An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers*

Mao Luo[1], **Chu-Min Li**[2†], **Fan Xiao**[1], **Felip Manyà**[3], **Zhipeng Lü**[1†]
[1]School of Computer Science, Huazhong University of Science and Technology, Wuhan, China
[2]MIS, University of Picardie Jules Verne, Amiens, France
[3]Artificial Intelligence Research Institute (IIIA-CSIC), Barcelona, Spain
{maoluo,fanxiao,zhipeng.lv}@hust.edu.cn, chu-min.li@u-picardie.fr, felip@iiia.csic.es

Abstract

Learnt clauses in CDCL SAT solvers often contain redundant literals. This may have a negative impact on performance because redundant literals may deteriorate both the effectiveness of Boolean constraint propagation and the quality of subsequent learnt clauses. To overcome this drawback, we define a new inprocessing SAT approach which eliminates redundant literals from learnt clauses by applying Boolean constraint propagation. Learnt clause minimization is activated before the SAT solver triggers some selected restarts, and affects only some learnt clauses during the search process. Moreover, we conducted an empirical evaluation on instances coming from the hard combinatorial and application categories of recent SAT competitions. The results show that a remarkable number of additional instances are solved when the approach is incorporated into five of the best performing CDCL SAT solvers (Glucose, TC_Glucose, COMiniSatPS, MapleCOMSPS and MapleCOMSPS_LRB).

1 Introduction

Modern Conflict-Driven Clause Learning (CDCL) SAT solvers are routinely used as core solving engines in many real-world applications. Their ability to solve challenging problems comes from the combination of different ingredients: variable selection heuristics, Boolean constraint propagation, clause learning, restarts, clause database management, data structures, preprocessing and inprocessing.

Formula simplification techniques applied during preprocessing have proven useful in enabling efficient SAT solving for real-world application domains (e.g. [Bacchus and Winter, 2003; Eén and Biere, 2005; Piette *et al.*, 2008]). The most successful preprocessing techniques include variants of bounded variable elimination, addition or elimination of redundant clauses, detection of subsumed clauses and suitable combinations of them. They aim mostly at reducing the number of clauses, literals and variables in the input formula.

More recently, some solvers have significantly boosted their performance by interleaving formula simplification techniques with CDCL search. Among such *inprocessing techniques* [Järvisalo *et al.*, 2012], we mention local and recursive clause minimization [Beame *et al.*, 2004; Sörensson and Biere, 2009], which remove redundant literals from learnt clauses immediately after their creation; clause distillation in a concurrent context [Wieringa and Heljanko, 2013]; and on-the-fly clause subsumption [Han and Somenzi, 2009; Hamadi *et al.*, 2009], which efficiently removes clauses subsumed by the resolvents derived during clause learning. Nevertheless, there is a common belief that clause simplification techniques that bring substantial gains in preprocessing do not increase significantly the performance of solvers when they are applied to learnt clauses during the search [Biere, 2011; Wotzlaw *et al.*, 2013].

Given the impact of clause learning on the resolution of practical instances, the objective of this paper is to devise and implement a new inprocessing learnt clause minimization approach able to remove redundant literals from learnt clauses for CDCL solvers. Our approach is based on Boolean constraint propagation, or more precisely unit (clause) propagation, which is notably time-consuming on large instances. Hence, in what follows, we must carefully decide how to deal with the following issues: (1) when should the clause minimization procedure be activated? (2) which learnt clauses should be minimized? and (3) how should unit propagation be applied?. By adequately answering these three questions, the proposed approach reaches a good trade-off between effectiveness and efficiency. Observe that such an approach was missing in the literature, explaining the fact that the winners of the 2016 SAT competition did not use unit propagation to minimize learnt clauses.

We evaluated our approach on five of the best performing CDCL SAT solvers: Glucose [Audemard and Simon, 2009], COMiniSatPS [Oh, 2016], MapleCOMSPS [Liang *et al.*, 2016c], MapleCOMSPS_LRB [Liang *et al.*, 2016b] and TC_Glucose [Moon and Mary, 2016]. The experimental results show that the proposed approach allows them to solve a remarkable number of additional instances coming from the hard combinatorial and application categories of the 2014 and 2016 SAT competitions.

*Research partially supported by the National Natural Science Foundation of China (Grants no. 61370183, 61370184, 61472147), the Generalitat de Catalunya grant AGAUR 2014-SGR-118, and the MINECO-FEDER project RASO TIN2015-71799-C2-1-P.

†Corresponding author

We quote a statement of two leading SAT solver developers [Audemard and Simon, 2012] to better appreciate the significance of our contributions:

"We must also say, as a preliminary, that improving SAT solvers is often a cruel world. To give an idea, improving a solver by solving at least ten more instances (on a fixed set of benchmarks of a competition) is generally showing a critical new feature. In general, the winner of a competition is decided based on a couple of additional solved benchmarks."

The previous statement was also quoted in [Liang *et al.*, 2016a] to appreciate the advance brought about by the CHB branching heuristic in CDCL solvers.

The paper is organized as follows: Section 2 gives some basic concepts about SAT and CDCL solvers. Section 3 presents some related works on inprocessing and preprocessing clause minimization. Section 4 describes the new learnt clause minimization approach, as well as how it is implemented in different CDCL solvers. Section 5 reports on the experimental investigation. Section 6 contains the concluding remarks.

2 Preliminaries

Let $V = \{x_1, \ldots, x_n\}$ be a set of propositional variables. A literal is a propositional variable x or its negation $\overline{x}$. A clause is a disjunction of literals. The size of a clause refers to the number of literals it contains. A unit clause contains exactly one literal. A Conjunctive Normal Form (CNF) formula is a conjunction of clauses. A CNF formula is also represented by the set of its clauses, and a clause by the set of its literals. A clause C' subsumes a clause C iff $C' \subset C$.

Let $\rho : V \rightarrow \{0, 1\}$ be a truth assignment. The assignment ρ satisfies a literal x ($\overline{x}$) iff $\rho(x) = 1$ ($\rho(x) = 0$), satisfies a clause iff it satisfies any of its literals, and satisfies a CNF formula iff it satisfies all its clauses. A CNF formula F is satisfiable iff there is at least one satisfying assignment. The empty clause ($\square$) is always unsatisfiable, and represents a conflict or a failure. SAT is the problem of deciding whether a given CNF formula is satisfiable. Two formulas are logically equivalent iff they are satisfied by the same assignments. A literal l is redundant in a clause C of a formula F if removing l from C results in a formula logically equivalent to F.

