A Case for Simple SAT Solvers[*]

Jinbo Huang

Logic and Computation Program
National ICT Australia
Canberra, ACT 0200 Australia
jinbo.huang@nicta.com.au

Abstract. As SAT becomes more popular due to its ability to handle large real-world problems, progress in efficiency appears to have slowed down over the past few years. On the other hand, we now have access to many sophisticated implementations of SAT solvers, sometimes boasting large amounts of code. Although low-level optimizations can help, we argue that the SAT algorithm itself offers opportunities for more significant improvements. Specifically, we start with a no-frills solver implemented in less than 550 lines of code, and show that by focusing on the central aspects of the solver, higher performance can be achieved over some best existing solvers on a large set of benchmarks. This provides motivation for further research into these more important aspects of SAT algorithms, which we hope will lead to future significant advances in SAT.

1 Introduction

Modern clause learning technology coupled with data structures for fast unit propagation has greatly enhanced the efficiency and scalability of SAT, making it practical to tackle real-world instances with millions of clauses. However, over the past few years progress appears to have slowed down considerably. For example, MiniSat 2.0, winner of SAT Race 2006 [13], solved no more instances in the race than SatELiteGTI from 2005, and Cadence MiniSat v1.14 actually solved three fewer instances than MiniSat v1.13 from 2005. Even more unfortunately, our benchmarking shows that MiniSat 2.0 solves 59 fewer instances than BerkMin, a solver written in 2002 based on the same clause learning framework [4], on a set of 251 instances distributed by Miroslav Velev[1] given a one-hour time limit.

In this paper we would like to investigate what contributes to such substantial differences in performance between solvers. To begin with, clearly, difference in the efficiency of implementation alone would not explain these differences in performance we are witnessing, as that would imply, for example, that BerkMin would outperform MiniSat 2.0 in general, which is not true. Given a set of 311 instances from IBM [14] also used in our benchmarking, MiniSat 2.0 solves 103 more instances than BerkMin.

[*] National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

[1] http://www.miroslav-velev.com/sat_benchmarks.html.

In fact, we venture to argue that the sophistication of implementation, although responsible for some of the solver efficiency, may not be as critical as the choices made in the central aspects of the solver. In support of this argument, we start with a simple clause learning SAT solver written in less than 550 lines of C++ (excluding comments and blank lines), called Tinisat [5], and show that by improving its decision heuristic, a consistently high performance across different benchmark families, including IBM and Velev, can be achieved.

While our experiments with Tinisat speak to the importance of the decision heuristic, there are other important aspects of clause learning, such as the backtracking scheme, restart policy, and clause deletion policy. To offer a more complete picture, and to illustrate how a simple and effective implementation is possible, we devote the following section to a story that attempts to unify the various aspects of clause learning around a central theme so that their importance can be better understood. We then describe the experiments done regarding the decision heuristic, and finally the experiments that demonstrate the performance of Tinisat on a large number of industrial benchmarks.

2 A Unifying View of Clause Learning

Clause learning originated in SAT as a means of pruning DPLL search trees. The early SAT solver GRASP [9], for example, used clause learning to achieve a form of *nonchronological backtracking* where branches known to contain no solutions were skipped over during backtracking. Although the learning method introduced by GRASP has been adopted by later SAT solvers, a subtle yet critical change has occurred in how backtracking is done after learning a clause, starting with Chaff [10]. In fact, the backtracking scheme used by Chaff, and most of its successors, is so different from GRASP's that the overall SAT algorithm is no longer a binary DPLL search.

Repeated Probing To better understand this change, we have previously proposed an alternative formulation of clause learning as a simple 3-step cycle [5], which we now refer to as *repeated probing* and give as Algorithm 1.

This formulation makes it explicit that clause learning is no traditional binary search enhanced with pruning, but something very different and, in fact, simpler: It is a sequence of resolutions (line 2) and all components of the solvers serve a common purpose of guiding the resolution process. Using this formulation as the basis, we now embark on a unifying account of clause learning SAT algorithms. We start with a review of the learning process itself [9], now understood as a resolution process of which the learned clause is the final resolvent [2].

Algorithm 1 Repeated Probing

```
1: set variables till hitting a conflict; return SAT if all variables are set without conflict
2: derive a new clause by resolution; return UNSAT if empty clause is derived
3: unset some variables and go back to step 1.
```

Clause Learning as Resolution The process of learning commences upon the generation of an empty clause under a particular assignment of truth values to a subset of the variables, and ends with the derivation of a new clause that is entailed by the set of existing clauses. Recall that there are two types of assignments: decision assignments and implications produced by unit propagation. As is customary, we label each (decision or implication) assignment with the decision level in which it is made. In this paper we assume that the decision level is initially 1, and is incremented before each decision is made—hence the very first decision is made in level 2. This will allow us to label the literals that are implied by the CNF formula with level 1, and use level 0 to signify a fatal conflict. Note that the last case arises when a conflict occurs in decision level 1, at which point the CNF formula is immediately declared unsatisfiable without the need to go through the usual learning process.

