ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of the Franklin Institute ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of The Franklin Institute

www.elsevier.com/locate/jfranklin

Search techniques for SAT-based Boolean optimization

Fadi A. Aloul*

Department of Computer Engineering, American University of Sharjah, Sharjah, UAE

Received 31 January 2006; accepted 31 January 2006

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 are very competitive with generic integer linear programming (ILP) solvers.

Keywords: Optimization algorithms; Integer linear programming; Backtrack search; Boolean satisfiability

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

*Tel.: +971 6 5152784.
E-mail address: faloul@aus.edu.

0016-0032/$30.00
doi:10.1016/j.jfranklin.2006.01.003

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 [1], recent algorithmic advances along with highly efficient solver implementations [2–6] have enabled the successful deployment of SAT solvers to a wide range of application domains. Such applications include formal verification [7], FPGA routing [8], global routing [9], timing analysis [10], logic synthesis [11], and sequential equivalence checking [12]. SAT has also been extended to a variety of applications in Artificial Intelligence including other well known NP-complete problems such as graph colorability [13], vertex cover, Hamiltonian path, and independent sets [14]. 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" [9,15–19]. PB constraints are more expressive and can replace an exponential number of CNF constraints [9]. 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 techniques to solve Boolean optimization problems. The algorithms we present can be adapted to any SAT solver. The first technique is based on a *linear* sweep search and the second is based on a *binary* sweep search. Both techniques are implemented on top of the SAT-based 0–1 ILP solver PBS [9]. Experiments are conducted on a variety of instances from FPGA routing, N-queens, and graph coloring. The performance of both techniques is compared to the performance of the *generic* commercial ILP solver CPLEX 7.0. We present experimental evidence showing that (i) SAT-based Boolean optimization solvers can outperform generic ILP solvers and (ii) both linear and binary sweep search techniques are competitive. Therefore, we propose a simple flow that analyzes the instance's properties and 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 many 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 techniques to solve Boolean optimization problems. Both techniques 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 conjunction (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, the CNF instance $f(a,b,c) = (a \vee b)(\bar{b} \vee c)$ consists of three variables, two clauses, and four 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 [2–6] are based on the original Davis–Putnam–Logemann–Loveland (DPLL) backtrack search algorithm [20]. 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 DPLL engines or the data structure used to represent the SAT instance. We review some of the best methods next.

2.1. Conflict diagnosis

In general, whenever a SAT solver encounters a conflict it unassigns all variable assignments at the most recent decision level and flips the value of the most recent decision variable. This is typically known as *chronological backtracking*. In 1997, the GRASP SAT solver proposed the use of conflict diagnosis whenever a conflict is detected [4]. The idea is to identify a set of variable assignments that cause one or more clauses to become unsatisfied. These assignments can be used to construct a *conflict-induced clause* that once added to the clause database will prevent regenerating the same conflict in future parts of the search process. This form of clause learning has been shown to significantly prune the search space.

Another advantage of conflict diagnosis is non-chronological backtracking. Rather than backtracking to the previous level (i.e. chronological backtracking), the solver can backtrack directly to the decision variable that led to the conflict. Backtracking to earlier levels can help in potentially pruning large portions of the search space.

2.2. Decision heuristics

Besides conflict diagnosis, recent studies have shown that decision heuristics can be very effective in solving hard SAT instances. Decision heuristics work on selecting a sequence of decision variables that are likely to identify a satisfying assignment faster. Several researchers have proposed intelligent decision heuristics that can be classified as static [21,4] or dynamic [4–6]. We briefly review some of the most common heuristics next. For a comprehensive review of SAT decision heuristics, see [22].

