Du glucose en goutte à goutte pour les coeurs inconsistants

Gilles Audemard[1] **Jean-Marie Lagniez** [1] **Laurent Simon**[2]

[1] Univ. Lille-Nord de France – CRIL/CNRS UMR8188 – Lens, F-62307, France
[2] Univ. Paris-Sud – LRI/CNRS UMR 8623 / INRIA Saclay – Orsay, F-91405, France

audemard@cril.fr lagniez@cril.fr simon@lri.fr

Résumé

Les dernières avancées réalisées 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. Pour la plupart, leur résolution 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 nécessaire de concevoir les nouveaux démonstrateurs SAT afin de prendre en considération ce nouveau paradigme d'utilisation. Dans cet article, nous proposons d'améliorer le démonstrateur SAT `Glucose` afin d'en faire un démonstrateur incrémental efficace. Pour cela, nous étendons la notion de qualité de clauses apprises dans le cas intensif de démonstrateur incrémental utilisant des hypothèses. Afin de valider expérimentalement nos contributions, nous avons étudié ces performances sur une utilisation typique des démonstrateurs SAT incrémentaux : la recherche de noyaux (coeurs) minimaux inconsistants. Nous pensons que ces améliorations peuvent directement bénéficier à la plupart des autres applications basées sur les démonstrateurs SAT incrémentaux.

1 Introduction

Décider de la satisfiabilité d'une formule propositionnelle fait partie des questions fondamentales en intelligence artificielle. Ce problème, nommé *problème de SATisfiabilité* (SAT), a grandement été étudié pratiquement et théoriquement depuis les années 70. Étant donné sa place centrale dans la théorie de la complexité (premier problème a avoir été prouvé NP-complet par Cook [8]), il est l'un des problèmes de référence car il permet de capturer la difficulté d'un large panel d'autres problèmes provenant d'applications très diverses. Ainsi, les progrès réalisés dans le contexte de la résolution pratique du problème SAT ont un impact direct sur la résolution en pratique de nombreux autres problèmes pouvant s'y réduire ou pouvant être résolu par plusieurs appels successifs à un oracle NP.

Depuis l'avénement des solveurs SAT modernes [17, 10], initialement introduits par Marques-Silva et Sakallah [23], les démonstrateurs peuvent résoudre des problèmes de plus en plus conséquent (des millions de clauses et de variables). Puisque de nombreux problèmes peuvent se réduire facilement au problème SAT, l'amélioration de ces solveurs a eu un impact direct dans 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. De plus, ce gain de performance laisse envisager la possibilité de traiter pratiquement, à l'aide de solveurs SAT, des problèmes au-delà de NP nécessitant l'appel à plusieurs oracles NP afin de pouvoir être traités [6, 24].

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` [10], est aujourd'hui une technique utilisée dans des applications très diverses : vérification formelle bornée [11], extraction de noyaux inconsistants [18], MAXSAT [12], ou encore en vérification inductive [7]. Dans ce contexte, les démonstrateurs SAT ne sont pas lancés une seule fois sur des formules potentiellement énormes, mais sont potentiellement 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 pos-

sibilité de ré-utiliser un maximum les informations pouvant être collectées entre deux appels successifs au démonstrateur. Malgré tout, il est clair que lorsque des clauses sont supprimées en cours de route, les informations pouvant être dérivées (clauses apprises) à partir de ces dernières ne peuvent plus être ré-utilisées. Pour palier ce problème, il est nécessaire d'ajouter des hypothèses au démonstrateur SAT. Ces hypothèses sont des littéraux assignés à chaque clauses et qui sont affectés au début de la recherche.

Dans cet article, nous prenons un cas d'utilisation typique des démonstrateurs SAT incrémentaux, qui est la recherche de noyaux minimum inconsistants [14, 18, 22], et nous nous focalisons uniquement à perfectionner le moteur SAT afin d'améliorer les performances globales de l'extracteur de MUS. Pour effectuer cela, nous avons choisi d'étudier expérimentalement notre travail sur `Muser` [4], qui est un extracteur MUS open source très efficace et facilement modifiable. Nous nous intéressons plus particulièrement à améliorer les performances de `glucose` [2] lorsque celui-ci est utilisé comme moteur SAT. Ainsi, nous espérons que les améliorations induites pourront directement être utilisées dans d'autres applications typiques de SAT incrémental.