Given a CNF formula F and a literal l, we define $F|_l$ to be the CNF formula resulting from F after removing the clauses containing an occurrence of l and all the occurrences of $\bar{l}$. We recursively define Boolean constraint propagation, or more precisely Unit Propagation (UP), using the following two rules: (R1) $UP(F) = F$ if F does not contain any unit clause; and (R2) $UP(F) = UP(F|_l)$ if there is a unit clause $\{l\}$ in F. In the latter case, we say that the literal l is asserted by UP, and it must hold that $\rho(l) = 1$. The literal l is deduced or asserted by $UP(F)$ if the unit clause $\{l\}$ is obtained by applying R2. When $UP(F \cup \{l\})$ is computed, we say that l is propagated in F.

In the sequel, we use $UP(F)$ to denote the CNF formula obtained from F after repeatedly applying R2 until there is no unit clause or the empty clause is derived. Concretely, $UP(F) = \square$ if repeatedly applying R2 derives the empty clause. Otherwise, $UP(F)$ denotes the CNF formula in which all unit clauses have been propagated using R2.

A CDCL solver [Marques-Silva and Sakallah, 1999; Moskewicz *et al.*, 2001] performs a non-chronological backtrack search in the space of partial truth assignments. Concretely, the solver repeatedly picks a decision literal l_i and applies unit propagation in $F \cup \{l_1, l_2, \ldots, l_i\}$ (i.e., computes $UP(F \cup \{l_1, l_2, \ldots, l_i\})$ until the empty formula or the empty clause are derived. If the empty clause is derived, the reasons are analyzed and a clause (nogood) is learnt using a particular method, usually the First UIP (Unique Implication Point) scheme [Zhang *et al.*, 2001]. The learnt clause is then added to the clause database.

Let F be a CNF formula, let l_i be the i^{th} decision literal picked by a CDCL solver, and let $F_i = UP(F \cup \{l_1, l_2, \ldots, l_i\})$. We say that l_i and all the literals asserted when computing $UP(F_{i-1} \cup \{l_i\})$ belong to level i. Literals asserted in $UP(F)$ do not depend on any decision literal and form level 0. When UP deduces the empty clause and a learnt clause is extracted from the conflict analysis, the solver cancels the literal assertions in the reverse order until the level where the learnt clause contains one non-asserted literal, and continues the search from that level after propagating the non-asserted literal. Under certain conditions, the solver cancels the literal assertions until level 0 and restarts the search from level 0. Since too many learnt clauses slow down the solver and may overflow the available memory, the solver periodically removes a subset of learnt clauses using a particular clause deletion strategy.

The literals of a learnt clause C are partitioned w.r.t. their assertion level. The number of sets in the partition is called the Literal Block Distance (LBD) of C. As Audemard and Simon (2009) showed, LBD measures the quality of learnt clauses. Clauses with small LBD values are considered to be more relevant. The best performing CDCL solvers of the recent SAT competitions use LBD as a measure of the quality of learnt clauses to decide which clauses must be removed or retained. Moreover, solvers like Glucose and its derivatives use the LBD of recent learnt clauses to decide when a restart must be triggered.

3 Related Work on Clause Minimization

Clause minimization (see e.g. [Eén and Biere, 2005; Han and Somenzi, 2007; Piette *et al.*, 2008; Sörensson and Biere, 2009; Wotzlaw *et al.*, 2013; Marques-Silva, 2000; Han and Somenzi, 2009; Hamadi *et al.*, 2009; Wieringa and Heljanko, 2013]) both before and during the search is crucial for the performance of CDCL SAT solvers for two reasons: (i) shorter clauses need less memory and, more importantly, (ii) shorter clauses are easier to become unit, and thus increase the power of unit propagation.

The most effective approach to remove redundant literals in learnt clauses probably is *recursive clause minimization* [Sörensson and Biere, 2009], which can be specified using the notion of involved literal. Let $C = l_1 \vee l_2 \vee \cdots \vee l_k$ be a newly created learnt clause. Then, (i) all literals in C are involved; and (ii) for any other literal l, if there is a clause $C' = l'_1 \vee l'_2 \vee \ldots \vee l'_{k'} \vee \bar{l}$ such that $l'_1, l'_2, \ldots, l'_{k'}$ are involved, then l is involved. Recursive clause minimization tests each literal l_i of C. If there is a clause $C' = l'_1 \vee l'_2 \vee \ldots \vee l'_{k'} \vee \bar{l}_i$

such that $l'_1, l'_2, \ldots, l'_{k'}$ are involved, then l_i is removed from C. MiniSAT 1.13 implements recursive clause minimization in a very ingenious way and applies it to each new learnt clause. This approach is now part of many SAT solvers, and frequently removes more than 30% of literals.

Another effective inprocessing clause minimization approach is based on binary implication graphs [Heule *et al.*, 2011]. Given a CNF formula F, the algorithm first performs a depth-first search over the binary implication graph to assign a time stamp to each literal in the graph. It then uses these time stamps to discover and remove various kinds of redundancy, including redundant literals in learnt clauses. The algorithm is linear in the total number of literals in F and is realized in a function named $Stamp$ in Lingeling [Biere, 2010] and MapleCOMSPS_LRB. As an inprocessing, the function is executed upon some restarts to minimize learnt clauses.

In solvers such as Glucose and MapleCOMSPS, a restriction of the resolution rule is applied to minimize a newly created learnt clause $C = l_1 \vee l_2 \vee \cdots \vee l_k$ whose LBD or size is smaller than a prefixed limit. The first literal l_1 of C is called the asserting literal. For any $i > 1$, if $l_1 \vee \bar{l}_i$ is a clause in F, then Glucose and MapleCOMSPS remove l_i from C.

Conflict analysis consists in performing successive resolution steps from a conflicting clause. At each step, a new resolvent is derived from two clauses C' and C'', where C' is the conflicting clause or a previous resolvent, and C'' is an existing clause. If $C'' = l \vee C$ and $C' = \bar{l} \vee D$, where l is a literal, and C and D are disjunction of literals with $D \subseteq C$, the resolvent of C' and C'' is C and subsumes C'', meaning that l is redundant in C'' and can be removed from C''. This approach allows to minimize both original clauses and learnt clauses during the search and was proposed independently in [Han and Somenzi, 2009] and [Hamadi *et al.*, 2009]. It was implemented in CryptoMinisat 2.5.0 [Soos, 2010] for on-the-fly clause improvement.

