On the Glucose SAT solver

Gilles Audemard, Laurent Simon

▶ **To cite this version:**

Gilles Audemard, Laurent Simon. On the Glucose SAT solver. International Journal on Artificial Intelligence Tools (IJAIT), 2018, 27 (1), pp.1-25. hal-03299473

HAL Id: hal-03299473
https://univ-artois.hal.science/hal-03299473

Submitted on 28 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Glucose SAT Solver

Gilles Audemard
Univ. Lille-Nord de France. CRIL/CNRS UMR 8188, Lens, France
audemard@cril.fr

Laurent Simon
LaBRI, University Bordeaux, France
lsimon@labri.fr

The set of novelties introduced with the SAT solver Glucose is now considered as a standard for practical SAT solving. In this paper, we review the different strategies and technologies added in Glucose over the years. We detail each technique and discuss its impact on the final performances reached by Glucose. We also come back on one of the main developments of the solver over the very last years: its efficient parallelization. We extensively tested different versions of Glucose and Syrup (its parallel version) on all the benchmarks since 2011. By including, as a reference, the SAT solver Lingeling (and its parallel version Plingeling), we show that Glucose and Syrup are significantly faster than other solvers, even if they can solve fewer instances.

1. Introduction

The quest for practical solving of SAT formulas has a number of milestones, beginning in the early sixties with the Davis-Putnam-Logeman-Loveland procedure, a systematic backtrack-search algorithm over partial solutions (DPLL). In the early 2000's, with the introduction of CDCL (Conflict-Driven Clause Learning) algorithms, a major breakthrough was reached even if the CDCL paradigm, by its own, would probably not have been sufficient to demonstrate such an impressive progress. A number of additional ingredients are mandatory to unleash the full power of CDCL approaches. Even if the theoretical reasons behind the practical success of SAT solvers are largely unclear, we know in practice what ingredients are needed, and how they must be incorporated in any SAT solver targeting application instances.

All the following data structures and algorithms will be formally defined later. Nevertheless, let us introduce here a number of intuitions behind each CDCL component. By definition, the first cornerstone of any CDCL algorithm is conflict analysis (described in the next section) [53]. However, it is essential to build the solver on top of the 2-Watched Literal scheme, that allows a very efficient BCP (Boolean Constraint Propagation) thanks to a cheap and lazy way of handling formula simplifications under partial assignment and backtracking [42]. This data structure has a main drawback: the search for a solution is partially blind, and only unit and

empty clauses are detected. This is certainly one of the reasons VSIDS was introduced [54,21]. This heuristics is only updated during conflict analysis and thus perfectly fits the above constraints (it is state-independant and thus does not need any effort after assignments/unassignments). Conflict Analysis, 2 Watched-Literal and VSIDS are still the most important ingredients of any CDCL implementation. Later, it was stressed out how important the restart policy was. If restarts were firstly introduced to fight the heavy tailed phenomenon observed on DPLL implementations [26], the very fast restart policies (following for instance the Luby series [32]), used with the phase caching scheme [47] was probably one of the most important improvements since the first release of MINISAT[21], the reference implementation of the CDCL algorithm.

With GLUCOSE, we proposed to renew the vision of CDCL solvers. Instead of seeing them as an improvement of a DPLL search, we saw them as clauses producers. For this, we introduced a new quality measure, called LBD (Literal Block Distance) that has been proven to be very informative, in practice, to predict clauses usefulness. In most of the cases (for so-called "Industrial" problems) it seems crucial to aggressively remove many learnt clauses, even during the first conflicts (up to 95% of them). Two factors seem to play a crucial role in managing the learnt clause database: keeping as few clauses as possible ensures a very fast BCP, and removing bad clauses allows to guide the solver search around a shorter UNSAT proof. Moreover, the restart policy introduced in GLUCOSE, and based on the quality of recently learnt clauses, has also been proven to be very efficient in practice.

In this paper, we review the set of ingredients that were typically introduced in the series of GLUCOSE releases and try to propose some explanations of its efficiency. We particularly focus on identifying the impacts of these ingredients on our solver performances. We also develop the ideas behind SYRUP, the parallel version of GLUCOSE, that uses a special data structure with its associated policy for sharing clauses between cores.

We also show, in this paper, that besides the fact that GLUCOSE does not integrate inprocessing and sophisticated preprocessing techniques, it is still one of the best SAT solvers, even on recent problems, as soon as CPU time is taken into account. When using Allen van Gelder "careful ranking" [23] used in the first cycle of competitions (from 2002 to 2007, see `http://www.satcompetition.org/`), GLUCOSE and SYRUP are clearly the solvers of choice: they are faster, even if they can solve fewer problems.

The paper is organized as follows. After a necessary recall of the CDCL algorithm, we introduce, in Section 3, the notion of Literal Block Distance, on top of which all the ingredients of Glucose are built. Its usefulness in detecting *good* clauses is also discussed. In Section 4, we describe the dynamic restart strategy introduced in GLUCOSE, which is probably the second important novelty of our solver. In section 5 we introduce the notion of lazy exportation/importation of clauses, which is one of the signatures of our parallel version of GLUCOSE, called SYRUP. In Section 6 describes the special data structure used for clauses when incremental SAT solving

is used with GLUCOSE. At last, Section 7, we extensively test 4 versions of GLUCOSE, SYRUP, and LINGELING/PLINGELING on all the problems of the SAT Competition since 2011. The last section of this paper concludes it.

2. Preliminaries

Before focusing on the novelties introduced in GLUCOSE, let us recall here some basic notations. A propositional formula in Conjunctive Normal Form (CNF) is a conjunction of clauses (interpreted as a set), where a clause is a disjunction of literals (interpreted as a set). A literal is a propositional variable x or its negation $\neg x$. A unit clause is a clause with only one literal. The empty clause, denoted $\bot$, is false, while the empty formula, denoted $\top$ is true. An interpretation $\mathcal{I}$ of a propositional formula Σ associates to some variables x of the formula a value $\mathcal{I}(x)$ equal to true or false. An interpretation is complete if all variables of the formula are interpreted; it is partial otherwise. An interpretation can also be represented as a set of literals (literals that are assigned to true). A model of a formula Σ is an interpretation that satisfies the formula, *i.e.*, that satisfies all clauses of the formula. The SAT problem is the decision problem that consists in verifying if a formula Σ in CNF has a model.

The resolution rule is essential in SAT. Two clauses c_1 and c_2 can be resolved if and only if it exists a unique literal ℓ, such that $\ell \in c_1$ and $\neg\ell \in c_2$. Clause $(c_1 - \{\ell\}) \cup (c_2 - \{\neg\ell\})$ is called the resolvent in ℓ of c_1 and c_2. It is denoted $c_1 \otimes_\ell c_2$. Let us just notice that adding resolvents of clauses that are part of a given formula does not change its satisfiability.

2.1. *Conflict Driven Clause Learning solvers*

When they were introduced fifteen years ago [42,21], CDCL SAT solvers (Conflict Driven Clause Learning) were presented as an extension of the DPLL algorithm [20] with additional features such as clause learning [53,54], based on top of an efficient data structure (2 Watched Literals) for Unit Propagation detections, giving an efficient Boolean Constraint Propagation engine (BCP). Now, it is well admitted that they have to be seen as a mix of backtrack algorithms and resolution engines. Furthermore, is has been proved that CDCL SAT solvers are more powerful than DPLL ones, i.e. there exist formulas on which the proof (the proof can be seen as a special trace of solver's run) can be polynomial (w.r.t. to the number of variables) for CDCL solvers whereas, it is necessary exponential for DPLL ones. The opposite is not true [11,49].