A simple decision heuristic is to *randomly* select the next decision variable from among the unassigned variables. This heuristic is commonly known as RAND. Other heuristics, such as the *maximum occurrences on minimum sized clauses* (MOM) [23], employ greedy algorithms that select the decision variable that satisfies the maximum number of clauses or leads to the maximum number of implications. In 1997, the GRASP SAT [4] solver proposed the use of the *dynamic largest individual sum* (DLIS) decision heuristic which selects the *literal* that appears in the largest number of unresolved clauses. It also proposed the *dynamic largest combined sum* (DLCS) decision heuristic which selects the *variable* that appears in the largest number of unresolved clauses. With so many proposed heuristics, it is difficult to determine which heuristic is the best. Each heuristic performs well on different types of problems.

Recently, the Chaff SAT solver proposed the use of the *variable state independent decaying sum* (VSIDS) heuristic [5]. This has been found to be cheap to maintain and quite effective on a variety of problems. VSIDS keeps a counter for each literal in the clause database. These counters are incremented as new conflict-induced clauses are added. The counters are also periodically divided by a constant to emphasize variables identified in recent conflicts.

2.3. Random restarts

Random restarts have played an important role in enhancing SAT solvers performance [24,5]. A SAT solver often selects a sequence of decision assignments that gets it stuck in a hard region of the search space. Random restarts can extract the solver from such regions by *periodically* halting the search process, resetting all decisions and implications made, and randomly selecting a new sequence of decision variables that will explore a new region in the search space. Although the current search space is abandoned, the SAT solver keeps *all* previously learned clauses. Hence, the SAT solver avoids repeating earlier analysis of the search space.

2.4. Optimized Boolean constraint propagation

In practice, most of the SAT solver's runtime (almost 90%) is spent in the BCP procedure. Therefore, an efficient BCP procedure is crucial to any SAT solver. A simple and intuitive implementation for BCP is to traverse all clauses containing a literal of a variable that got assigned [4]. The clause is checked if it has become unit or unsatisfied. This implication step requires time bounded by the number of existing literals of the assigned variable. In 2001, the authors of the SAT solver Chaff [5] noted that there is no need to traverse a clause with N literals until $N-1$ literals are assigned to 0. They proposed the use of *watched literals* that keep track of any two unassigned literals in each

clause. The idea is that the clause can never be unsatisfied or unit as long as the two watched literals are unassigned. Hence, the clause can only be visited when either of the two watched literals is assigned to 0. Empirical results show that the watched literals method can lead to great improvements over conventional BCP implementations, especially for problem instances containing large numbers of large clauses.

3. Pseudo Boolean constraints

Boolean satisfiability problems can also include PB expressions, which are expressions of the form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leqslant b, \tag{1}$$

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

- $\bar{x}_i = (1 - x_i)$,
- $(Ax = b) \Leftrightarrow (Ax \leqslant b)(Ax \geqslant b)$,
- $(Ax \geqslant b) \Leftrightarrow (-Ax \leqslant -b)$,

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

Table 1 shows an example of a scheduling problem expressed using CNF and PB constraints. The problem assumes a company that manages three employees and offers three working shifts per day. The goal is to identify a valid schedule for one day that satisfies the following two conditions: (i) At least one employee must be working during each shift and (ii) each employee can work up to one shift per day. Variable E_{ij} denotes employee i working during shift j. Clearly, the PB encoding, consisting of six PB constraints and 18 literals, is more efficient than the CNF encoding, consisting of 12 clauses and 27 literals. Hence, significant memory savings can be achieved by using PB instead of CNF encodings.

Table 1
Two possible encodings of the scheduling problem consisting of three employees and three working shifts per day