Preprocessing clause minimization can devote more time to each clause and still improve the performance of solvers. Piette *et al.* (2008) applied the following rules in a vivification procedure after sorting the literals of a clause $C = l_1 \vee l_2 \vee \cdots \vee l_k$ of the input formula F using a MOMS-style [Jeroslow and Wang, 1990] ordering:

1. If for some $i \in \{1, 2, \ldots, k-1\}$ and some $j > i$, $UP((F \setminus \{C\}) \cup \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_i\})$ deduces l_j, then $F \leftarrow (F \setminus \{C\}) \cup \{l_1 \vee l_2 \vee \ldots \vee l_i \vee l_j\}$.
2. If for some $i \in \{1, 2, \ldots, k-1\}$ and some $j > i$, $UP((F \setminus \{C\}) \cup \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_i\})$ deduces $\bar{l}_j$, then $F \leftarrow (F \setminus \{C\}) \cup \{l_1 \vee l_2 \vee \ldots \vee l_i \vee \cdots \vee l_{j-1} \vee l_{j+1} \vee \cdots \vee l_k\}$.
3. If for some $i \in \{1, 2, \ldots, k-1\}$, $UP((F \setminus \{C\}) \cup \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_i\}) = \Box$, then extract a nogood C_l from the conflict using the First UIP scheme. If $C_l \subset C$ or $|C_l| < |C|$ then $F \leftarrow (F \setminus \{C\}) \cup C_l$, otherwise $F \leftarrow (F \setminus \{C\}) \cup \{l_1 \vee l_2 \vee \ldots \vee l_i\}$.

Independently, Han and Somenzi (2007) proposed a similar approach called distillation.

Vivification and distillation are usually applied during preprocessing, because they need many time-consuming unit propagations. One exception is CryptoMinisat, which implements a limited distillation as inprocessing. In the context of parallel solvers, Wieringa and Heljanko (2013) use a concurrent thread to repeatedly select and minimize the best learnt clause (according to some criteria such as clause size or LBD) among the newest 1000 learnt clauses using unit propagation.

Nevertheless, to the best of our knowledge, none of the awarded CDCL solvers in the 2016 SAT Competition uses a distillation or vivification approach to minimize learnt clauses. In particular, the solver Riss6, the silver medal winner of the main track, disables a vivification based learnt clause minimization used in Riss 5.05, because it turned out to be ineffective for formulas of more recent years according to [Manthey *et al.*, 2016]. In fact, it is generally believed that inprocessing clause minimization based on unit propagation can hardly be made effective for CDCL solvers. Wotzlaw *et al.* (2013) applied distillation to minimize the 100 most active clauses during the search, and concluded that inprocessing distillation is generally not as effective as preprocessing distillation, as well as that developing effective inprocessing techniques is non-trivial and requires in-depth knowledge about how different techniques can be combined and integrated efficiently into the solvers.

4 Minimizing Learnt Clauses During Search

The main objective of this paper is to devise and implement an effective and efficient inprocessing approach to minimize learnt clauses using unit propagation. This inprocessing was missing in the literature, as is illustrated by the fact that the winners of the 2016 SAT competition did not use unit propagation to minimize learnt clauses. We first describe the general principle used to efficiently minimize learnt clauses, and then present its implementation in Glucose, TC_Glucose, COMiniSatPS, MapleCOMSPS and MapleCOMSPS_LRB.

4.1 General Principle

In order to make learnt clause minimization effective and efficient, we have to answer the following questions:

1. When should we minimize the learnt clauses? It seems clear enough that learnt clause minimization should be activated at level 0 to ensure that the minimization is independent of any branching decision. In other words, it should be activated upon a restart. So, the relevant questions are: should we activate learnt clause minimization for every restart? If not, how do we determine the restarts upon which we activate learnt clause minimization?
2. Should we minimize each learnt clause? If not, which learnt clauses should be minimized?
3. Clause minimization based on unit propagation clearly depends on the ordering selected to propagate the literals. What is the best order in which the literals of a learnt clause should be propagated during learnt clause minimization?

The most satisfactory answers to these questions depend on other techniques implemented in the solver. Nevertheless, we want to argue the following general principle for guiding the implementation of learnt clause minimization:

Algorithm 1: minimization$(F, L, nbNewLearnts, \sigma)$, a generic learnt clause minimization algorithm

```
Input: $F$: A CNF formula; $L$: a set of learnt clauses such
       that $L \cap F = \emptyset$; $nbNewLearnts$: the number of
       clauses learnt since the last execution of function
       $minimizeL$; $\sigma$: the number of times function
       $minimizeL$ was executed so far.
Output: A set of learnt clauses.
1 begin
2     if $liveRestart(nbNewLearnts, \sigma)$ then
3         $L \leftarrow minimizeL(F, L)$; /* call Algorithm 2 */
4         $nbNewLearnts \leftarrow 0$; $\sigma + +$;
5     return $L$;
6 end
```

Algorithm 2: minimizeL(F, L): minimizing a set of learnt clauses

```
Input: $F$: A CNF formula; $L$: a set of learnt clauses such
       that $L \cap F = \emptyset$.
Output: A set of learnt clauses.
1  begin
2      foreach $C = l_1 \vee l_2 \vee \cdots \vee l_k \in L$ do
3          if $!liveClause(C)$ then continue;
4          $L \leftarrow L \setminus \{C\}$; $C \leftarrow sort(C)$;
5          $C' \leftarrow \emptyset$; /* $C'$ will be the minimized clause */
6          for $i \leftarrow 1$ to $k$ do
7              if $(R \leftarrow UP(F \cup L \cup \overline{C}' \cup \{\bar{l}_i\})) = \Box$ then
8                  $C' \leftarrow conflAnalysis(F, L, \overline{C}' \cup \{\bar{l}_i\}, R)$;
9                  break;
10             else if $UP(F \cup L \cup \overline{C}' \cup \{l_i\}) \neq \Box$ then
11                 $C' \leftarrow C' \vee l_i$; /* add $l_i$ into clause $C'$ */
12         $L \leftarrow L \cup \{C'\}$;
13     return $L$;
14 end
```

