Vers une gestion fine et dynamique de la base de clauses apprises

Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, Lakhdar Saïs *

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens

{audemard,lagniez,mazure,sais}@cril.fr

Résumé

Dans ce papier, nous proposons une nouvelle heuristique dynamique pour la gestion de la base de clauses apprises dans le cadre des solveurs SAT modernes. Cette approche est basée sur le principe d'activation et de désactivation de clauses. Etant donnée une étape de la recherche, nous utilisons une fonction pour activer certaines clauses et en geler d'autres. De cette manière certaines clauses inutiles à un moment de la recherche pourront de nouveau utilisées dans le futur. Notre stratégie tente d'exploiter certaines informations du passé pour prédire si une clause sera ou non utilisée dans le futur. Elle diffère des autres stratégies par le fait que les clauses ne sont pas supprimées définitivement. Les expérimentations faites sur les instances de la dernière compétition SAT montrent l'efficacité de l'approche proposée.

Abstract

In this paper, we propose a new dynamic management policy of the learnt clauses database in modern SAT solvers. It is based on a dynamic freezing and activation principle of the learnt clauses. At a given search state, using a relevant selection function, it activates the most promising learnt clauses while freezing irrelevant ones. In this way, clauses learned in previous steps can be discarded at the current step and might be activated again in future steps of the search process. Our strategy tries to exploit pieces of information gathered from the past to deduce the relevance of a given clause for the remaining search steps. This policy contrasts with all the well-known deletion strategies, where a given learned clause is definitely eliminated. Experiments on SAT instances taken from the last competitions demonstrate the efficiency of our technique.

*Ce travail est supporté par l'ANR « UNLOC », ANR 08-BLAN-0289-01.

1 Introduction

Le problème SAT est le problème de décision qui consiste à vérifier si une formule booléenne, souvent représentée sous Forme Normale Conjonctive (CNF), admet ou non une affectation de ses variables qui associe « vrai » à la formule. Ce problème est très important en théorie de la complexité (problème NP-Complet de référence) et a beaucoup d'application en intelligence artificielle (planification, raisonnement non monotone, bioinformatique, vérification formelle ...).

Au cours de ces deux dernières décennies le problème SAT, avec l'arrivée d'une nouvelle génération de solveurs, a énormément gagné en intérêt. Ces solveurs, appelés CDCL [11, 5] (Conflict Driven, Clause Learning) ont pour la première fois été capables de résoudre des instances conséquentes codant des problèmes de la vie réelle. Leur architecture est basée sur les notions suivantes : l'heuristique de choix de variables VSIDS (Variable State Independant, Decading Sum) [11], l'utilisation de redémarrage [6, 7] et d'une structure de données efficace (*ex.* Watched literals) ainsi que l'apprentissage de clauses [9, 10, 15]. Cette dernière est une composante très importante de ce type de solveur. En effet d'un point de vue théorique, K. Pipatsrisawat et A. Darwiche [13] ont récemment prouvé que l'apprentissage de clauses tel qu'il est fait dans les démonstrateurs SAT modernes permet de simuler la résolution générale. Ce résultat montre que l'apprentissage rend les démonstrateurs de type DPLL [4] aussi puissants que la résolution générale. Mais en pratique, cet ajout de clauses peut poser pro-

blème. En effet, dans le pire des cas, la taille de la base de connaissance peut croître de manière exponentielle. Pour maintenir une base des clauses apprises de taille polynomiale - permettant ainsi un coût raisonnable pour la propagation unitaire - un ensemble de stratégies a été proposé pour supprimer les clauses supposées inutiles pour la suite de la recherche. Parmi ces stratégies, deux ont montré des résultats pratiques intéressants. La première, la plus populaire, est basée sur la notion de *first fail*. Cette stratégie considère qu'une clause apparaissant peu dans l'analyse des conflits récents est inutile et qu'elle doit être supprimée. La seconde est basée sur une mesure nommée LBD (*Litetal Block Distance*) [1]. Cette mesure statique correspond au nombre de niveaux différents intervenant dans la génération de la clause apprise. Les auteurs considèrent alors que les clauses ayant un LBD élevés sont inutiles pour la suite de la recherche et qu'elles peuvent donc être supprimées. Le principal inconvénient de ces stratégies est qu'il est impossible d'éviter la suppression de clauses potentiellement utiles pour la suite de la recherche. Leur élimination de manière définitive rend possible la dérivation répétitive de la même clause.

Dans ce papier, nous proposons une nouvelle heuristique dynamique pour la gestion de la base des clauses apprises dans le cadre des solveurs SAT modernes. Cette approche s'appuie sur le principe d'activation et de désactivation de clauses. Étant donnée une étape de la recherche, nous utilisons une fonction pour activer certaines clauses et en désactiver d'autres. Cette fonction, basée sur le *progress saving* [12], tente d'exploiter certaines informations du passé pour prédire si une clause sera ou non utilisée dans le futur. L'idée de notre approche consiste alors à geler une clause quand elle est jugée inutile pour la recherche en cours, et à la réactiver lorsqu'elle peut jouer un rôle dans l'établissement de la preuve. Cette stratégie diffère des stratégies habituelles puisque les clauses ne sont pas définitivement supprimées.

La suite de ce papier est organisée de la manière suivante : après quelques définitions préliminaires (section 2), nous introduirons une nouvelle mesure, basé sur le *progress saving*, permettant d'évaluer l'utilité d'une clause (section 3). Ensuite, nous présentons une approche dynamique de nettoyage de la base des clauses apprises basée sur la notion d'activation et de désactivation de clauses (section 4). Avant de conclure et de donner quelques perspectives, nous comparons expérimentalement notre politique de nettoyage des clauses apprises avec les politiques de nettoyage faisant parti de l'état de l'art (section 5).

2 Définitions, notations et notions préliminaires

Dans cette section, après quelques définitions et notations, nous introduisons le schéma général d'un solveur CDCL.

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. Finalement, SAT est le problème de décision qui consiste à vérifier si une formule Σ sous Forme Normale Conjonctive (CNF) possède ou non un modèle.

