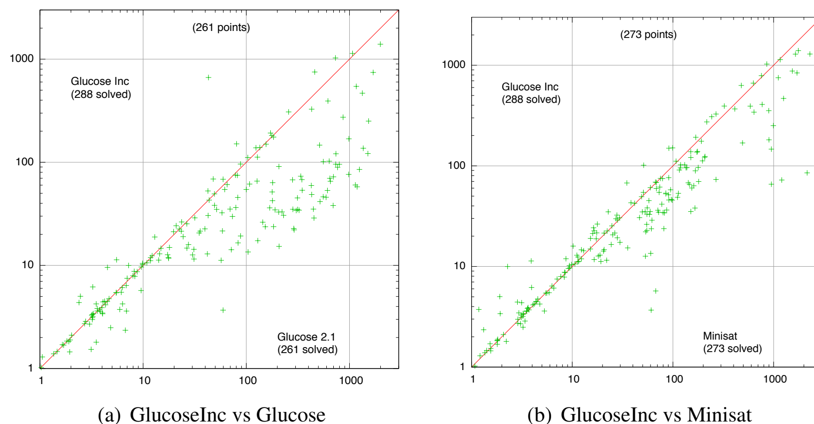
(a) `Glucose` vs. `Glucose` avec le nouveau LBD

(b) `Minisat` vs. `Glucose` avec le nouveau LBD

FIGURE *4. Comparaison, au sein de `Muser`, des performances de `Glucose` (2.1) en le nouveau LBD. La figure 4(a) compare cette version avec `Glucose` classique. La figure 4(b) compare cette version avec `Minisat`.*

Par contre, en utilisant la nouvelle définition des LBD, nous pouvons constater que le temps nécessaire est fortement amélioré. De plus, la valeur des LBD n'est plus du tout reliée à la taille des clauses apprises, mais bien plus petite.

Malgré tout, comme nous pouvons le constater sur la figure 4, même si nous améliorons les performances de `Glucose`, cette nouvelle version obtient des résultats très proches de `Minisat` (par rapport au nombre d'instances résolues), mais plus lente. À ce niveau de nos expérimentations, les résultats sont assez décevants : `Glucose` est supposé être beaucoup plus efficace que `Minisat` dans le cadre non incrémental. Afin d'en améliorer les performances globales, nous proposons d'y apporter de nouvelles optimisations.



(a) GlucoseInc vs Glucose

(b) GlucoseInc vs Minisat

FIGURE 5. *Comparaison, au sein de* `Muser`*, des performances de* `GlucoseInc`*. La figure 4(a) compare cette version avec* `Glucose` *classique. La figure 4(b) compare cette version avec* `Minisat`*.*

Amélioration des performances

Comme le montre le tableau 1, les clauses apprises peuvent rapidement devenir extrêmement grandes. Par exemple, les clauses apprises ont une taille moyenne de 1729 littéraux pour l'instance `g7n` (avec certaines clauses contenant quelques 10 000 littéraux !). Dès lors, même une opération simple comme le parcours d'une clause peut rapidement avoir un surcout prohibitif. Ceci est très problématique, puisque de nombreuses opérations fondamentales des solveurs SAT sont justement basées sur le parcours de clauses (propagation unitaire, calcul du LBD, calcul des clauses apprises, suppression des clauses satisfaites...). Nous proposons des améliorations spécialement dédiées à ces opération simples mais cruciales.

Parcours efficaces des clauses contenant des sélecteurs

Comme nous venons de le voir, dans de nombreux cas, les clauses apprises vont contenir majoritairement des variables sélecteurs qui n'ont pas à être prises en compte dans le calcul du nouveau LBD. Comme ces littéraux peuvent être utilisés comme témoin (*watchers*) nous ne pouvons malheureusement partitionner ces derniers en deux sous ensembles indépendants (sélecteurs/non sélecteurs) : nous devons être à même de visiter ces littéraux particuliers à volonté. Il est néanmoins possible d'enregistrer dans chaque clause sa taille et le nombre de sélecteurs qu'elle contient. De plus, lors de la création de chaque clause, nous poussons les sélecteurs à la fin. Nous pouvons espérer qu'un tel arrangement accélérera la mise à jour des LBDs : nous pouvons stopper le parcours de la clause dès lors que tous le littéraux initiaux ont été visités.