2 SAT et SAT Incrémental

Les démonstrateurs SAT "*modernes*" sont basés sur le paradigme CDCL (*conflict driven clause learning*) [17]. Bien qu'à l'origine, ils ont été proposés comme une extension de l'algorithme DPLL [9] comprenant l'apprentissage de clauses [23, 25], il est maintenant admis qu'ils doivent être considérés comme un mélange d'algorithmes de retour arrière et de moteurs de résolution. Ces démonstrateurs intègrent également de nombreuses techniques telles que : l'heuristiques de choix de variable dynamique [17], le choix de la polarité des variables [21], la stratégie de suppression des clauses apprises [13, 2, 1] et de redémarrages [5, 3] (le lecteur pourra se référer à [16] pour une description détaillée des démonstrateurs CDCL). Leur haut degré de sophistication leur permet aujourd'hui de traiter des problèmes applicatifs conséquents [15].

2.1 SAT incrémental avec sélecteurs

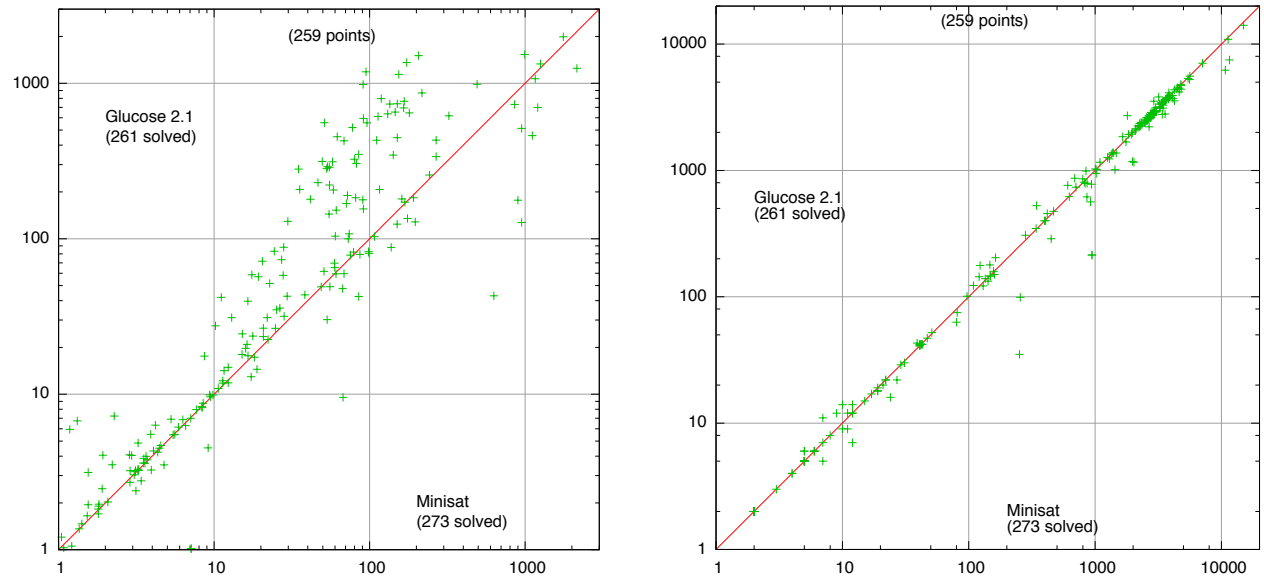
Dans cet article, nous nous concentrons sur l'amélioration des démonstrateurs SAT incrémentaux (voir [19] pour une description détaillée). Dans ce type d'approche, le même démonstrateur est lancé sur un certain nombre d'instances proches les unes des autres et l'état du solveur est mémorisé entre chaque appel. Par exemple, les heuristiques de choix de variables [17], les choix de polarité [21] ou encore les stratégies de redémarrages et de nettoyage de la base de clauses apprises [13, 2, 1] sont maintenus en l'état entre deux appels successifs. Dans le cadre incrémental, certaines clauses peuvent être supprimées entre deux appels au démonstrateur. Il n'est alors pas possible de ré-utiliser directement les clauses apprises générées à l'aide de ces dernières. Pour palier 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 [19]). Ainsi, si durant la recherche une variable issue des hypothèses doit absolument changer de polarité, alors le problème est UNSAT suivant l'ensemble de littéraux $\mathcal{A}$.

