Building State-of-the-Art SAT Solvers

Inês Lynce and **João Marques-Silva**[1]

Abstract. The area of Propositional Satisfiability (SAT) has been the subject of intensive research in recent years, with significant theoretical and practical contributions. From a practical perspective, a large number of very effective SAT solvers have recently been proposed, most of which based on improvements made to the original Davis-Putnam-Logemann-Loveland (DPLL) backtrack search SAT algorithm. The new solvers are capable of solving very large, very hard real-world problem instances, which more traditional SAT solvers are totally incapable of. Despite the significant improvements in state-of-the-art backtrack search SAT solvers, a few relevant questions remain. Is a well-organized and well-implemented DPLL algorithm enough per se, or should the algorithm definitely include additional search techniques? Which search techniques are indeed effective for most problem instances? Which search techniques cooperate effectively and which do not? This paper is a first step towards answering the previous questions. We start by describing the search techniques that have been proposed in recent years for backtrack search SAT solvers. Afterwards, we empirically evaluate the different techniques, using representative real-world problem instances. Finally, and to conclude, we address the problem of organizing effective DPLL-based SAT solvers.

1 Introduction

Propositional Satisfiability (SAT) is a well-known NP-complete problem of theoretical and practical relevance, with application in many fields of Computer Science and Engineering, including Artificial Intelligence (AI) and Electronic Design Automation (EDA). In the past few years, Propositional Satisfiability has been the subject of intensive research, from which very significant improvements have resulted [2, 9, 16, 11, 12, 7]. These improvements range from new search strategies, to new search pruning and reasoning techniques, and to new fast implementations.

State-of-the-art SAT solvers can now very easily solve problem instances that more traditional SAT solvers are known to be totally incapable of. As a result, a thorough understanding of the organization, the strategies, the techniques, and the implementation of state-of-the-art SAT solvers is essential to help focus future SAT research, to help devise effective new ideas for the next generation solvers, able to solve the next generation problem instances, and finally to help developing innovative modeling approaches, more capable of exploiting the organization of state-of-the-art SAT solvers.

1.1 SAT Algorithms

Over the years a large number of algorithms has been proposed for SAT, from the original Davis-Putnam procedure [4], to recent backtrack search algorithms [2, 9, 16, 11, 12, 7], to local search algorithms [14], among many others. Local search algorithms have allowed solving extremely large satisfiable instances of SAT. These algorithms have also been shown to be very effective in randomly generated instances of SAT. On the other hand, several improvements to the backtrack search Davis-Putnam-Logemann-Loveland algorithm [3] have been introduced. These improvements have been shown to be crucial for solving large instances of SAT derived from real-world applications, and in particular for those where local search cannot be applied, i.e. for unsatisfiable instances. Indeed, the most effective algorithms are able to prove that an instance cannot be satisfied if given enough time. We should note that in a large number of significant real-world applications, proving unsatisfiability is most often the objective.

1.2 Objectives

This paper proposes to further investigate the different improvements that have been introduced by recent state-of-the-art SAT solvers. Given a representative set of instances, from different origins, several questions can be posed. Which SAT techniques are effective? How should the techniques be combined? How should a DPLL-based SAT solver be organized to be competitive with other state-of-the-art SAT solvers? This paper represents a first study to answer these questions. We will start by describing the basic, more well-known algorithms, will detail recent, more advanced SAT strategies and techniques, and will address the implementation of fast SAT solvers. In addition, we will highlight current SAT research topics. Throughout the paper, another key underlying objective is to address some often perceived misconceptions in organizing SAT algorithms.

The remainder of the paper is organized as follows. Section 2 introduces the definitions and the experimental setup. Afterwards, we review backtrack search algorithms for SAT. Section 4 analyzes existing SAT implementations and Section 5 describes different search strategies for SAT algorithms. Finally, we point out recent research trends and the paper concludes in Section 7. Within each of the different sections, the proposed approaches are empirically evaluated in a common SAT framework.