Accéleration de la propagation

Si les modifications des structures de données décrites précédemment permettent d'améliorer le calcul de la mise à jour du LBD, nous avons également à faire face à un autre problème : la propagation unitaire nécessite également de parcourir les clauses pour détecter de nouveaux littéraux témoins. Supposons que, durant la propagation d'une hypothèse (sélecteur), on cherche un nouveau témoin pour une clause c. Supposons également que le nouveau témoin s_i de la clause c est choisi parmi les autres selecteurs. Ainsi, si s_i est également choisi comme hypothèse (tous les sélecteurs le sont au fur et à mesure), alors la clause c sera à nouveau parcourue. Ceci peut facilement être évité : Lors de la propagation d'une hypothèse, nous parcourons la clause en entier pour trouver un nouveau témoin qui est vrai (la clause est alors satisfaite) ou qui n'est pas un sélecteur. De cette manière, nous espérons limiter le nombre de clauses à visiter lors de la recherche de nouveaux littéraux témoins. De plus, comme `Glucose` effectue de nombreux redémarrages, nous limitons également le retour arrière pour qu'il ne remonte pas au dessus du premier niveau de décision qui n'est pas une hypothèse.

Simplification de la base de clauses

La dernière, et importante, modification que nous avons introduite est reliée à la suppression définitive des clauses satisfaites. C'est très important dans le cas de `Muser` : à chaque fois que l'on détermine qu'une clause est hors d'un MUS, son sélecteur associé est définitivement affectée à vrai (au nivau de décision 0). Ainsi, toutes les clauses apprises où il apparait sont également définitivement satisfaites. Parcourir toutes les clauses apprises pour détecter ces littéraux satisfaits peut s'avérer contre-productif. Dans cette nouvelle version de `Glucose`, nous supprimons uniquement les clauses apprises dont un des deux littéraux témoins est définitivement satisfait. Comme nous avons modifié le processus de propagation unitaire (voir plus haut), nous pouvons espérer que cela soit suffisant pour supprimer la majorité des clauses apprises définitivement satisfaites.

4.1. Factoriser les hypothèses

Comme nous avons déjà pu le voir précédemment, la résolution du problème SAT sous hypothèses conduit à la génération de clauses apprises contenant de nombreux littéraux. Ainsi, comme remarqué dans la figure 3, les clauses apprises ont tendances à avoir une taille moyenne importante. Nous décrivons maintenant une approche qui utilise des compteurs et effectue des permutations afin de considérer de manière paresseuse (*lazy*) ces hypothèses.

Pour cela, nous nous inspirons de la résolution étendue (Huang, 2010 ; Audemard *et al.*, 2010) afin de factoriser les hypothèses présentes dans les clauses apprises.

4.1.1. Graphe de définition

Lorsqu'une clause est apprise nous proposons de remplacer la « partie hypothèse » pas un nouveau littéral, appelé *abréviation*. Cette dernière est constituée de l'ensemble des littéraux hypothèses de la clause apprise. Afin de sauvegarder les relations entre les différentes abréviation nous sauvegardons leur définition dans un *graphe de définition* de la manière suivante.

$$(p_1 \vee \cdots \vee p_n \vee a_1 \vee \cdots \vee a_m)$$

est factorisée en

$$(p_1 \vee \cdots \vee p_n \vee \ell) \qquad \text{et} \qquad \ell \mapsto \underbrace{a_1 \vee \cdots \vee a_m}_{\mathcal{G}[\ell]}$$

FIGURE 6. *Factoriser les hypothèses en introduisant un nouveau littéral d'abréviation ℓ.*

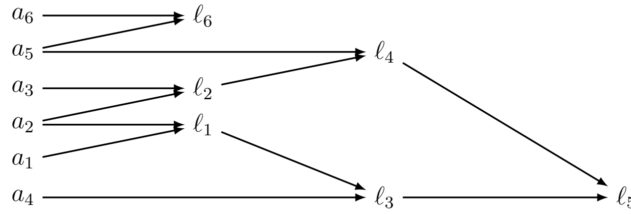
Soit $p_1 \vee \cdots \vee p_n \vee a_1 \vee \cdots \vee a_m$ une nouvelle clause apprise, où $p_1, \ldots, p_n$ sont des littéraux initiaux et $a_1, \ldots, a_m$ sont soit des *assumptions* ou des abréviations. Au lieu de la garder telle qu'elle, un nouveau littéral ℓ est créé la clause est modifiée de manière à incorporer le littéral ℓ à la place des hypothèses de celle-ci. Nous obtenons donc la clause $p_1 \vee \cdots \vee p_n \vee \ell$. Puis, nous sauvegardons $a_1 \vee \cdots \vee a_m$ comme une étant une définition $\mathcal{G}[\ell]$ de ℓ dans le graphe de définition $\mathcal{G}$ (voir Fig. 6). Remarquons que lorsque $m \leq 1$ ce remplacement n'a pas de sens et donc la clause apprise n'a pas besoin d'être modifiée.

