Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction

Gilles Audemard[1,⋆], Jean-Marie Lagniez[2,⋆⋆], and Laurent Simon[3]

[1] CRIL, Artois University, France
audemard@cril.fr
[2] FMV, JKU University, Austria
Jean-Marie.Lagniez@jku.at
[3] LRI, University Paris Sud, France
simon@lri.fr

Abstract. Beside the important progresses observed in SAT solving, a number of applications explicitly rely on incremental SAT solving only. In this paper, we focus on refining the incremental SAT Solver `Glucose`, from the SAT engine perspective, and address a number of unseen problems this new use of SAT solvers opened. By playing on clause database cleaning, assumptions managements and other classical parameters, we show that our approach immediately and significantly improves an intensive assumption-based incremental SAT solving task: Minimal Unsatisfiable Set. We believe this work could bring immediate benefits in a number of other applications relying on incremental SAT.

1 Introduction

The Satisfiability (SAT) status of a propositional formula is one of the most fundamental question in computer science, heavily studied since the 70's, both theoretically and practically. This problem captures the hardness of a large set of difficult – but practically interesting –problems that could arise in many applications. Following the famous proof of SAT NP-Completeness [1], a number of work have reduced many other problems to SAT. In 2001 [2], based on the ideas of Marques-Silva and Sakallah [3], a new era of SAT solvers was born, called "Modern" SAT Solvers [4]. The tremendous progresses observed in SAT had an important impact in many other problems of computer science: the State of the Art in solving a number of NP-Complete problems is now to use a SAT solver engine. The efficiency of data structures used in SAT solvers also allows to work with problems of millions of variables and clauses, which typically correspond to SAT encodings of problems above NP [5,6].

However, since a few years, a new use of SAT solvers has emerged, called "Incremental SAT Solving". Even if this mechanism was already proposed in earlier versions of `Minisat` [4], its importance has grown in the very last years. It is now the state of the art to rely on this new paradigm in many applications, like in bounded model checking [7], extraction of a Minimal Unsatisfiable Set (MUS) [8], Maximum Satisfiability (MaxSAT) [9] or even inductive verification [10]. In this context, SAT solvers

⋆ Partially supported by CNRS and OSEO, under the ISI project "Pajero".
⋆⋆ Partially supported by FWF, NFN Grant S11408-N23 (RiSE), an EC FEDER grant.

M. Järvisalo and A. Van Gelder (Eds.): SAT 2013, LNCS 7962, pp. 309–317, 2013.

are not run on a single, potentially huge, SAT problem. They are rather called thousands times on a number of instances close to each other (with removed/added clauses), which allows to reuse as much informations as possible between successive SAT calls. However, it is clear that, for some applications (when constraints can be removed), the information held by the learned clauses can not be directly reused. For allowing the removal of clauses, it is necessary to add "assumptions" to the SAT solver. Assumptions are literals that are assumed to be true and which are always picked first for decisions during a single run.

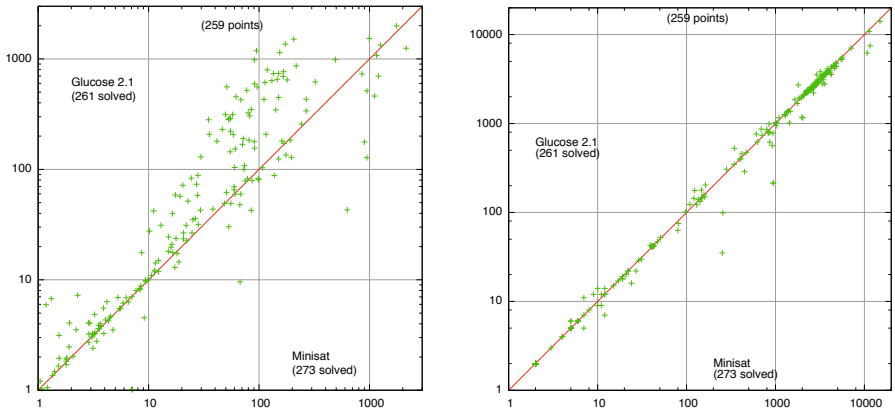
In this paper, we take a typical use of incremental SAT solving, heavily relying on assumptions, hoping that our results will be immediately useful for other incremental uses. We base our work on the highly tunable and open source `Muser` [11] (see also [12,8,13] for MUS extraction). We focus here on the SAT engine and demonstrate how we can improve the result of the overall incremental SAT solving by carefully taking into account the essential ingredients of SAT solvers. In particular, we are interested in pushing to the incremental case the solver `glucose` [14]. Memory consumption is indeed a strong limitation of SAT solving with assumptions, because each clause may contain hundreds of literals. Being able to remove useless clauses is essential and successfully applying `glucose` strategies may be crucial.