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 au cours de la recherche 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 associée mais également toutes les clauses apprises dérivées à partir de celle-ci. Ceci est illustré dans l'exemple suivant :

Exemple 1 *On considère une formule contenant (entre autres) les clauses $c_1 = a \vee b$ et $c_2 = \neg a \vee b \vee c$. On ajoute donc deux sélecteurs s_1 et s_2 associés aux clauses c_1 et c_2. Les clauses c_1 et c_2 de la formule initiale sont supprimées et remplacées par $(c_1 \vee s_1)$ et $(c_2 \vee s_2)$. Lorsque l'on choisi d'affecter comme hypothèse s_1 et $\neg s_2$ alors la clause c_1 est activée pendant cet appel à SAT et la clause c_2 est désactivée. Si lors des précédents appels au moteur SAT on a appris la clause (les sélecteurs de ces deux clauses étaient alors faux) issue de la résolution entre c_1 et c_2 et égale à $b \vee c \vee s_1 \vee s_2$ alors cette clause apprise ne sera activée que lorsque les hypothèses s_1 et s_2 seront fausses.*

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

Nous allons nous focaliser sur une application typique utilisant SAT incrémental ainsi que les sélecteurs. Il s'agit du problème de calcul de MUS (*Minimum Unsatisfiable Set*) sur une formule insatisfiable [20]. Ce problème consiste à extraire une sous formule également insatisfiable de la formule initiale, telle que toute sous formule plus petite soit satisfiable. Dans cette approche, 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



(a) Temps de résolution

(b) Nombre d'appels au moteur SAT

FIGURE 1 – 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

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` [4], 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 [4] 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.

Si, entre chaque appel au démonstrateur SAT, on veut pouvoir ré-utiliser des clauses apprises, il est alors essentiel de déterminer quelles clauses seront utiles pour les futurs appels. Dans ce contexte, nous avons choisi d'utiliser le démonstrateur `Glucose` [2], une variante de `Minisat` [10], basé sur un score (appelé LBD) évaluant la qualité des clauses apprises. Nous avons donc démarré notre travail en comparant `Glucose` contre `Minisat` comme moteur SAT de `Muser`. Les résultats, étonnamment mauvais, sont résumés dans la figure 2 (pour chaque figure de ce type, nous donnons le nombre d'instances résolues par chaque méthode, le nombre d'instances résolues par les deux (259 pour la figure 1(a) ; moins il y a de points dans la région d'une méthode, meilleure est cette dernière comparée à l'autre). Malgré des résultats très mauvais pour `Glucose`, les deux parties de la figure 2 sont intéressantes à observer. Le temps CPU est clairement en faveur de `Minisat`. Mais la figure 1(b), comparant les deux approches du point de vue du nombre d'appel au moteur SAT, montre que la plupart du temps, les deux approches nécessitent approximativement autant d'appels démonstrateur 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 obtenir de très bons résultats.

Dès lors, 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 surcoût 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, on

		LBD					Nouveau LBD				
			taille		LBD			taille		LBD	
Instance	#C	temps	moyenne	max	moyenne	max	temps	moyenne	max	moyenne	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.

va lancer le moteur SAT 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`.

3 Améliorer les moteurs SAT

LBD et sélecteurs

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. 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), nous devons prendre en compte les sélecteurs dans le calcul du LBD. 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. 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.