Considérons l'exemple de la figure 7 qui illustre le comportement de notre approche sur l'exécution d'un solveur SAT incrémental avec les hypothèses $a_1, \ldots, \overline{a_6}$. La partie gauche de la figure 7(a) illustre l'ensemble des clauses apprises α_1, …, α_7 par le solveur à travers ces différentes appels, où $p_1, \ldots, p_7$ sont les littéraux initiaux et $a_1, \ldots, a_6$ sont les littéraux hypothèses. Afin de simplifier l'exemple nous ne reportons pas dans les antécédents l'ensemble des clauses utilisées pour la génération des clauses apprises. Par exemple, la clause α_3 est dérivée par résolution *via* la clause α_1 et des clauses non reportées de la formule originale (les "...").

Le résultat de l'introduction d'abréviations pour la factorisation des clauses apprises est reporté dans la partie droite de la figure 7(a). La première clause α_1 est factorisée en α'_1 et la définition $a_1 \vee a_2$ du nouveau littéral abréviation ℓ_1 est ajoutée au graphe de définition reporté dans la figure 7(b). Cette sauvegarde se traduit par l'ajout dans le graphe $\mathcal{G}$ du nœud ℓ_1 et s'ils ne sont pas déjà présents, des nœuds a_1 et a_2. ℓ_1 est lié avec sa définition par les deux arcs entrants (a_1, ℓ_1) et (a_2, ℓ_1). Remarquons aussi que $\alpha'_5 = \alpha_5$, puisqu'il n'y a qu'un seul littéral non-initial, ceci (comme discuté précédemment) permet de réduire le nombre de nouveaux littéraux à introduire. Pour terminer, remarquons que comme pour ℓ_3, ℓ_4 ou ℓ_5, certaines définitions

clauses apprises	antécédents		clauses factorisées
$\alpha_1 : p_2 \vee p_7 \vee a_1 \vee a_2$	$\{\ldots\}$		$\alpha'_1 : p_2 \vee p_7 \vee \ell_1$
$\alpha_2 : p_2 \vee a_2 \vee a_3$	$\{\ldots\}$		$\alpha'_2 : p_2 \vee \ell_2$
$\alpha_3 : p_7 \vee p_4 \vee \overline{p_6} \vee a_1 \vee a_2 \vee a_4$	$\{\alpha_1, \ldots\}$	*factorisation*	$\alpha'_3 : p_7 \vee p_4 \vee \overline{p_6} \vee \ell_3$
$\alpha_4 : p_6 \vee p_8 \vee a_3 \vee a_2 \vee a_5$	$\{\alpha_2, \ldots\}$	$\Longrightarrow$	$\alpha'_4 : p_6 \vee p_8 \vee \ell_4$
$\alpha_5 : p_2 \vee p_5 \vee a_2$	$\{\ldots\}$		$\alpha'_5 : p_2 \vee p_5 \vee a_2$
$\alpha_6 : p_7 \vee p_4 \vee a_1 \vee a_2 \vee a_4 \vee a_5$	$\{\alpha_3, \alpha_4, \ldots\}$		$\alpha'_6 : p_7 \vee p_4 \vee \ell_5$
$\alpha_7 : \overline{p_2} \vee a_6 \vee a_5$	$\{\ldots\}$		$\alpha'_7 : \overline{p_2} \vee \ell_6$

(a) Clauses apprises (clauses originales à gauche, version factorisée à droite)



(b) Graphe de définition

FIGURE 7. *Factorisation des hypothèses.*

peuvent dépendre d'autres définitions et donc comme pour α_3, α_4, et α_6 factoriser récursivement les clauses apprises.

