On Implementing More Efficient SAT Data Structures

Inês Lynce and João Marques-Silva

IST/INESC-ID/CEL, Technical University of Lisbon, Portugal
{ines,jpms}@sat.inesc.pt

Abstract. The dramatic improvements made in recent years to state-of-the-art SAT solvers have in part been motivated by new very effective data structures. This paper proposes to further improve existing data structures, namely the watched literals data structure. Experimental results, obtained on representative problem instances, illustrate the effectiveness of the proposed optimizations, and suggest further directions for additional improvements.

1 Introduction

SAT solvers have seen dramatic improvements in recent years [2, 4, 5, 7–9], being currently capable of tackling large and hard real-world problem instances, and being actively applied in industrial settings. Nevertheless, the improvements in SAT solvers also motivated users of SAT technology to devise more complex applications, that yield problem instances of ever increasing size and hardness. As a result, applied research in SAT solver technology is currently a very active field, with new ideas and promises being proposed at a very fast pace.

One of the key areas of improvement in SAT technology has been the development of new data structures. These data structures, exemplified by the head and tail literals in SATO [9] and by the watched literals in Chaff [8], are characterized by being lazy, reducing the overhead of applying boolean constraint propagation to formulas with very large clauses (as for example the ones that often result from recording clauses during conflict analysis).

This paper proposes improvements to the watched literals data structure, which are motivated by earlier work on solvers for the Binate Covering Problem [3]. The key idea is to filter from the list of literals in the clause the literals that have been assigned at lower decision levels.

The paper is organized as follows. We start in Section 2 by reviewing previous work on efficient data structures for backtrack search SAT solvers. Afterwards, in Section 3 we propose improvements to the watched literals data structure. Next, some preliminary experimental results are presented and analyzed. Finally, the paper concludes in Section 5.

2 Overview of SAT Data Structures

Recent state-of-the-art SAT solvers are characterized by using very efficient implementations, intended to reduce the CPU time required per each node in the search tree. As a result, the often used standard implementations (i.e. *counter-based*), have been replaced by the *lazy* implementations, characterized by saving memory and requiring less computational effort.

2.1 Counter-Based Data Structures

Most backtrack search SAT algorithms represent clauses as lists of literals, and associate with each variable x a list of the clauses that contain a literal in x. In addition, literal counters are associated with each clause, to keep track of unsatisfied, satisfied and unit clauses. These literal counters indicate how many literals are unsatisfied, satisfied and, indirectly, how many are still unassigned. A clause is unsatisfied if the unsatisfied literal counter equals the number of literals; it is satisfied if the counter of satisfied literals is greater than one; finally, it is unit if the unsatisfied literal counter equals the number of literals minus one, which means there is still one unassigned literal. When a clause is declared unit, the list of literals is traversed to identify which literal needs to be assigned. Examples of a SAT solvers that utilize a counter-based implementation include GRASP [7], relsat [2] and satz [5].

2.2 Lazy Data Structures

Lazy implementations are characterized by each variable keeping a reduced set of clauses' references, for each of which the variable can effectively be used for declaring the clause as unit or unsatisfied. These data structures are based on the observation that if a clause has more than one unassigned literal, it can be neither a unit nor an unsatisfied clause. As a result, instead of keeping counters for each clause, each variable only keeps a list of references to *some* of its clause literals. Since such literals are the *last* to be assigned value 0, unit and unsatisfied clauses are detected by examining the references in the clause. Examples of efficient lazy data structures include the head/tail lists used in SATO [9] and the watched literals used in Chaff [8]. In Chaff, for each clause there are solely two references to literals, which are said to be *watched*. Moreover, watched literals do not have to be updated in the backtrack step.

3 Improving Data Structures

In this section we describe the new techniques applied for obtaining more efficient data structures. All these techniques are based on lazy data structures (*watched literals*, WL), although the implementation for small clauses is a kind of *back to basics*. Indeed, the operations on small clauses take constant time, as well as operations on counters. In addition, pointers on watched literals are now fixed, and consequently literals to be watched have to be swapped.

3.1 Handling Small Clauses: Back to Basics

As a simple optimization to lazy data structures, we propose the special handling of the clauses that are more common in problem instances: binary and ternary clauses. In our implementation, we decided to have dedicated structures for binary and ternary clauses, even though watched literals are also used in these clauses. Nevertheless, both binary and ternary clauses can be identified as unit, satisfied or unsatisfied in constant time. Since the vast majority of the initial number of clauses for most real-world problem instances are either binary or ternary, the average CPU time required to handle each clause may be noticeably reduced. In this situation, more sophisticated data structures are solely applied to large clauses and to clauses recorded during the search process.