2 SAT and Incremental SAT

So-called "modern SAT solvers" are based on the *conflict driven clause learning* paradigm (CDCL) [2]. If they were initially introduced as an extension of the DPLL algorithm [15], with a powerful conflict clause learning [3,16] scheme, it is acknowledged that they must be described as a mix between backtrack search and plain resolution engines. They integrate a number of effective techniques including clause learning, highly dynamic variable ordering heuristic [2], polarity heuristic [17], clause deletion strategy [18,14,19] and search restarts [20,21] (see [22] for a detailed survey).

2.1 Incremental SAT Solving with Selectors

In this paper, we focus on improving SAT engines for *incremental SAT Solving* (see [23] for a detailed introduction). In incremental SAT solving, the same solver is ran on a number of instances close to each others and the solver state is memorized between each call. For instance, the final state of variable ordering [2], polarity cache [17] and clause deletion/restarts strategies [18,14,19] can be easily saved for the next call. However, in most of the cases, *i.e.* as soon as initial constraints can be removed, learned clauses can not be directly reused. This is why it is necessary to add "assumptions" to the SAT solver. Assumptions in SAT solvers were already proposed in early versions of `Minisat` [4] and are daily used in many distinct applications (*s.t.* bounded model checking [7], extraction of a minimal unsatisfiable set [8] ...). A set of assumptions $\mathcal{A}$ is defined as a set of literals that are assumed to be true and which are picked for decisions first, always in the top of the search tree ([23] proposed another way of handling them, which can be easily used with our approach). Then, if during the search, it is needed to flip the assignment of one of these assumptions to false, the problem is unsatisfiable under the initial assumptions.



(a) running time (b) number of SAT solver calls

Fig. 1. Comparison of `Glucose` 2.1 performances against `Minisat` in `Muser`. Figure (Left) us the loglog plot of `Minisat` (x-axis) against `Glucose` (y-axis) in CPU time (seconds). Figure (Right) is the loglog plot of `Minisat` (x-axis) agains `Glucose` (y-axis) in number of calls to the SAT solver by `Muser`.

When an assumption is used for activating/deactivating a clause (one fresh assumption for each clause), it is called a *selector* for this clause: if the literal is false (resp. true) under the current assumptions, then the clause is activated (resp. deactivated). Therefore, since these additional literals appear only positively in the formula, the learned clauses obtained during the search process keep track of all the initial clauses used to produce them. Then, removing the set of clauses which are derived from an initial clause with selector a can be easily performed by simply assuming a to true.

2.2 Using Glucose in Minimal UNSAT Set Extraction Problem

As a typical application of incremental SAT solving with selectors, we focus on the *minimal unsatisfiable set* (MUS) extraction problem [24]. In this approach, a unique selector is added to *each clause* and the SAT solver is incrementally called with most of these additional literals as assumptions. This application is quite typical and challenging for incremental SAT solvers: the number of assumptions is usually quite large (equal to the number of clauses in the formula) and the SAT solver is called many times. We rely our work on the highly tunable and open source `Muser` [11] but we focus only on the SAT engine. Let us precise that we use `Muser` with default options, *i.e.* the hybrid algorithm is used (essentially deletion-based) with clause-set refinement and model rotation (see [11] for more details). Note also that we use the 300 benchmarks from the MUS track of the SAT 2011 competition. Experiments are done on Intel XEON X5550 quad-cores 2.66 GHz with 32Gb RAM. CPU time limit is 2400s.