To give a precise description of the learning process, we enlist the notion of the *antecedent clause* [9] for an implication, which is defined to be the clause that became unit and produced the implication. For example, after the assignments $[A, \overline{B}]$, the clause $c_1 : \overline{A} \vee B \vee \overline{C}$ becomes unit, and $\overline{C}$ is implied; clause c_1 is then the antecedent clause for the implication $\overline{C}$. Note that given any clause cl, if a literal of cl has become false due to an implication, then clause cl can always be resolved against the antecedent clause for that implication.

The learning process maintains a clause cl that is initialized to be the clause in which a conflict has just occurred (that is, whose literals have all been assigned false). Clause cl is then repeatedly updated to be the resolvent of itself and the antecedent clause for some implication that has made one of its literals false. Learning methods generally agree on the order of the resolution steps, but differ on the condition for terminating the loop. One of the most polular methods is 1-UIP [15], which terminates the resolution loop when clause cl contains exactly one literal whose value is assigned in the current decision level.

A Common Purpose Once one commits to a particular learning method, it becomes clear that the sequence of clauses learned is completely determined by the sequence of assignments made. Now, note that the latter is determined in turn by nothing other than the combination of the decision heuristic, backtracking scheme, restart policy, and (recursively) the learned clauses that have been added to the CNF formula (assuming a fixed implementation of unit propagation). For unsatisfiable instances, the sequence of learned clauses ends in the empty clause and determines the time and space efficiency of the algorithm. For satisfiable instances, the sequence of assignments made determines the exact point at which a solution is discovered.

Hence in all cases, the various components of the SAT solver can be understood as serving a single, common purpose—that of determining the sequence of assignments made and the sequence of clauses learned. Accordingly, the effectiveness of these components can be understood solely in terms of their contributions to the early termination of these sequences.

Backtracking Schemes To understand the role of backtracking, let us now go back to the *repeated probing* process given in Algorithm 1. Observe that each probing is a flat sequence of (decision and implication) assignments leading to a conflict (line 1), and each pair of consecutive probings share a prefix by means of partially (or completely as a special case) erasing the assignments of the previous probing in reverse chronological order (line 3)—this erasing of assignment is exactly what is known as *backtracking*.

Backtracking in systematic depth-first search has traditionally been understood as a means of (recursively) exploring each branch of a search node until a solution is found in one. This is no longer the case in modern clause learning. In the formulation of Algorithm 1, backtracking amounts to a heuristic for deciding what prefix of the current sequence of assignments should carry over to the next. The only property required of the heuristic is, naturally, that there should not be an empty clause after backtracking.

The weakest condition that guarantees this requirement is to erase all assignments in levels $\geq \beta$, where β is the highest decision level among literals in the clause just learned. This is in fact the backtracking scheme used in GRASP.

Any decision level $< \beta$, therefore, is a perfectly legal level to backtrack to. Hence Algorithm 1 subsumes the framework of *unrestricted backtracking* described in [8]. Although unrestricted backtracking can potentially result in better performance, one has yet to come up with heuristics that compute good backtracking points. Most current clause learning SAT solvers backtrack to (i.e., erase assignments in levels down to but excluding) the *assertion level*, which is the *second highest* decision level among literals in the learned clause, resulting in a generally more aggressive jump[2] than GRASP's backtracking.

Restarts A restart is a backtrack to decision level 1, where all assignments are erased except those implied by the CNF formula. In the context of Algorithm 1, the utility of a restart can be best understood as giving the solver an opportunity to make better decisions than it did in the last probing now that new information is available in the form of the newly learned clauses [5].

Practical restart policies, however, do not guarantee that new decisions made after a restart will be better than those made in the previous probing. One must therefore also take into account the overhead of too frequent restarts (e.g., a restart after each conflict), which arises from the erasing and redoing of assignments that may not have been necessary.

Completeness Using 1-UIP learning and under the assumption that all clauses are kept, repeated probing as given in Algorithm 1 is complete in that it will always terminate with an answer given sufficient resources.[3] The key reason is

[2] To our knowledge whether this type of backtracking can benefit solvers for the more general constraint satisfaction problems is an interesting open question.

[3] Zhang and Malik [17] proved the completeness of a restricted version of Algorithm 1 where the solver always backtracks to the assertion level, in which case learned clauses need not be kept to ensure completeness.

that each clause learned by 1-UIP is guaranteed to be distinct from all existing clauses. In fact a stronger statement holds: Each of these learned clauses is guaranteed to be subsumed by no existing clause [16]. Since there is only a finite number of distinct clauses over a given number of variables, and each iteration of Algorithm 1 produces a distinct clause, the loop must eventually terminate, on both satisfiable and unsatisfiable instances.

