Solving SMT Problems with a Costly Decision Procedure by Finding Minimum Satisfying Assignments of Boolean Formulas

Martin Babka, Tomáš Balyo, and Jaroslav Kezniki

Abstract An SMT-solving procedure can be implemented by using a SAT solver to find a satisfying assignment of the propositional skeleton of the predicate formula and then deciding the feasibility of the assignment using a particular decision procedure. The complexity of the decision procedure depends on the size of the assignment. In case that the runtime of the solving is dominated by the decision procedure it is convenient to find short satisfying assignments in the SAT solving phase. Unfortunately most of the modern state-of-the-art SAT solvers always output a complete assignment of variables for satisfiable formulas even if they can be satisfied by assigning truth values to only a fraction of the variables. In this paper, we first describe an application in the code performance modeling domain, which requires SMT-solving with a costly decision procedure. Then we focus on the problem of finding minimum-size satisfying partial truth assignments. We describe and experimentally evaluate several methods how to solve this problem. These include reduction to partial maximum satisfiability – PMAXSAT, PMINSAT, pseudo-Boolean optimization and iterated SAT solving. We examine the methods experimentally on existing benchmark formulas as well as on a new benchmark set based on the performance modeling scenario.

Martin Babka · Tomáš Balyo
Department of Theoretical Computer Science and Mathematical Logic, Faculty of Mathematics and Physics, Charles University, Malostranské nám. 2/25, 118 00 Prague, Czech Republic.
e-mail: {babka|balyo}@ktiml.mff.cuni.cz

Jaroslav Kezniki
Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University, Malostranské nám. 2/25, 118 00 Prague, Czech Republic.
e-mail: keznikl@d3s.mff.cuni.cz

Jaroslav Kezniki
Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodárenskou veží 2, 182 07 Prague, Czech Republic.
e-mail: keznikl@cs.cas.cz

1 Introduction

Boolean satisfiability (SAT) is one of the most important and most studied problems of computer science. It is important in theoretical computer science, it was the first NP-complete problem [12], as well as in practical applications. SAT has a lot of successful applications in many fields such as A.I. planning [19], automated reasoning [26] and hardware verification [29]. This is possible because of the high practical efficiency of modern SAT solvers.

An important extension of the SAT problem is the SMT (Sat Modulo Theories) problem [4, 24]. SMT is a combination of SAT and some theories, for example arithmetic, arrays, or uninterpreted functions. Like SAT, SMT has numerous applications for example bounded model checking [1] or performance modeling of software [10, 11]. SMT solving can be done by using a SAT solver to evaluate the propositional skeleton of the SMT formula and then checking the result of the SAT solver using the theory evaluation procedures. It might be the case, that the evaluation of the theory is very time consuming and therefore it is beneficial to try to find minimum satisfying assignments in the SAT solving phase.

Unfortunately, most of the current state-of-the-art SAT solvers always output a complete satisfying truth assignment even for formulas that can be satisfied by small partial truth assignments. It is because these solvers implement the conflict-driven clause learning (CDCL) DPLL algorithm [7] in a very efficient manner. The search for a satisfying assignment in these implementations is continued until all variables are assigned or an empty clause is learned. Therefore the output of the solvers is a complete truth assignment for satisfiable instances.

In this paper we first give a brief description and example of the challenge of solving SMT problems with a theory that has a very costly decision procedure. We show how it can be addressed using a special SAT solver, that gives minimum partial satisfying assignments. The rest of the paper is then dedicated to finding such assignments.

In the theory of Boolean functions the problem of finding a partial satisfying truth assignment with the minimal number of assigned variables is called the shortest implicant problem. The decision version of this problem has been shown to be Σ_2^P – complete for general formulas [28]. However, for CNF formulas, it is in NP (see below), thus, theoretically, it is not harder than SAT.

This problem is also referred to as finding minimum-size implicants. It is sometimes confused with the problem of finding minimal-size implicants (implicants that cannot be shortened, i.e., prime implicants). A minimum-size implicant is always a minimal-size (prime) implicant but not vice versa. The problem of finding prime implicants is well studied and there are many papers devoted to this topic, see e.g. [25]. On the other hand, methods for finding minimum-size implicants are often hidden inside papers dealing with other problems, where they are only briefly mentioned as a possible application. There are however some papers dealing directly with minimum-size implicants such as [23] and [22].