If we want to be able to efficiently reuse learned clauses, it is essential to be able to distinguish which clauses may be useful for futures calls. In this context, it may be a good choice to use `Glucose` [14], a variant of `Minisat` [4] based on an efficient scoring mechanism of clauses, called Literal Block Distance (LBD). So, as a starting

point for our work, we compared the integration of Glucose against Minisat in Muser. Our surprisingly bad results are summarized Fig. 2.2 (for each scatter plot of the paper, we provide the number of benchmarks solved by each method, the number of benchmarks solved by both methods (259 for Fig 1(a)) and the less points are located in the area of a given method, the best this last one is). The two figures are quite interesting, however. On one side, the CPU time comparison, Figure 1(a), is in clear favor of Minisat. On the other side, Figure 1(b) shows this comparison from a number of SAT calls point of view. On this last figure, the comparison is not so clear, and some interesting points show that Glucose can sometimes require less SAT calls. Moreover, we found that, over the 259 run both solved by Glucose and Minisat, 103 runs of Glucose have fewer calls than Minisat, 68 runs of Glucose have more calls than Minisat and 88 runs have the same number of calls. Using Glucose instead of Minisat seems a good option if we can fix the CPU time problem. Intuitively, we think that Glucose is able to derive proofs with fewer initial clauses, thus helping Muser to quickly converge. Together with the idea of proposing a better clause database management, this was our initial motivations for this work.

How can we explain such a disappointing result? Glucose is updating the scores of clauses during unit-propagation. Thus, when clauses tend to have thousands of literals, it adds a prohibitive penalty. However, it is also important to notice that, on hard MUS problems, Minisat showed (not reported here) some limitations for scaling-up, mainly due to memory consumption problems (6 Memory out). On hard MUS problems, one may have to handle hundreds of hard UNSAT calls, and the ability of Glucose to handle long runs with many learned clauses may be crucial here. In the following, we will show how it is possible to adapt Glucose mechanisms to incremental SAT solving.

3 Improving Incremental SAT Engines

3.1 Assumptions and LBD

In Minisat, each assumption has its own decision level, except when it was propagated by other assumptions assignments. Thus, in most of the cases, the LBD score will be dominated by the number of assumptions in the clause. For most of the learned clauses, this value will be quickly approximating its size. Given the fact that, in addition, the LBD score is very discriminating (clauses of LBD $n + 1$ are significantly less important than clauses of LBD n), we must get rid of assumption literals during LBD scoring. We propose to adapt the LBD by simply skipping assumption literals during its computation. This new LBD scoring will be called hereafter "New LBD".

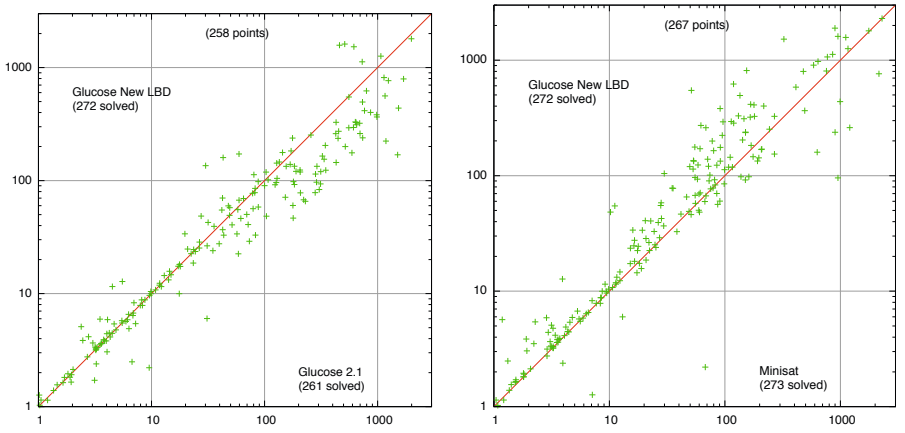
Tab. 1 shows some statistics about the above remark. On four representative instances, we detail the initial LBD score obtained and the new LBD score as previously described. As we can see, it is clear that using the initial LBD score is not meaningful in the context of SAT solving with selectors. The second part of Tab. 1 shows the same statistics with the New LBD definition. As one may observe, the CPU time is significantly improved. Moreover, LBD scores are smaller and are no more related to the size of learned clauses (when counting assumptions). Furthermore, average size of learned