We recall Algorithm 1 an overview of a typical CDCL solver, focusing on the necessary notions needed to understand GLUCOSE in the later. For more details, the reader can refer to [40]. A typical branch of a CDCL solver is a sequence of decision/propagations repeated until it reaches a conflict, *i.e.* the empty clause. Each decision literal (line 20-22) is assigned at a *decision level* (d), literals that are implied (by unit propagation, line 6) by this decision have the same level (the notation $\ell@d$ denotes that literal ℓ is assigned at level d). Then, an interpretation can be written

$\mathcal{I} = \{\langle \ell_{k_{1_1}}@1, \ldots \ell_{k_{1_i}}@1\rangle, \langle \ell_{k_{2_1}}@2, \ldots\ \ell_{k_{2_j}}@2\rangle \ldots \langle \ell_{k_{d_1}}@d, \ldots \ell_{k_{d_t}}@d\rangle\}$. Symbols $\langle$ and $\rangle$ partition $\mathcal{I}$ with respect to the different decision levels and literals $\ell_{k_{1_1}}$, $\ell_{k_{2_1}}$, ... are the decision literals of each level. If all variables are assigned, then a model is found (line 17). Each time a conflict is reached by unit propagation (clause c is the empty clause, line 6-7), a *nogood* is computed, line 9 (usually using the First Unique Implication Point principle [54]), together with a correct backtrack level (bl). The nogood is in fact a simple clause derived using a sequence of resolutions (described precisely later). It provides an explanation for the conflict and has special properties such as empowerment [48]. Indeed, it provides a new unit literal (according to the partial assignment) that can be propagated at decision level bl. So, a backtrack (sometimes called backjump) is performed (last sequences of literals are deleted from the interpretation $\mathcal{I}$) (line 14). Note that, if the nogood is the empty clause, then the unsatisfiability of the formula is proven (line 11). Once in a while, restarts are fired (line 13), *i.e.*, the interpretation is cleared (except from literals propagated at decision level 0). We will discuss restart strategies in Section 4. Finally, and it is one of the main novelty introduced in GLUCOSE, periodically, some learnt clauses, *useless* ones, are removed from the database (line 19). We will come back on this step Section 3.

Let us illustrate one step of this algorithm with a small example. Let Σ be a formula containing clauses:

$c_1 = x_1 \vee x_4$ $\quad c_2 = x_1 \vee \neg x_3 \vee \neg x_8$ $\quad c_3 = x_1 \vee x_8 \vee x_{12}$
$c_4 = x_2 \vee x_{11}$ $\quad c_5 = \neg x_3 \vee \neg x_7 \vee x_{13}$ $\quad c_6 = \neg x_3 \vee \neg x_7 \vee \neg x_{13} \vee x_9$
$c_7 = x_8 \vee \neg x_7 \vee \neg x_9$ $\quad c_8 = \neg x_1 \vee \neg x_{12} \vee x_2 \vee x_7$

Suppose function `pickBranchLit` chooses literal $\neg x_1$ as the first decision literal. Then, $\mathcal{I}_p = \{\langle \neg x_1@1, x_4@1[c_1]\rangle\}$. Indeed, since x_1 is false, c_1 propagates x_4 to true (this is the goal of the function `unitPropagation`, x_4 must be true in order to satisfy c_1). Furthermore, we store the reason of the propagation inside the partial interpretation (here the clause c_1, inside brackets). Note that, of course, the decision literal has no reason (reasons are unisatisfied clauses). At decision level 2, the literal x_3 is chosen. At decision level 3 it is the literal $\neg x_2$. Then, the partial interpretation is $\{\langle \neg x_1@1, x_4@1[c_1]\rangle, \langle x_3@2, \neg x_8@2[c_2], x_{12}@2[c_3]\rangle, \langle \neg x_2@3, x_{11}@3[c_4]\rangle\}$. Finally, if the next decision literal is x_7, then, the solver reaches a conflict. Indeed, the following propagations are added to the partial interpretation: $\langle x_7@4, x_{13}@4[c_5], x_9@4[c6], \neg x_9@4[c_7]$. One tries to assign x_9 to values true and false. The conflict clause (clause c in algorithm 1) is c_7.

The function `conflictAnalysis` produces a sequence of resolutions starting from the variable in conflict (x_9) and using clauses that are reason of previous propagations. Then, the following resolutions are done (note that only literals that are part of the intermediary clauses are used in this process):

- $d^* = c_7 \otimes_{x_9} c_6 = \neg x_3 \vee x_8 \vee \neg x_7 \vee \neg x_{13}$
- $\gamma = d^* \otimes_{x_{13}} c_5 = \neg x_3 \vee x_8 \vee \neg x_7$

Algorithm 1: CDCL solver

Input: Σ une formule CNF
Output: SAT ou UNSAT

```
1  I_p = ∅ ;                                              /* interpretation */
2  d = 0 ;                                                /* Ddeision Level */
3  nb_c = 0 ;                                             /* conflict number */
4  Γ = ∅ ;                                                /* set of learnt clauses */
5  while (true) do
6      c = unitPropagation(Σ ∪ Γ,I_p);
7      if (c ≠null) then
8          nb_c = nb_c + 1 ;                              /* new conflict */
9          γ =conflictAnalysis(Σ ∪ Γ,I_p,c);
10         bl = ComputeBacktrackLevel(γ,I_p);
11         if (γ = ⊥) then return UNSAT;
12         Γ = Γ ∪ {γ};
13         if (restart(nb_c)) then bl = 0;
14         cancelUntil(Σ ∪ Γ,I_p,bl);                     /* update I_p */
15         d = bl;
16     else
17         if (all variables are assigned) then
18             return SAT;
19         if (timeToReduce(nb_c)) then reduceDB();
20         ℓ = pickDecisionLiteral(Σ ∪ Γ);
21         d = d + 1;
22         I_p = I_p ∪ {⟨ℓ@d⟩};
23 end
```

The clause γ is the first clause (during conflict analysis) containing only one literal from the last decision literal (here literal $\neg x_7$). This clause is added to the learnt clause database. This is the First UIP (Unique Implication Point) scheme [54], that can be formulated using a implication graph. It has to be noticed that all intermediary clauses derived (here clause d^*) before reaching the FUIP clause are not stored in the database. As you can see, this clause is falsified since level 2. It is clear that, if we had this clause before performing the sequence of decision/propagations, we should have propagated $\neg x_7$ at decision level 2 (using this clause as reason). Thus, the backtrack level is set to 2, the partial interpretation $\mathcal{I}_p$ is reduced accordingly, and the propagation function can be called again. At the end, we have $\mathcal{I}_p = \{\langle \neg x_1@1, x_4@1[c_1]\rangle, \langle x_3@2, \neg x_8@2[c_2], x_{12}@2[c_3], \neg x_7@2[\gamma]\ldots\rangle\}$.

It is very important to note that this learning process is the key point of CDCL solvers. At each conflict (typically more than 5,000 times per seconds) the algorithm adds only one new clause, but derived by many resolutions steps (sometimes more

than hundreds).

In a typical CDCL implementation, the main effort is spent in the function `unitPropagation`. It is thus crucial to be able to detect unit clauses as efficiently as possible. To this end, a data structure called 2-watched scheme [42] is commonly used. For a given clause, two non falsified literals are the witnesses that this clause is at least binary (or satisfied). Each time one of the witnesses is falsified, the propagation engine tries to find a new witness to replace it. If none is found, then the clause is unit or empty (depending on the value of the other witness). This data structure is very efficient for mainly two reasons. First, it is just needed to update the clauses in which a newly falsified literal is one of the two witnesses and, second, touched clauses during propagations do not need to be touched again during backtrack. Thus, this last operation is almost free (it is just needed to update variable values).