Rappelons que les hypothèses sont *toujours* assignées. De plus, les algorithmes d'extraction de MUS, comme pour `Muser`, pour lesquels notre méthode est applicable, respecte toujours la propriété suivante : *l'ensemble des* **variables** *utilisées dans les hypothèse ne change pas entre les différents appels aux solveurs SAT*, excepté pour les hypothèses décidées au niveau zéro [1] (ces dernières sont cependant bien assignées). Par conséquent, il est *toujours* possible de définir, *via* une phase d'initialisation discutée dans la section suivante, la valeur de vérité de toutes les variables abréviations [2]. Ainsi, dans la suite, nous focalisons de nouveau nos expérimentations sur le problème d'extraction de MUS.

4.1.2. *Initialisation*

Après avoir factorisé les hypothèses et ajouté les abréviations associées, toutes les clauses apprises α contiennent *au plus une* hypothèse ou abréviation. Ce dernier sera dénoté par $r(\alpha)$ dans la suite. Remarquons que $r(\alpha)$ peut être indéfini dans le cas où il a été assigné au niveau zéro ou lorsqu'une clause apprise ne posséde pas d'hypothèses à sa création.

1. Les littéraux décidés au niveau zéro sont ceux prouvés unitaires.
2. Remarquons que notre approche fonctionne toujours même si cette propriété n'est pas vérifiée. Cependant, dans ce cas cette dernière serait moi efficace. En effet, elle aurait pour conséquence de ne plus considérer les clauses apprises contenant des littéraux non totalement définis (i.e. un littéral qui possède un antécédent qui n'est pas dans l'ensemble des hypothèses)

Algorithme 4.1 : assigneAbreviation

Input : ℓ: littéral; **var** $\mathcal{I}$: interprétation; $\mathcal{G}$: graphe de définition
1 supprimeUnit($\mathcal{G}[\ell]$);
2 **while** $\mathcal{I}(\mathcal{G}[\ell])$ *non assigné* **do**
3 | choisir $\ell' \in \mathcal{G}[\ell]$ *non assigné* ;
4 | assigneAbreviation($\mathcal{G}, \ell', \mathcal{I}$);
5 **end**
6 **if** $\mathcal{I}(\mathcal{G}[\ell]) = \bot$ **then** $\mathcal{I} \leftarrow \mathcal{I} \cup \{\neg\ell\}$ **else** $\mathcal{I} \leftarrow \mathcal{I} \cup \{\ell\}$;

Comme le graphe de définition $\mathcal{G}$ peut être interprété comme un circuit (acyclique), il permet de calculer la valeur de vérité de chaque abréviation une fois que toutes les hypothèses ont été assignées.

Afin d'associer la bonne valeur de vérité à une abréviation, nous avons besoin d'assigner les variables hypothèses et, récursivement, toutes les abréviations présentes dans sa définition. Cette initialisation est décrite dans l'algorithme 4.1. Il prend en entrée les arguments suivants : le littéral ℓ qui doit être assigné, (par référence) l'interprétation courante [3] $\mathcal{I}$ ainsi que le graphe de définition $\mathcal{G}$. Premièrement, les littéraux prouvés unitaires au niveau zéro sont supprimés de $\mathcal{G}[\ell]$. Remarquons que l'impact de l'affection des hypothèses ou des abréviations au niveau zéro (ce qui signifie qu'elles ont été *supprimées*) est pris en considération *a priori* par le solveur durant la phase de propagation unitaire. Ensuite, tant qu'il existe un littéral ℓ' dans $\mathcal{G}[\ell]$ tel que $\mathcal{G}[\ell]$ n'est pas défini par l'interprétation courante $\mathcal{I}$, nous appelons récursivement la procédure afin d'assigner ℓ'. Lorsque la valeur de $\mathcal{G}[\ell]$ est déterminée sous l'interprétation $\mathcal{I}$, nous pouvons alors ssigner ℓ à $\mathcal{I}(\mathcal{G}[\ell])$.

Par construction du graphe de définition, $\mathcal{G}$ ne possède pas de cycle. De plus, nous supposons que toutes les hypothèses sont assignées dans $\mathcal{I}$, comme discuté précédemment. Par conséquent, notre algorithme termine et assigne la valeur de chaque abréviation ℓ en fonction de sa définition $\mathcal{G}[\ell]$.