Constraint	CNF encoding	PB encoding
#1	$(E_{11} \vee E_{21} \vee E_{31})$	$(E_{11} + E_{21} + E_{31} \geqslant 1)$
	$(E_{12} \vee E_{22} \vee E_{32})$	$(E_{12} + E_{22} + E_{32} \geqslant 1)$
	$(E_{13} \vee E_{23} \vee E_{33})$	$(E_{13} + E_{23} + E_{33} \geqslant 1)$
#2	$(\bar{E}_{11} \vee \bar{E}_{12})(\bar{E}_{11} \vee \bar{E}_{13})$	$(E_{11} + E_{12} + E_{13} \leqslant 1)$
	$(\bar{E}_{12} \vee \bar{E}_{13})(\bar{E}_{21} \vee \bar{E}_{22})$	$(E_{21} + E_{22} + E_{23} \leqslant 1)$
	$(\bar{E}_{21} \vee \bar{E}_{23})(\bar{E}_{22} \vee \bar{E}_{23})$	$(E_{31} + E_{32} + E_{33} \geqslant 1)$
	$(\bar{E}_{31} \vee \bar{E}_{32})(\bar{E}_{31} \vee \bar{E}_{33})$	
	$(\bar{E}_{32} \vee \bar{E}_{33})$	

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 up to 1 shift per day.

4. Boolean optimization problems

Besides solving decision problems, handling PB constraints expands the ability of SAT solvers to solve Boolean optimization problems that were traditionally handled as instances of ILP. These so-called 0–1 ILP problems call for the *minimization* or *maximization* of a linear objective function:

$$\sum_{i=1}^{n} a_i x_i \tag{2}$$

subject to a set of m linear PB constraints $Ax \leqslant b$ where $b \in Z^n$, $A \in Z^m \times Z^n$, and $x \in \{0,1\}^n$. We describe two common SAT-based techniques to solve Boolean optimization problems. Both techniques 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 technique is based on a *linear* sweep search and the second technique is based on a *binary* sweep search. Both techniques are described next.

4.1. Linear sweep search algorithm

The technique 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 g_i be the goal for the ith instance. Initially the goal of I_0 is set to be

$$g_0 = \left(\sum_{j=1}^{n} a_j\right) + 1, \tag{3}$$

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 ith instance is satisfiable, a new goal value $\tilde{g}_i$ is identified by substituting the instance's solution in the objective function constraint. Note that the resulting goal $\tilde{g}_i \leqslant g_i$. The goal for instance $I_i + 1$ is now set to $\tilde{g}_i - 1$ and the instance is passed to the SAT solver. The process is repeated until the SAT solver proves unsatisfiability. The goal reached in the last satisfiable instance is returned by the SAT solver as the *optimal* value of the objective function. Note that most SAT-based 0–1 ILP solvers use the linear sweep search method [9,15–19].

4.2. Binary sweep search algorithm

Another concept that can be used for solving Boolean optimization problems is the binary sweep search algorithm. The technique is based on the idea of testing the SAT instance I_i whose goal g_i is the mean value of all possible optimal values of the objective function. 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 Eq. (3) and *bestUns* is set to -1. If the instance is satisfiable, its solution is substituted in the objective function constraint yielding a new goal value $\tilde{g}_i \leqslant g_i$. The *bestSat* variable is set to $\tilde{g}_i$. On the other

hand, if the instance is unsatisfiable, the *bestUns* variable is set to 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 (i) *bestSat* is equal to 0 or (ii) the difference between *bestSat* and *bestUns* is equal to 1.

5. Experimental evaluation

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

Graph coloring [25]: 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 is set to 20.

N-queens [26]: maximize the number of queens that can be placed on an $N \times N$ chess board as long as no two queens can attack each other.

FPGA routing (SAT) [27]: minimize the number of wires used to route a given number of nets.

FPGA routing (UNS) [27]: maximize the number of routable nets in an unsatisfiable FPGA instance.

In order to speed up the search process, all instances were *pre-processed* with ShatterPB [28] which augments the SAT instance with symmetry-breaking predicates (SBPs). The work in [27–30] show that the use of SBPs can significantly prune the search space and speed up the SAT search process. Pre-processing time was negligible in most cases. We use the recent SAT solver PBS [9] 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 generic ILP solver CPLEX version 7.0. All experiments are performed on a 750 MHz Sun-Blade 1000 workstation with 2 GB RAM running the Solaris operating system. All time-outs are set to 1000 s.

