ON SOLVING OPTIMIZATION PROBLEMS USING BOOLEAN SATISFIABILITY

Fadi A. Aloul

American University of Sharjah
Department of Computer Engineering
P.O. Box 26666, Sharjah, UAE
faloul@ausharjah.edu

ABSTRACT

The last few years have seen significant advances in Boolean satisfiability (SAT) solving. This has lead to the successful deployment of SAT solvers in a wide range of problems in Engineering and Computer Science. In general, most SAT solvers are applied to Boolean decision problems that are expressed in conjunctive normal form (CNF). While this input format is applicable to some engineering tasks, it poses a significant obstacle to others. One of the main advances in SAT is generalizing SAT solvers to handle stronger representation of constraints. Specifically, pseudo-Boolean (PB) constraints which are efficient in representing counting constraints and can replace an exponential number of CNF constraints. Another significant advantage of PB constraints is its ability to express Boolean optimization problems. This allows for new applications that were never handled by SAT solvers before. In this paper, we describe two methods to solve Boolean optimization problems using SAT solvers. Both methods are implemented and evaluated using the SAT solver PBS. We develop an adaptive flow that analyzes a given Boolean optimization problem and selects the solving method that best fits the problem characteristics. Empirical evidence on a variety of benchmarks shows that both methods are competitive. The results also show that SAT-based methods tend to outperform generic integer linear programming (ILP) solvers.

KEYWORDS

Optimization Algorithms, Boolean Satisfiability, CNF, pseudo-Boolean, backtrack search.

1. INTRODUCTION

The Boolean Satisfiability (SAT) problem has been the topic of intensive research over the past few decades. Given a set of Boolean variables and a set of constraints expressed in product-of-sum form (also known as conjunctive normal form (CNF)), the goal is to find a variable assignment that satisfies all constraints or prove that no such assignment exists. Despite the SAT problem's worst-case exponential complexity [11], recent algorithmic advances along with highly efficient solver implementations [7, 17, 19, 21, 28] have enabled the successful deployment of SAT solvers to a wide range of application domains. Such applications include formal verification [8], FPGA routing [22], global routing [5], power leakage estimation [1], timing analysis [24], logic synthesis [20], and sequential equivalence checking [9]. SAT has also been extended to a variety of applications in Artificial Intelligence including other well known NP-complete problems such as graph colorability [23], vertex cover, hamiltonian path, and independent sets [13]. In general, SAT solvers require that the problem be represented in CNF form. While this is applicable to some Engineering tasks, it poses a significant obstacle to many others. In particular to tasks that need to express "counting constraints" which impose a lower or upper bound on a certain number of objects. Expressing such constraints in CNF cannot be efficiently done. Recently, SAT solvers were extended to handle pseudo-Boolean (PB) constraints which can easily represent "counting constraints" [5, 6, 10, 16, 27]. PB constraints are more expressive and can replace an exponential number of CNF constraints [5]. Besides expressing Boolean *decision* problems, a key advantage of using PB constraints is the ability to express Boolean *optimization* problems. These problems were traditionally handled as instances of Integer Linear Programming (ILP). They represent 0-1 ILP problems that call for the minimization or maximization of a linear objective function subject to a set of linear PB constraints.

In this paper, we describe two SAT-based approaches to solve Boolean optimization problems. The algorithms we present can be adapted to any SAT solver. The first approach is based on a *linear* sweep search and the second is based on a *binary* sweep search. Both approaches are implemented on top of the SAT solver PBS [5]. Experiments are conducted on a variety of instances from FPGA routing, N-queens, and graph coloring. The performance of both approaches is compared to the performance of the generic commercial ILP solver CPLEX 7.0. We present experimental evidence showing that (i) SAT solvers can outperform *generic* ILP solvers and (ii) both linear and binary sweep search are competitive. Therefore, we propose a flow that selects the type of search that is best suited to the problem in question. We perform an empirical evaluation comparison of linear vs. binary sweep search and point out that the adaptive flow we propose picks the best configuration in most cases.