4.2. Expérimentation des méthodes proposées

Toutes les modifications proposées on été implantés dans deux solveurs SAT. La première partie (sans utilisation d'abréviations) a été inclue dans `GlucoseInc`. La figure 5 compare `GlucoseInc` avec `Minisat`, et avec `Glucose`. Nous avons par ailleurs ajouté les résultats de l'utilisation seule des abréviations (Lagniez, Biere, 2013) comme décrite ci-dessus. Cette dernière méthode, alternative et complémentaire à la notre, est intéressante car elle améliore également l'oracle SAT.

3. Cette interprétation représente l'état des littéraux prouvés unitaires au niveau zéro ainsi que le choix de l'utilisateur concernant l'activation et la désactivation de certaines clauses.

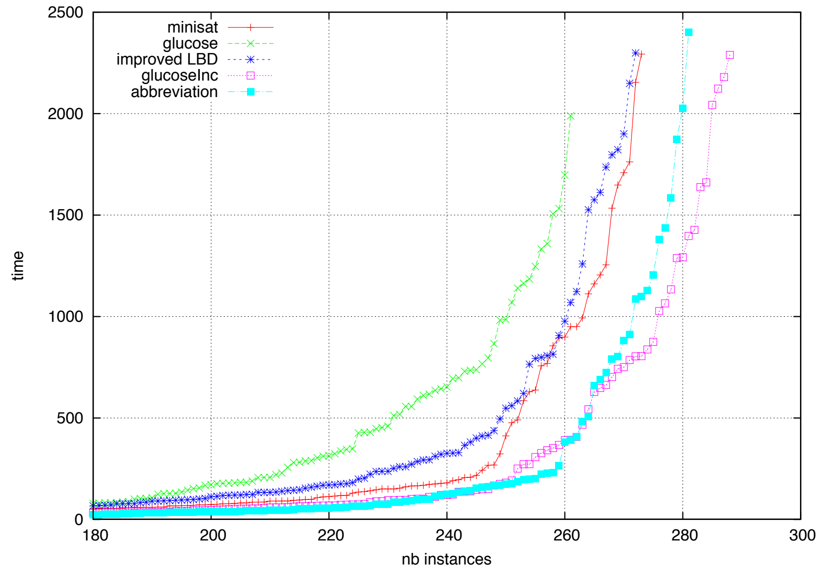


FIGURE 8. *Cactus plot. L'axe de x représente le nombre d'instances. L'axe des y représente le temps nécessaires pour les résoudre si elles sont toutes exécutées en parallèle.*

Les résultats sont clairs : `GlucoseInc` dépasse les performances des deux autres moteurs SAT de `Minisat` et `Glucose`. C'est la version qui résout le plus d'instance et le plus rapidement possible. L'oracle SAT de (Lagniez, Biere, 2013) (appelé `abréviations`) obtient de très bons résultats, bien meilleur que ceux de `Minisat`, et relativement proches de `GlucoseInc`, même si cette dernière s'avère être celle qui résout le plus d'instances. Cela est aussi explicite sur la fameuse courbe cactus de la figure 8.

Comme nous l'avons écrit dans la section 2.2, notre objectif consistait à améliorer les performances des moteurs SAT incrémentaux, et donc, les performances de chaque appel au moteur SAT. La table 2 montre que cet objectif est atteint. Elle montre, pour `Minisat Glucose` et `GlucoseInc`, et pour les 4 mêmes instances que précédemment, le nombre d'appels SAT et le temps moyen nécessaire pour chacun d'entre eux. Même dans le cas où plus d'appels à SAT sont réalisés, le temps total pour extraire le MUS est toujours amélioré. Sur ces instances le moteur SAT basé sur les abréviations est appelé moins de fois pour des résultats en temps relativement similaires voir

		Minisat		Glucose		GlucoseInc			abréviations		
Instance	#C	#SAT calls	avg	#SAT calls	avg	time	#SAT calls	avg	time	#SAT calls	avg
fdmus_b21_96	8541	2103	0.009	2134	0.02	11	2153	0.004	10	2071	0.004
longmult6	8853	706	0.01	1027	0.03	13	748	0.01	9	695	0.001
dump_vc950	360419	7	135	11	11.5	65	9	7.2	5	8	0.625
g7n	70492	4791	0.02	4393	0.08	67	4779	0.01	70	4597	0.01

TABLE 2. *Pour des instances représentatives, nous rapportons le nombre de clauses, et, pour chaque moteur SAT incrémental, nous rapportons le nombre d'appels au démonstrateur ainsi ques le temps moyen pour chacun de ces appels. Pour* `GlucoseInc` *et* `abréviations`*, nous rapportons aussi le temps total nécessaire à la recherche du MUS (pour les autres moteurs SAT, cela est donné dans la table 1*

plus rapide. Pour conclure, dans tous les cas, l'amélioration du moteur SAT permet d'améliorer l'extracteur de MUS.