1. Clause minimization should not be activated at each restart. In fact, the number of newly learnt clauses per restart may not be sufficient for unit propagation to easily deduce a conflict. So, we adopt the clause reduction strategy of Glucose to activate learnt clause minimization. More precisely, we define function $liveRestart(nbNewLearnts, \sigma)$ to determine if learnt clause minimization has to be activated at the beginning of a restart. In the function, $nbNewLearnts$ and σ are global variables representing, respectively, the number of clauses which were learnt since the last learnt clause minimization, and the number of learnt clause minimizations performed so far.
2. Chanseok Oh (2016) empirically showed that learnt clauses with high LBD values are not very useful to solve practical SAT instances. Indeed, the LBD of a learnt clause is correlated to the minimum number of decisions needed to make the clause failed, meaning that it is much harder to reduce the LBD of a learnt clause than to reduce its size. Moreover, clauses with high LBD are generally long, needing more unit propagations to be minimized. In Section 5, we will give empirical evidence showing that minimizing clauses with high LBD is very costly and useless. So, we should minimize learnt clauses with small LBD values. More precisely, we define a function $liveClause(C)$ to determine if a learnt clause C has to be minimized according to its LBD.
3. A function $sort(C)$ is defined to sort the literals before minimizing the learnt clause C.

Algorithm 1 realizes this general principle and is called at the beginning of each restart. Function $minimizeL(F, L)$ is defined in Algorithm 2. We will precisely define function $liveRestart(nbNewLearnts, \sigma)$ later when we implement Algorithm 1 in a solver.

Algorithm 2 works as follows:

1. Let $C = l_1 \vee l_2 \vee \cdots \vee l_k$ be a learnt clause in L such that $liveClause(C)$ is true. If UP$(F \cup \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_i\}) = \Box$, then $F \cup \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_i\}$ is unsatisfiable and the clause $l_1 \vee l_2 \vee \cdots \vee l_i$ is a logical consequence of F and can be added to F to subsume C without affecting satisfiability. Let $\overline{C}' = \{\bar{l}'_1, \bar{l}'_2, \ldots, \bar{l}'_{i-1}\}$. Function $conflAnalysis(F, L, \overline{C}' \cup \{\bar{l}_i\}, R)$ retraces the implication graph from the conflicting clause R derived by UP until the literals in $\overline{C}' \cup \{\bar{l}_i\}$, in order to collect the literals in $\overline{C}' \cup \{\bar{l}_i\}$ from which there is a path to the conflicting clause R in the implication graph. The function returns a disjunction of the negation of the collected literals. Note that Function $conflAnalysis(F, L, \overline{C}' \cup \{\bar{l}_i\}, R)$ does not use the first UIP scheme, which is different from Rule 3 in the vivification approach in [Piette *et al.*, 2008]. Algorithm 2 (line 7) applies UP$(F \cup L \cup \overline{C}' \cup \{\bar{l}_i\})$ incrementally for efficiency reasons: $\bar{l}_i$ is propagated in the formula returned by UP$(F \cup L \cup \overline{C}')$ after checking if l_i or $\bar{l}_i$ is not asserted. UP$(F \cup L \cup \overline{C}' \cup \{\bar{l}_i\})$ is implemented using the unit propagation function and the watched literal data structures of the solver.
2. If UP$(F \cup \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_{i-1}, l_i\}) = \Box$, then $C'' = l_1 \vee l_2 \vee \cdots \vee l_{i-1} \vee \bar{l}_i$ is a logical consequence of F. The resolvent of $C = l_1 \vee l_2 \vee \cdots \vee l_k$ $(k \geq i)$ and C'' does not contain l_i and subsumes C. So, Algorithm 2 does not add l_i to the new clause C' in this case. Actually, Algorithm 2 (line 10) computes UP$(F \cup L \cup \overline{C}' \cup \{l_i\})$ by checking if UP$(F \cup L \cup \overline{C}')$ deduces $\bar{l}_i$, instead of propagating l_i, for efficiency reasons.

The implementation of Algorithm 2 in a solver needs the definition of functions $liveClause(C)$ and $sort(C)$, which will be discussed later. A common constraint we impose is that if a learnt clause C has been minimized in a previous call of Algorithm 2, $liveClause(C)$ returns false. In this way, a learnt clause is minimized at most once in our approach.

4.2 Minimizing Learnt Clauses in Glucose and TC_Glucose

Glucose is a very efficient CDCL solver developed from Minisat [Eén and Sörensson, 2003]. It was habitually awarded in SAT competitions between 2009 and 2014, and is the base solver of many other awarded solvers.

Glucose was the first solver which incorporated the LBD measure in the clause learning mechanism and adopted an aggressive strategy for clause database reduction. We used Glucose 3.0, in which the reduction process is fired once the number of clauses learnt since the last reduction reaches $first + 2 \times inc \times \sigma$, where $first = 2000$ and $inc = 300$ are parameters, and σ is the number of database reductions performed so far. The learnt clauses are first sorted in decreasing order of their LBD values, and then the first half of learnt clauses are removed except for the binary clauses, the clauses whose LBD value is 2, and the clauses that are reasons of the current partial assignment. Note that the reduction process is not necessarily fired at level 0.

Glucose 3.0 also features a fast restart mechanism which is independent of the clause database reduction. Roughly speaking, Glucose restarts the search from level 0 when the average LBD value in recent learnt clauses is high compared with the average LBD value of all the learnt clauses.

Solver TC_Glucose is like Glucose 3.0 but it uses a tie-breaking technique for VSIDS, and the CHB branching heuristic [Liang *et al.*, 2016a] instead of the VSIDS branching heuristic for small instances. It was the best solver of the hard combinatorial category in the 2016 SAT Competition.

Learnt clause minimization in Glucose and TC_Glucose is implemented by defining the three functions in Algorithm 1 and Algorithm 2 as follows:

Function $liveRestart(nbNewLearnts, \sigma)$ returns true iff the learnt clause reduction process was fired in the preceding restart. In other words, the learnt clause minimization in Glucose and TC_Glucose follows their learnt clause database reduction, after which only the first half of learnt clauses remains.

Function $liveClause(C)$ returns true if C has not yet been minimized and belongs to the second half of learnt clauses after sorting the clauses in decreasing order of their LBD values.