2 Preliminaries

This section introduces the notational framework used in the paper. Moreover, we describe the experimental setup that will be used to obtain the different experimental results presented throughout the paper.

2.1 Definitions

Propositional variables are denoted $x_1, \ldots, x_n$, and can be assigned truth values 0 (or F) or 1 (or T). The truth value assigned to a vari-

[1] Technical University of Lisbon, IST/INESC/CEL, R. Alves Redol, 9, 1000-029 Lisboa, Portugal, email: {ines,jpms}@sat.inesc.pt

Table 1. Example Instances

Application Domain	Selected Instance	# Variables	#Clauses	Satisfiable?
Circuit Testing (Dimacs)	bf0432-079	1044	3685	N
	ssa2670-141	4843	2315	N
Inductive Inference(Dimacs)	ii16e1	1245	14766	Y
Parity Learning(Dimacs)	par16-1-c	317	1264	Y
"Flat" Graph Colouring	flat200-39	600	2237	Y
"Morphed" Graph Colouring	sw100-49	500	3100	Y
Blocks World	4blocksb	410	24758	Y
Planning	logistics.c	1027	9507	N
	facts7hh.13.simple	2809	48920	Y
Bounded Model Checking	barrel5	1407	5383	N
	queueinvar16	1168	6496	N
Superscalar Processor Verification	dlx2_aa	490	2804	N
	dlx2_cc_a_bug17	4847	39184	Y
	2dlx_cc_mc_ex_bp_f2_bug005	4824	48233	Y
Data Encryption Standard	cnf-r3-b4-k1.2	939040	35963	Y

able x is denoted by $\nu(x)$. (When clear from context we use $x = \nu_x$, where $\nu_x \in \{0, 1\}$). A variable whose binary value has already been determined is considered to be *assigned*; otherwise it is *unassigned*. A literal l is either a variable x_i or its negation $\neg x_i$. A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. A clause is said to be *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is *unsatisfied* if at least one clause is unsatisfied, which means there is a *conflict*. A *truth assignment* for a formula is a set of assigned variables and their corresponding truth values. An assignment is *complete* if all variables are assigned; otherwise, it is *partial*. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

2.2 Experimental Setup

In order to experimentally evaluate the different approaches, in a controlled experiment that ensures that only specific differences are evaluated, a dedicated SAT solving framework is needed. Besides differing data structures and coding styles, each existing SAT solver implements its own set of search techniques, strategies and heuristics. Hence, a comparison between state-of-the-art SAT solvers hardly guarantees meaningful results.

Consequently, we developed the JQUEST SAT framework, a Java framework of SAT algorithms, that implements a significant number of the most well-known SAT techniques, and that can be used to conduct unbiased experimental evaluations of SAT techniques and algorithms.

In order to perform this comparison using the JQUEST SAT solver, instances were selected from several classes of instances (see Table 1)[2]. In all cases, the problem instances chosen can be solved with several thousand decisions by the most effective solvers, usually taking a few tens of seconds, and thus being significantly hard. For this reason, different algorithms can provide significant variations on the time required for solving a given instance. In addition, we should also observe that the problem instances selected are intended to be *representative*, since each resembles, in terms of hardness for SAT solvers, the typical instance in each class of problem instances.

For the results shown a P-III@866 MHz Linux machine with 1 GByte of physical memory was used. The Java Virtual Machine used was SUN's HotSpot JVM for JDK1.3. With respect to the organization of the algorithm, we decided to randomize the variable selection heuristic, since this is required for applying some strategies (see Section 5), and since randomization usually provides improvements on most if not all algorithms. For the results shown, the number of runs for each instance and for each algorithm was set to 100. Moreover, the obtained results correspond to the median values for all the runs. The CPU time was limited to 3000 seconds.