Algorithm 1: CDCL solver

Input: a CNF formula Σ
Output: SAT or UNSAT

1 $\Delta = \emptyset$; `/* learnt clauses database */`
2 **while** *(true)* **do**
3 **if** *(!propagate())* **then**
4 **if** *($(c = analyzeConflict()) == \emptyset$)* **then**
 return UNSAT ;
5 $\Delta = \Delta \cup \{c\}$;
6 **if** *(timeToRestart())* **then** backtrack to level 0;
7 **else**
8 backtrack to the assertion level of c;
9 **else**
10 **if** *($(l$ = decide()) == null)* **then** return SAT ;
11 assert l in a new decision level;
12 **if** *(timeToReduce())* **then** clean(Δ);

Nous présentons brièvement le schéma général d'un solveur SAT de type CDCL [11, 5]. 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.

L'algorithme 1 donne le schéma général d'un solveur de type CDCL. Typiquement, un tel solveur peut être assimilé à une séquence de décision suivie de propagations des littéraux unitaires, jusqu'à l'obtention d'un conflit. Chaque littéral choisi comme littéral de décision (ligne 10) est affecté suivant une certaine polarité à un niveau de déci-

sion donné (ligne 11). Si tous les littéraux sont affectés ($decide() == null$), alors $\mathcal{I}$ est un modèle de Σ (lignes 10). À chaque fois qu'un conflit est atteint par propagation (ligne 3), une clause (*nogood*) c est calculée en utilisant une méthode d'analyse de conflits donnée, le plus souvent selon le schéma « First UIP » (*« Unique Implication Point »* [15]), et un niveau de *backjump* est calculé en fonction de la clause c. À ce moment, il est possible de prouver l'inconsistance (c est la clause vide) de la formule (ligne 4). Si ce n'est pas le cas, un saut arrière est effectué et le niveau de décision devient égal au niveau de *backjump* (ligne 8). Ensuite, certains solveurs CDCL forcent le redémarrage (diverses stratégies sont possibles [7]) et dans ce cas, on remonte en haut de l'arbre de recherche (ligne 6). Finalement, la base de clauses apprises peut être réduite lorsque celle-ci est considérée comme trop volumineuse (ligne 12).

Cette dernière composante, bien que souvent omise de la description des démonstrateurs CDCL, est pourtant cruciale pour leurs performances. En effet, conserver un grand nombre de clauses peut avoir des conséquences néfastes sur l'efficacité de la propagation unitaire, tandis que supprimer trop de clauses peut faire perdre le bénéfice de l'apprentissage. Par conséquent, identifier les bonnes clauses apprises (*i.e.* importantes pour la dérivation de la preuve) est un véritable challenge. La première mesure proposée pour quantifier la qualité d'une clause utilise l'idée sous-jacente à l'heuristique VSIDS. Plus précisément, une clause apprise est considérée comme utile pour la preuve, si elle apparaît fréquemment dans l'analyse des conflits récents. On remarque donc qu'une telle stratégie suppose qu'une clause utile dans le passé le sera dans le futur. Plus récemment, une nouvelle mesure, nommée LBD, a été utilisée pour estimer la qualité d'une clause apprise. L'implémentation de cette mesure dans un solveur CDCL (le solveur GLUCOSE [1]) a permis d'en améliorer sensiblement les performances. Cette nouvelle mesure est basée sur le calcul du nombre de niveaux de décision apparaissant dans une clause apprise lors de sa génération. Les auteurs on démontré expérimentalement que les clauses possédant un faible LBD sont plus souvent utilisées pendant la recherche que les clauses ayant un LBD élevé.

Un autre point crucial des solveurs CDCL concerne la phase d'affectation selon laquelle la prochaine variable choisie par l'heuristique VSIDS sera affectée. Habituellement, une phase par défaut (*e.g.* fausse) est définie et utilisée chaque fois qu'une variable de décision est assignée. Récemment, Pipatsrisawat et Darwiche [12] ont observé expérimentalement que les solveurs implémentant les redémarrages et les retours arrière non chronologique, peuvent être amenés à résoudre de manière répétitive les mêmes sous-problèmes. Pour éviter cela, les auteurs ont proposé une heuristique dynamique consistant à sauvegarder la phase selon laquelle une variable a été propagée, et à utiliser cette information lors des prochaines affectations de cette variable. Cette heuristique de phase, nommée *progress saving*, permet d'éviter la résolution répétitive des mêmes sous-formules satisfiables. Cette mémorisation de la phase peut être représentée par une interprétation complète $\mathcal{P}$. De cette manière, lorsqu'une nouvelle variable v est choisie comme décision, le littéral ℓ qui lui est associé (pour la phase) est choisi tel que $\ell \in \mathcal{P}$.

Dans la section suivante, nous exploitons la notion de *progress saving* ($\mathcal{P}$) pour estimer si une clause peut potentiellement être utile pour dériver une clause apprise (*i.e.* la possibilité qu'elle soit utilisée pour propager un littéral menant au conflit).

3 Une nouvelle mesure pour identifier les bonnes clauses

Comme précisé précédemment les solveurs SAT de type CDCL peuvent formaliser un système de preuve par résolution [13, 2]. Le principal inconvénient des méthodes basées sur la résolution générale est lié à leur complexité spatiale. D'un certain point de vue, les solveurs SAT modernes peuvent être vus comme une méthode de preuve par résolution avec une stratégie de suppression de clauses. Par conséquence, la complétude des solveurs SAT modernes est fortement liée à la politique de suppression de clauses. Par exemple, supposons que la stratégie de nettoyage de la base de clauses apprises soit très agressive, dans ce cas il nous serait impossible de garantir la complétude du solveur. Définir quelle clause sera utile pour la preuve est d'une grande importance pour l'efficacité des solveurs. Répondre à une telle question est cependant calculatoirement difficile et est très proche du problème qui consiste à trouver une preuve de taille minimale. Pour apporter une solution à ce problème nous définissons une mesure simple permettant d'évaluer la pertinence d'une clause et montrons expérimentalement sont efficacité. La mesure proposée dans cette section est basée sur la notion de *progress saving* [12], habituellement utilisée pour déterminer la valeur de vérité dont sera affectée la prochaine variable de décision. Cette mesure, nommée psm (*Progress Saving based quality Measure*), est définie de la manière suivante : étant données une clause c et une interprétation complète $\mathcal{P}$ représentant l'ensemble des polarité associées à chaque variable, nous définissons $psm_{\mathcal{P}}(c) = |\mathcal{P} \cap c|$.