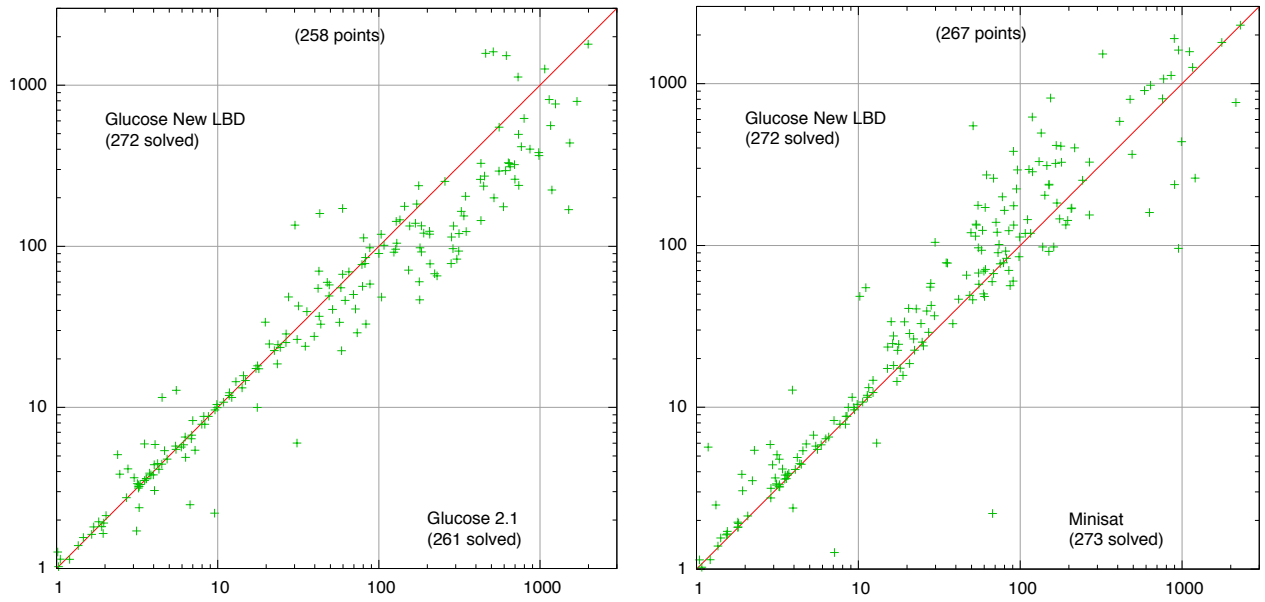
Malgré tout, comme nous pouvons le constater sur la figure 2, même si nous améliorons les performances de `Glucose`, cette nouvelle version obtient des résultats très proches de `Minisat` (par rapport au nombre d'instances résolues), mais plus lente. À ce niveau de nos expérimentations, les résultats 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 par la suite quelques modifications importantes.

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 surcoût 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 opérations.

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 splitter ces dernières en deux sous ensembles indépendants (sélecteurs/non sélecteurs) : nous devons être à même de visiter ces littéraux particuliers lorsque nécessaire. 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



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

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

FIGURE 2 – Comparaison, au sein de `Muser`, des performances de `Glucose` (2.1) en le nouveau LBD. La figure 2(a) compare cette version avec `Glucose` classique. La figure 2(b) compare cette version avec `Minisat`.

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 pouvons espérons limiter le nombre de clauses à visiter lors de la recherche de nouveaux littéraux témoins. De plus, comme `Glucose` effectue de nombreux restarts, nous limitons également le backtrack au premier niveau de décision qui n'est pas une hypothèse.