4.3. Autres idées aux résultats négatifs

Habituellement, seuls les « *bons* » résultats sont publiés. Nous avons décidé de passer en revue un certains nombre d'idées prometteuses que nous avons testées, mais qui se sont malheureusement avérées mauvaises au cours des expérimentations. Par exemple, nous avons essayé de détecter les vieilles clauses apprises (créées lors d'un ancien run) et qui n'ont pas été utilisées dans le processus de propagation unitaire depuis un certain temps et avons alors donné une pénalité à leur LBD. Si elles sont à nouveau utilisées, alors le LBD sera remis à jour, sinon ces clauses seront supprimées lors du prochain nettoyage de la base. Nous avons également essayé de favoriser les clauses avec peu de littéraux initiaux en utilisant cette valeur comme second critère de tri. Nous avons finalement essayé de prédire le résultat (SAT/UNSAT) de l'appel courant afin de modifier la stratégie de suppression des clauses apprises. En effet, la plupart des appels SAT sont fait avec très peu de conflits. Nous avons donc essayé d'être plus agressif et de supprimer plus de clauses lorsque de nombreux conflits étaient atteints.

Malheureusement, aucune de ces idées n'a donné de bons résultats. Il faut quand même noter que la nouvelle version de `Muser` obtenue avec `GlucoseInc` est capable de trouver un MUS pour 96% des instances (en 2400 secondes). Il y a donc des chances (nous avons augmenté le temps CPU à 15000 secondes et n'avons été en mesure de résoudre que 4 instances supplémentaires) que les instances restantes soient très dures et nécessitent alors de nouvelles stratégies quand à la recherche des MUS proprement dite.

5. Conclusion

Une des raisons du succès des démonstrateurs SAT est certainement leur capacité à être utilisés comme des boites noires dans de nombreuses applications. Néanmoins, lorsqu'une nouvelle et très spécifique utilisation est proposée il peut s'avérer nécessaire d'en adapter les démonstrateurs. Ceci est typiquement le cas dans le cadre SAT incrémental. Dans cet article, nous nous sommes focalisés sur une des applications SAT incrémentales les plus représentatives, car utilisant de nombreux sélecteurs : la recherche de noyaux minimum inconsistants. Nous avons uniquement travaillé du coté du moteur SAT, c'est-à-dire que nous nous sommes attelé à en améliorer ses performances, afin d'obtenir des meilleures performances au niveau de l'extracteur de noyau. Nous pensons que cette amélioration peut avoir des applications directes dans d'autres domaines utilisant la technologie SAT incrémental.

Ce travail préliminaire offre de nombreuses et intéressantes perspectives de recherche. Par exemple, nous comptons étudier les dépendances entre les sélecteurs afin de les prendre en compte pour améliorer la recherche. Une autre piste de recherche est reliée à l'adaptation des démonstrateurs (par rapport aux heuristiques, redémarrages...) par rapport aux lancement précédents.

Bibliographie

Audemard G., Katsirelos G., Simon L. (2010). A restriction of extended resolution for clause learning SAT solvers. In *Proc. aaai'10.*

Audemard G., Lagniez J. M., Mazure B., Saïs L. (2011). On freezing and reactivating learnt clauses. In *Proc. sat'11*, p. 188–200.

Audemard G., Simon L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *Proc. ijcai'09*, p. 399-404.

Audemard G., Simon L. (2012). Refining restarts strategies for SAT and UNSAT. In *Proc. cp'12*, vol. 7514, p. 118-126. Springer.

Bakker R. R., Dikker F., Tempelman F., Wognum P. M. (1993). Diagnosing and solving over-determined constraint satisfaction problems. In *Ijcai*, vol. 93, p. 276–281.

Belov A., Järvisalo M., Marques-Silva J. (2013). Formula preprocessing in mus extraction. In *Tools and algorithms for the construction and analysis of systems*, p. 108–123. Springer.