The paper is organized as follows. In Section 2 we review recent advances in Boolean satisfiability. Pseudo-Boolean constraints are defined in Section 3. We then describe, in Section 4, the two SAT-based approaches to solve Boolean optimization problems. Both approaches are analyzed and compared against the performance of the generic ILP solver in Section 5. We conclude in Section 6 with a summary of the paper's main contributions.

2. BOOLEAN SATISFIABILITY

The satisfiability problem involves finding an assignment to a set of binary variables that satisfies a given set of constraints. In general, these constraints are expressed in *conjunctive normal form* (CNF) or as is commonly known as product-of-sum form. A CNF formula φ on n binary variables $x_1, \ldots, x_n$ consists of the con-

junction (AND) of m clauses $\omega_1, \ldots, \omega_m$ each of which consists of the disjunction (OR) of k literals. A literal l is an occurrence of a Boolean variable or its complement. We will refer to a CNF formula as a clause database.

A variable x is said to be *assigned* when its logical value is set to 0 or 1 and *unassigned* otherwise. A literal l is a *true* (*false*) literal if it evaluates to 1 (0) under the current assignment to its associated variable, and a *free literal* if its associated variable is *unassigned*. A clause is said to be *satisfied* if at least one of its literals is true, *unsatisfied* if all of its literals are set to false, *unit* if all but a single literal are set to false, and *unresolved* otherwise. A formula is said to be satisfied if all its clauses are satisfied, and unsatisfied if at least one of its clauses is unsatisfied.

As an example, a CNF instance $f = (a \vee b)(\bar{b} \vee c)$ consists of 3 variables, 2 clauses, and 4 literals. The assignment $\{a = 0, b = 1, c = 0\}$ leads to a conflict, whereas the assignment $\{a = 0, b = 1, c = 1\}$ satisfies f.

Most modern SAT solvers [7, 17, 19, 21, 28] are based on the original Davis-Putnam backtrack search algorithm [14]. The algorithm performs a search process that traverses the space of 2^n variable assignments until a satisfying assignment is found (the formula is satisfiable), or all combinations have been exhausted (the formula is unsatisfiable). It maintains a *decision tree* to keep track of variable assignments and can be viewed as consisting of three main engines: *Decision, Deduction, Diagnosis* engines.

Originally, all variables are unassigned. The algorithm begins by choosing a decision assignment to an unassigned variable. After each decision, the deduction engine determines the implications of the assignment on other variables. This is obtained by forcing the assignment of the variable representing an unassigned literal in an unresolved clause, whose all other literals are assigned to 0, to satisfy the clause. This is referred to as the *unit clause* rule and the repeated application of the unit clause rule over the given clause database is known as Boolean constraint propagation (BCP). If no conflict is detected, the algorithm makes a new decision on a new unassigned variable. Otherwise, the diagnosis engine backtracks by unassigning one or more recently assigned variables and the search continues in another area of the search space.

Several powerful methods have been proposed to expedite the backtrack search algorithm. These methods have focused on improving the DLL engines or the data structure used to represent the SAT instance. We review some of the best methods next.

2.1 Conflict Diagnosis

One of the best methods is known as the conflict diagnosis procedure [19] and has been implemented in almost all SAT solvers. Whenever a conflict is detected, the procedure analyzes the variable assignments that cause one or more clauses to become unsatisfied. Such analysis can identify a small subset of variables whose current assignments can be blamed for the conflict. These assignments are turned into a *conflict-induced clause* and augmented with the clause database to avoid regenerating the same conflict in future parts of the search process. In essence, the procedure performs a form of learning from the encountered conflicts. Recognizing that the current conflict is caused by variable assignments from earlier levels in the decision tree enables *non-chronological backtracking*, i.e. backtracking to earlier levels rather than the previous level, potentially pruning large portions of the search space. Significant speedups have been achieved with the addition of conflict-induced clauses, as they tend to effectively prune the search space.

2.2 Random Restarts

Besides conflict diagnosis, recent studies have shown that using random restarts can be very effective in solving hard SAT instances [18, 21]. A SAT solver may often get stuck in a "bad" region of the search space because of the sequence of decision assignments it had made. The restart process helps to extricate the solver from such regions by periodically resetting all decision and implication assignments and randomly selecting a new sequence of decisions, thus insuring that different subtrees are explored each time. Additionally, all learned clauses in the various probes of the search space are kept and help boost the effectiveness of subsequent restarts.