3.2 Literal Displacement

This improvement is inspired in the data structures of the Scherzo [3] solver for the Binate Covering Problem. In this solver, assigned literals are displaced whenever possible. We generalize this idea for the watched literals data structure. When searching for an unassigned literal, each literal assigned at a decision level d is displaced from the list of literals to a table of lists of (assigned) literals, indexed by decision level. Only when the search process backtracks to decision level d is this assigned literal returned to its clause. The advantages of this solution are clear. By displacing assigned literals the subsequent search for unassigned literals becomes faster.

Figure 1 illustrates the operation of new lazy data structures. Literal displacement clauses are next to WL clauses. Observe that WL clauses have positioned watched literals, as proposed by Chaff [8]. In practise, literal swapping is applyed, in order to guaranteeing the first two positioned literals to be the watched literals.

With literal displacement, the idea is to have a double representation of the clause: the *static literals* and the *dynamic literals*. Static literals have no pointers and their contents correspond to the original clause, i.e., with all the literals, which are never swapped. Dynamic literals have a dynamic size and also the two first literals being (positioned) watched. Dynamic literals are discarded whenever a non-watched unsatisfied literal is found when searching for unassigned non-watched literals. In the given example, three literals are discarded when looking for an unassigned non-watched literal. Observe that only one of these literals is returned to the clause when the search backtracks, since the others were assigned in a previous decision level.

3.3 Combining Different Data Structures

As was done in the past with other data structures, the most promising approach to literal displacement is to use it jointly with the default watched literal data structure and the special handling of small clauses. Hence, for small enough clauses a dedicated data structure is used. For larger clauses, the default watched

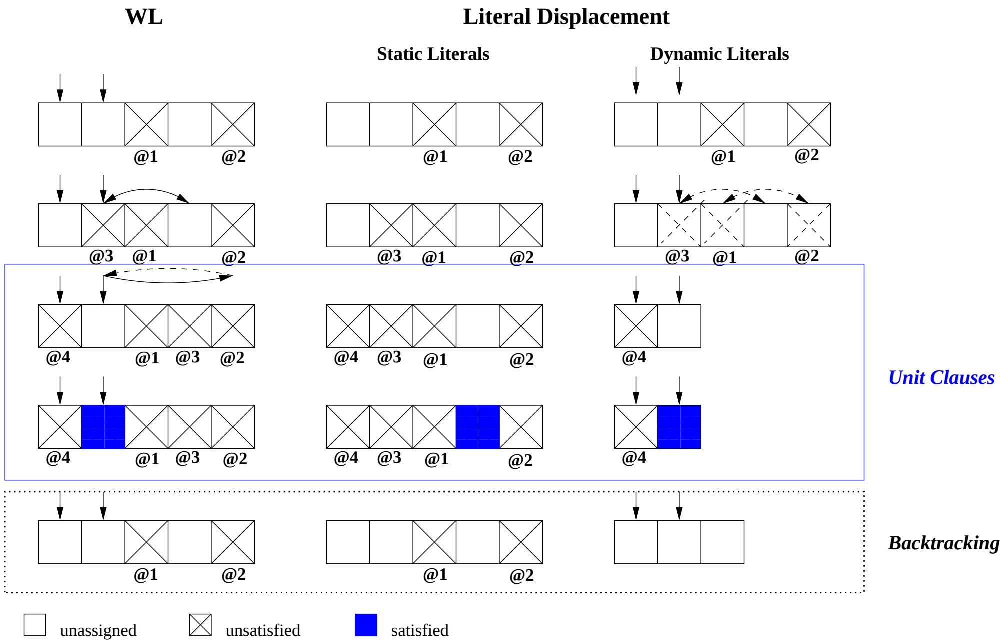

Fig. 1. Operation of new lazy data structures

literal data structure is used. Finally, for the even larger clauses, the watched literals data structure with literal displacement is used.

3.4 Controlled Literal Displacement

The utilization of literal displacement can in some cases have a negative performance impact. When most backtracking is chronological, displacing literals associated with neighbor decision levels can introduce a significant and unnecessary overhead. As a result, we propose a mechanism to control when literal displacement can actually take place. Basically we specify a decision level threshold, and only displace literals assigned at decision levels lower than the current decision level minus the decision level threshold. Hence, when most backtracking is chronological, literals assigned at neighbor decision levels are not displaced, since they are likely to become unassigned in the short term.