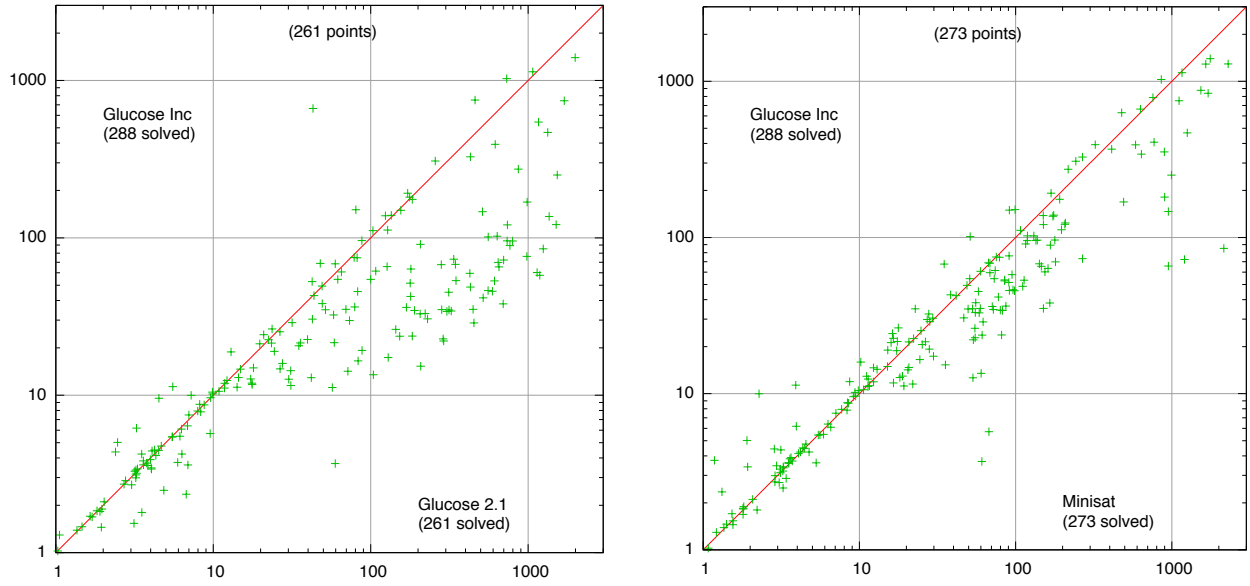
Simplification de la base de clauses

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.

3.1 Comparaison globale

Toutes les modifications proposées on été implantés et nous avons appelé le moteur SAT résultant `GlucoseInc`. La figure 3 compare `GlucoseInc` avec `Minisat`, mais aussi avec `Glucose`. Les résultats sont clairs : cette nouvelle version dépasse les performances des deux autres moteurs SAT. Cela est aussi explicite sur la fameuse courbe cactus de la figure 4.

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 né-



(a) running time

(b) number of SAT solver calls

FIGURE 3 – Comparaison, au sein de `Muser`, des performances de `GlucoseInc`. La figure 2(a) compare cette version avec `Glucose` classique. La figure 2(b) compare cette version avec `Minisat`.

cessaire 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.

3.2 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 essayer 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 essayer 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 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.

4 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 somme focalisé 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