2.3 Decision Heuristics

Intelligent decision heuristics also played an important role in enhancing SAT solvers performance. Several studies have proposed various decision heuristics that can be classified as static [3, 19] or dynamic [19, 21, 28]. For example, the GRASP SAT solver [19] is typically used with the *dynamic largest individual sum* (DLIS) decision heuristic which selects the *literal* that appears in the largest number of unresolved clauses. It also implements the *dynamic largest combined sum* (DLCS) decision heuristic which selects the *variable* that appears in the largest number of unresolved clauses. A recent heuristic that has been found to be particularly effective in a variety of problems is the *variable state independent decaying sum* (VSIDS) heuristic introduced in Chaff [21]. This heuristic maintains two counters for every variable that are incremented if a positive (respectively negative) literal is identified in a new conflict-induced clause. The variable with the highest counter is selected for the next decision. Counters are also periodically divided by a constant to emphasize variables identified in recent conflicts.

2.4 Efficient Boolean Constraint Propagation

SAT solvers typically spend most of their times in the BCP procedure [21]. In a conventional implementation of BCP, an assignment to a variable triggers a traversal of all clauses that contain literals of that variable to check whether they have become unit or unsatisfied. In other words, an implication step requires time bounded by the number of existing literals of the assigned variable. This overhead can be significantly reduced by adopting a form of "lazy" evaluation that avoids unnecessary traversals of the clause database. In 2001, Moskewicz et al. proposed the concept of *watch literals* that keeps track of any two unassigned literals per clause [21]. The basic idea is that a clause can never get unsatisfied or lead to an implication as long as the two watched literals are unassigned. Unlike the conventional approach, an assignment to a variable triggers a traversal to all clauses that contain *watched* literals of that variable. Empirically, such BCP optimizations show great improvements over conventional BCP implementations, especially for problem instances containing large numbers of large clauses. For example, in a SAT instance consisting of n k-literal clauses, this enhancement reduces BCP's overhead from kn to $2n$, which is substantial for typical instances with $k \gg 2$.

3. PSEUDO BOOLEAN CONSTRAINTS

Boolean Satisfiability problems can also include pseudo-Boolean (PB) expressions, which are expressions of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b \quad (1)$$

where $a_i, b \in Z^+$ and x_i are literals of Boolean variables. Note that any CNF clause can be viewed as a PB constraint, e.g. clause $(a \vee b \vee c)$ is equivalent to $(a + b + c \geq 1)$. Using the relations:

- $\overline{x_i} = (1 - x_i)$
- $(Ax = b) \Leftrightarrow (Ax \leq b)(Ax \geq b)$
- $(Ax \geq b) \Leftrightarrow (-Ax \leq -b)$

any arbitrary PB constraint can be converted to the *normalized* form of (1) consisting of only positive coefficients. This normalization facilitates more efficient algorithms.

Figure 1 shows an example of a scheduling problem expressed using CNF and PB constraints. The problem assumes a company consisting of 3 employees and 3 working shifts per day. At least one employee must be working during each shift. Each employee can work upto 1 shift per day. Variable E_{ij} denotes employee *i* working during shift *j*. Clearly, the PB encoding, consisting of 6 PB constraints and 18 literals, is more efficient than the CNF encoding, consisting of 12 clauses and 27 literals.