Premièrement, il est important de noter que notre mesure *psm* est dynamique. En effet, puisque l'ensemble des littéraux $\mathcal{P}$ associés à la polarité des variables évolue durant la recherche, le *psm* d'une clause donnée évolue par la même occasion. Par exemple, quand une clause est apprise son *psm* est égal à zéro, et lorsque le retour-arrière est effectué son *psm* devient égal à un. Il est aussi important de noter que lorsqu'une clause est à l'origine de la propagation d'un littéral, son *psm* est aussi égal à un. Ces remarques préliminaires suggèrent que les clauses possédant une petite valeur

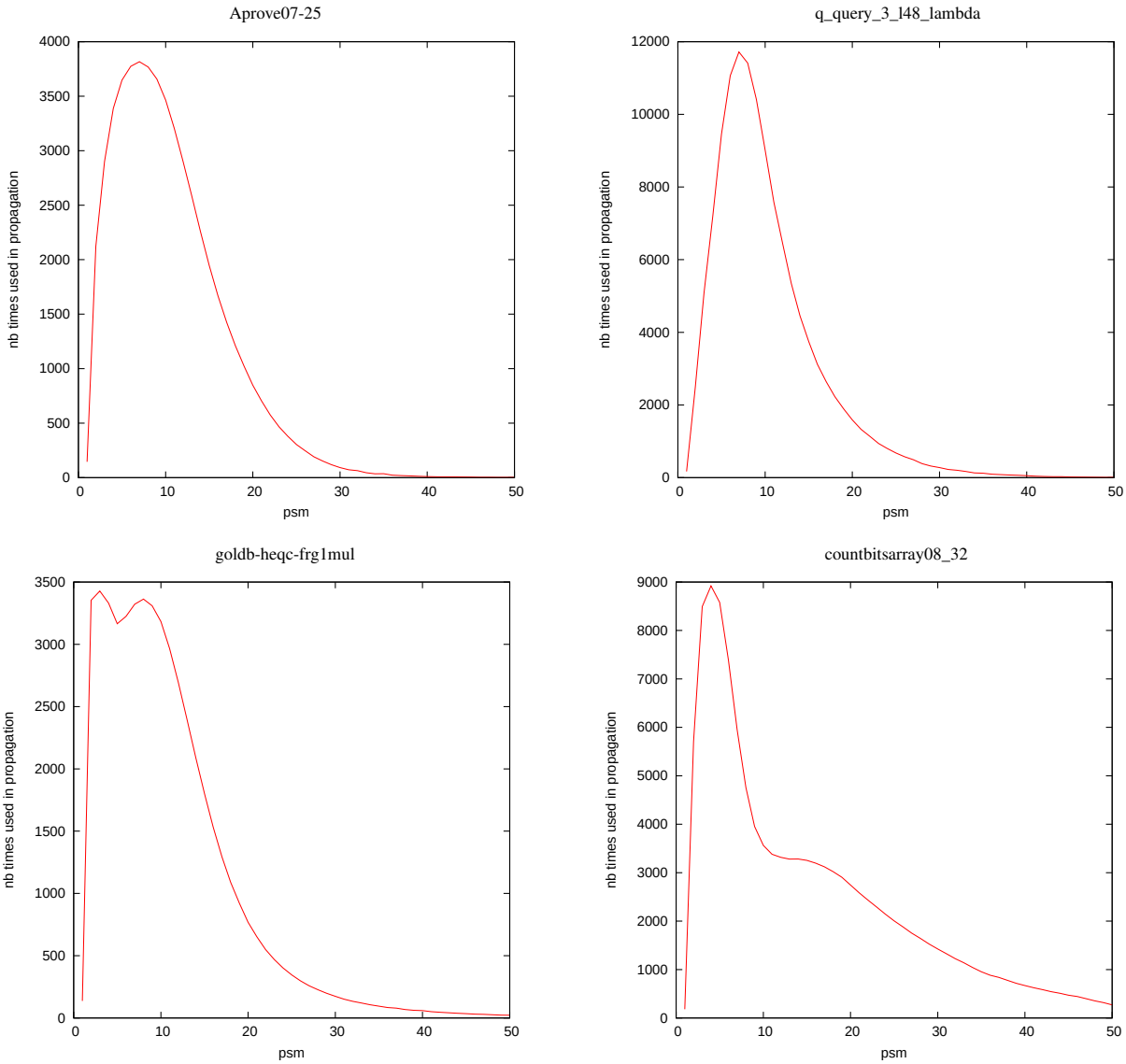

FIGURE 1 – Progress Saving mesure / utilité dans la PU

de *psm* sont plus importantes pour la suite de la recherche. En effet, considérons $\mathcal{I}$ l'interprétation partielle courante, $\mathcal{P}$ l'interprétation complète représentant le *progress saving* et c une clause. Comme $\mathcal{I} \subset \mathcal{P}$, $psm_{\mathcal{P}}(c)$ représente le nombre de littéraux affectés à vrai par $\mathcal{I}$ ou qui seront affectés à vrai par $\mathcal{P} \backslash \mathcal{I}$. Donc, une clause avec un petit *psm* a de fortes chances d'être utilisée pour la propagation unitaire ou d'être falsifiée. Au contraire, une clause avec un grand *psm* a plus de chance d'être satisfaite par plus d'un littéral et donc d'être inutile pour la suite de la recherche.