Function $sort(C)$ returns C without changing the order of its literals. This order is roughly the reverse order in which these literals are involved to deduce a literal in the conflicting level according to the First UIP scheme.

Algorithm 1 is executed before every restart. It calls Algorithm 2 if $liveRestart(nbNewLearnts, \sigma)$ returns true.

4.3 Minimizing Learnt Clauses in COMiniSatPS, MapleCOMSPS and MapleCOMSPS_LRB

COMiniSatPS is a SAT solver created by applying a series of small diff patches to MiniSAT 2.2.0. Its initial prototypes (SWDiA5BY and MiniSat_HACK_xxxED) won six medals in the 2014 SAT Competition and the 2014 Configurable SAT Solver Challenge. MapleCOMPS and MapleCOMSPS_LRB are based on COMiniSatPS, and were the winners of the main track and the application category in the 2016 SAT Competition, respectively.

The clause database reduction policy of COMiniSatPS, MapleCOMSPS and MapleCOMSPS_LRB is quite different from that of Glucose. In these solvers, the learnt clauses are divided into three subsets: (1) clauses whose LBD value is smaller than or equal to a threshold t_1 are stored in a subset called $CORE$; (2) clauses whose LBD value is greater than t_1 and smaller than or equal to another threshold t_2 are stored in a subset called $TIER2$; and (3) the remaining clauses are stored in a subset called $LOCAL$. If a clause in $TIER2$ is not involved in any conflict for a long time, it is moved to $LOCAL$.

Periodically, the clauses of $LOCAL$ are sorted in increasing order of their activity in recent conflicts, and the learnt clauses in the first half are removed (except for the clauses that are reasons of the current partial assignment).

The three solvers interleave Glucose-style restart phases with no restart or Luby restart phases.

Learnt clause minimization in COMiniSatPS, MapleCOMSPS and MapleCOMSPS_LRB is implemented by defining the three functions in Algorithm 1 and Algorithm 2 as follows (recall that Algorithm 1 is executed before each restart):

Function $liveRestart(nbNewLearnts, \sigma)$ returns true iff $nbNewLearnts$, the number of clauses learnt since the last learnt clause minimization, is greater than or equal to $\alpha + 2 \times \beta \times \sigma$. We empirically fixed α=1000, β=1000 for the three solvers. Note that function $liveRestart(nbNewLearnts, \sigma)$ does not follow the clause database reduction in any of the three solvers.

Function $liveClause(C)$ returns true iff C has not yet been minimized and belongs to $CORE$ or $TIER2$.

Function $sort(C)$ returns C without changing the order of its literals.

5 Experimental Investigation

We implemented the learnt clause minimization approach described in Section 4 in solvers Glucose 3.0, TC_Glucose, COMiniSatPS (COMSPS for short), MapleCOMSPS (Maple for short) and MapleCOMSPS_LRB (MapleLRB for short). The resulting solvers[1] are named Glucose+, TC_Glucose+, COMSPS+, Maple+ and MapleLRB+, respectively. Besides, we created the solvers MapleLRB/noSp and MapleLRB+/noSp by disabling in MapleLBR and MapleLRB+, respectively, the inprocessing technique of [Heule *et al.*, 2011], based on binary implication graphs, that implements function $Stamp$ (noSp means that function $Stamp$ is removed in MapleLRB/noSp and MapleLRB+/noSp).

The test instances include the application and hard combinatorial tracks of the 2014 and 2016 SAT competitions, and the community attachment formulas proposed in [Giráldez-Cru and Levy, 2015]. The experiments were performed on a computer with Intel Westmere Xeon E7-8837 processors at 2.66 GHz and 10 GB of memory under Linux. The cutoff time is 5000 seconds for each instance and each solver.

Table 1 compares the solvers Glucose, Glucose+, COMSPS, COMSPS+, Maple, Maple+, MapleLRB, MapleLRB+, MapleLRB/noSp and MapleLRB+/noSp on instances of the application category of the 2014 and 2016 SAT competitions.

[1]Available at http://home.mis.u-picardie.fr/~cli/

	SAT 2014 (300 instances)			SAT 2016 (300 instances)		
Solver	total	Sat	Unsat	total	Sat	Unsat
Glucose	213	100	113	146	61	85
Glucose+	224	105	119	152	63	89
COMSPS	221	105	116	146	65	81
COMSPS+	236	112	124	155	66	89
Maple	235	114	121	154	72	82
Maple+	250	117	133	161	70	91
MapleLRB	234	111	123	152	68	84
MapleLRB+	248	114	134	167	74	93
MapleLRB/noSp	231	111	120	150	67	83
MapleLRB+/noSp	247	113	134	164	73	91

Table 1: Comparison of Glucose, Glucose+, COMSPS, COMSPS+, Maple, Maple+, MapleLRB and MapleLRB+, MapleLRB/noSp and MapleLRB+/noSp on application instances of the 2014 and 2016 SAT competitions.

Our learnt clause minimization consistently improves the performance of all the solvers. In particular, MapleLRB was the best solver in this category in the 2016 Competition, but it solved only 4 instances more than the 4th and the 5th solvers of this category. Our approach allows MapleLRB+ to solve 15 instances more than MapleLRB.

Figure 1 shows a sample of scatter plots comparing MapleLRB+ and MapleLRB on 3 benchmark families of the application category of the 2016 SAT competition. A family consists of instances with similar names. A point (x, y) in the plots corresponds to an instance, where x (y) represents the solving time in seconds of MapleLRB+ (MapleLRB). A point (x, y) where $x = 5000$ ($y = 5000$) means that the instance was not solved by MapleLRB+ (MapleLRB) within a cutoff time of 5000s.

Table 1 also compares our inprocessing approach with the inprocessing approach of [Heule *et al.*, 2011], by means of four solvers: MapleLRB, MapleLRB+, MapleLRB/noSp and MapleLRB+/noSp. The only difference of these four solvers is specified in their names:

MapleLRB+: with our approach and with Stamp, the function implementing the inprocessing approach of [Heule *et al.*, 2011];

MapleLRB+/noSp: with our approach but without Stamp;

MapleLRB: without our approach but with Stamp;

MapleLRB/noSp: without our approach and without Stamp.