Constraint	CNF Encoding	PB Encoding
#1	$(E_{11} \vee E_{21} \vee E_{31})$ $(E_{12} \vee E_{22} \vee E_{32})$ $(E_{13} \vee E_{23} \vee E_{33})$	$(E_{11} + E_{21} + E_{31} \geq 1)$ $(E_{12} + E_{22} + E_{32} \geq 1)$ $(E_{13} + E_{23} + E_{33} \geq 1)$
#2	$(\overline{E_{11}} \vee \overline{E_{12}})(\overline{E_{11}} \vee \overline{E_{13}})$ $(\overline{E_{12}} \vee \overline{E_{13}})(\overline{E_{21}} \vee \overline{E_{22}})$ $(\overline{E_{21}} \vee \overline{E_{23}})(\overline{E_{22}} \vee \overline{E_{23}})$ $(\overline{E_{31}} \vee \overline{E_{32}})(\overline{E_{31}} \vee \overline{E_{33}})$ $(\overline{E_{32}} \vee \overline{E_{33}})$	$(E_{11} + E_{12} + E_{13} \leq 1)$ $(E_{21} + E_{22} + E_{23} \leq 1)$ $(E_{31} + E_{32} + E_{33} \leq 1)$

Figure 1. Two possible encodings of the scheduling problem consisting of 3 employees and 3 working shifts per day. E_{ij} denotes employee i working during shift j. Constraint #1 indicates that at least one employee is working during each shift. Constraint #2 indicates that each employee can work upto 1 shift per day.

4. BOOLEAN OPTIMIZATION PROBLEMS

Besides solving consistency problems, handling PB constraints expands the ability of SAT solvers to solve Boolean optimization problems that were traditionally handled as instances of Integer Linear Programming (ILP) [23, 87]. These so-called 0-1 ILP problems call for the *minimization* or *maximization* of a linear objective function:

$$\sum_{i=1}^{n} a_i\dot{x}_i \quad (2)$$

subject to a set of m linear PB constraints $Ax \leq b$ where $b \in Z^n$, $A \in Z^m \times Z^n$, and $x \in \{0, 1\}^n$. We describe two common SAT-based approaches to solve Boolean optimization problems. Both approaches convert the objective function to a PB constraint with a sliding right-hand-side (RHS) goal and proceed to solve a sequence of SAT instances that differ only in the value of that goal. The first approach is based on a *linear* sweep search and the second approach is based on a *binary* sweep search. Both approaches are described next.

4.1 Linear Sweep Search Algorithm

The approach performs a linear sweep search on the possible values of the objective function, starting from the initial goal, requiring at each step that the computed solution have a better value than the last computed value. To illustrate, assume a *minimization* scenario, denote the sequence of SAT instances by $I_0, I_1, I_2, \ldots$ and let $\hat{g}_i$ be the goal for the *i*th instance. Initially the goal of I_0 is set to be:

$$\hat{g}_0 = \left(\sum_{j=1}^{n} a_j \right) + 1 \quad (3)$$

where n is total number of literals in the objective function. The process proceeds to solve all I_i instances starting with instance I_0. If the *i*th instance is satisfiable, substituting its solution in the objective function constraint should yield a new goal value $g_i \leq \hat{g}_i$. The goal for instance I_{i+1} is now set to $g_i - 1$ and the process is repeated. The goal reached in the last satisfiable instance is returned by the solver as the optimal value of the objective function.

4.2 Binary Sweep Search Algorithm

Another concept that can be used for solving Boolean optimization problems is the binary sweep search algorithm. The method is based on the idea of testing the SAT instance I_i whose goal $\hat{g}_i$ is the center value of all possible optimal values. Two variables, *bestSat* and *bestUns,* are maintained that store the best identified satisfiable value and the best identified unsatisfiable value, respectively. Assuming a *minimization* scenario, the approach solves a sequence of SAT instances $I_0, I_1, I_2, \ldots$. Initially the goal and *bestSat* are set to the value computed in *(3)* and *bestUns* is set to 0. If the instance is satisfiable, its solution is substituted in the objective function constraint yielding a new goal value $g_i \leq \hat{g}_i$. The *bestSat* variable is set to g_i. On the other hand, if the instance is unsatisfiable, the *bestUns* variable is set to $\hat{g}_i$. The goal for the instance I_{i+1} is then set to $((bestSat + bestUns)/2)$ and the process is repeated. The optimal value of the objective function is *bestSat* whenever the *bestUns* value is equal to $bestSat - 1$.

Table 1. Results for optimization instances using the SAT-based 0-1 ILP PBS solver and generic ILP CPLEX solver. PBS was configured to use linear and binary sweep search. Best results for each instance are shown in bold.