Pour analyser et valider cette hypothèse, un ensemble d'expérimentations a été mené. La Figure 1 reporte, pour quelques instances représentatives, la moyenne du nombre de fois où une clause avec une certaine valeur de *psm* a été utilisée durant le processus de propagation. Dans cette expérimentation, nous considérons comme intervalle de temps le moment de la recherche où la base de clauses apprises est nettoyée. Cet intervalle de temps est noté t_k avec $k > 0$ (le début de la recherche est à t_0). Soient $\mathcal{P}_{t_k}$ et $\mathcal{P}_{t_{k+1}}$ les interprétations représentant respectivement le *progress saving* aux étapes t_k et t_{k+1}. Considérons la fenêtre de temps entre t_k et t_{k+1} : lorsqu'une clause c provenant de la base de clauses apprises est utilisée dans le processus de propagation unitaire nous calculons $psm = psm_{\mathcal{P}_{t_k}}(c)$ et incrémentons la variable $\alpha(psm)$ (qui correspond au nombre de fois où une clause avec une telle valeur de *psm* est utilisée pour propager un littéral). Le nombre moyen de fois où une clause possédant un *psm* donné (axe des x) est utilisé dans le processus de propagation unitaire (axe des y), correspond alors à $\alpha(psm)$ divisé par le nombre total de fois où la base de clauses apprises a été nettoyée.

D'après la Figure 1, les clauses apprises possédant une petite valeur de *psm* sont plus souvent utilisées dans le processus de propagation que les clauses possédant une grande valeur de *psm*. Si nous regardons plus en détail, nous pouvons voir que les clauses les plus utilisées ont une valeur de *psm* proche de 10. Nous pouvons assurer, puisque l'expérimentation a été menée sur un très large panel d'instances, que dans la majorité des instances considérées la distribution de la valeur de *psm* ressemble à celle des deux courbes hautes de la Figure 1.

Cette première expérimentation illustre le fait que les clauses possédant une faible valeur de *psm* sont importantes dans le processus de propagation. Pour valider de manière expérimentale notre mesure nous avons choisi de l'intégrer dans une politique de réduction de la base de clauses apprises (clean(Δ) - ligne 12 de l'algorithme 1) et de comparer celle-ci avec les approches de réduction de clauses apprises existantes dans la littérature. Cette nouvelle stratégie de réduction de base a été introduite dans le solveur MINISAT 2.2. Comme pour les approches de réduction classique, l'ensemble des clauses apprises est tout d'abord trié par ordre croissant (ici, en fonction de leur *psm*). Lorsque deux clauses possèdent le même *psm*, nous utilisons la stratégie classique basée sur l'activité (VSIDS) pour les départager. Une fois les clauses triées la base de clauses apprises est réduite de moitié. Comme pour les autres stratégies, nous conservons toujours les clauses binaires.

Dans la suite, toutes nos expérimentations ont été menées sur un Quad-core Intel XEON X5550 avec 32Gb de mémoire. 292 instances ont été utilisées, elles sont issues de la compétition SAT 2009, catégorie industrielle. Pour l'ensemble des expérimentations le temps CPU est limité à 900 secondes.

Pour chaque solveur, nous indiquons le nombre d'instances résolues (#Résolues) ainsi que le nombre d'instances résolues satisfiables (#SAT) et insatisfiables (#UNSAT). Nous donnons aussi la moyenne de temps nécessaire pour résoudre une instance (tps moy).

Solveur	#Résolues (#SAT- #UNSAT)	tps moy
MINISAT^d	174 (68 - **106**)	142
MINISAT^d + *psm*	**177** (**73** - 104)	130
MINISAT^d + LBD	173 (71 - 102)	132

TABLE 1 – Résultats avec la séquence de réduction utilisée par défaut dans le solveur MINISAT.

La tableau 1 résume les résultats obtenus pas MINISAT^d [5], MINISAT^d + LBD [1] et MINISAT^d + psm utilisant la séquence de réduction proposée par défaut (notée MINISAT^d) dans MINISAT 2.2. Comme nous pouvons le voir, MINISAT^d+psm est le solveur qui résout le plus d'instances. C'est également le meilleur sur les instances SAT. Cette expérimentation montre l'efficacité de notre mesure dans le cas d'une politique utilisant la séquence de réduction proposée par défaut dans MINISAT 2.2.

Solveur	#Résolues (#SAT- #UNSAT)	tps moy
MINISAT^a	162 (68 - 94)	136
MINISAT^a + *psm*	163 (70 - 93)	140
MINISAT^a + LBD	**168** (**72** - **96**)	128

TABLE 2 – Résultats avec la séquence de réduction utilisée agressive.

Pour être plus juste dans la comparaison des trois approches, nous présentons aussi dans le tableau 2 les résultats obtenus lorsque la séquence de réduction est plus agressive, comme celle utilisée dans [1] (noté MINISAT^a). Dans cette expérimentation la séquence de réduction de la base de clauses apprises Δ est calculée de la manière suivante : $t_0 = 4000$ conflits et $t_k = t_{k-1} + 300$ conflits pour $k > 0$. En utilisant une stratégie de réduction plus agressive, les résultats obtenus par le LBD sont meilleurs que ceux obtenus à l'aide de la stratégie VSIDS et par notre mesure *psm*.

Pour résumer, ces premières expérimentations ont clairement montré la viabilité de notre nouvelle mesure. Cette dernière étant compétitive avec les autres méthodes existant dans la littérature.

4 Geler pour ne pas supprimer : une politique dynamique de réduction de la base de clauses

Dans la section 3, nous avons défini une nouvelle mesure basée sur la notion de *progress saving* pour identifier les clauses apprises importantes. Dans cette section, nous décrivons une approche dynamique de gestion de la base de clauses apprises. Cette approche est basée sur deux notions-clés. Premièrement, le *progress saving* évolue durant la recherche. Par conséquent, une clause peut être considérée comme inutile (avec une grande valeur de *psm*) à un instant de la recherche et devenir intéressante (avec une petite valeur de *psm*) à un autre moment de la recherche. Deuxièmement, déterminer si une clause donnée sera importante pour la preuve est un problème calculatoirement difficile. Toutes les politiques de nettoyage de la base de clauses apprises proposées dans la littérature ne sont pas sûres de ne pas supprimer une clause importante pour la suite de la recherche. Pour ces deux raisons, l'approche proposée introduit un nouveau concept qui consiste à geler certaines clauses. Lorsqu'une clause est considérée comme inutile à un instant de la recherche celle-ci devient gelée et sera réactivée lorsqu'elle sera de nouveau considérée comme importante. Plus précisément, geler (respectivement activer) une clause signifie que la clause est déconnectée (respectivement connectée) à la base des clauses

apprises et ne participe donc pas à la recherche (propagation unitaire...).