Table 2 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),[1] the maximum theoretical value of the objective function (i.e. $g_0 - 1$ or initial upper bound), the initial solution (i.e. $\tilde{g}_0$) when solving the instance using PBS, and the optimal value of the objective function. 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 instances).
- Even when the solver times-out, the obtained value of the objective function is substantially smaller than the maximum theoretical value of the objective function.
- Both the linear and binary sweep search techniques are competitive.
- The linear sweep search tends to beat the binary sweep search for instances whose initial solution, $\tilde{g}_0$, is relatively close to the optimal solution. For example, the SAT solver's

[1] All *maximization* instances were converted to *minimization* instances using the process described in Section 3. For example, the following maximization objective $(x + y + 2z \geqslant 0)$ is expressed as the minimization objective $(\bar{x} + \bar{y} + 2\bar{z} \leqslant 4)$.

Table 2
Results for optimization instances (after pre-processing them with ShatterPB) using the SAT-based 0–1 ILP PBS solver and generic ILP CPLEX solver

Family (obj type)	Instance name	Max soln	Initial soln	Optimal soln	PBS				CPLEX 7.0	
					Linear		Binary		Time	Best soln
					Time	Best soln	Time	Best soln		
Graph coloring (Min)	anna	20	20	11	**24.5**	11	67.8	11	774	11
	david	20	20	11	**2.85**	11	6.47	11	>1000	19
	DSJC125.1	20	20	5	39.73	5	**17.4**	5	>1000	20
	games120	20	20	9	11.18	9	**3.96**	9	>1000	19
	jean	20	20	10	28.4	10	**14.4**	10	>1000	11
	miles250	20	20	8	1.94	8	**1.83**	8	605	8
	myciel3	20	11	4	**0.07**	4	0.08	4	0.19	4
	myciel4	20	20	5	**0.22**	5	0.33	5	20.8	5
	myciel5	20	20	6	29.5	6	**10.5**	6	>1000	6
	queen5_5	20	20	5	0.22	5	**0.18**	5	0.26	5
	queen6_6	20	20	7	**0.59**	7	2.65	7	174	7
	queen7_7	20	20	7	**8.31**	7	9.17	7	475	7
	queen8_12	20	20	12	**1.08**	12	3.3	12	>1000	15
	Total				**149**		**138**		**>8050**	
FPGA-SAT (Min)	fpga20_15	450	30	30	**0.61**	30	1.46	30	200	30
	fpga25_15	563	30	30	**1.86**	30	4.07	30	>1000	26
	fpga30_15	675	30	30	**0.99**	30	2.49	30	>1000	18
	fpga25_20	750	40	40	**3.71**	40	13.5	40	>1000	39
	fpga30_20	900	40	40	**34.9**	40	149	40	>1000	25
	fpga35_20	1050	40	40	**4.35**	40	14.6	40	>1000	24
	Total				**46**		**185**		**>5200**	