Our goal is to give an overview of several methods for the minimum satisfying assignment problem based on reducing this problem to other well known problems.

Two of the described reductions (PMAXSAT and PMINSAT) have not been described elsewhere. The others are mentioned in the literature. For more information please see Section Related Work. In the paper we also do experimental comparison of the described methods using relevant benchmark problems and state-of-the-art solvers.

2 Motivation

2.1 SMT *Solving With a Costly Decision Procedure*

In general, the main motivation for short satisfying assignments is the case of an SMT-solving [4, 24] algorithm with a costly decision procedure. SMT-solving is a technique for finding satisfying assignments of predicate-logic formulas. The basic idea of one of the approaches to SMT-solving is to employ a SAT solver for finding a satisfying assignment of the propositional skeleton of a given predicate formula. Having such a satisfying assignment, a decision procedure (specific to the particular predicate logic) is employed in order to decide the feasibility of the assignment with respect to the predicates. If the assignment is not feasible, the SAT solver is (incrementally) asked for another satisfying skeleton assignment until the assignment is feasible or there are no undecided assignments left. As an aside, the state-of-the-art SMT solvers operate incrementally; i.e., they call the decision procedure already for partial skeleton assignments. Nevertheless, since this potentially increases the number of expensive decision procedure calls, we will consider the non-incremental case. Note, that the unsatisfiability of a propositional skeleton implies unsatisfiability of the associated predicate formula (the opposite does not hold). Additionally, the satisfiability of a predicate formula implies the satisfiability of its propositional skeleton.

A typical decision procedure of an SMT-solving algorithm is designed to work with the conjunctive fragment of the predicate logic (i.e., conjunctions of predicates and their negations). A formula in the conjunctive fragment can be easily obtained from a satisfying skeleton assignment. Therefore, while deciding feasibility of a skeleton assignment, it is necessary to evaluate some of the associated predicates; in the case of a feasible assignment all of them.

Taking into account a decision procedure where an evaluation of a predicate is a costly operation [11], it is beneficial to minimize the number of evaluated predicates while deciding feasibility of a skeleton assignment. However, this minimization has to be performed by the SAT solver by providing small satisfying assignments (as the decision procedure works with the conjunctive fragment and thus has to evaluate all the corresponding predicates).

2.2 Stochastic Performance Logic

To illustrate this problem, we describe the Stochastic Performance Logic (SPL) [10, 11], for which evaluating the predicates is a very time-consuming operation and which will thus greatly benefit from minimization of the satisfying skeleton assignments during SMT-solving. Specifically, it is a predicate logic designed for expressing assumptions about performance of code and is motivated by the challenges in the performance modeling domain. In particular, according to [11], it is beneficial to provide means for performance testing similar to functional unit-testing approaches – that is, being able to express performance-related developer assumptions or intended usage in code in a platform-independent way and test or verify them automatically.

The main goal of SPL is thus to capture performance conditions that should be met by software (expressing performance-related developer assumptions or intended usage) in a form of predicate formulas, semantics of which is platform-independent. Specifically, the approach of SPL is based on capturing performance conditions on a given function relatively to performance of a baseline function (rather than on absolute metrics); e.g., in case of an encryption function, the baseline can be the memory-copying function (i.e., no encryption). In practice, SPL formulas are inserted into code (e.g., as Java annotations) and automatically validated [18].

The semantics of the predicates expressing the relative performance is based on instrumentation and monitoring of the execution times of both the tested and baseline function and performing a statistical test in order to validate or invalidate the statistical hypothesis determined by the predicate. Therefore, the decision procedure in SPL has to perform (expensive) execution-time measurements and a statistical test in order to evaluate a single performance predicate. Thus, it is an extremely time-consuming operation.