Cette politique de nettoyage de la base de clauses apprises basée sur le principe de désactivation et d'activation de clauses ne peut pas être utilisée avec les mesures habituelles. En effet, le LBD d'une clause apprise est statique car défini à la création de la clause, et ne change pas durant la recherche. Tandis que la mesure basée sur l'activité (basée sur VSIDS) est dynamique mais peut seulement mettre à jour les clauses attachées (les clauses doivent être utilisées pour être pondérées).

À présent, décrivons de manière formelle notre politique de nettoyage de la base de clauses apprises. Premièrement, comme le *psm* d'une clause est fortement dynamique, nous introduisons la notion de *déviation* entre deux *progress saving* successifs. Soit V_{t_k} l'ensemble des variables assignées par le solveur entre l'étape t_{k-1} et t_k. La déviation d_{t_k} est définie de la manière suivante : $d_{t_k} = \frac{h(\mathcal{P}_{t_k}, \mathcal{P}_{t_{k-1}})}{|V_{t_k}|}$, où h donne la distance de Hamming entre deux interprétations. Cette notion de déviation est une normalisation de la distance de Hamming pour ne prendre en compte que les variables réellement utilisées par le solveur. Elle permet de calculer l'évolution de l'interprétation liée au *progress saving* entre deux nettoyages de base successifs. Une déviation qui tend vers zéro indique que le solveur continue d'explorer le même espace de recherche, alors qu'une déviation importante indique que le solveur explore une partie différente de l'espace de recherche.

Pour obtenir une vue plus précise du comportement de la déviation, nous introduisons la notion de déviation minimale d^m. À l'instant t_k la déviation minimale est donnée par : $d^m_{t_k} = min\{d_{t_i} | 0 \leq i \leq k\}$.

L'utilisation de cette notion de déviation minimale nous permet d'affiner notre mesure *psm*. En effet, soit c une clause évaluée à l'étape t_k, si $psm_{\mathcal{P}_{t_k}}(c) > d^m_{t_k} \times |c|$ alors la clause c à de fortes chances d'être satisfaite dans le futur, sinon elle sera sûrement utilisée dans le processus de propagation.

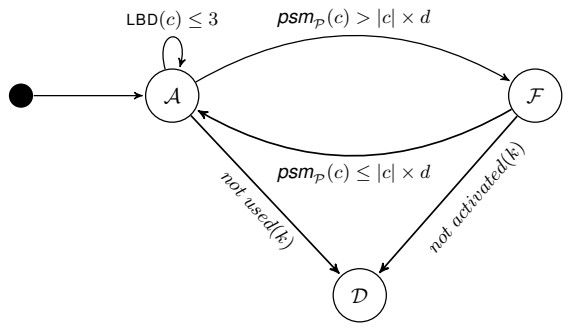

FIGURE 2 – Cycle de vie d'une clause apprise.

La Figure 2 décrit notre approche sous la forme d'un automate d'état. À chaque étape de nettoyage t_k, les clauses peuvent passer d'un état à un autre suivant certaines conditions.

Pour commencer, notons qu'une clause apprise c possède trois états :

1. *État activé* $\mathcal{A}$: c est activée et attachée, elle est utilisée par le solveur ;
2. *État gelé* $\mathcal{F}$: c est gelée *i.e.* c n'est pas attachée ;
3. *État mort* $\mathcal{D}$: c est supprimée.

À présent, décrivons les différentes transitions de l'automate à état.

- Chaque fois qu'une nouvelle clause est apprise celle-ci entre dans l'état $\mathcal{A}$;
- Une clause $c \in \mathcal{A}$ avec une valeur de LBD ($lbd(c) \leq 3$ sur la figure) reste dans l'état $\mathcal{A}$ jusqu'à la fin du processus de recherche ;
- Une clause $c \in \mathcal{A}$ telle que $\frac{psm_{\mathcal{P}_{t_k}}(c)}{|c|} > d^m_{t_k}$ passe à l'état gelé $\mathcal{F}$;
- Une clause $c \in \mathcal{F}$ telle que $\frac{psm_{S_{n_i}}(c)}{|c|} \leq d^m_{n_i}$ passe à l'état activé $\mathcal{A}$;
- Une clause $c \in \mathcal{F}$ qui n'a pas été activée pendant k étapes de nettoyage est supprimée. De manière similaire, une clause $c \in \mathcal{A}$ restant active pendant plus de k étapes sans participer à la recherche est aussi supprimée. Dans les deux cas, cette clause entre dans l'état $\mathcal{D}$. Pour notre étude expérimentale la valeur de k a été fixée à 7.

Un des principaux avantages de notre méthode réside dans le fait qu'il est possible d'augmenter la fréquence de nettoyage de la base de clauses apprises sans l'inconvénient de supprimer une clause importante pour la suite de la recherche. Nous pouvons donc utiliser une politique de nettoyage de base très agressive. Nous avons choisi $t_0 = 500$ et $t_{k+1} + 100$ conflits.