3 Backtrack Search Algorithms

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland [3]. The backtrack search algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n problem variables. A *decision level* is associated with each variable selection and assignment. The first variable selection corresponds to decision level 1. For each new decision assignment, the decision level is incremented by 1. In addition, the *unit clause rule* [4] is applied. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfied. In this case, the value of the literal and of the associated variable are said to be *implied*. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation.

In chronological backtracking strategies, the search algorithm keeps track of which decision assignments have been toggled. Given a conflict that occurs at decision level d, the algorithm checks whether the corresponding decision variable x has already been toggled. If not, it erases the variable assignments which are implied by the assignment on x, including the assignment on x, and assigns the opposite value to x. In contrast, if the value of x has already been

[2] All the instances are available from Sat-Ex web site (`http://www.lri.fr/~simon/satex/satex.php3`), with the exception of the *superscalar processor verification* instances.

toggled, the search backtracks to level $d-1$. Satz [9] is an example of a modern SAT solver that applies chronological backtracking.

Non-chronological backtracking strategies were originally proposed by Stallman and Sussman [15] in the area of Truth Maintenance Systems (TMS), and further studied by J. Gaschnig [6] and others (see for example [5, 13] in the context of Constraint Satisfaction Problems (CSP)). Non-chronological backtracking strategies attempt to identify the variable assignments causing a conflict and backtrack directly to a point so that at least one of those variable assignments is modified. GRASP [11] and relsat [2] are examples of SAT solvers that successfully implement non-chronological backtracking.

For all SAT algorithms implementing non-chronological backtracking, new clauses are recorded to explain and prevent identified conflicting conditions. Basically, given a set of variable assignments, that is identified as representing a sufficient condition causing an identified conflict, clause recording consists in the creation of a new clause that prevents the same assignments from occurring simultaneously again during the subsequent search. In general, recorded clauses are used for computing the backtracking decision level, which is defined as the highest decision level of all variable assignments of the literals in each newly recorded clause [3].

Table 2. Chronological vs Non-Chronological Backtracking

Instance	CB	NCB
bf0432-079	>3000	3.73
ssa2670-141	>3000	101.69
ii16e1	>3000	0.53
par16-1-c	165.75	1362.53
flat200-39	656.55	1472.53
sw100-49	>3000	17.15
4blocksb	>3000	639.87
logistics.c	>3000	38.96
facts7hh.13.simple	>3000	8.31
barrel5	189.88	635.12
queueinvar16	>3000	22.9
dlx2_aa	>3000	32.35
dlx2_cc_a_bug17	>3000	4.2
2dlx..._bug005	>3000	>3000
cnf-r3-b4-k1.2	>3000	18.69

The results presented in Table 2 compare the required CPU time for solving the chosen example instances with both chronological backtracking (CB) and non-chronological backtracking (NCB). Definitely, the results trend is clear: the use of non-chronological backtracking is crucial for solving a large number of instances from different domains. Nonetheless, the few instances that chronological backtracking is able to solve need more time to be solved when non-chronological backtracking is used. This is due to the fact that non-chronological backtracking does not apply often to these instances, and consequently making an additional effort for analyzing the causes of conflicts does not compensate.

The main conclusion of the previous comparison is that applying non-chronological backtracking is most often crucial in solving real-world instances of SAT.

[3] It would be interesting to distinguish between results obtained with non-chronological backtracking and with clause recording, since these techniques do not necessarily have to be jointly applied. This evaluation is not within the scope of the paper, but a preliminary study can be found in [10].

4 Fast Implementations

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.

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 [11], relsat [2] and satz [9].

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 [16] and the watched literals used in Chaff [12]. 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.