Belov A., Lynce I., Marques-Silva J. (2012). Towards efficient mus extraction. *AI Communications*, vol. 25, nº 2, p. 97–116.

Belov A., Marques-Silva J. (2011). Accelerating MUS extraction with recursive model rotation. In *Proc. fmcad'11*, p. 37–40. FMCAD.

Biere A. (2008). Adaptive restart strategies for conflict driven sat solvers. In *Proc. sat'08*, vol. 4996, p. 28-33. Springer.

Biere A., Cimatti A., Clarke E. M., Fujita M., Zhu Y. (1999). Symbolic model checking using sat procedures instead of bdds. In *Proc. dac'99*, p. 317–320.

Bradley A. (2012). IC3 and beyond: Incremental, inductive verification. In *proc. of cav.*

Cook S. (1971). The complexity of theorem-proving procedures. In *Proc. stoc'71*, p. 151–158. ACM.

Davis M., Logemann G., Loveland D. (1962). A machine program for theorem-proving. *Commun. ACM*.

Eén N., Sörensson N. (2003a). An Extensible SAT-solver. In *Proc. sat'03*, p. 333–336.

Eén N., Sörensson N. (2003b). Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, vol. 89, n° 4, p. 543 - 560.

Fu Z., Malik S. (2006). On solving the partial MAX-SAT problem. In *Proc. sat'06*, vol. 4121, p. 252-265. Springer.

Goldberg E., Novikov Y. (2002). BerkMin: A fast and robust SAT-solver. In *Proc. date'02*, p. 142-149.

Grégoire E., Mazure B., Piette C. (2006). Extracting muses. In *Proc. ecai'06*, vol. 141, p. 387-391. IOS Press.

Hemery F., Lecoutre C., Sais L., Boussemart F. *et al.* (2006). Extracting mucs from constraint networks. In *Ecai*, vol. 6, p. 113–117.

Huang J. (2010). Extended clause learning. *AI*, vol. 174, n° 15, p. 1277–1284.

Kautz H. A., Selman B. (1992, août). Planning as satisfiability. In, p. 359-363. Vienna, Austria.

Lagniez J.-M., Biere A. (2013). Factoring out assumptions to speed up mus extraction. In *16th international conference on theory and applications of satisfiability testing(sat'13)*, vol. 7962, p. 276-292. Springer.

Marques-Silva J., Lynce I., Malik S. (2009). Conflict-driven clause learning sat solvers. In A. Biere, M. J. H. Heule, H. van Maaren, T. Walsh (Eds.),, vol. 185, p. 980. IOS Press.

Moskewicz M., Conor C., Zhao Y., Zhang L., Malik S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proc. dac'01*.

Nadel A. (2010). Boosting minimal unsatisfiable core extraction. In *Proc. fmcad'10*, p. 221-229.

Nadel A., Ryvchin V. (2012). Efficient SAT solving under assumptions. In *Proc. sat'12*.

Nam G.-J., Aloul F., Sakallah K. A., Rutenbar R. A. (2004). A comparative study of two boolean formulations of fpga detailed routing constraints. *Computers, IEEE Transactions on*, vol. 53, n° 6, p. 688–696.

Papadimitriou C. H., Wolfe D. (1988). The complexity of facets resolved. *Journal of Computer and System Sciences*, vol. 37, n° 1, p. 2–13.

Pipatsrisawat K., Darwiche A. (2007). A lightweight component caching scheme for satisfiability solvers. In *Proc. sat'07*, p. 294–299.

Ryvchin V., Strichman O. (2011). Faster extraction of high-level minimal unsatisfiable cores. In *Proc. sat'11*, vol. 6695, p. 174-187.

Silva J. P. M., Sakallah K. (1996). Grasp – a new search algorithm for satisfiability. In *Proc. cad'96*, p. 220–227.

Siqueira N. J. L. de, Puget J.-F. (1988). Explanation-based generalisation of failures. In *Ecai*, p. 339-344.

Soh T. (2011). *Studies on applying incremental sat solving to optimization and enumeration problems*. Thèse de doctorat non publiée, School of Multidisciplinary Sciences.

Velev M., Bryant R. (2003). Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, vol. 35, nº 2, p. 73 – 106.

Wieringa S. (2014). *Incremental satisfiability solving and its applications*. Thèse de doctorat non publiée, Aalto University.

Zhang L., Madigan C., Moskewicz M., Malik S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. cad'01*, p. 279–285.