Family (Obj - Type)	Instance name	Optimal Soln	Initial Soln	PBS				CPLEX	
				Linear		Binary			
				Time	Best Soln	Time	Best Soln	Time	Best Soln
Graph Coloring (Min)	**anna**	11	20	**24.5**	11	67.8	11	774	11
	david	11	20	**2.85**	11	6.47	11	>1000	n/a
	DSJC125.1	5	20	39.73	5	**17.4**	5	>1000	n/a
	games120	9	20	11.18	9	**3.96**	9	>1000	n/a
	jean	10	20	28.4	10	**14.4**	10	>1000	n/a
	miles250	8	20	1.94	8	**1.83**	8	605	8
	myciel3	4	11	**0.07**	4	0.08	4	0.19	4
	myciel4	5	20	**0.22**	5	0.33	5	20.8	5
	myciel5	6	20	29.5	6	**10.5**	6	>1000	n/a
	queen5_5	5	20	0.22	5	**0.18**	5	0.26	5
	queen6_6	7	20	**0.59**	7	2.65	7	174	7
	queen7_7	7	20	**8.31**	7	9.17	7	475	7
	queen8_12	12	20	**1.08**	12	3.3	12	>1000	n/a
N-Queen (Max)	**NQueens8**	8	8	0.1	8	0.35	8	**0.01**	8
	NQueens9	9	9	0.69	9	2.71	9	**0.04**	9
	NQueens10	10	10	5	10	20.5	10	**0.6**	10
	NQueens11	11	11	69.2	11	228	11	**0.12**	11
FPGA-SAT (Min)	**fpga20_15**	30	30	**0.61**	30	1.46	30	200	30
	fpga25_15	30	30	**1.86**	30	4.07	30	>1000	n/a
	fpga30_15	30	30	**0.99**	30	2.49	30	>1000	n/a
	fpga25_20	40	40	**3.71**	40	13.5	40	>1000	n/a
	fpga30_20	40	40	**34.9**	40	149	40	>1000	n/a
	fpga35_20	40	40	**4.35**	40	14.6	40	>1000	n/a
FPGA-UNS (Max)	**chnl7_8**	406	268	0.43	406	**0.38**	406	25.1	406
	chnl7_9	518	317	**1.68**	518	1.74	518	194	518
	chnl7_10	644	383	8.01	644	**7.2**	644	464	644
	chnl7_11	784	474	40.9	784	**35.7**	784	>1000	n/a
	chnl8_9	592	382	5.35	592	**1.65**	592	92.8	592
	chnl8_10	736	442	**3.47**	736	3.53	736	>1000	n/a
	chnl8_11	896	546	**20.8**	896	32.2	896	>1000	n/a
	chnl8_12	1072	606	211	1072	**156**	1072	>1000	n/a
	chnl9_10	828	506	3.05	828	**2.33**	828	362	828
	chnl9_11	1008	604	11.9	1008	**5.5**	1008	>1000	n/a
	chnl9_12	1206	700	**60.2**	1206	70.4	1206	>1000	n/a
	chnl9_13	1422	850	>1000	1422	**765**	1422	>1000	n/a

5. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the two proposed methods for solving Boolean optimization problems. Our benchmarks include optimization instances from

- Graph coloring [15]: minimize the number of colors used to color each node in an undirected graph such that no two nodes sharing the same edge have the same color. Initial number of colors in all instances was set to 20.
- N-queens [25]: maximize the number of queens that can be placed on an NxN chess board as long as no two queens can attack each other.
- FPGA routing (SAT) [3]: minimize the number of wires used to route a given number of nets.
- FPGA routing (UNS) [3]: maximize the number of routed nets in an unsatisfiable FPGA instance.