Clause Deletion When employed, clause deletion adds another dimension to what determines the sequences of assignments and conflicts, as it affects both the unit propagation and the learning process. As a direct result, different clauses will be learned and different decisions made. This means that clause deletion can be a double-edged knife: The reduced memory requirement and unit propagation cost must be balanced against the potentially larger number of probings required. In practice, solvers that employ clause deletion use heuristics to estimate the usefulness of clauses and decide which clauses should be deleted at what time.

The completeness of the algorithm can also be impacted by clause deletion. Some solvers intentionally ignore this issue as it does not seem to matter in most practical cases [4]. Other solvers provide for increasingly long periods between deletions, or delete clauses in such a way that the total number of clauses monotonically increases, thus ensuring completeness [10].

Concluding Remarks In this section we have presented a comprehensive and unifying view of modern clause learning SAT algorithms based on formulating them as the process of repeated probing instead of DPLL search. This view exposes the true semantics of these SAT algorithms as pure resolution procedures, rather than DPLL searches enhanced with pruning. With this understanding, the various components of a clause learning SAT solver, including the decision heuristic, backtracking scheme, restart policy, and implementation of unit propagation, are all unified behind a single, common purpose—that of guiding the resolution process, which provides new insights into the roles played by and the effectiveness of the respective components.

3 A Simple SAT Solver

Tinisat is directly implemented in the framework of repeated probing (Algorithm 1). Pseudocode for its top-level procedure can be found in [5]. The final version (with the new decision heuristic described below) has about 550 lines of code, which breaks down as follows: CNF parsing (92 lines), clause pool maintenance (83 lines), unit propagation (53 lines), clause learning (55 lines), decision heuristic (67 lines), main loop (17 lines), miscellaneous (183 lines).

Restart Policy Given the set of experiments reported in [5] where different restart policies are compared, we decided to use Luby's policy [7] (where a "unit run" is 512 conflicts; see [5] for details), and devote our first set of experiments to the discovery of a more robust decision heuristic.

Decision Heuristic Our quest for a good decision heuristic was motivated by the comparative performance of MiniSat and Berkmin described in the beginning of the paper: MiniSat drastically outperforms BerkMin on the IBM benchmarks, but equally drastically underperforms it on the Velev benchmarks.

Upon studying the documentations of the two solvers, we found that one major difference between them was the decision heuristic: BerkMin favors variables from recent conflict clauses, while MiniSat selects variables based on their scores only. To verify if this was indeed a major reason for the drastic difference of solver performance between benchmark groups, we did the following experiment: We implemented a BerkMin-type heuristic in TINISAT, ran the solver on the same IBM and Velev benchmarks, and compared its performance with that of MiniSat. We used two versions of MiniSat for this purpose, MiniSat 2.0 and MiniSat 1.14, as the former employs preprocessing while the latter and TINISAT don't. This helps ensure that what we observe is not just due to whether preprocessing is used.

In the results, we saw a shift of advantage similar to the case of BerkMin vs. MiniSat: TINISAT with the BerkMin-type heuristic solved significantly fewer IBM instances, but significantly more Velev instances, than both versions of MiniSat. We then changed the decision heuristic of TINISAT to the MiniSat type, so that it simply selected a variable of the highest score, and started to repeat our experiment. It quickly became evident that TINISAT was now doing very poorly on the Velev benchmarks.

The conclusion we drew from these experiments was that a new decision heuristic was needed if the solver was to achieve a more robust performance across different types of problems. After an extensive set of further experiments, we found that the BerkMin type heuristic can be more robust if coupled with a type of phase selection heuristic suggested in [12, 11].

The overall heuristic now used in TINISAT is as follows. For each literal a score is kept that is initially the number of its occurrences in the original clauses. On learning a clause, the score of every literal is incremented by 1 for each of its occurrences in clauses involved in resolution. The scores of all literals are halved once every 128 conflicts. When a decision is called for, we pick a free variable with the highest score (sum of two literal scores) from the most recently learned clause that has not been satisfied; if no such clause exists (at most 256 clauses are searched for this purpose) we pick any free variable with the highest score.

The variable is then set to a value as follows: Each variable has a field called *phase*, initially set to the value with the higher score. On backtracking, every variable whose assignment is erased has its *phase* set to that assignment, except for variables set in the latest decision level. When chosen for decision, a variable is set to its *phase*. However, this heuristic is bypassed if the two values differ in score by more than 32, in which case the value with the higher score is used. The intuition behind this phase selection heuristic is that the assignments made prior to the last decision level did not lead to any conflict and may have satisfied some subsets of the CNF, and hence repeating those same assignments may tend to avoid solving some subproblems multiple times [12, 11].