3. Literal Block distance

3.1. *Motivations*

Given the high learning rate of CDCL solvers (they typically learn more than 5,000 clauses per second, and more than 15,000 on some particular problems), the question of managing the learnt clauses database was identified since their early ages (first implementations were typically consuming too much memory). However, the first answer was just about trying to contain the combinatorial explosion by simply periodically cleaning the learnt clause database. The policy for this was essentially to try to make sufficient room for new learnt clauses by removing unused previously-learnt clause (see line 19 of Algorithm 1). Even if it was not really identified as the most crucial component of CDCL solvers, it appeared more and more important over the last years. Indeed, keeping too many learnt clauses slows down the unit propagation process, while deleting too many of them breaks the overall learning benefit.

As mentioned above a CDCL-based SAT solver can be formulated as a resolution proof system [49,11]. Consequently, the practical incarnation of modern SAT solvers can be seen as a clauses producer procedure with a deletion strategy. However, for unsatisfiable instances, many around half of the learnt clauses are useless. By useless, we mean that they do not occur in the final proof of unsatisfiability (a clause can be unused for the final proof, but essential for the heuristics, by forcing, by unit propagation a literal value). This is highlighted Figure 3.1, where 55% of learnt clauses are useless, when clause database reduction is never done (keeping all clauses allows to prevent an interesting clause from being (wrongly) judged useless because of its early removal). For satisfiable instances, of course, the notion of clause usefulness has to be defined, but clearly enough, it cannot simply be related to the clause occurrence in the current proof. Indeed, for SAT instances, the role of learnt clauses is to conduct the solver on some particular search space, and also, probably, to explicit some dependencies between variable assignments. Consequently, defining

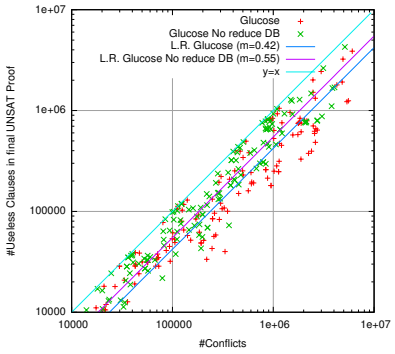


Fig. 1. Useless clauses in the final UNSAT proof w.r.t the total number of generated clauses. Glucose no reduce DB is a hacked version of GLUCOSE that do not perform any learnt clause removal. Experiments are done on a set of 250 UNSAT problems from competitions 2011 and 2013. Only successful run are collected. LR stands for linear regression

what is a relevant clause before completing the proof itself is not simple (Occurring on the proof for UNSAT? Forcing a value for SAT?). If we want to answer this question more formally, it is easy to show that the usefulness of a learnt clause is computationally harder than the initial SAT question: it is related to finding a proof of minimal size.

Trying to maintain only a relevant set of clauses is naturally older than CDCL algorithms [24]. However, in the early years of CDCL, the importance of clause deletion strategies was not clearly identified as one of their essential components. The aim was just to help the solver managing too many clauses. Nevertheless, a few proposal were made. In [37], authors proposed the idea of bounding the size of learnt clauses, keeping only clauses smaller than a certain threshold. The authors of Berkmin [25] considered that recent clauses were better than older ones. They used a FIFO to remove old clauses while keeping in memory short clauses (smaller than 8). In Chaff [42], a clause was marked to be deleted when a certain number of literals became unassigned (between 100 and 200). In Minisat [21], learnt clauses are deleted based on an activity heuristic. Each time a clause is used in conflict analysis, its activity is increased. Then, periodically, half of the clauses are deleted (the ones with small activities). This technique is very interesting, because it also deletes without additional cost subsumed learnt clauses (they will never be used in the future, then their activity will always decrease).

Finally, it can be noticed that, for almost all these techniques, the number of deleted clauses is often as small as possible. Minisat is a little bit more complicated. The number of clauses to keep is increasing following a geometrical series (with common ratio $r = 1.1$), first term is $m/3$ (m is the initial number of clauses). The series is increased each time the number of conflicts reaches another geometric series of first term 100 and common ratio $r' = 1.5$. For instance, on a formula with 10,000 clauses, the maximum size of learnt clauses is firstly set to 3,333. But,

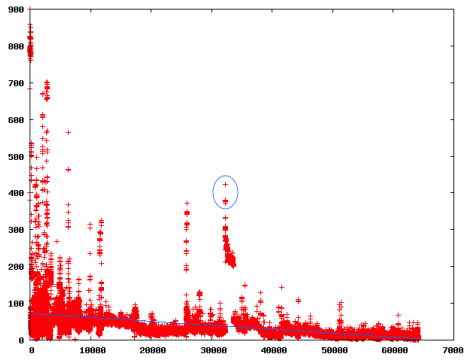


Fig. 2. Decision levels for reaching a conflict for a representative instance. For each conflict (x-axis) we report the number of decision levels (y-axis). The line represents the simple least-square linear regression that fits the set of points. The circle will be explain in Section 4.

after 100 conflicts, it is increased to 3,666 ($r = 1.1$). Then after another 150 conflicts (here $r' = 1.5$), it is increased to $4,033$. So, the limit would be 3,666 after 100 conflicts, 4,033 after 250 conflicts, 4,436 after 475 conflicts, then 4,819/812, 5,367/1,318, 5,904/2,078, 6,495/3,217, 7,144/4,925, 7,959/7,488... So, the first reduction of the database will be triggered when it will reach 7,959 clauses. Because the second series is increasing much more quickly than the first, the first series will only slowly increase (the limit of 7,959 clauses will remain between conflicts 7,488 and 11,333). At the end, MINISAT will keep a number of clauses that will be roughly the square root of the number of conflicts. However, it quickly reaches a number of learnt clauses that are near the initial size of the formula, which can be a lot on typical industrial instances.

This being said, the crucial questions remain: *what are the good clauses to learn? How to identify them during the search?* These questions were at the origin of our solver GLUCOSE. Indeed, if one is able to determine good clauses, then, one can often remove useless ones. We introduce in the next section our measure to identify good learnt clauses.

3.2. *The Literal Block Distance (LBD) measure*

GLUCOSE was proposed after an extensive series of experimental studies about CDCL solvers. We noticed that, for most of the instances, the number of decisions made before reaching a conflict globally decreases during the search [5]. This can be seen as a natural phenomenon for unsatisfiable instances (thanks to learning), but it was somehow surprising to observe this phenomenon also on satisfiable ones. Figure 2 shows a typical observation for this. For conflict number x, we report the decision level (y) where the conflict occurs. The line ($y = m \times x + n$) represents the simple least-square linear regression that fits this set of points. When m is negative, then the global behavior consists in a decreasing of decision levels along the

search. In [5], we measured that this is the case for 83% of instances coming from a large panel (around 300 instances). Then, the lower the value of m would be, the faster we could expect to find the solution. Note that this was already pointed out, but from a general perspective only, in earlier papers on CDCL solvers: " *A good learning scheme should reduce the number of decisions needed to solve certain problems as much as possible.*" [54].

As we saw in Section 2.1, each decision can potentially create a lot of propagated literals (what we call *blocks*), especially in the deepest parts of the search tree. Those variables will probably be propagated together again and again, because they are intuitively semantically related. Then, if we want to reduce the number of decisions, we need to add dependencies between independent blocks, and, then, to add the strongest possible constraints between them. This observation is at the origin of our measure LBD (Literal Block Distance), formalized in the following.

Definition 3.1. Given a clause c, and a partition of its literals into n subsets according to the current assignment, such that literals are partitioned w.r.t their decision level. The LBD of c is exactly n.