In order to speed up the search process, all instances were pre-processed with ShatterPB [2] which augments the SAT instance with symmetry-breaking predicates (SBPs). The work in [2, 3, 4, 12] showed that the use of SBPs can significantly speed up the solution of SAT instances. We use the recent SAT solver PBS [5] (with settings "-D 1") which incorporates modern SAT techniques described in Section 2 and also handles PB constraints. PBS was modified to solve Boolean optimization problems using the linear and binary sweep search. We also compare the performance of PBS against the commercial ILP solver CPLEX version 7.0. All experiments were performed on a 750MHz Sun-Blade 1000 workstation with 2GB RAM running the Solaris operating system. All time-outs are set to 1000 seconds.

Table 1 lists the results of solving 35 instances. For each instance the table lists the instance name and family, its objective type (e.g. *Min* for minimization and *Max* for maximization), the optimal value of the objective function, and the initial solution $\hat{g}_0$ when solving the instance using PBS. The remaining columns show the run times and the best reached objective value of PBS (using the linear and binary sweep search methods) and CPLEX solvers, respectively. We observe the following:

- Except for the N-queens instances, PBS outperforms CPLEX and in some cases with a substantial margin (e.g. FPGA-SAT instances).

- Both the linear and binary sweep search methods are competitive.
- The linear sweep search tends to beat the binary sweep search for instances whose initial solution is close to the optimal objective value. For example, the SAT solver's initial solution for all the FPGA-SAT and N-queens instances is equal to the optimal value of the objective function.
- The binary sweep search was on average faster for instances whose initial solution is far from the optimal solution.

Overall, the above results show that SAT-based methods are in general faster than generic ILP solvers, such as CPLEX, since SAT-based solvers are expected to take advantage of the Boolean nature of the problem. The only exception, in Table 1, was the N-queens set which was solved with CPLEX in a fraction of a second. Given the black-box nature of the CPLEX solver it was hard to justify its exceptional performance on these instances. However, analysis of the constraints in these instances showed that they are highly structured. We conjecture that CPLEX incorporates advanced algorithms that detect and simplify certain structured instances, such as the N-queens instances.

In terms of selecting whether to use a linear or binary sweep search, we propose to use the linear (binary) method whenever the initial solution is close (far) from the *upper* bound of the optimal solution. This poses the question of how to identify the upper bound of the optimal solution for an optimization instance? An obvious answer is to use the information provided with the instance to guess the upper bound. For example, the 8-*queens* instance consists of an $8x8$ grid and is likely to fit upto 8 queens only. The SAT solver's initial solution is 8. The remaining processing time is spent on proving that this is the optimal solution. Another example is the $FPGAi\_j$ which represents an FPGA instance with j nets. According to Table 1, each FPGA-SAT instance needs at least 2 wires per net. Hence, the *FPGA20_15* instance is likely to use at least 30 wires which turns out to be the initial and optimal solution. An alternative solution to identify the upper bound is to use some form of branch-and-bound which gives an estimate of the optimal value of the objective function.

6. CONCLUSIONS

This work is motivated by the observation that SAT solvers can be extended to handle Boolean optimization problems, which is useful in many applications. We describe two methods to solve Boolean optimization problems using SAT solvers. One method is based on a *linear* sweep search and the other is based on a *binary* sweep search. Both methods can be easily adapted to any SAT solver. Empirically, we observe that both approaches are competitive. Therefore, we propose an adaptive flow that picks either the linear or binary search configuration, depending on the instance's characteristics, to achieve the most effective Boolean optimization for a given instance. We also show that SAT-based methods can outperform generic ILP solvers in many cases. Our on-going work deals with identifying new metrics (e.g. branch-and-bound) that can help improve our adaptive flow.

7. REFERENCES

[1] F. A. Aloul, S. M. Hassoun, K. A. Sakallah, D. Blaauw, "Robust SAT-Based Search Algorithm for Leakage Power Reduction," in *Proc. of the International Workshop on Power and Timing Modeling, Optimization and Simulation* (PATMOS), 167-177, 2002.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "ShatterPB: Symmetry-Breaking for Pseudo-Boolean Formulas," in *Proc. of the Asia South Pacific Design Automation Conference* (ASPDAC), 884-887, 2004.

[3] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Efficient Symmetry-Breaking for Boolean Satisfiability," in *Proc. of the International Joint Conference on Artificial Intelligence* (IJCAI), 271-282, 2003.