Pour valider notre approche, nous avons étudié expérimentalement le comportement des clauses au sein de l'automate à état. La Figure 3 montre, pour les mêmes instances que celles considérées dans la Figure 1, le nombre de clauses actives, gelées et supprimées ainsi que le nombre de transitions de l'état actif à l'état gelé et vice versa. Ces différentes données sont représentées sur l'axe y des ordonnées tandis que sur l'axe des abscisses x représente le nombre d'opérations de nettoyage de la base de clauses apprises. Pour des raisons de lisibilité, l'ensemble des courbes a été lissé. Pour l'ensemble de ces instances, le nombre de clauses gelées ($Frozen$) et le nombre de clauses actives ($Active$) sont relativement similaires. En ce qui concerne le taux de transfert d'un état à un autre, nous pouvons remarquer que le nombre de clauses passant de l'état actif à l'état gelé ($\mathcal{A} \rightarrow \mathcal{F}$) est plus grand que le nombre de clauses passant de l'état gelé à l'état actif ($\mathcal{F} \rightarrow \mathcal{A}$). Ceci s'explique par le fait que la base des clauses apprises est

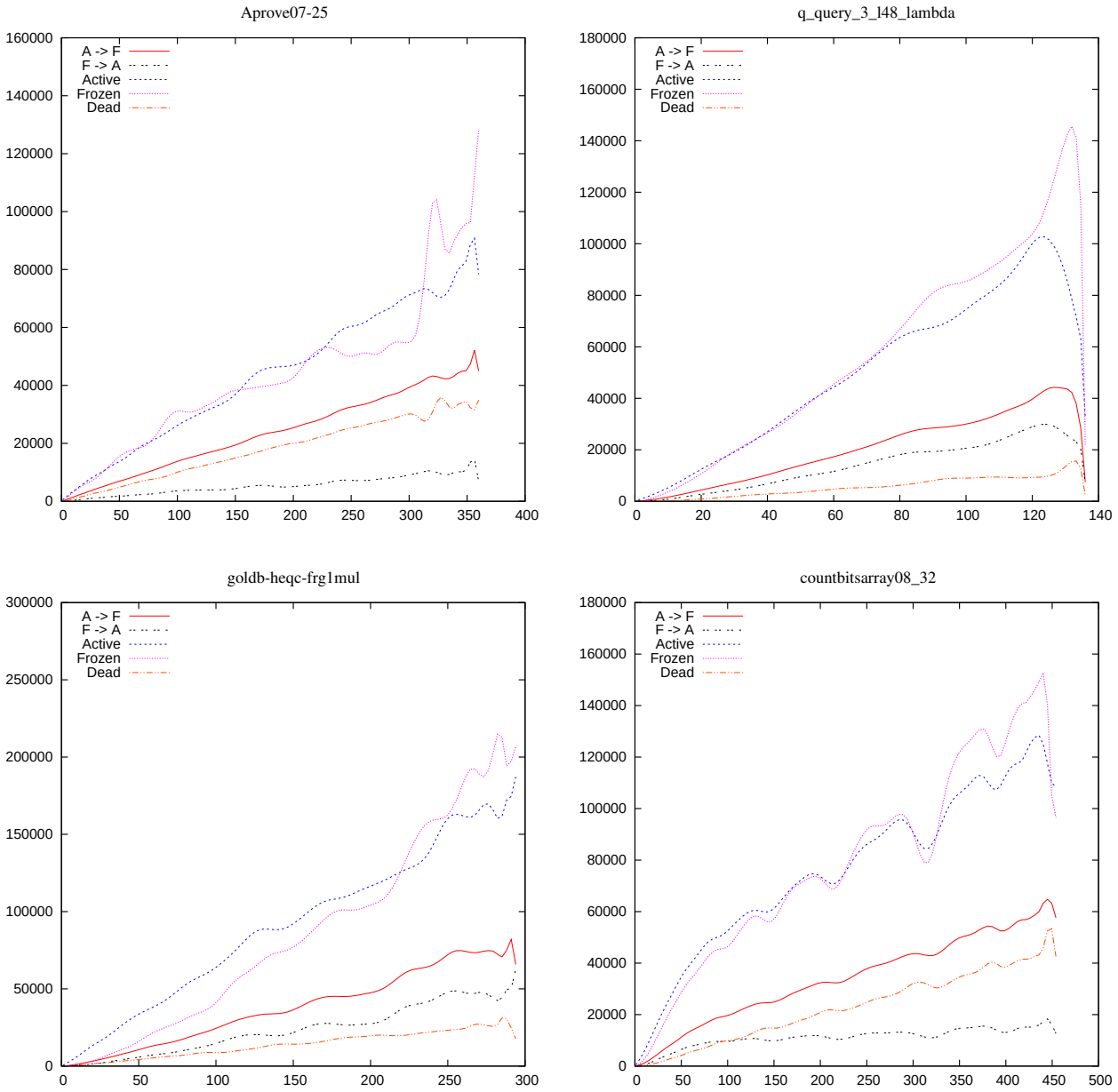

FIGURE 3 – Étude du taux de transfert.

sans cesse alimentée par de nouvelles clauses (ces clauses sont initialement dans l'état actif $\mathcal{A}$). Pour terminer, nous pouvons observer qu'à chaque étape du nettoyage certaines clauses sont définitivement supprimées ($Dead$).

5 Expérimentations

Cette section est divisée en deux parties. Dans la première partie, nous comparons notre approche dynamique de nettoyage de la base de clauses apprises (psm_{dym}) vis-à-vis des approches de la littérature (LBD, VSIDS) décrites dans la section 3. Dans la seconde partie, nous comparons notre approche avec trois autres solveurs de l'état de l'art : (i) le solveur GLUCOSE qui utilise le LBD pour choisir les clauses à conserver ou à supprimer lors de la réduction de la base de clauses apprises, une stratégie de redémarrage dynamique et d'autres composantes décrites dans [1] ; (ii) le solveur LINGELING, dont un ensemble d'améliorations permet d'augmenter la puissance de raisonnement (*ex :* les clauses bloquées [8]) ; et finalement, (iii) le solveur CRYPTOMINISAT, qui implémente nombre des améliorations se trouvant dans la littérature (*ex :* vivification, raisonnement sur les clauses xor ...). La description de ces trois solveurs est disponible sur le site de la SATRACE 2010 (`http://baldur.iti.uka.de/sat-race-2010`).