From a practical point of view, we compute and store the LBD score of each learnt clause when it is produced. This measure is static, even if it is possible (we will see in the later, see Section 3.3) to update it during search. Intuitively, it is easy to understand the importance of learnt clauses of LBD 2: they only contain one variable of the last decision level (they are FUIP), and, later, this variable will be "glued" with the block of literals propagated above, no matter the size of the clause. We suspected all those clauses to be very important during search, and we gave them a special name: "Glue Clauses" (which gave the name of GLUCOSE, which is much more simpler to pronounce).

As an example, take a look at the learnt clause γ in the example of Section 2.1. Its size is equal to 3, whereas its LBD value is 2. As soon as x_1 is false, the propagation of either x_3 or $\neg x_8$ will propagate $\neg x_7$. On the contrary, suppose now that c_8 is also a learnt clause. In order to be used as a reason, one previously needs to assign many literals as decision ones. This clause has less chances to be useful during the search (currently, its LBD value is equal to 4).

We would like here to point out the extensive experimental study about different scoring of learnt clauses (including random scoring) conducted in [34]. An interesting remark that can be found in its study, although intuitive at this stage, is that, by removing learnt clauses, the CDCL solver is improving its diversification.

3.3. *Aggressive deletion of learnt clauses*

One of the novelties introduced in GLUCOSE is the notion of aggressive cleaning strategy, which aims to drastically reduce the learnt clauses database. This is possible especially if the LBD measure introduced above is accurate enough. In the first version (2009) of GLUCOSE we removed half of the learnt clauses every $20,000 + 500 \times x$

(x is the number of times this action was previously performed). We decided to keep forever all glue clauses. In the second version (2011), we introduced an even more aggressive policy by removing half of learnt clauses every $2,000 + 300 \times x$. Thus, just after the first $2,000$ conflicts, we remove around $1,000$ of learnt clauses. Note that the call of `reduceDB` (Algorithm 1, line 19) is done during the search (not at a restart), like in MINISAT. Thus, some clauses may be the reason of some propagations, and can not be removed, whatever their score. We added a last feature to GLUCOSE in order to have more dynamicity to the LBD scoring. It is indeed essential to identify good clauses. Keeping a bad clause does not have a very big impact in general, but removing a good clause could arm the solver performances. Thus, we add the possibility for a clause to decrease its score along the search (good clauses have a small LBD), in order to be sure to keep all good learnt clauses. More precisely, an alternative LBD value can be computed during unit propagation (we need all the literals to be assigned), for each clause that are reasons. However, in the last version of GLUCOSE, we only update the LBD score of clauses occurring in conflict analysis if the new score is better than the older one. Furthermore, we block this clause, i.e., we protect it for one cleaning round: it will not be deleted during the next cleaning process, whatever its score.

3.4. *Explaining usefulness of LBD*

Now that we have defined the LBD and its usage in GLUCOSE learnt clause database cleaning strategies, it is natural to try to explain why it shows so good results in practice. In this section, we provide 3 possible reasons for this.

First of all, we ran some experiments with a classical solver, that does not perform database clause reduction. For each learnt clause, we measured the number of times it was useful in unit-propagation and conflict analysis. Figure 3 shows the cumulative distribution function of all values gathered over 100 benchmarks coming from SAT competitions. For instance, 40% of the unit propagations on learnt clauses are done on glue clauses, and only 20% are done on clauses of size 2. Half of the learnt clauses used in the resolution mechanism during all conflict analysis have LBD < 6, whereas we need to consider clauses of size smaller than 13 for the same result. Such results show that LBD allows a better estimation of usefulness of a clause.

From a theoretical point of view, it is interesting to recall, as mentioned above, that LBD of FUIP learnt clause is optimal over all other possible UIP learning schemas [35]. This theoretical result cast a potentially good explanation for the efficiency of First UIP over all other UIP mechanisms: FUIP efficiency would then be partly explained by its ability to produce clauses of small LBD (it was also showed that FUIP is optimal in the backjump size).

Property 3.1. Given a conflict graph, any First UIP asserting clause has the smallest LBD value over all other UIPs.

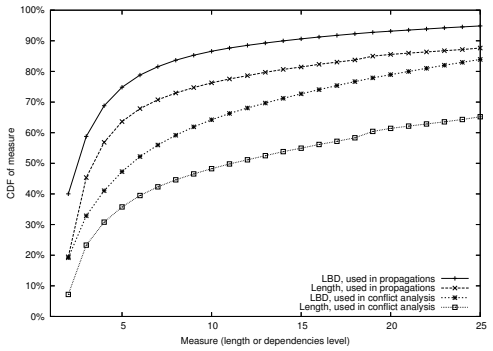


Fig. 3. Cumulative Distribution Functions of LBD scores and size of clauses used in propagations and conflict analysis.

A lot of efforts have been made in the last years to understand why CDCL solvers are so efficient on real-world instances [1,2]. In this series of studies, the notion of communities, defined at the CNF level, was used: a community is a subset of vertices of the graph, and analyzing the graph structures relies on finding a good partition of the graph into communities, such that there is a dense connection inside a group (community) and few connections between distinct communities. The notion of community is useful in graph theory, network or in social networks. In [2], authors showed that industrial SAT instances (coming form BMC, bio-info...) have a very strong community structure. They also made some experiments showing, that, in many cases, learning does not destroy much such community structure. This work should, partially, explain the efficiency of CDCL solvers on such huge instances, by suggesting that they can exploit this structure efficiently. In [46], we showed that there is a strong correlation between the LBD value of a learnt clause and the number of distinct communities it contains. We also showed that the correlation is clearly weaker if we only compare the number of communities and the size of the clause. Learnt clauses of small LBD add stronger constraints between fewer communities. We believe that this allows the solver to focus its search on a smaller part of the search tree, avoiding scattering, when the solver jumps between a number of communities creating learnt clauses that link these communities together, thus making the structure of the SAT instance worse and consequently harder to solve. We think this study gives a new point of view of LBD and provides a new explanation of its efficiency.

These reasons can shade a new light on LBD's efficiency. We believe that LBD is a good indicator of learnt clause usefulness.

4. Restarts

4.1. *Previous works*

Even if restarts strategies were introduced as a way to escape the Heavy-Tailed phenomenon [26] observed in backtrack solvers CPU time (typically on look ahead DPLL solvers), the reasons underlying the importance of restarts CDCL solvers are more complex. In CDCL solvers, restarting is not exactly restarting. The solver maintains all its heuristic values between them and restarts must be seen as dynamic rearrangements between dependencies of variables. This is especially true with the use of phase saving heuristic [47]. Additionally, the learning mechanism forces some hidden restarts, triggered in all solvers: when a unit clause is learnt (which can be very frequent in many huge problems, especially in the first conflicts), the solver usually cancels all its decisions without trying to recover them in the same order, in order to immediately consider this new fact. Thus, on problems with a lot of learnt unit clauses, the solver will restart more often, but without following the restarts policy.

Restarts schemes were first introduced without aggressive laws, often with increasing search windows. For example, restarts are done every x conflicts for some solvers (16,000 for SIEGE [51], 700 for CHAFF[42], 550 for BERKMIN[25]). MINISAT[21] uses a geometric series (starting from 100, with a factor equal to 1.5). Later, the Luby strategy [31] was proved to be very efficient. This series follows a slow increasing law 1 1 2 1 1 2 4 1 1 2 4 8 (multiplied by a factor, typically from 16 to 256) that has a very interesting property: it is exponentially increasing, but exponentially slowly, thus limiting the risk of searching for a long time in the wrong search space. An experimental evaluation of static and geometric restart schemes can be found in [17].