4 Experimental Results

This section evaluates the different SAT data structures described in the paper. For the experimental evaluation, we used JQuest2, a new release of JQuest [6].

JQuest2 is a Java SAT solver [1] that implements the different data structures, as well as the most competitive SAT strategies: clause recording with non-chronological backtracking [7] and opportunistic clause deletion, VSIDS heuristic [8] and search restarts [1].

With Table 1 we compare Chaff with JQuest2 (default configuration) performance, including also the ratio between both. This comparison was performed on *hard real-world* instances selected from different classes of instances [2]. For the results shown a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used. The Java Virtual Machine used was SUN's HotSpot JVM for JDK1.4. The CPU time was limited to 1500 seconds. For each aborted instance (X), 1500s were added to the total time. Observe that JQuest2 tends to be less than a factor of 2 slower than Chaff. Hence, JQuest2 performance is very very close to Chaff's, and therefore is certainly a useful platform for prototyping new data structures and new algorithms.

Table 1. Comparison Chaff/JQuest2.

Class of Instances	#	`JQuest2`		`Chaff`		`Ratio`
		Time	X	Time	X	
sss-sat-1.0	100	929.73	0	301.09	0	3.09
vliw-sat-1.0	100	14050.59	0	6271.24	0	2.24
vliw-sat-1.1	100	12648.07	0	6588.11	0	1.42
fvp-unsat-1.0	4	1071.16	0	731.12	0	1.47
bounded model checking	34	8954.27	0	3383.24	0	2.64
equivalence checking	25	4953.91	2	3636.46	2	1.36
beijing	16	3453.96	2	3150.05	2	1.10

Table 2 compares the obtained results for watched literals – `WL`, literal displacement – `Lit Displ` (see Section 3.2) and combining both data structures – `Both` (see Section 3.3). For all the given results, dedicated data structures were used for binary and ternary clauses. Moreover, and for the `Both` results, clause having less than 40 literals were implemented as positioned watched literals, and larger clauses were implemented as clauses with literal displacement. As a final remark, observe that the given results refer to the *time per decision* (in miliseconds). Since different data structures can yield slightly different algorithms, the most fair comparision seems to be based on the values obtained for the time per decision.

[1] Even though Java yields a necessarily slower implementation, it is also plain that it allows fast prototyping of new algorithms. Moreover, well-devised Java implementations can be used as the blueprint for faster C/C++ implementations.

[2] All the instances are available from `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/` (SATLIB web site), with the exception of the *superscalar processor verification* instances.

Table 2. Results on benchmark examples.

Family	Instance	SAT?	WL	Lit Displ	Both
Bounded Model Checking	barrel7	No	2.71	2.69	2.53
	queueinvar4	No	2.57	2.54	2.53
	longmult5	No	2.16	2.23	2.17
Des-Encryption	cnf-r2-b3-k1.1	Yes	8.43	8.55	8.41
	cnf-r2-b3-k1.2	Yes	0.74	0.75	0.75
Equivalence Checking	c2670_bug	Yes	0.36	0.38	0.35
	c2670	No	0.90	0.94	0.80
Superscalar Processor Verification	2dlx_cc_mc_ex_bp_f2_bug005	Yes	1.10	0.94	0.82
	2dlx_cc_mc_ex_bp_f2_bug009	Yes	0.91	0.95	0.86
	2dlx_cc_mc_ex_bp_f2_bug013	Yes	1.21	0.95	0.86

The presented results show that improvements can be obtained for combining both data structures. Moreover, the two data structures when used isolated seem to have similar results. We believe that advantages and disadvantages can be found in both, depending on the search performed, as well as on the size of original and recorded clauses.

5 Conclusions

This paper describes improvements to existing data structures, namely the watched literals data structure. We describe new data structures based on the key idea of filtering from the list of literals in the clause the literals that have been assigned at lower decision levels. Moreover, we propose to combine different data structures based on the clauses size. Alternatively, other criteria should be elected to decide on which data structure to use.

Experimental results, obtained on a variety of problem instances, compare the effectiveness of the proposed optimizations. Moreover, the given results suggest further directions for additional improvements.

References

1. L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In R. Dechter, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494. Springer Verlag, September 2000.
2. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, July 1997.
3. O. Coudert. On Solving Covering Problems. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 197–202, June 1996.
4. E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.

5. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 341–355, October 1997.
6. I. Lynce and J. Marques-Silva. Efficient data structures for backtrack search SAT solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, May 2002.
7. J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, November 1996.
8. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.
9. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.