The inprocessing approach of [Heule *et al.*, 2011] implemented in MapleLRB allows MapleLRB to solve 3 (2) instances of 2014 (2016) more than MapleLRB/noSp, which has disabled that inprocessing. However, our approach is more effective, allowing MapleLRB+/noSp to solve 16 (14) instances of 2014 (2016) more than MapleLRB/noSp, and MapleLRB+ to solve 14 (15) instances of 2014 (2016) more than MapleLRB.

Figure 2 shows the cactus plots of the four solvers on the application instances of the 2014 SAT competition (top) and of the 2016 SAT competition (bottom). The two solvers using our approach perform clearly better than the two solvers using the approach of [Heule *et al.*, 2011]. Note that the inprocessing approach of [Heule et.al. 2011] is also used in Lingeling and subsumes (at least partly) many inprocessing techniques.

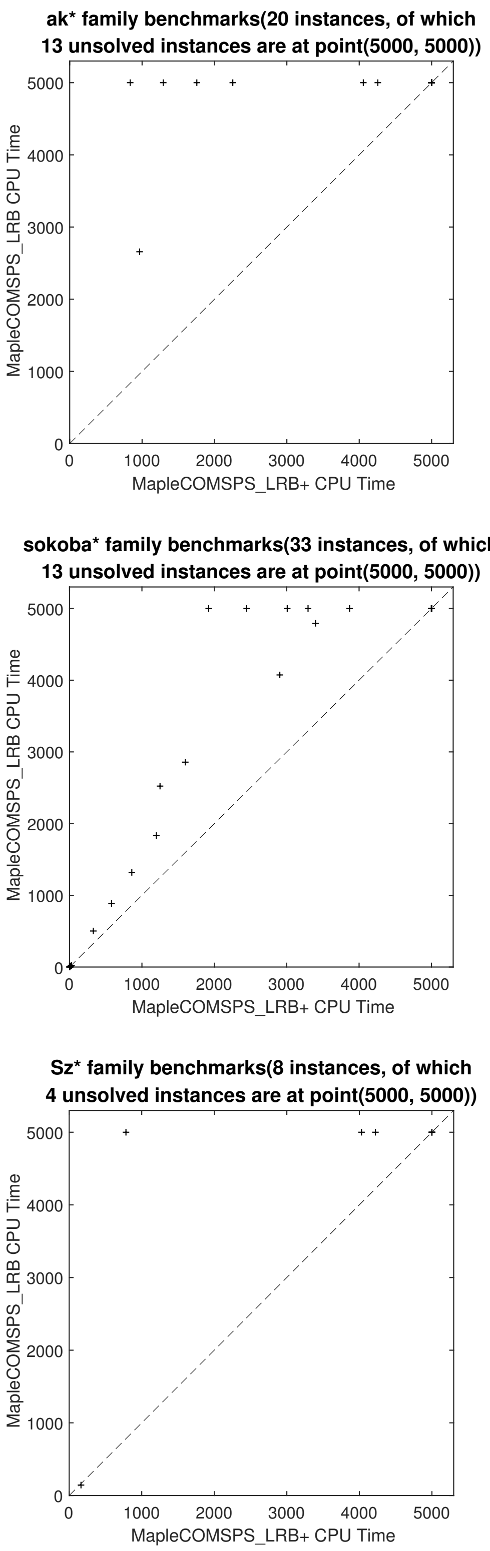
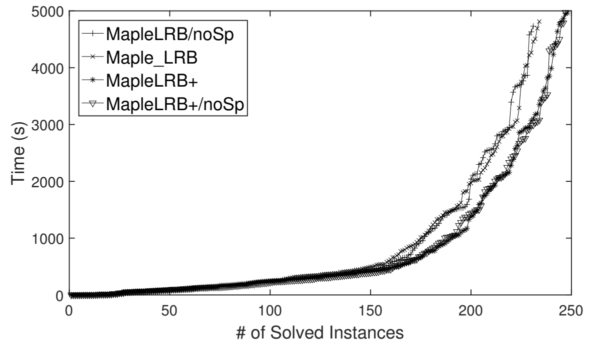


Figure 1: Scatter plots comparing MapleLRB+ runtime (x-axis) and MapleLRB runtime (y-axis) on 3 benchmark families of the application category of the 2016 SAT competition.



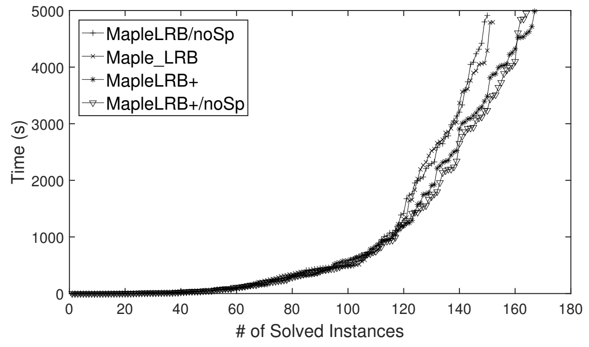


Figure 2: Cactus plots of MapleLRB+, MapleLRB, MapleLRB+/noSP and MapleLRB/noSp on application instances of the 2014 (top) and 2016 (bottom) SAT competitions.

	SAT 2014 (300 instances)			SAT 2016 (200 instances)		
Solver	total	Sat	Unsat	total	Sat	Unsat
Maple	217	97	120	47	8	39
Maple+	221	96	125	46	6	40
TC_Glucose	172	80	92	58	3	55
TC_Glucose+	184	79	105	57	3	54
MapleLRB	206	93	113	40	8	32
MapleLRB+	212	95	117	44	9	35
MapleLRB/noSp	206	92	114	40	8	32
MapleLRB+/noSp	214	97	117	44	9	35

Table 2: Comparison of Maple, Maple+, TC_Glucose, TC_Glucose+, MapleLRB, MapleLRB+, MapleLRB/noSp and MapleLRB+/noSp on hard combinatorial instances of the 2014 and 2016 SAT competitions.

Table 2 compares Maple, Maple+, TC_Glucose, TC_Glucose+, MapleLRB, MapleLRB+, MapleLRB/noSp and MapleLRB+/noSp on the hard combinatorial instances of the 2014 and 2016 SAT competitions. Observe that the instances of the 2016 SAT Competition are very special, because the best solver for these instances (TC_Glucose) performs significantly less well than the other solvers on the 2014 instances. The performance of Maple+, TC_Glucose+ and MapleLRB+ is similar to that of Maple, TC_Glucose and MapleLRB, respectively, on the 2016 instances. However, their performance is significantly better than that of Maple, TC_Glucose and MapleLRB on the 2014 instances.