It was later proposed, in [14], to nest two series, a geometric one and a Luby one. The idea was to ensure that restarts were guaranteed to increase (allowing the search to, theoretically, terminate) and that fast restarts occurred more often than the geometric series. In [13], it was proposed to postpone scheduled restarts by observing the "agility" of the solver. This measure is based on the polarity of the phase saving mechanism. If most of the variables are forced against their saved polarity, then the restart is postponed: the solver might find a refutation soon. If polarities are stalling, the scheduled restart is triggered. To our knowledge, it is the first dynamic restart strategy. That is, the restart scheme does not follow a deterministic law, but reacts on the search state. Finally, a width-based policy was proposed in [50], that is, each time the learnt clause has a size greater or equal to a threshold, a penalty is set. After a given number of penalties, a restart is triggered. The threshold and the number of penalties can be static or dynamic.

4.2. *Glucose restarts*

The idea behind our restart strategy is the following: since we want to produce as many good clauses as possible (w.r.t. LBD), we simply trigger a restart when the

last produced clauses have a too high LBDs (in which case the search is considered to be on a bad path). To this end, we compare the current average (defined on a slicing window of the last conflicts) of LBDs with the global average of all LBDs of learnt clauses so far. If the current average (called avg^{lbd}_{cur}) is substantially greater than the global one (called avg^{lbd}_{glo}), a restart is performed. Formally, a restart is performed if $avg^{lbd}_{cur} \times K > avg^{lbd}_{glo}$.

In order to compute avg^{lbd}_{cur} we use a bounded queue of size X. Of course, whenever the bounded queue is not full, we can not compute the moving average, then at least X conflicts are performed before making a restart. The "magic" constant K, called the *margin ratio*, provides different behaviors. The larger K is, the fewer restarts are performed. In the first version of GLUCOSE (2009), we used $X = 100$ and $K = 0.7$. These values were experimentally fixed to give good results on both SAT and UNSAT problems. This strategy was not changed in GLUCOSE 2.0. However, following the good results obtained by a GLUCOSE-based solver [43], we changed these values ($X = 50$ and $K = 0.8$) since the version 2.1 of GLUCOSE (2012).

This restart scheme is particularly efficient on unsatisfiable instances. The problem we have is the following. GLUCOSE is firing aggressive clause deletion (see Section 3) and fast restarts. On some instances, restarts are really triggered every 50 conflicts. So, if the solver is trying to reach a global assignment, it has only a few tries before reaching it. Additionally, the aggressive clauses deletion strategy may have deleted few clauses that are needed to reach the global assignment directly. Moreover, some satisfiable instances require larger restarts intervals [6,17]. Thus, in the version 2.1 of GLUCOSE [6], we added the possibility of postponing restarts. A restart is postponed by simply emptying the bounded queue. A new restart will only be possible when the bounded queue will be full again. Postponing occurs when the number of total assignment grows suddenly. For example, take a look at Figure 2, and focus on the small circle. The number of decisions before reaching a conflict suddenly increases. It seems that the solver goes trough a difficult part of the search space, and is closed to a full assignment. We can safely suppose that the solver needs to stay on this search space in order to find a solution. To this end, we compute the global average of assigned literals when a conflict occurs (avg^{ass}_{glo}) and an average of the recent ones (avg^{ass}_{cur}). A restart is postponed if $avg^{ass}_{glo} > R \times avg^{ass}_{cur}$. Here again, in order to compute avg^{ass}_{cur}, we use a bounded queue (of size 5000). In our implementation, we use $R = 1.4$.

Recently, the authors of [17] proposed an original and elegant implementation of our restart strategy using an exponential moving average. This approach allows a few implementation advantages: it is computed using only a few integers variables and does not need bounded queues. More importantly, it favors the weights (LBD or number of assigned literals) of recent values, similarly to the VSIDS heuristic.

5. Parallel solving

5.1. *Introduction*

Following the progresses made in computer architectures, a number of attempts have been made for parallelizing SAT solvers [3,33,10]. Computers may have now many cores and distributed architectures are also easily available but parallelizing remains a challenge. The first idea of simply exploring different branches is not efficient (industrial instances may have hundreds of thousands of variables, and thus only splitting on 10 variables (using already 1024 cores) does not change the initial problem much). Moreover, it is harder and harder to improve single core SAT solvers [8], and thus exploiting multicores architecture may be a good way of significantly improving them. Two main approaches are commonly used to parallelize SAT solvers, the divide and conquer one[19,30,52] and the portfolio one[28,10,3]. In this paper, we only focus on portfolio solvers. The main idea is to exploit the complementarity between different sequential CDCL strategies to let them compete on the same formula with more or less cooperation between them [28,18,3]. Each thread deals with the whole formula and cooperation is achieved through the exchange of learnt clauses. Even if finding good different strategies is important for deploying a good portfolio solver, the key point is the clause exchange. When a solver learns a clause, it (directly or not) has to decide if this clause has to be shared or not. Usually, solvers send clauses of small size [28,15] or of small LBD [3] but may consider additional dynamic limits [27]. Less works have been conducted on importation policies. Some solvers do not directly use imported clauses. For example, PENELOPE uses the concept of freezing mechanism to add or not the imported clause in the search space [3].

In the next section, we explain the novel strategies used in our parallel portfolio version of GLUCOSE (called SYRUP: a lot of GLUCOSE) [7].

5.2. *Glucose (Syrup) strategy*

First of all, note that our approach tries to restrict the "orthogonal" search to its minimum. Thus, we rather see our approach as a simple parallelization effort of the same engine rather than a "portfolio" one. Our final goal is to see all solvers working together to produce a single proof, as short as possible. So, we must carefully manage shared clauses.

In SYRUP, we have a strategy for clauses exportations and for clauses importations. Let us first define our exportation policy. Exporting clauses can be highly harmful, especially if we target a high number of cores. On computers with 64 cores, it is easy to flood a core with too many clauses from the other cores. We based our strategies on the following observations. A sequential solver (here GLUCOSE) learns a lot of clauses that are seen only once (in conflict reasoning) after their computation (see Figure 4). If we carefully observe this figure, we can see that only 34% of them are seen at least one time during conflict analysis, which is surprisingly low. If we focus on clauses seen at least twice, this ratio falls to 22%. So, if we consider a

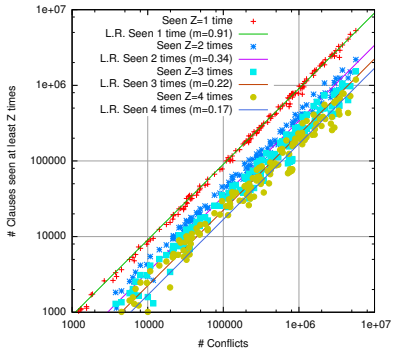


Fig. 4. Scatter plot of the number of conflicts (X axis) against the number of clauses seen at least Z={1, 2, 3, 4} times for all the successful launches (on SAT 2011 problems) in GLUCOSE

clause useful if it is used many times during conflict analysis, this simple measure is highly discriminating. It seems not realistic to send to the other cores clauses that are useless even for a single engine. The key point of SYRUP is thus to not send clauses immediately after their creation. The solver waits until the clause is seen a given number of times during conflict analysis before sending it. Starting from these observations, we choose the following filters for clauses exportation:

- Unaries and glues: these clauses are immediately sent after their computation. They are assumed to be good clauses.
- Clauses seen at least twicein during conflict analysis are sent. Furthermore, we add a limit on their LBD (less than the median of all LBDs) and their size (less than the average of all size).