Table 3. Counter-based vs Lazy Implementations

Instance	CBcb	CBwl	NCBcb	NCBwl
bf0432-079	>3000	>3000	3.73	3.11
ssa2670-141	>3000	>3000	101.69	22.91
ii16e1	>3000	>3000	0.53	0.48
par16-1-c	165.75	175.59	1362.53	233.53
flat200-39	656.55	769.66	1472.53	396.95
sw100-49	>3000	>3000	17.15	12.19
4blocksb	>3000	>3000	639.87	248.03
logistics.c	>3000	>3000	38.96	27.39
facts7hh.13.simple	>3000	>3000	8.31	9.09
barrel5	189.88	203.29	635.12	146.74
queueinvar16	>3000	>3000	22.9	13.06
dlx2_aa	>3000	>3000	32.35	11.74
dlx2_cc_a_bug17	>3000	>3000	4.2	3.92
2dlx..._bug005	>3000	>3000	>3000	>3000
cnf-r3-b4-k1.2	>3000	>3000	18.69	16.48

The results presented in Table 3 reveal interesting trends. (We should note that the different implementations (`cb` for counter-based, `wl` for (lazy) watched literals) perform the same search for the same backtracking strategy (CB or NCB)). For CB, the best results are obtained with a `cb` implementation. In contrast, the best results for NCB are obtained with a `wl` implementation. The explanation for this difference is simple. NCB performs clause recording and therefore creates a clause for each identified conflict. Usually, these clauses have a significant number of literals, especially when compared to the original clauses. Hence, `wl` is more efficient for the large (usually recorded) clauses. On the other hand, `cb` is more adequate for the small (usually original) clauses.

Interestingly, this experimental evidence strongly suggests utilizing mixed data structures. For reasonably small clauses (e.g. with a number of literals less than or equal to k) use a `cb` data structure, whereas for larger clauses (e.g. with a number of literals greater than k) use a `wl` data structure.

5 Search Strategies

Search strategies are used to implement different organizations of the search process. The most well-known strategy consists in randomizing the variable selection heuristic used for selecting variables and also the values to assign to them [2].

Although intimately related with randomizing variable selection heuristics, randomization is also a key aspect of search restart strategies [1, 8]. Randomization ensures with high probability that different sub-trees are searched each time the backtrack search algorithm is restarted.

Current state-of-the-art SAT solvers already incorporate some of the above forms of randomization [1, 12]. In these SAT solvers variable selection heuristics are randomized and search restart strategies are utilized.

Table 4. Search Restart Strategies

Instance	CBcb	CBcbRST	NCBwl	NCBwlRST
bf0432-079	>3000	>3000	3.11	3.17
ssa2670-141	>3000	>3000	22.91	23.22
ii16e1	>3000	>3000	0.48	0.5
par16-1-c	165.75	135.55	233.53	198.51
flat200-39	656.55	539.89	396.95	125.95
sw100-49	>3000	>3000	12.19	10.3
4blocksb	>3000	>3000	248.03	323.74
logistics.c	>3000	>3000	27.39	30.99
facts7hh.13.s...	>3000	>3000	9.09	9.91
barrel5	189.88	157.14	146.74	175.74
queueinvar16	>3000	>3000	13.06	13.36
dlx2_aa	>3000	>3000	11.74	12.31
dlx2_cc_a_bug17	>3000	>3000	3.92	3.58
2dlx..._bug005	>3000	>3000	>3000	197.55
cnf-r3-b4-k1.2	>3000	>3000	16.48	16.78

Experimental results on restarts were obtained considering the best implementation for CB and for NCB. Hence, Table 4 applies restarts (RST) to chronological backtracking with a counter-based implementation (CBcb) and to non-chronological backtracking implemented with watched literals (NCBwl). In general, applying restarts improves the results, even though the improvements are not remarkable. However, there is an exception with respect to the 2dlx_cc_mc_ex_bp_f2_bug005 instance, that would not be solved in the allowed CPU time if it was not for the restarts. It should be observed that this instance is representative of a very large number of instances from the formal verification domain, all known to be hard and structured problem instances, for which search restarts are crucial and yield similar improvements.