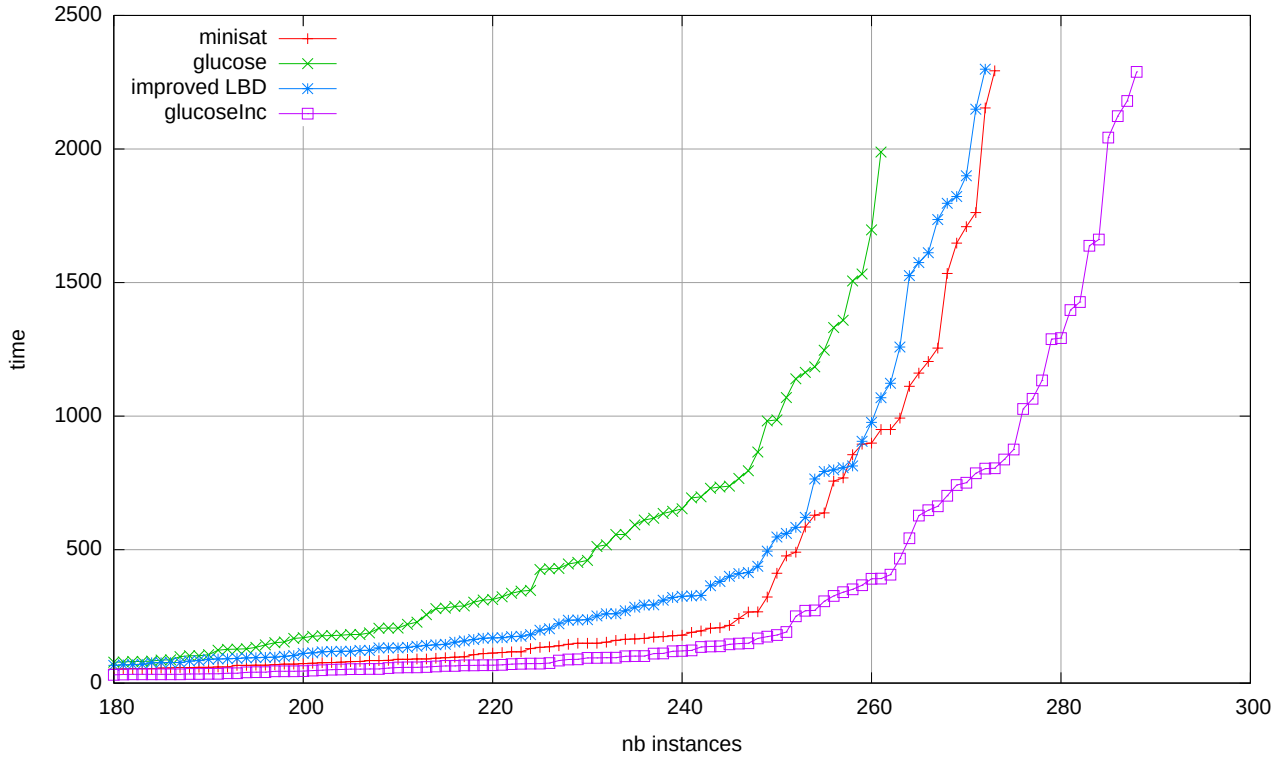


FIGURE 4 – Cactus plot. L'axe de x représente le nombre d'instances. L'axe des y représente le temps nécessaires pour les résoudre si elles sont toutes exécutées en parallèle.

rapport aux lancement précédents.

Références

[1] G. Audemard, J. M. Lagniez, B. Mazure, and L. Saïs. On freezing and reactivating learnt clauses. In *Proc. SAT'11*, pages 188–200, 2011.

[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI'09*, pages 399–404, 2009.

[3] G. Audemard and L. Simon. Refining restarts strategies for SAT and UNSAT. In *Proc. CP'12*, volume 7514 of *LNCS*, pages 118–126. Springer, 2012.

[4] A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proc. FMCAD'11*, pages 37–40. FMCAD, 2011.

[5] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In *Proc. SAT'08*, volume 4996 of *LNCS*, pages 28–33. Springer, 2008.

[6] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proc. DAC'99*, pages 317–320, 1999.

[7] A. Bradley. IC3 and beyond : Incremental, inductive verification. In *proc. of CAV*, 2012.

[8] S. Cook. The complexity of theorem-proving procedures. In *Proc. STOC'71*, pages 151–158. ACM, 1971.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 1962.

[10] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. SAT'03*, pages 333–336, 2003.

[11] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4) :543 – 560, 2003.

[12] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *Proc. SAT'06*, volume 4121 of *LNCS*, pages 252–265. Springer, 2006.

[13] E. Goldberg and Y. Novikov. BerkMin : A fast and robust SAT-solver. In *Proc. DATE'02*, pages 142–149, 2002.

[14] E. Grégoire, B. Mazure, and C. Piette. Extracting muses. In *Proc. ECAI'06*, volume 141 of *Frontiers in Artificial Intel. and Applications*, pages 387–391. IOS Press, 2006.

		Minisat		Glucose		GlucoseInc		
Instance	#C	#SAT calls	avg	#SAT calls	avg	time	#SAT calls	avg
fdmus_b21_96	8541	2103	0.009	2134	0.02	11	2153	0.004
longmult6	8853	706	0.01	1027	0.03	13	748	0.01
dump_vc950	360419	7	135	11	11.5	65	9	7.2
g7n	70492	4791	0.02	4393	0.08	67	4779	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 que le temps moyen pour chacun de ces appels. Pour GlucoseInc, nous rapportons aussi le temps total nécessaire à la recherche du MUS (pour les autres moteurs SAT, cela est donné dans la table 1

[15] D. LeBerre and L. Simon, editors. *Special Volume on the SAT 2005 competitions and evaluations*, volume 2. Journal on Satisfiability, Boolean Modeling and Computation, 2006.

[16] J. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, page 980. IOS Press, 2009.

[17] M. Moskewicz, C. Conor, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an Efficient SAT Solver. In *Proc. DAC'01*, 2001.

[18] A. Nadel. Boosting minimal unsatisfiable core extraction. In *Proc. FMCAD'10*, pages 221–229, 2010.

[19] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *Proc. SAT'12*, 2012.

[20] Y. Oh, M.N. Mneimneh, Z.S. Andraus, K.A. Sakallah, and I.L. Markov. AMUSE : a minimally-unsatisfiable subformula extractor. In *Design Automation Conference*, pages 518–523, 2004.

[21] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proc. SAT'07*, pages 294–299, 2007.

[22] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Proc. SAT'11*, volume 6695 of *LNCS*, pages 174–187, 2011.

[23] J. P. Marques Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proc. CAD'96*, pages 220–227, 1996.

[24] M. Velev and R. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2) :73 – 106, 2003.

[25] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. CAD'01*, pages 279–285, 2001.