Table 1. For some characteristics instances, we provide the number of clauses and, for each LBD score, we also report the time to compute the MUS, the maximum and average size, and the LBD of learned clauses

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

clauses may be larger for the new LBD score than for the initial one: good LBD clauses are not necessarily short ones.

However, as we can see Fig. 2, even if we increase `Glucose` performances, this new version is very close to `Minisat` (in number of solved instances), but slower. At this stage, such results are relatively disappointing: `Glucose` is supposed to be more efficient than `Minisat` in non incremental SAT solving. In order to improve its overall performances, we propose hereafter some very important improvements.



(a) `Glucose` vs. `Glucose` with New LBD (b) `Minisat` vs. `Glucose` with New LBD

Fig. 2. Comparison of `Glucose` 2.1 performances against `Glucose` with a new LBD scoring in `Muser`. Fig. 2(a) is the loglog plot of `Glucose` 2.1 (x-axis) against `Glucose` with New LBD (y-axis) in CPU time (seconds). Fig. 2(b) is the loglog plot of `Minisat` (x-axis) against `Glucose` with New LBD (y-axis) in CPU time (seconds).

3.2 Improving Performances

As pointed out Tab 1, the produced learned clauses can be quite large. For example, the average size of learned clauses on instance `g7n` is 1729 (with clauses with more

than 10,000 literals). Thus, even a simple operation like clause traversal may quickly induce a prohibitive overhead when applied with selectors. This is very problematic as fundamental operations of SAT solvers are based on clauses traversals, *i.e.* BCP, updates of LBD scores or even simplification of clauses database (when removing satisfied clauses). Considering this operation as one of the main bottleneck for SAT solving with selectors, we propose several improvements specialized for these operations.

Efficient Traversal of Clauses with Many Selectors. As previously noticed, in most of the cases, learned clauses will be dominated by selector literals but, as they are not taken into account in the new LBD scoring, we do not have to check them during clauses traversals. Moreover, since these literals can be used as watchers during the BCP process, it is impossible to split the set of clause's literals into two independent sets (selectors/non selectors): we must be able to visit selector literals when needed. However, it is possible to store, in each learned clause, the number of initial literals and the total size of the clause (obviously, the number of selectors is redundant). In addition, we propose to push all the selectors at the end of the learned clauses. As a first, immediate, impact of this new clause arrangement, we can hope to speed up LBD score updates: we can stop the traversal as soon as all initial literals have been checked (number of initial literals in the clause is known).

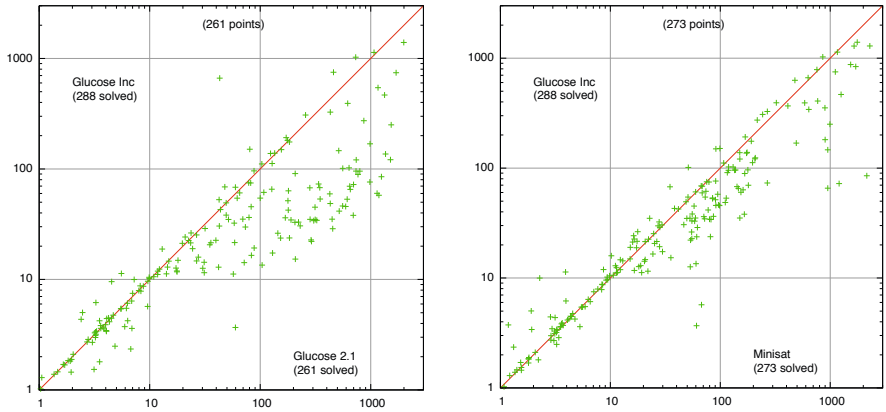
Improving BCP. If the above data structure modification allows to speed up LBD updates, the BCP engine also needs to traverse clauses when looking for new watchers. Suppose that, during the propagation of an assumption, the new watcher a for a clause c is chosen among the other selectors of the clause. Then, if a is assigned to false, c will be traversed again. This can be easily avoided: when propagating an assumption, we always traverse the entire clause in order to find a new watcher which is true (the clause is satisfied) or which is not a selector. Because `Glucose` is firing very frequent restarts, we also limit any restart to backtrack only until the decision level of the first non-selector decision level.