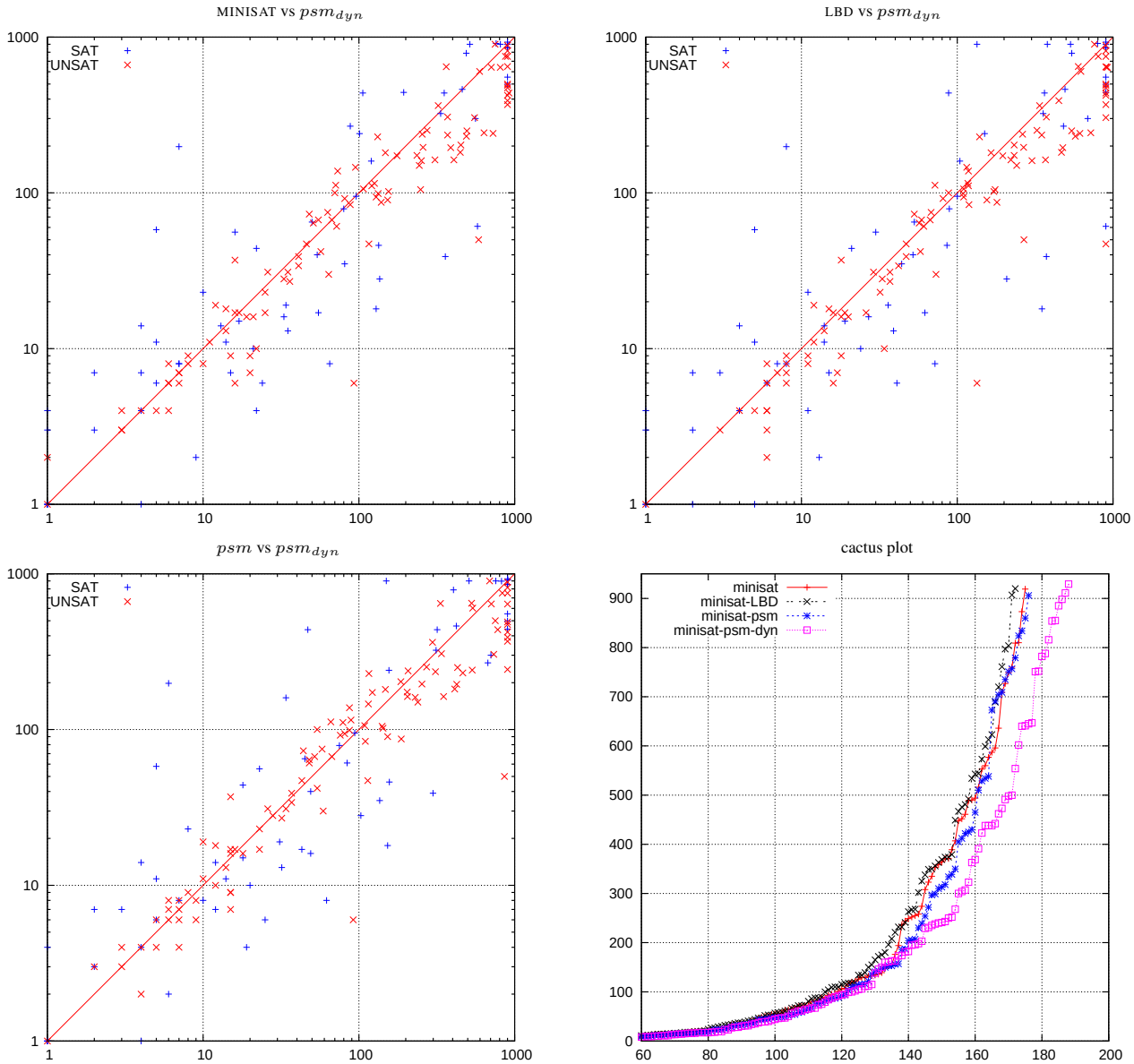

FIGURE 4 – Comparaison avec différentes stratégies de netoyage de la base de clauses apprises.

Pour l'ensemble des expérimentations, nous avons choisi d'utiliser MINISAT 2.2 comme solveur de base pour implémenter notre stratégie dynamique de réduction de la base de clauses apprises. Le solveur résultant est nommé MINISAT-psm_{dyn}. Le code source ainsi que des données additionnelles sont disponibles sur `http://www.cril.fr/~lagniez/ressource.html`.

5.1 Comparaison avec différentes stratégie de réduction

Nous comparons notre politique dynamique de nettoyage de la base de clauses apprises, appelée MINISAT-psm_{dyn}, avec MINISAT classique, et MINISAT avec la stratégie de réduction basée sur le *psm* (MINISAT-*psm*) et sur le LBD (MINISAT-LBD) (comme pour la section 3). La Figure 4 résume les résultats obtenus. Les trois nuages de points correspondent aux comparaisons de MINISAT-$psm_{psm_{dyn}}$ avec les trois autres solveurs. Chaque point correspond à une instance. Un point en dessous de la diagonale signifie que cette instance a été résolue plus rapidement avec MINISAT-$psm_{psm_{dyn}}$. Les instances SAT et UNSAT sont respectivement représentées sur la figure par le signe plus et multiplié. Cette figure contient aussi les courbes permettant de comparer le nombre d'instances résolues en fonction du temps pour les quatre solveurs.

Il est clair que notre approche de réduction de la base des clauses apprises permettant de geler des clauses est nettement meilleure que les trois autres. Le solveur résout 189 instances (76 SAT et 113 UNSAT), ce qui est meilleur que les autres solveurs (voir tableau 1). De plus, comme nous pouvons le voir sur le nuage de point, MINISAT-$psm_{psm_{dyn}}$