[4] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetries," in *IEEE Trans. on Computer Aided Design*, 22(9), 1117-1137, 2003.

[5] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ILP versus Specialized 0-1 ILP: An Update," in *Proc. of the International Conference on Computer-Aided Design* (ICCAD), 450-457, 2002.

[6] P. Barth, "A Davis-Putnam based Enumeration Algorithm for Linear Pseudo-Boolean Optimization," *Technical Report MPI-I-95-2-003, Max-Planck-Institut Für Informatik*, 1995.

[7] R. J. Bayardo Jr. and R. C. Schrag, "Using CSP Look-Back Techniques to Solve Real World SAT Instances," in *Proc. of the 14th National Conference on Artificial Intelligence*, 203-208, 1997.

[8] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proc. of the Design Automation Conference* (DAC), 317-320, 1999.

[9] P. Bjesse and K. Claessen, "SAT-based Verification Without State Space Traversal," in *Proc. of Formal Methods in Computer-Aided Design* (FMCAD), 372-389, 2000.

[10] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," in *Proc. of the Design Automation Conferenc*e (DAC), 830-835, 2003.

[11] S. Cook, "The Complexity of Theorem Proving Procedures," in *Proc. of the 3rd Annual ACM Symposium on Theory of Computing*, 151-158, 1971.

[12] J. Crawford, M. Ginsberg, E. Luks and A. Roy, "Symmetry-breaking Predicates for Search Problems," in *Proc. of the International Conference on Principles of Knowledge Representation and Reasoning*, 148-159, 1996.

[13] N. Creignou, S. Kanna, and M. Sudan, "Complexity Classifications of Boolean Constraint Satisfaction Problems," *SIAM Press*, 2001.

[14] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," in *Journal of the ACM*, (5)7, pp. 394-397, 1962.

[15] DIMACS Graph Coloring Instance. *http://mat.gsia.cmu.edu/COLOR/instances.html*

[16] H. Dixon and M. Ginsberg, "Inference Methods for a Pseudo-Boolean Satisfiability Solver," in *Proc. of the National Conference on Artificial Intelligence* (AAAI), 635-640, 2002.

[17] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Proc. of the Design Automation and Test Conference in Europe* (DATE), 142-149, 2002.

[18] C. P. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search Through Randomization," in *Proc. of the National Conference on Artificial Intelligence* (AAAI), 431-437, 1998.

[19] J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," in *IEEE Transactions on Computers,* (48)5, 506-521, 1999.

[20] S. Memik and F. Fallah, "Accelerated Boolean Satisfiability-Based Scheduling of Control/Data Flow Graphs for High-Level Synthesis," in *Proc. of the International Conference on Computer Design* (ICCD), 2002.

[21] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of the Design Automation Conference* (DAC), 530-535, 2001.

[22] G. Nam, F. A. Aloul, K. A. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," in *the Proc. of the International Symposium on Physical Design* (ISPD), 222-227, 2001.

[23] A. Ramani, F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Breaking Instance-Independent Symmetries in Exact Graph Coloring," in *Proc. of the Design, Automation, and Test in Europe Conference* (DATE), 324-329, 2004.

[24] L. Silva, J. Silva, L. Silveira and K. Sakallah, "Timing Analysis Using Propositional Satisfiability," in *IEEE International Conference on Electronics, Circuits and Systems*, 1998.

[25] R. Sosic and J. Gu, "Fast Search Algorithms for the N-Queens Problem," in *IEEE Trans. on Systems, Man, and Cybernetics*. 21(6), 1572-1576, 1991.

[26] J. Walsor, "Solving Linear Pseudo-Boolean Constraint Problems with Local Search," in *Proc. of the National Conference on Artificial Intelligence* (AAAI), 269-274, 1997.

[27] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *Proc. of Design Automation Conference* (DAC), 542-545, 2001.

[28] H. Zhang, "SATO: An Efficient Propositional Prover," in *Intl. Conference on Automated Deduction*, 272-275, 1997.

[29] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *Proc. of the International Conference on Computer Aided Design* (ICCAD), pp. 279-285, 2001.