N-queens (Max)	NQueens8	64	56 (8)	56 (8)	0.1	56 (8)	0.35	56 (8)	**0.01**	56 (8)
	NQueens9	81	72 (9)	72 (9)	0.69	72 (9)	2.71	72 (9)	**0.04**	72 (9)
	NQueens10	100	90 (10)	90 (10)	5	90 (10)	20.5	90 (10)	**0.6**	90 (10)
	NQueens11	121	110 (11)	110 (11)	69.2	110 (11)	228	110 (11)	**0.12**	110 (11)
	Total				75		**252**		0.77	
FPGA-UNS (Max)	chnl7_8	408	140 (268)	2 (406)	0.43	2 (406)	**0.38**	2 (406)	25.1	2 (406)
	chnl7_9	522	205 (317)	4 (518)	**1.68**	4 (518)	1.74	4 (518)	194	4 (518)
	chnl7_10	650	267 (383)	6 (644)	8.01	6 (644)	**7.2**	6 (644)	464	6 (644)
	chnl7_11	792	318 (474)	8 (784)	40.9	8 (784)	**35.7**	8 (784)	>1000	8 (784)
	chnl8_9	594	212 (382)	2 (592)	5.35	2 (592)	**1.65**	2 (592)	92.8	2 (592)
	chnl8_10	740	298 (442)	4 (736)	**3.47**	4 (736)	3.53	4 (736)	>1000	4 (736)
	chnl8_11	902	356 (546)	6 (896)	**20.8**	6 (896)	32.2	6 (896)	>1000	6 (896)
	chnl8_12	1080	474 (606)	8 (1072)	211	8 (1072)	**156**	8 (1072)	>1000	10 (1070)
	chnl9_10	830	324 (506)	2 (828)	3.05	2 (828)	**2.33**	2 (828)	362	2 (828)
	chnl9_11	1012	408 (604)	4 (1008)	11.9	4 (1008)	**5.5**	4 (1008)	>1000	4 (1008)
	chnl9_12	1212	512 (700)	6 (1206)	**60.2**	6 (1206)	70.4	6 (1206)	>1000	9 (1203)
	chnl9_13	1430	580 (850)	8 (1422)	>1000	8 (1422)	**765**	8 (1422)	>1000	11 (1419)
	Total				**>1367**		**1082**		**>8138**	
	Total				**>1637**		**1657**		**>21K**	

PBS is configured to use linear and binary sweep search. Best result for each instance is shown in bold. Maximization instances are converted to minimization instances. For maximization instances, the number between parenthesis reflects the value of the objective function before conversion.

initial solution for all the FPGA-SAT and N-queens instances is equal to the optimal value of the objective function. These instances are solved faster using the linear sweep search method.

- The binary sweep search tends to beat the linear sweep search for instances whose initial solution, $\tilde{g}_0$, is relatively far from the optimal solution. For example, the SAT solver's initial solution for all the FPGA-UNS and Graph coloring instances is far from the optimal value of the objective function. On average, these instances achieve better search runtimes using the binary sweep search method.

Overall, the above results show that SAT-based 0–1 ILP solvers, e.g. PBS, are in general faster than generic ILP solvers, e.g. CPLEX, since SAT-based 0–1 ILP solvers are expected to take advantage of the Boolean nature of the problem. The only exception, in Table 2, 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 relatively close (far) from the *lower* bound of the optimal solution. This poses the question of how to estimate the lower bound of the optimal solution for an optimization instance? A simple solution is to use the information provided with the instance to guess the lower bound. For example, the 8-*queens* instance consists of an 8×8 grid and is likely to fit up to 8 queens only. The SAT solver's initial solution is 8. The remaining processing time is spent on proving that 8 is the optimal solution. Another example is the *FPGAi_j* which represents an FPGA instance with *j* nets. According to Table 2, each FPGA-SAT instance needs at least two 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. In all other instances, the gap between the upper and estimated lower bound was large. Accordingly, the binary sweep search method was, on average, faster than the linear sweep search. An alternative solution to estimate the lower bound is to use some form of branch-and-bound. Other solutions can use a hybrid system that combines both binary and linear sweep search methods.

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 a simple 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.