Now, let us focus on the importation policy. An imported clause can have a bad impact on the search space. This is especially true for satisfiable instances where the solver tries to find a complete interpretation. A learnt clause can force the solver to go in the wrong direction. Importing clauses is done at each restart. This avoid extra works (for the 2-Watched literal scheme to work, the clause needs to be partially ordered according to the partial assignment), and this is possible thanks to our restart strategy, which is very aggressive (see Section 4). The novelty of our approach is the principle of probation for imported clauses (except binary clauses). When a clause is imported, we watch it only by one literal. This watching scheme does not guarantee anymore that all unit propagations are performed after each decision. However, this is sufficient to ensure that any conflicting clause will be detected during unit propagation. The interest of this technique is twofold. Firstly, the imported clauses will not pollute the current search of the solver, except when they are falsified. Indeed, the search of a solver is heavily guided by its own learnt clauses and external clauses will not have any negative impact until they became false. The classical learnt database deletion is also circumscribed to local learnt clauses. When an imported clause is falsified, it is “promoted”, i.e., it is watched

with 2 literals like any other internal clause. The clause is then part of the solver search strategy, and can be used for propagation. Some other advantages come from this strategy. Less efforts for propagations is due to keep track of the clause status until it is promoted and, more importantly, a LBD score is computed when the clause is promoted, giving to this clause a "*local*" score, according to the current core search space. This strategy is very restrictive: in average, only 10% of imported clauses are effectively promoted. Figure 5 shows a plot of the number of imported clauses that are finally promoted.

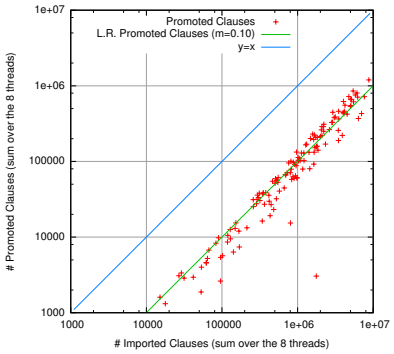


Fig. 5. Scatter plot of the number of conflicts (X axis) against the number of clauses that are finally promoted after being imported into a core, for all the successful launches (on SAT 2011 problems) in GLUCOSE. L.R. stands for Linear Regression.

6. Incremental SAT

In the last years, a new use of SAT solvers has emerged, called incremental SAT solving. This mode of operation was already implemented in MINISAT, but the number of applications using it increased a lot recently. It allows to use the same SAT engine to solve similar formulas (with only a few differences in the clauses set). In such a case, the SAT solver does not solve a single (potentially) hard instance, but may be used as an oracle and called thousands of times on a number of instances close to each others. Incremental SAT solving has many applications: Minimal unsatisfiable core extraction (MUS) [12], MAXSAT solving [41], planning [38], bounded model checking [16], or argumentation [39] are some examples. In general, the solver solves a formula Σ, then immediately after, it solves a formula Σ' such that Σ' has a subset of additional clauses (Σ^+) and a subset of clauses deleted (Σ^-), that is $\Sigma' = \Sigma \backslash \Sigma^- \cup \Sigma^+$.

Moreover, CDCL SAT solvers only consider their past activity to guide the search (heuristic VSIDS, learning, phase saving...). Then, it seems clear, in order to have the best possible performances, to keep the solver *alive* between two successive calls while giving him the possibility to add and remove clauses. This is what is called

incremental SAT solving [44,22]. Adding new clauses (and new variables) is very easy to implement and is natively included in all CDCL solvers. Deleting initial clauses is much more complex, but can be done in a smart way using assumptions. Let us explain how one can try to efficiently keep learnt clauses from the past while removing some initial clause.

When an original clause is temporarily (or not) deleted from the current formula, is not possible to directly re-use all learnt clauses that were computed using this last one. A simple and elegant solution to this is to add assumptions to solvers. Assumptions are characterized by a set $\mathcal{A}$ of literals assigned to true before all decision variables [22]. Then, one additional decision level can be needed per assumption, and the search really starts when all assumptions are assigned. The first *real* decision is made at level $d = |\mathcal{A}| + 1$.

Assumptions can be used to activate/disable a clause c. For that purpose, one has to associate to c a new variable s_i called selector. The literal s_i is added to the clause c. If the selector s_i is assigned to false (resp true) then the clause is activated (resp. disabled). Selectors appear only positively in the formula, and, thanks to the conflict analysis mechanism, all learnt clauses keep a footprint (the associated selector s_i) of all original clauses used to derivate them. When a selector s_i is assigned to true, the related clause is disabled, but also all learnt clause that were derivated from it: as it was considered by the solver as a decision, these literals occur in all these learnt clauses by construction.

Let us illustrate this with a modified version of the example introduced in Section 2.1. Suppose we add a selector s_i for all clauses $c_1 \dots c_8$ (remember that we add each selector to its clause). We choose to activate all clauses, then all selectors are assigned first (to true) and the partial interpretation is $\mathcal{I} = \langle \neg s_1 @ 1 \rangle \dots \langle \neg s_8 @ 8 \rangle$. Obviously, in this case, we obtain exactly the initial formula. The same decision literals produce the same partial interpretation, with different decision levels (for example x_1 is assigned at level 9). Thus, we obtain the same conflict to analyse, leading to the following learnt clause:

- $d^* = c_7 \otimes_{x_9} c_6 = \neg x_3 \vee x_8 \vee \neg x_7 \vee \neg x_{13} \vee s_7 \vee s_6$
- $\gamma = d^* \otimes_{x_{13}} c_5 = \neg x_3 \vee x_8 \vee \neg x_7 \vee s_7 \vee s_6 \vee s_5$

It is clear that learnt clauses will contain all selectors seen in conflict analysis, tracing their origin. On this simple example, γ contains 3 selectors. If on a next call to the SAT oracle, we choose to disable c_5, then the selector s_5 will be assigned to true. In such a case, we will disable c_5 and γ, a learnt clause that has c_5 from origin.

In [4], we modified GLUCOSE in order to efficiently cope with incremental SAT solving. We especially targeted applications needing a lot of selectors, like MUS extraction [12], where there is one selector per clause. All learnt clause have a footprint of all original clauses that were used in the resolution process to derive them. Of course, the number of selectors in learnt clauses can be very large but, additionally, each selector has its own decision level. We shown in [4] that the LBD of learnt

clauses in this context is approximatively the same as their size, leading to a weaker LBD measure (see Section 3). Then, in case of incremental SAT solving we changed the definition of LBD by not taking into account selectors for its computation. Furthermore, since learnt clauses with a lot of selectors can be very large, we also adapted some algorithms and techniques:

- Update LBD of clauses carefully, by recording the size without selectors.
- Change the traversal of learnt clause for checking if they are unit (or in conflict). For this, selectors are pushed to the end of the clause in memory.
- Check only few literals when removing satisfiable clauses

All this modifications improved a lot GLUCOSE in the context incremental SAT solving. Today, GLUCOSE is the SAT-based engine of many solvers that deal with incremental SAT solving: MUS (MUSER [12]), MAXSAT (OPEN-WBO [41], EVA500 [45]), argumentation (COQUIAAS[39])...

7. Summary

As it is shown in this paper, since its first release in 2009, GLUCOSE has been significantly improved. In order to illustrate this, let us report here the performances of the main released versions over the years. Before that, and once again, it is important to note that GLUCOSE is essentially based on the famous solver MINISAT [21] and an important part of this work was possible thanks to this solver.