6 Recent Trends

Regarding the evaluation of state-of-the-art SAT solvers, the experimental results, obtained on representative problem instances, indicate that lazy implementations may be preferable for the next generation SAT solvers.

More recently, and due to the introduction of lazy data structures (and consequently lazy knowledge of clause status), a different kind of variable selection heuristic (referred to as VSIDS, Variable State Independent Decaying Sum) has been proposed [12]. It selects the literal that appears most frequently over all clauses, which means that the metrics only have to be updated when a new recorded clause is created. More than to develop an *accurate* heuristic, the motivation has been to design a *fast* (but dynamically adaptive) heuristic. In fact, one of the key properties of this strategy is the very low overhead, due to being independent of the variable state.

Table 5. Branching Heuristics

Instance	NCBwlRSTslis	NCBwlRSTvsids
bf0432-079	3.17	2.24
ssa2670-141	23.22	1.06
ii16e1	0.5	0.42
par16-1-c	198.51	178.36
flat200-39	125.95	12.96
sw100-49	10.3	2.35
4blocksb	323.74	85.94
logistics.c	30.99	26.8
facts7hh.13.simple	9.91	7.28
barrel5	175.74	41.07
queueinvar16	13.36	16.39
dlx2_aa	12.31	10.12
dlx2_cc_a_bug17	3.58	2.86
2dlx..._bug005	197.55	41.66
cnf-r3-b4-k1.2	16.78	15.39

Table 5 shows the obtained results on using two different heuristics, named SLIS and VSIDS. SLIS (Static Largest Individual Sum) is an heuristic that selects the literal that appears most frequently in original clauses; in this case, the metrics are not dynamically changed during the search. This heuristic has been used in all the previous results, since it can be applied to both standard and lazy implementations [4]. Moreover, VSIDS consists in the heuristic described above. For some of the instances, and for the VSIDS heuristic, significant improvements can be observed.

7 Conclusions & Recommendations

In this paper we described and compared different techniques, implementations and strategies for backtrack search SAT algorithms. The plot from Figure 1 compares the obtained results for each algorithm described throughout the paper, based on the percentage of solved instances per unit of time. It is plain from the plot that we have achieved remarkable improvements, as we evolved from the standard DPLL algorithm (CBcb) to a non-chronological backtracking algorithm, with a lazy implementation, integrating restarts and using the VSIDS heuristic (NCBwlRSTvsids). Observe that each technique has been included in the order it was proposed in the literature.

The obtained experimental results are clear. Given the problem instances considered, which have been selected to be representative

[4] A more elaborated heuristic could have improved the obtained results for CB. Nevertheless, the experimental results of [10] are still far from being competitive with NCB.