Table 1. Performance of TINISAT.

Velev benchmarks (251)					
Time	TINISAT	TINISATELite	BM	MS 1.14	MS 2.0
10min	124	131	99	95	92
15min	136	146	115	98	93
20min	153	152	121	98	93
30min	164	163	139	106	97
60min	183	182	161	116	102
IBM benchmarks (311)					
Time	TINISAT	TINISATELite	BM	MS 1.14	MS 2.0
10min	238	248	172	238	269
15min	244	257	179	253	278
20min	253	267	183	257	290
30min	262	276	192	268	296
60min	279	289	201	280	304
SAT Race Q1 mixed benchmarks (50)					
Time	TINISAT	TINISATELite	BM	MS 1.14	MS 2.0
10min	40	45	37	38	40
15min	44	47	41	39	43
20min	46	48	42	41	43
30min	48	49	43	42	44
60min	49	49	47	42	44
SAT Race Q2 mixed benchmarks (50)					
Time	TINISAT	TINISATELite	BM	MS 1.14	MS 2.0
10min	31	36	28	32	32
15min	33	38	31	34	34
20min	35	39	31	38	35
30min	40	45	34	40	36
60min	43	45	37	41	39
SAT Race final mixed benchmarks (100)					
Time	TINISAT	TINISATELite	BM	MS 1.14	MS 2.0
10min	49	59	38	53	59
15min	58	70	43	58	68
20min	61	74	46	66	71
30min	70	81	49	72	71
60min	77	87	63	76	74

4 Final Experiments

We present here an extensive set of experiments conducted to evaluate the performance of TINISAT against BerkMin, MiniSat 1.14, and MiniSat 2.0. An additional solver called TINISATELite, created by coupling TINISAT with the SATELITE preprocessor [3], was also used. All experiments were run on a cluster of 16 AMD X2 Dual Core Processors running at 2GHz with 4GB of RAM under Linux, except those with BerkMin and MiniSat 1.14 on Velev, which were run on AMD Athlon 64 processors at 2GHz with 2GB of RAM (prior to a hardware upgrade). A one-hour time limit applied to all runs of all solvers.

Five benchmark groups have been used: 251 instances from Miroslav Velev, 311 instances from IBM Formal Verification [14], and the three groups of 50, 50, and 100 instances used in the first qualification round, second qualification round, and final round, respectively, of SAT Race 2006 [13]. The results are summarized in Table 1 (detailed results available at http://rsise.anu.edu.au/~jinbo/tinisat). For each benchmark group and each solver, we report the number of instances solved in 10, 15, 20, 30, and 60 minutes.

A high performance of TINISAT can be observed. For the 60-minute cut-off, for example, both TINISAT and TINISATELite outperform all other solvers in four of the groups. Most interestingly, the high degree of sensitivity to problem type exhibited by BerkMin and MiniSat on the Velev and IBM benchmarks appears to have been remedied in TINISAT, and a significant improvement in overall performance has been achieved: Out of the 762 instances, TINISAT and TINISATELite solves 68 and 89 more instances, respectively, than MiniSat 2.0.

5 Conclusion and Future Work

We have shown that by focusing on the central aspects of clause learning, a simple solver can significantly outperform the best existing solvers with more sophisticated implementations. Our successful quest for better decision heuristics also provides evidence that there is still considerable room for improvement even within the confines of the current clause learning framework. Some concrete opportunities for further improvement we see include more effective clause deletion policies, more flexible backtracking, dynamic restart policies, and hybridization of decision heuristics based on variable activity (as in typical clause learning solvers) and unit propagation lookahead (as in Satz [6] and its variants, which defeat clause learning solvers on many random and handmade problems).

References

1. Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *IJCAI*, pages 1194–1201, 2003.
2. Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *JAIR*, 22:319–351, 2004.
3. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.
4. Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *DATE*, pages 142–149, 2002.
5. Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.
6. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI*, pages 366–371, 1997.
7. Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.
8. Inês Lynce and João P. Marques Silva. Complete unrestricted backtracking algorithms for satisfiability. In *SAT*, 2002.
9. Joao Marques-Silva and Karem Sakallah. GRASP—A new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.
10. Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.
11. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.
12. Thammanit Pipatsrisawat and Adnan Darwiche. SAT solver description: Rsat. In *SAT-Race*, 2006.
13. Carsten Sinz. SAT Race. http://fmv.jku.at/sat-race-2006, 2006.
14. Emmanuel Zarpas. Benchmarking SAT solvers for bounded model checking. In *SAT*, pages 340–354, 2005.
15. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.
16. Lintao Zhang. On subsumption removal and on-the-fly CNF simplification. In *SAT*, pages 482–489, 2005.
17. Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.