Database Simplification. The last important modification we introduced is related to the removal of satisfied learned clauses. This is very important in the case of `Muser`: each time a clause is known to be out of a MUS, its associated selector is definitively set to true. Then, all learned clauses where it appears will be satisfied. However, scanning all literals of all learned clauses to find these true literals can break the potential benefit of this technique. In the new version of `Glucose`, we limit the search for satisfied learned clauses to the two watched literals only. Since we changed the BCP process (see above), we expect that this will be sufficient to find and remove most satisfied clauses.

As a last small modification of `Glucose`, we keep current indicators for clause database cleanings and restarts from one run to the other, allowing the behavior of the solver to reflect the very high frequency of runs in most incremental SAT solving.

3.3 Overall Evaluation

We implemented all the above described techniques and called it `GlucoseInc`. Fig. 3 compares `GlucoseInc` against `Minisat` and `Glucose`. Results are clear: this new version outperforms the other methods (see also Fig. 4).



(a) Improvement over `Glucose` (b) Improvement over `Minisat`

Fig. 3. Comparison of `Glucose` 2.1 (resp. `Minisat`) performances against `GlucoseInc` in `Muser`. Fig. 3(a) (resp. 3(b)) use the loglog plot of `Glucose` (resp. `Minisat`), x-axis, against `GlucoseInc` (y-axis) in CPU time (seconds) in `Muser`.

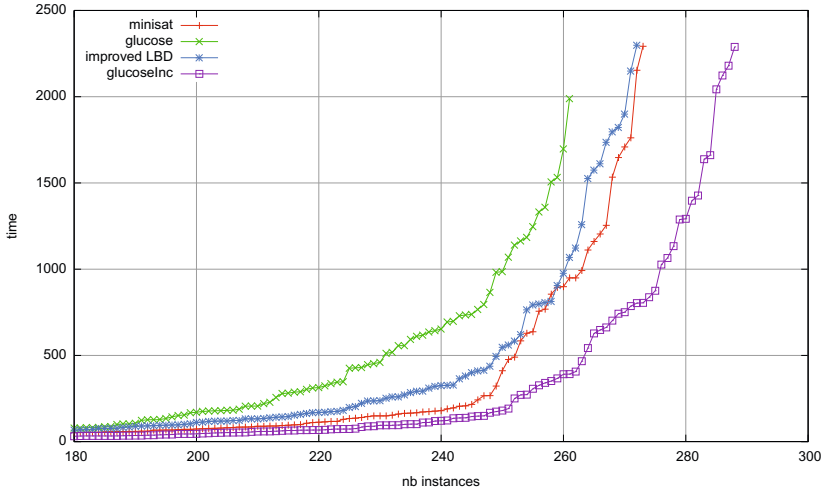


Fig. 4. Cactus plot. x-axis represents the number of instances and y-axis represents the time needed to solve them if they were ran in parallel

As we wrote in section 2.2, our goal was to improve performances of incremental SAT solving, and thus, performances for each call to the SAT solver. Tab. 2 shows that this goal is reached. It provides for `Minisat`, `Glucose` and `GlucoseInc` and for the same previous 4 instances, the number of SAT calls and the average time needed for each of them. Interestingly enough, one can notice that for each instance the total time needed to extract a MUS is improved, even if more SAT calls may be necessary. This is unfortunately a general remark: by dropping the quality of the LBD measure, we

Table 2. For representative instances, we provide the number of clauses, and for each incremental SAT engine, we provide the number of calls to the SAT solver and the average time needed for each call. For `GlucoseInc`, we also provide the total time (for other methods, this is done in Tab. 1)