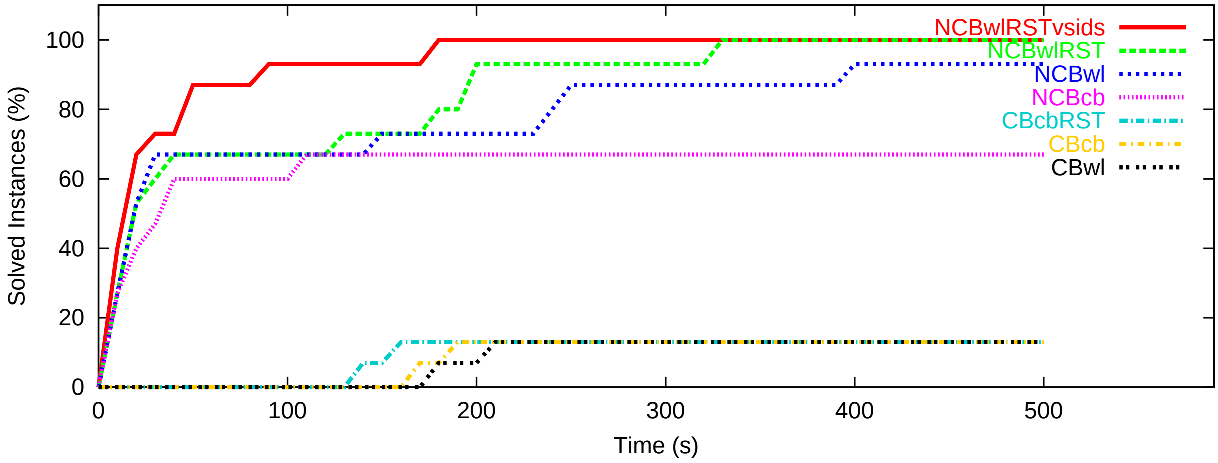

Figure 1. Experimental Results

of each class, from a relevant number of classes of problem instances, we can suggest answers to the questions formulated at the beginning of the paper:

1. A well-organized and well-implemented DPLL algorithm is not enough per se, and should definitely include additional search pruning techniques.
2. Non-chronological backtracking is indeed effective for most problem instances, when compared to chronological backtracking.
3. Search restart strategies can be crucial for solving some of the most hard and structured problem instances.
4. There are techniques that do cooperate effectively: chronological backtracking with counter-based implementations, and non-chronological backtracking with lazy implementations.

Moreover, we believe the experimental results clearly address some often perceived misconceptions in organizing SAT algorithms. To be successfully used in most classes of problem instances a state-of-the-art SAT solver should: (i) apply non-chronological backtracking with clause recording; (ii) utilize the watched literals data structure; (iii) apply a fast instead of an accurate decision heuristic; and (iv) implement the search restart strategy, and selectively use it.

ACKNOWLEDGEMENTS

This work is partially supported by the European research project IST-2001-34607 and by Fundação para a Ciência e Tecnologia under research projects PRAXIS/C/EEI/11249/98 and POSI/34504/CHS/2000.

REFERENCES

[1] L. Baptista and J. P. Marques-Silva, 'Using randomization and learning to solve hard real-world instances of satisfiability', in *International Conference on Principles and Practice of Constraint Programming*, ed., R. Dechter, volume 1894 of *Lecture Notes in Computer Science*, pp. 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*, pp. 203–208, (July 1997).

[3] M. Davis, G. Logemann, and D. Loveland, 'A machine program for theorem-proving', *Communications of the Association for Computing Machinery*, **5**, 394–397, (July 1962).

[4] M. Davis and H. Putnam, 'A computing procedure for quantification theory', *Journal of the Association for Computing Machinery*, **7**, 201–215, (July 1960).

[5] R. Dechter, 'Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition', *Artificial Intelligence*, **41**(3), 273–312, (January 1990).

[6] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, May 1979.

[7] E. Goldberg and Y. Novikov, 'BerkMin: a fast and robust sat-solver', in *Proceedings of the Design and Test in Europe Conference*, (March 2002).

[8] C. P. Gomes, B. Selman, and H. Kautz, 'Boosting combinatorial search through randomization', in *Proceedings of the National Conference on Artificial Intelligence*, (July 1998).

[9] 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*, (October 1997).

[10] I. Lynce and J. P. Marques-Silva, 'The effect of nogood recording in MAC-CBJ SAT algorithms', Technical Report RT/04/2002, INESC, (April 2002).

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

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, 'Engineering an efficient SAT solver', in *Proceedings of the Design Automation Conference*, (June 2001).

[13] Patrick Prosser, 'Hybrid algorithms for the constraint satisfaction problems', *Computational Intelligence*, **9**(3), 268–299, (August 1993).

[14] B. Selman and H. Kautz, 'Domain-independent extensions to GSAT: Solving large structured satisfiability problems', in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 290–295, (August 1993).

[15] R. M. Stallman and G. J. Sussman, 'Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis', *Artificial Intelligence*, **9**, 135–196, (October 1977).

[16] H. Zhang, 'SATO: An efficient propositional prover', in *Proceedings of the International Conference on Automated Deduction*, pp. 272–275, (July 1997).