Note that all the solvers in Table 1 and Table 2 implement the following inprocessing techniques: the clause minimization based on binary clause resolution of Glucose and the recursive learnt clause minimization of MiniSat. MapleLRB and MapleLRB+ include, in addition, the inprocessing approach of [Heule *et al.*, 2011]. The results show that our approach is compatible with all these techniques.

Table 3 compares different implementations of the learnt clause minimization in Glucose and MapleLRB on application instances of the 2014 SAT Competition, by varying the $liveRestart(nbNewLearnts, \sigma)$, $liveClause(C)$ and $sort(C)$ functions.

Glucose+/α-β: it is like Glucose+ but does not follow the learnt clause database reduction of Glucose any more. Instead, $liveRestart(nbNewLearnts, \sigma)$ returns true iff $nbNewLearnts \geq \alpha + 2 \times \beta \times \sigma$.

Glucose+H_lbd: it is like Glucose+ but $liveClause(C)$ is true iff C was never minimized before and is in the *first* half of learnt clauses when these clauses are sorted in decreasing order of their LBD value. In other words, the learnt clauses with higher LBD are minimized in Glucose+H_lbd.

Glucose+Δ: it is like Glucose+ but $liveClause(C)$ is true iff C was never minimized before and is in the last Δ fraction of learnt clauses when these clauses are sorted in decreasing order of their LBD value. Glucose+ is in fact Glucose+1/2.

MapleLRB+/Core: it is like MapleLRB+, but it minimizes every non-yet-minimized clause in $CORE$ upon every restart. It minimizes the clauses in $TIER2$ as in MapleLRB+.

Low2highLevel, High2lowLevel, Low2highActivity, High2lowActivity, Random and Reverse: all these solvers are like MapleLRB+, except that function $sort(C)$ is different. In solver Low2highLevel (Low2highActivity), literals in C are ordered from small level (activity) to high level (activity). In solver High2lowLevel (High2lowActivity), literals in C are ordered from high level (activity) to small level (activity). The level of a literal in C refers to the level of its last assertion. In solver Random, literals in C are randomly ordered. In solver Reverse, literals in C are reversed.

Table 3 shows the number of (Total, Sat, Unsat) instances solved by each solver within 5000s and exhibits some statistics about their runtime behavior. Column 5 (Impact) gives the clause size minimization measured as $(a - b)/a$, where a (b) is the total number of literals in the minimized clauses before (after) the minimization. Column 6 (Cost) gives the cost of the learnt clause minimization measured as the ratio of the total number of unit propagations performed by Algorithm 2 to the total number of other propagations performed in the search. Column 7 (LiveC) gives the ratio of the number of clauses minimized to the total number of learnt clauses. All data are averaged over the solved instances among the 300 application instances of the 2014 SAT Competition.

Several observations can be made from Table 3.

- All the described implementations of learnt clause minimization, except for the solvers Reverse and Glu-

Solver	Total	Sat	Unsat	Impact	Cost	LiveC
Glucose	213	100	113	/	/	/
Glucose+/10^3-10^3	216	96	120	27.04%	53.11%	11.79%
Glucose+/10^3-500	219	100	119	27.01%	60.73%	13.51%
Glucose+/10^3-300	215	100	115	26.69%	64.02%	13.98%
Glucose+H_lbd	206	98	108	33.20%	132%	24.05%
Glucose+	224	105	119	23.92%	29.90%	6.84%
Glucose+1/3	219	101	118	22.71%	19.43%	4.58%
Glucose+2/3	219	100	119	26.11%	42.45%	9.38%
MapleLRB	234	111	123	/	/	/
MapleLRB+/Core	238	108	130	26.73%	62.01%	22.61%
Low2highLevel	247	114	133	22.76%	57.11%	21.25%
High2lowLevel	237	105	132	24.41%	56.05%	22.59%
Low2highActivity	241	112	129	21.88%	54.45%	21.84%
High2lowActivity	237	107	130	24.28%	57.84%	21.47%
Random	241	109	132	24.07%	60.45%	21.99%
Reverse	233	106	127	18.37%	53.24%	21.24%
MapleLRB+	248	114	134	27.63%	58.11%	21.72%

Table 3: Comparison of different implementations of the learnt clause minimization in Glucose and MapleLRB on 300 application instances of the 2014 SAT Competition.

cose+H_lbd, improve their original solver, which provides evidence of the robustness of our approach.

- Both MapleLRB+ and MapleLRB+/Core minimize all the clauses in $CORE$ but at different times. The superior performance of MapleLRB+ over MapleLRB+/Core might be explained as follows: When MapleLRB+ minimizes a clause in $CORE$, there are usually more learnt clauses in the clause database, allowing unit propagation to deduce more easily the empty clause.
- Propagating the literals of C in the original order is the best option, whereas propagating the literals in the reverse order is the worst option. Recall that the original order of C is roughly the reverse order in which these literals are involved to deduce a literal in the conflicting level according to the First UIP scheme.
- Minimizing clauses with high LBD is very costly and useless, because Glucose+H_lbd is significantly worse than Glucose.
- The cost of the approach in Glucose+Δ is relatively small, because the solver just removed half of the learnt clauses when learnt clause minimization was activated, which is not the case for other solvers.
- Glucose+ and MapleLRB+ offer the best trade-off between cost and impact, explaining their superior performance.

It is known that industrial SAT instances exhibit community structure [Ansótegui *et al.*, 2012; Newsham *et al.*, 2014] and the LBD of a learnt clause is related to the community structure [Ansótegui *et al.*, 2015]. We analyze our approach using the community attachment formulas proposed in [Giráldez-Cru and Levy, 2015]. In these instances, variables are partitioned into communities and the community of a variable can be easily determined. We generated and solved 100 instances at each point with the same parameters as in Table 1 of [Giráldez-Cru and Levy, 2015] (modularity $Q = 0.9, 0.8$ or 0.7, clause size $k = 3$, number of communities $c = 40$, clause to variable ratio $r' = 4.06, 4.11$ or 4.13, and number of variables $n = 5000, 2000$, or 1200), using the generator in [Giráldez-Cru and Levy, 2015]. Recall that formulas with higher modularity exhibit clearer community structure.