2009 (Version 1.0). Introduction of LBD. First implementation (based on MINISAT 2.0). Dynamic restarts. See [5] for more details.
2011 (Version 2.0). Based on MINISAT 2.2. More agressive deletion strategy. Add additional features.
2012 (Version 2.3). Add new restart strategies that blocks restarts (see Section 4).
2013 (Version 3.0). Add incremental features (see Section 6). Add certified unsat proofs in DRUP format[29] (thanks to M. Heule to provide us the patch).
2014 (Version 4.0) Add the multi-thread portfolio approach (see Section 5).
2016 (Version 4.1) Add additional features. The solver *learns* from the first conflicts and try to adapt its search in consequence. Add new heuristic for the phase of decision variables [8].

We propose now to evaluate versions 1.0, 2.0, 3.0, 4.1 of GLUCOSE and SYRUP (with version 4.1 of GLUCOSE as core engine, 8 cores) on all application/industrial benchmarks from competitions 2011 to 2016. Each pool of instances contains 300 problems coming from the application track. The time limit is set to 2500 seconds. In order to contextualize these results, we also add the last version of LINGELING (version bbc that participated to the SAT competition 2016), one of the best SAT solvers over the years. Its sequential version (LGL) and parallel one (PLGL) are part of these experiments. Results are shown Table 7.

Table 1. Comparison of different versions of GLUCOSE on SAT competitions benchmarks. Sequential (LGL and parallel version (PLGL of LINGELING are also present in this comparison. Time limit is set to 2500 seconds. Each column (T U/S) gives the total (T) number of problem solved with the number of UNSAT (U) and SAT (S) ones. Last column represents the total number of problem solved on all competitions. Best results for a given competition are underlined.

Solver	2011		2013		2014		2015		2016		Total	
1.0	183	98/85	166	71/95	163	88/75	196	85/111	113	59/54	821	401/420
2.0	199	110/89	191	87/104	194	98/96	233	95/138	127	69/58	944	459/485
3.0	202	114/88	191	91/100	199	104/95	226	97/129	132	74/58	950	480/470
4.1	205	115/90	231	108/123	211	111/100	232	100/132	132	72/60	1011	506/505
LGL	207	123/84	191	98/93	226	129/97	241	106/135	140	85/55	1005	541/464
SYRUP	231	134/97	263	132/131	236	128/108	262	109/153	160	96/64	1152	599/553
PLGL	239	137/102	254	133/121	250	142/108	264	112/152	169	103/66	1176	627/549

Let us start to analyse these results by looking at the total number of problems solved (last column). First of all, the improvements over the year with GLUCOSE is clear. Each revision increases the number of problems solved. A big improvement was done with version 2.0. It is important to note that version 2.0 uses a new version of the solver MINISAT and a more agressive restart scheme (see Section 4). Last version of GLUCOSE also provides a big improvement, partly because of the analysis of the search behavior [8]. Solvers GLUCOSE 4.1 and LINGELING are comparable. Solver GLUCOSE solves the largest number of problems but LINGELING solves much more unsatisfiable ones.

Now, if we compare each competitions set of benchmarks, we can note that GLUCOSE becomes better and better version after version. One exception here, the version 2.0 solves the biggest number of satisfiable problems in 2015. LINGELING solves slightly more problems than GLUCOSE 4.1 for all competitions, except in 2013 with a big gap for GLUCOSE 4.1. This difference between competition results highlights the problem of benchmarks selection. For instance, GLUCOSE 4.1 is not the best on cryptographic problems (this was already pointed in our paper [5]), whereas LINGELING performs very well on such benchmarks (inprocessing techniques are, in this case, useful [36,9]).

Let us now focus on parallel versions, SYRUP and PLINGELING. One can note huge improvements comparing to their single core engine. Here again these two solvers obtain comparable results and here again SYRUP is better on satisfiable instances whereas PLINGELING is better on unsatisfiable ones.

However, the rules of the SAT competition allow only a loose comparison of solvers. In the early versions of the competition, the comparison of two solvers was not only based on the number of solved instances. CPU time was considered. It was trivially believed a few years ago that a fast solver would be able to solve more instances, and thus simply comparing the number of solved instances was a good way of giving a simple and robust measure. However, as it is common as soon as a benchmarking measure is proposed, candidates to the SAT competition adapted their solvers to this rule. This has allowed a family of solvers that are slower but

Table 2. Careful ranking. For each competition and for all benchmarks), we compute the careful ranking by taking into account these 4 solvers. The higher is the number, the better is the solver.

Solver	2011	2013	2014	2015	2016	Total
GLUCOSE 4.1	-131	-163	-230	-269	-176	-856
LGL	-195	-399	-183	-209	-110	-1067
SYRUP	161	**350**	**228**	**274**	136	**1131**
PLGL	**165**	212	185	204	**150**	792

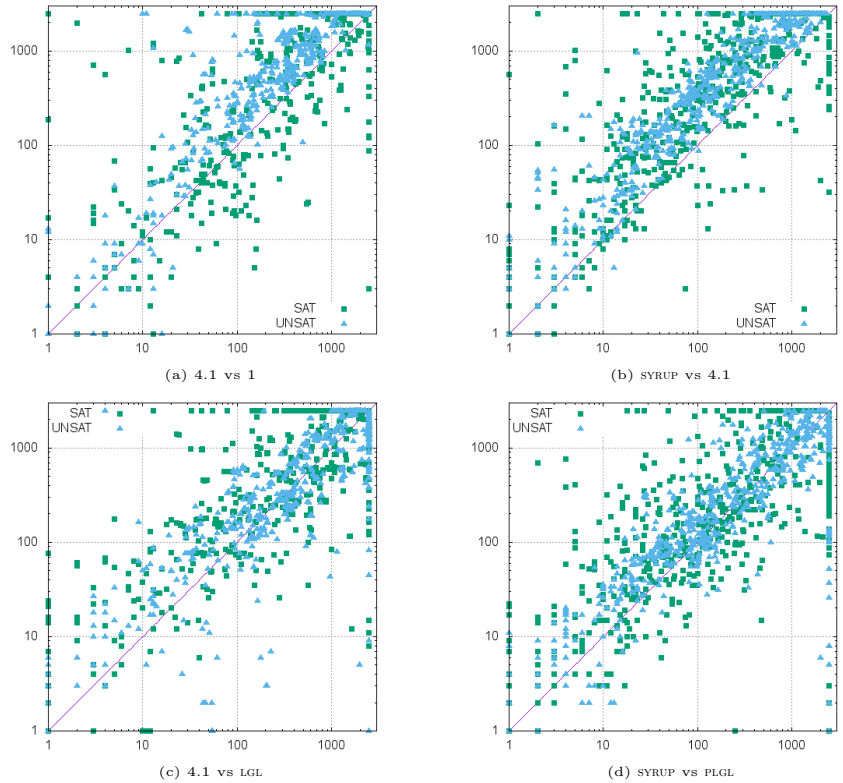
that can solve more instances if the CPU time stays in the same order of magnitude (typically 5000s). For instance, the 2016 winner has a strategy that switches the branching strategy after 2500s. Inprocessing techniques are also, to some extend, targeting long run. The comparison of CPU time is not trivial, essentially because of the timeout parameter. Here, we report the same results as above by using the Careful Ranking proposed by Alen Van Gelder [23], that, intuitively, tries to take the CPU time and the timeouts into account. We consider only version 4.1 of GLUCOSE and SYRUP. We restrict this table to only 4 solvers to make the comparison stronger (having more GLUCOSE than LINGELING may introduce an bias in the final score, even if the relative ranking should remain). Results are available Table 7.