References

[1] S. Cook, The complexity of theorem proving procedures, in: Proceedings of the Third Annual ACM Symposium on Theory of Computing, 1971, pp. 151–158.
[2] R.J. Bayardo Jr., R.C. Schrag, Using CSP look-back techniques to solve real world SAT instances, in: Proceedings of the 14th National Conference on Artificial Intelligence, 1997, pp. 203–208.
[3] E. Goldberg, Y. Novikov, BerkMin: a fast and robust SAT-solver, in: Proceedings of the Design Automation and Test Conference in Europe (DATE), 2002, pp. 142–149.
[4] J. Marques-Silva, K. Sakallah, GRASP: a search algorithm for propositional satisfiability, IEEE Trans. Comput. 5 (48) (1999) 506–521.
[5] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of the Design Automation Conference (DAC), 2001, pp. 530–535.
[6] H. Zhang, SATO: an efficient propositional prover, in: International Conference on Automated Deduction, 1997, pp. 272–275.
[7] Biere, A. Cimatti, E. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: Proceedings of the Design Automation Conference (DAC), 1999, pp. 317–320.
[8] G. Nam, F.A. Aloul, K.A. Sakallah, R. Rutenbar, A comparative study of two Boolean formulations of FPGA detailed routing constraints, in: The Proceedings of the International Symposium on Physical Design (ISPD), 2001, pp. 222–227.
[9] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, Generic ILP versus specialized 0–1 ILP: an Update, in: Proceedings of the International Conference on Computer-Aided Design (ICCAD), 2002, pp. 450–457.
[10] L. Silva, J. Silva, L. Silveira, K. Sakallah, Timing analysis using propositional satisfiability, IEEE Int. Conf. Electronics, Circuits Syst., September 1998, pp. 95–98.
[11] S. Memik, F. Fallah, Accelerated Boolean satisfiability-based scheduling of control/data flow graphs for high-level synthesis, in: Proceedings of the International Conference on Computer Design (ICCD), 2002.
[12] P. Bjesse, K. Claessen, SAT-based verification without state space traversal, in: Proceedings of Formal Methods in Computer-Aided Design (FMCAD), 2000, pp. 372–389.
[13] Ramani, F.A. Aloul, I.L. Markov, K.A. Sakallah, Breaking instance-independent symmetries in exact graph coloring, in: Proceedings of the Design, Automation, and Test in Europe Conference (DATE), 2004, pp. 324–329.
[14] N. Creignou, S. Kanna, M. Sudan, Complexity Classifications of Boolean Constraint Satisfaction Problems, SIAM Press, Philadelphia, PA, 2001.
[15] 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.
[16] D. Chai, A. Kuehlmann, A fast pseudo-Boolean constraint solver, in: Proceedings of the Design Automation Conference (DAC), 2003, pp. 830–835.
[17] H. Dixon, M. Ginsberg, Inference methods for a pseudo-Boolean satisfiability solver, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 2002, pp. 635–640.
[18] V. Manquinho, P. Flores, J. Marques-Silva, A. Oliveira, Prime implicant computation using satisfiability algorithms, in: Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, 1997, pp. 232–239.
[19] J. Whittemore, J. Kim, K. Sakallah, SATIRE: a new incremental satisfiability engine, in: Proceedings of Design Automation Conference (DAC), 2001, pp. 542–545.
[20] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, J. ACM 7 (5) (1962) 394–397.
[21] F.A. Aloul, I.L. Markov, K.A. Sakallah, Faster SAT and smaller BDDs via common function structure, in: Proceedings of the International Conference on Computer Aided Design (ICCAD), 2001, pp. 443–448.
[22] J. Marques-Silva, The impact of branching heuristics in propositional satisfiability algorithms, in: Proceedings of the Ninth Portuguese Conference on Artificial Intelligence, 1999.
[23] J. Freeman, Improvements to propositional satisfiability algorithms, Ph.D. Thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
[24] C.P. Gomes, B. Selman, H. Kautz, Boosting combinatorial search through randomization, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1998, pp. 431–437.
[25] DIMACS graph coloring instance ⟨http://mat.gsia.cmu.edu/COLOR/instances.html⟩.

[26] R. Sosic, J. Gu, Fast search algorithms for the N-queens problem, IEEE Trans. Systems, Man, Cybernetics 21 (6) (1991) 1572–1576.
[27] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, Efficient symmetry-breaking for Boolean satisfiability, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2003, pp. 271–282.
[28] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, ShatterPB: symmetry-breaking for pseudo-Boolean formulas, in: Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC), 2004, pp. 884–887.
[29] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, Solving difficult instances of Boolean satisfiability in the presence of symmetries, IEEE Trans. Comput. Aided Des. 22 (9) (2003) 1117–1137.
[30] J. Crawford, M. Ginsberg, E. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, 1996, pp. 148–159.