		`Minisat`		`Glucose`		`GlucoseInc`		
Instance	#C	#SAT calls	avg	#SAT calls	avg	time	#SAT calls	avg
fdmus_b21_96	8541	2103	0.009	2134	0.02	11	2153	0.004
longmult6	8853	706	0.01	1027	0.03	13	748	0.01
dump_vc950	360419	7	135	11	11.5	65	9	7.2
g7n	70492	4791	0.02	4393	0.08	67	4779	0.01

loose the first observation we made. Now, using `Glucose` instead of `Minisat` does not show any improvements in the number of SAT calls by `Muser`.

4 Conclusion

One of the reasons for the success of SAT solvers is certainly their ability to be used as Black Boxes in many distinct applications. However, when a very specific new use is proposed, it may be important to adapt SAT solvers technologies accordingly, directly at the engine level. Incremental SAT solving is typically a new field of research for SAT solvers technologies. In this paper, we proposed to focus on an intensive incremental SAT solving task, Minimal UNSAT Set extraction, but by working only on the SAT engine level. This study led to an impressive improvement in MUS extraction performances, and we believe that our approach can also lead to substantial – and direct – improvements in many other applications relying on incremental SAT solving.

By playing only on the SAT engine, we were able to drastically improve the performances of `glucose` on MUS extraction. It has to be pointed out that the set of problems we based our study on is now solved at 96% (on 2400s). So, there is a chance that the remaining problems are very hard, and may not be a good set of problems for further improvements. For instance, we were able to solve only 4 additional problems by increasing the CPU time to 15000 seconds on the unsolved problems. Moreover, the efficient handling of clauses by `glucose` allows it to have no Memory Out problems where `Minisat` had 6 Memory Out under the same conditions.

This preliminary work has a lot of promising perspectives. For example, we plan to study dependencies beween selectors and to take them into account in order to improve performances. An other interesting future work is related to the adaptation of the SAT solver (in term of heuristics, restarts, ...) with respect to previous calls.

References

1. Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. STOC 1971, pp. 151–158 ACM (1971)
2. Moskewicz, M., Conor, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. DAC 2001 (2001)

3. Silva, J.P.M., Sakallah, K.: Grasp – a new search algorithm for satisfiability. In: Proc. CAD 1996, pp. 220–227 (1996)
4. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using sat procedures instead of bdds. In: Proc. DAC 1999, pp. 317–320 (1999)
6. Velev, M.N., Bryant, R.E.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. Journal of Symbolic Computation 35(2), 73–106 (2003)
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
8. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Proc. FMCAD 2010, pp. 221–229 (2010)
9. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)
10. Bradley, A.R.: IC3 and beyond: Incremental, inductive verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, p. 4. Springer, Heidelberg (2012)
11. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: Proc. FMCAD 2011, pp. 37–40 (2011)
12. Grégoire, É., Mazure, B., Piette, C.: Extracting muses. In: Proc. ECAI 2006. Frontiers in Artificial Intel. and Applications, vol. 141, pp. 387–391. IOS Press (2006)
13. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)
14. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proc. IJCAI 2009, pp. 399–404 (2009)
15. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM (1962)
16. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proc. CAD 2001, pp. 279–285 (2001)
17. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
18. Goldberg, E.I., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proc. DATE 2002, pp. 142–149 (2002)
19. Audemard, G., Lagniez, J.-M., Mazure, B., Saïs, L.: On freezing and reactivating learnt clauses. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 188–200. Springer, Heidelberg (2011)
20. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
21. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 118–126. Springer, Heidelberg (2012)
22. Marques-Silva, J., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. Frontiers in Artificial Intelligence and Applications, vol. 185, p. 980. IOS Press (2009)
23. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 242–255. Springer, Heidelberg (2012)
24. Oh, Y., Mneimneh, M., Andraus, Z., Sakallah, K., Markov, I.: AMUSE: a minimally-unsatisfiable subformula extractor. In: Design Automation Conference, pp. 518–523 (2004)