What can be concluded from this set of results is that PLINGLING and SYRUP are very comparable, depending on the set of selected benchmarks. More surprisingly, if we sum up all the benchmarks, the picture is now not the same as above. If we consider only the number of solved instance, PLINGLING is winning. Now, if we consider the CPU time, the picture is clearly in favor of SYRUP and GLUCOSE. That means that, even if SYRUP solves fewer benchmarks, it is generally faster than PLINGING. This is the same for LINGELING and GLUCOSE.

Now, let us finish this section with 4 scatter plots, Figure 6. The first one (a) compares GLUCOSE 4.1 with the initial revision. This scatter confirms that version 4.1 is much more faster, but provides new insights: for many satisfiable instances, version 1.0 is faster than version 4.1. This is probably due to the agressive policies: satisfiable instances need long restarts and larger learnt clauses database. The second one (b), provides the comparison between SYRUP and GLUCOSE 4.1. Here, results are obvious: except for few satisfiable instances, the portfolio approach outperforms the sequential one. Finally, let us compare our solver with LINGELING sequential (c) and parallel (d) versions). This shows, graphically, what was pointed out above thanks to the careful ranking: GLUCOSE (resp. SYRUP) is faster than LINGELING (resp. PLINGELING) even if it solves fewer instances.

8. Conclusion and future works

In this paper, we retraced the different enhancements added to the original GLUCOSE, introduced in 2009. We showed that all the mechanisms underlying the Literal Block Distance (LBD), a central mesure of clause usefulness, can be used in many



(a) 4.1 vs 1 (b) SYRUP vs 4.1

(c) 4.1 vs LGL (d) SYRUP vs PLGL

Fig. 6. Different scatter plots. Each dot represents an instance. All competition instances (1500) are represented. When the dot is above the diagonal then the first solver (version 4.1 for fig. (a)) is faster than the second one (version 1 for fig. (a)) to solve the given instance.

components of a core Conflict Driven Clause Learning algorithm. This measure is crucial to identify good clauses but also when to restart.

However, given the number of new features added to GLUCOSE over the years, it may be difficult to understand which contributions are the most important, for someone that did not follow all the history of our solver. This paper aims at detailing the main techniques that can be found in the last version of GLUCOSE, focusing on the most important ones. We particularly focus also on the parallel version of GLUCOSE, that introduced the concept of Lazy sharing of clauses.

Thanks to an exhaustive experimental study of the main releases of GLUCOSE, over the years, tested against a large set of benchmarcks, we thus demonstrated that the notion of LBD is still very important for SAT solvers, and illustrated the performance gain. We showed that, GLUCOSE was constantly offering the same performances as the last LINGELING solver, taken as a reference in this paper. This observations were extended to the parallel version of each solver. Moreover, we showed that, as soon as a more precise measure for solver comparison is used, taking into account the CPU time, the balance is clearly in favor of Glucose.

References

1. C. Ansótegui, ML. Bonet, J. Giráldez-Cru, and J. Levy. The fractal dimension of SAT formulas. In *proceedings of IJCAR*, pages 107–121, 2014.
2. C. Ansótegui, J. Giráldez-Cru, and J. Levy. The community structure of sat formulas. In *Theory and Applications of Satisfiability Testing SAT*, pages 410–423, 2012.
3. G. Audemard, B. Hoessen, S. Jabbour, JM. Lagniez, and C. Piette. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing - SAT*, pages 200–213, 2012.
4. G. Audemard, JM. Lagniez, and L. Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Theory and Applications of Satisfiability Testing - SAT*, pages 309–317, 2013.
5. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI'09*, pages 399–404, 2009.
6. 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.
7. G. Audemard and L. Simon. Lazy clause exchange policy for parallel SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 197–205, 2014.
8. G. Audemard and L. Simon. Extreme cases in SAT problems. In *Theory and Applications of Satisfiability Testing - SAT*, pages 87–103, 2016.
9. T. Balyo, A. Fröhlich, M. Heule, and A. Biere. Everything you always wanted to know about blocked sets (but were afraid to ask). In *Theory and Applications of Satisfiability Testing - SAT*, pages 317–332, 2014.
10. T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing - SAT*, pages 156–172, 2015.
11. P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
12. A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proc. FMCAD'11*, pages 37–40. FMCAD, 2011.
13. A. Biere. Adaptive restart strategies for conflict driven sat solvers. In *11th international conference on Theory and Applications of Satisfiability Testing - SAT*, pages 28–33, 2008.
14. A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
15. A. Biere. Lingeling and friends at the SAT Competition 2011. FMV Report Series Technical Report 11/1, Johannes Kepler University, Linz, Austria, 2011.
16. 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.
17. A. Biere and A. Fröhlich. Evaluating CDCL restart schemes (preliminary version). In *Workshop on Pragmatics of SAT (POS)*, 2015.
18. Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 51, 2013.
19. W. Chrabakh and R. Wolski. GrADSAT: A parallel SAT solver for the grid. Technical report, UCSB, 2003.
20. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 1962.
21. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. SAT'03*, pages 333–336, 2003.

22. N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
23. Allen Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In *Proceedings of Theory and Applications of Satisfiability Testing*, pages 317–328, 2011.
24. D. Gelperin. Deletion-directed search in resolution-based proof procedures. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence.3*, pages 47–50, 1973.
25. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. DATE'02*, pages 142–149, 2002.
26. C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of AAAI*, pages 431–437, 1998.
27. Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 499–504, 2009.
28. Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.
29. M. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 181–188, 2013.
30. M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 50–65, 2011.
31. J. Huang. The effect of restarts on the effectiveness of clause learning. In *Proc. IJCAI'07*, 2007.
32. J. Huang. The effect of restarts on the efficiency of clause learning. In *Proc. IJCAI*, pages 2318–2323, 2007.
33. A. Eero Johannes Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, pages 372–386, 2010.
34. S. Jabbour, J. Lonlac, L. Sais, and Y. Salhi. Revisiting the learned clauses database reduction strategies. Technical report, coRR, arXiv:1402.1956, 2014.
35. S. Jabbour and L. Sais. personnal communication, February 2008.
36. M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. In *proceedings of International Joint Conference of Automated Reasoning, IJCAR*, pages 355–370, 2012.
37. R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208, 1997.
38. H. A. Kautz and B. Selman. Planning as satisfiability. In *European Conferance on Artificial Intelligence*, pages 359–363, 1992.
39. JM. Lagniez, E. Lonca, and JG. Mailly. Coquiaas: A constraint-based quick abstract argumentation solver. In *International Conference on Tools With Artificial Intelligence*, pages 928–935, 2015.
40. 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.
41. R. Martins, V. M. Manquinho, and I. Lynce. Open-wbo: A modular maxsat solver,. In *17th international conference on Theory and Applications of Satisfiability Testing - SAT*, pages 438–445, 2014.

42. M. Moskewicz, C. Conor, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC'01*, 2001.
43. H. Nabeshima, K. Iwanuma, and K. Inoue. Glueminisat2.2.5. In *SAT competition, system description*, 2011.
44. A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *Proc. SAT'12*, 2012.
45. N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2717–2723, 2014.
46. Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon. Impact of community structure on SAT solver performance. In *Theory and Applications of Satisfiability Testing - SAT*, pages 252–268, 2014.
47. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proc. SAT'07*, pages 294–299, 2007.
48. K. Pipatsrisawat and A. Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1481–1484, 2008.
49. K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers with restarts. In *proceedings of CP*, pages 654–668, 2009.
50. 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*, pages 341–355, 2009.
51. L. Ryan. *Efficient algorithms for clause learning SAT solver*. PhD thesis, Simon Fraser University, School of Computing Science, 2004.
52. A. Semenov and O. Zaikin. Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 222–230, 2015.
53. J. P. Marques Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proc. CAD'96*, pages 220–227, 1996.
54. 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.