Q	r'	n	comm_red_ratio	lbd_red_ratio	size_red_ratio
0.9	4.06	5000	0.70%	11.45%	15.15%
0.8	4.11	2000	9.21%	10.48%	16.37%
0.7	4.13	1200	10.53%	10.08%	15.66%
SAT Competition 2014: Application Instances				16.71%	27.63%

Table 4: Impact of our approach in MapleLRB+ on the size, the LBD and the number of communities of a learnt clause.

The impact of our approach on the size, the LBD and the number of communities of a learnt clause is measured as $ratio = (value\_before\_reduction - value\_after\_reduction)/value\_before\_reduction$, and is averaged over the 100 instances at each point. As shown in Table 4 with the data from the resolution of the community attachment formulas and the application instances of the 2014 SAT competition by MapleLRB+, the impact of our approach on the clause size (size_red_ratio) is significantly higher than the impact on the LBD (lbd_red_ratio) and on the number of communities (comm_red_ratio) of a clause. This means that reducing the LBD and the number of communities of a clause is harder than reducing its size. The same observation applies to the application instances of the 2014 SAT competition.

Moreover, the impact of our approach on the number of communities is very low for instances with high modularity. This phenomenon might be explained as follows: to minimize a learnt clause C, our approach has to deduce the empty clause by successively propagating the negation of the literals of C. In an instance with modularity 0.9, the number of communities in C is roughly the number of decisions needed to make C failed. So, Algorithm 2 probably has to propagate the negation of at least one literal in each community of C to deduce the empty clause. Consequently, the reduced clause C' probably contains the same number of communities as C.

6 Conclusions

We described a new inprocessing approach to minimize learnt clauses based on applying Boolean constraint propagation to a reduced number of learnt clauses when a restart is triggered under certain conditions, controlled using three functions whose exact definition depends on other techniques in the solver. The approach is simple and can be easily implemented in a CDCL SAT solver. We implemented it in five of the best performing state-of-the-art CDCL solvers, and conducted an empirical evaluation on instances of recent SAT competitions. The results provide evidence of the significance and the robustness of the contributions of this paper. According to Audemard and Simon (2012), we are in front of a new critical feature of SAT solvers, because the proposed approach was able to solve up to 16 additional instance of a same category of the SAT Competition. We believe that future SAT solvers will incorporate our new inprocessing technique.

References

[Ansótegui *et al.*, 2012] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of SAT formulas. In *SAT-2012*, pages 410–423, 2012.

[Ansótegui *et al.*, 2015] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using community structure to detect relevant learnt clauses. In *SAT-2015*, pages 238–254, 2015.

[Audemard and Simon, 2009] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI-2009*, pages 399–404, 2009.

[Audemard and Simon, 2012] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *CP-2012*, pages 118–126, 2012.

[Bacchus and Winter, 2003] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT-2003*, pages 341–355, 2003.

[Beame *et al.*, 2004] 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.

[Biere, 2010] Armin Biere. Lingeling, plingeling, picosat and precosat at SAT race 2010. *FMV Report Series TR 10/1, Johannes Kepler University, Linz, Austria*, 2010.

[Biere, 2011] Armin Biere. Preprocessing and inprocessing techniques in SAT. In *Haifa Verification Conference*, page 1, 2011.

[Eén and Biere, 2005] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT-2005*, pages 61–75, 2005.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT-2013*, pages 502–518, 2003.

[Giráldez-Cru and Levy, 2015] Jesús Giráldez-Cru and Jordi Levy. A modularity-based random sat instances generator. In *IJCAI-2015*, pages 1952–1958, 2015.

[Hamadi *et al.*, 2009] Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Learning for dynamic subsumption. In *ICTAI-2009*, pages 328–335, 2009.

[Han and Somenzi, 2007] HyoJung Han and Fabio Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *DAC-2007*, pages 582–587, 2007.

[Han and Somenzi, 2009] Hyojung Han and Fabio Somenzi. On-the-fly clause improvement. In *SAT-2009*, pages 209–222, 2009.

[Heule *et al.*, 2011] Marijn JH Heule, Matti Järvisalo, and Armin Biere. Efficient CNF simplification based on binary implication graphs. In *SAT-2011*, pages 201–215, 2011.

[Järvisalo *et al.*, 2012] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *IJCAR-2012*, pages 355–370, 2012.

[Jeroslow and Wang, 1990] Robert G Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1):167–187, 1990.

[Liang *et al.*, 2016a] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *AAAI-16*, pages 3434–3440, 2016.

[Liang *et al.*, 2016b] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT-2016*, pages 123–140, 2016.

[Liang *et al.*, 2016c] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, and Krzysztof Czarnecki. CHBR glucose. In *SAT Competition 2016: Solver and Benchmark Descriptions*, pages 52–53, 2016.

[Manthey *et al.*, 2016] Norbert Manthey, Aaron Stephan, and Elias Werner. Riss 6 solver and derivatives. In *SAT Competition 2016: Solver and Benchmark Descriptions*, pages 56–57, 2016.

[Marques-Silva and Sakallah, 1999] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Marques-Silva, 2000] Joao Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *CP-2000*, pages 537–542, 2000.

[Moon and Mary, 2016] Seongsoo Moon and Inaba Mary. CHBR glucose. In *SAT Competition 2016: Solver and Benchmark Descriptions*, page 27, 2016.

[Moskewicz *et al.*, 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC-2001*, 2001.

[Newsham *et al.*, 2014] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *SAT-2014*, pages 252–268, 2014.

[Oh, 2016] Chanseok Oh. *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL*. PhD thesis, New York University, NY, USA, 2016.

[Piette *et al.*, 2008] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying propositional clausal formulae. In *ECAI-2008*, pages 525–529, 2008.

[Soos, 2010] Mate Soos. Cryptominisat 2.5. 0. *SAT Race competitive event booklet*, 2010.

[Sörensson and Biere, 2009] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *SAT-2009*, pages 237–243, 2009.

[Wieringa and Heljanko, 2013] Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In *SAT-2013*, pages 116–132, 2013.

[Wotzlaw *et al.*, 2013] Andreas Wotzlaw, Alexander van der Grinten, and Ewald Speckenmeyer. Effectiveness of pre- and inprocessing for CDCL-based SAT solving. *arXiv preprint arXiv:1310.4756*, 2013.

[Zhang *et al.*, 2001] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD-2001*, pages 279–285, 2001.