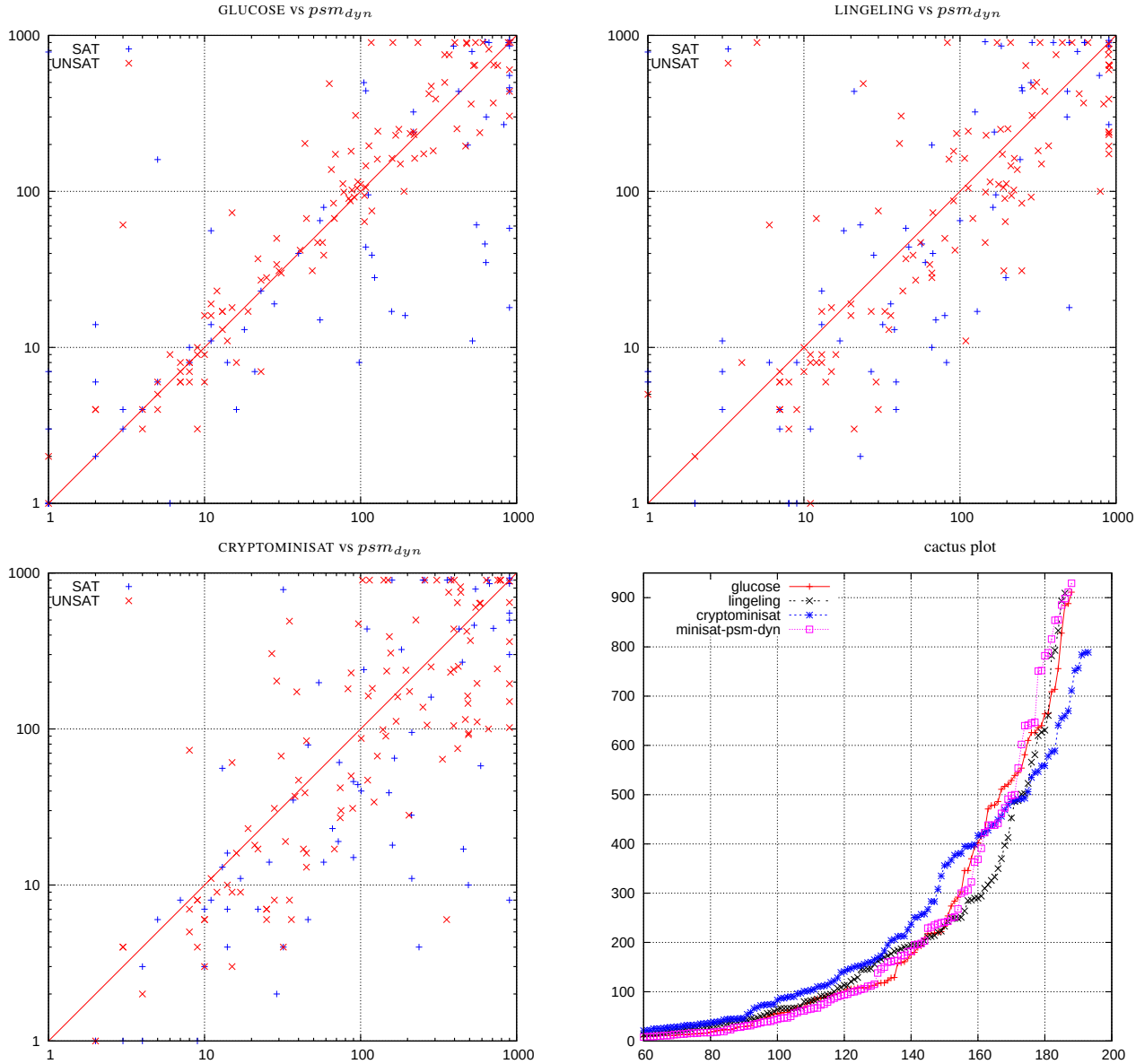

FIGURE 5 – Comparaison avec les solveurs de l'état de l'art : GLUCOSE, LINGELING and CRYPTOMINISAT.

résout les instances plus rapidement que les autres solveurs.

5.2 Comparaison avec les solveurs de l'état de l'art

Les résultats obtenus par les différents solveurs sont : LINGELING résout 187 instances (77 SAT et 110 UNSAT), GLUCOSE 189 instances (70 SAT et 119 UNSAT) et CRYPTOMINISAT 194 instances (74 SAT et 120 UNSAT). Donc notre approche basée sur la notion de réduction dynamique de la base de clauses apprises est compétitive avec les solveurs de l'état de l'art (rappelons qu'elle résout 189 instances (76 SAT et 113 UNSAT)). Ces résultats sont très encourageants puisque dernièrement (*ex* : redémarrages dynamiques, clauses bloquées, ...).

6 Conclusion

Dans ce papier, nous avons introduit une nouvelle mesure permettant d'identifier les clauses importantes pour la suite de la recherche. Cette mesure est dynamique (contrairement à la mesure basé sur le LBD) et peut être calculée sur des clauses ne participant pas à la recherche (contrairement à la mesure basée sur VSIDS). Grâce à ces propriétés, une nouvelle stratégie dynamique de réduction de la base de clauses apprises a été proposée. Cette nouvelle stratégie est basée sur l'activation et la désactivation de clauses, alors que avec les stratégies de réduction de l'état de l'art où les clauses sont définitivement supprimées. Un ensemble d'expérimentations a montré que cette approche a un très bon comportement.

En perspective, nous comptons étudier plus finement la

manière selon laquelle évolue l'interprétation $\mathcal{P}$, ce qui nous permettrait de décider quand une réduction de la base de connaissance serait souhaitable. De plus, comme la notion de *progress saving* a déjà été utilisée dans d'autres travaux [3, 14] nous pensons qu'il serait intéressant de combiner ces différentes approches.

Références

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of International Joint Conference on Artificial Intelligence*, pages 399–404, 2009.

[2] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22 :319–351, 2004.

[3] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, pages 28–33, 2008.

[4] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[5] N. Een and N. Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.

[6] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[7] J. Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.

[8] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *TACAS*, pages 129–144, 2010.

[9] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI/IAAI*, pages 203–208, 1997.

[10] J. Marques-Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *proceedings of ICCAD*, pages 220–227, 1996.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.

[12] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.

[13] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. In *cp09*, pages 654–668, 2009.

[14] Knot Pipatsrisawat and Adnan Darwiche. Width-based restart policies for clause-learning satisfiability solvers. In *12th International Conference on Theory and Applications of Satisfiability Testing - SAT 2009*, pages 341–355, 2009.

[15] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD*, pages 279–285, 2001.