To provide a clearer perspective on this issue, we present a brief summary of the SPL-solving algorithm (Fig. 1). Before going into detail, we first describe the notation. For a given SPL formula F, the $MakeSkeleton$ function returns its propositional skeleton F_S. A_P is a partial assignment of F_S enforcing the results of the previous decision-procedure runs. The $ApplyAssignment(F,A)$ function returns formula F after applying the assignment A ; i.e., with all variables from A replaced by their assigned values. The *PartSAT* function returns for the given formula a satisfying assignment with only some variables assigned (i.e., a partial satisfying assignment). The tuple (var,val) denotes a variable and its value in an assignment. The $FilterAssigned$ function returns the assigned variables in the given assignment. *MeasureAndTest* is the very expensive decision procedure deciding validity of a single performance predicate associated with the given skeleton variable. Finally, m is the result of the procedure (i.e., true or false).

After the propositional skeleton F_S is created and the partial assignment A_P is initialized (lines 1-2), a partial satisfying truth assignment A_{temp} of F_S after applying A_P is obtained via the *PartSAT* function (line 3). If *PartSAT* indicates that F_S after applying A_P is unsatisfiable, the algorithm returns “false” (lines 4-6), because it implies that the original SPL formula is unsatisfiable with respect to measurements

```
1: F_S ← MakeSkeleton(F)
2: A_P ← ∅
3: A_temp ← PartSAT(ApplyAssignment(F_S, A_P))
4: if A_temp = false then
5:     return false
6: end if
7: for all (var, val) ∈ FilterAssigned(A_temp) do
8:     m ← MeasureAndTest(var)
9:     A_P ← A_P ∪ {(var, m)}
10:    if val ≠ m then
11:        goto line 3
12:    end if
13: end for
14: return true
```

Fig. 1 SPL-solving algorithm

dictating A_P. Otherwise, the algorithm sequentially processes assignments of all assigned variables; i.e., those which were not yet checked by the decision procedure (line 7). Note that the order in which the variables are processed may depend on further optimization; e.g., the variable corresponding to the "cheapest to measure" performance predicate will be processed first. For each assigned variable, it is necessary to call the decision procedure *MeasureAndTest* (line 8). The result of the decision procedure is added to A_P to be enforced in the subsequent *PartSAT* runs (line 9). If the stored result conforms to the current skeleton valuation A_{temp}, the next variable is processed. Otherwise (lines 10-12), A_{temp} is infeasible with respect to the measurements and a new skeleton valuation has to be obtained from *PartSAT*.

It is important to stress, that each call of the decision procedure *MeasureAndTest* for a typical performance predicate usually takes a non-trivial amount of time; i.e., hundreds of milliseconds. Thus, it is obvious that employing a *PartSAT* function that supports partial satisfying assignments with the minimum number of assigned variables (and thus minimizes the number of performance predicates to be evaluated) would significantly reduce the execution time of the whole SPL-solving algorithm.

The rest of the paper is devoted to the computation of the *PartSAT* function i.e. finding minimum satisfying truth assignments of Boolean formulas.

3 Preliminaries

A *Boolean variable* is variable with two possible values *True (1)* and *False (0)*. A *literal* of a Boolean variable x is either x or $\bar{x}$ (*positive* or *negative literal*). A *clause* is a disjunction (OR) of literals. A *conjunctive normal form (*CNF*) formula* is a conjunction (AND) of clauses. The number of variables of a formula will be denoted by n. A (partial) truth assignment ϕ of a formula F assigns a truth value to (some of) its variables. The assignment ϕ satisfies a positive(negative) literal if

it assigns the value true (false) to its variable and ϕ satisfies a clause if it satisfies any of its literals. Finally, ϕ satisfies a CNF formula if it satisfies all of its clauses. A formula F is said to be satisfiable if there is a (partial) truth assignment ϕ that satisfies F. Such an assignment is called a *satisfying assignment*. The satisfiability problem (SAT) is to find a satisfying assignment of a given CNF formula. We will call ϕ_{min} a *minimum-size satisfying assignment* of a formula F if there is no other satisfying assignment ϕ of F, such that ϕ assigns truth values to fewer variables than ϕ_{min}.

A conjunction (AND) of literals is called a *term*. An *implicant* I of a formula F is a term, such that any truth assignment that satisfies I also satisfies F. I is a *shortest implicant* of a formula F if there is no other implicant I' of F such that I' contains fewer literals than I. I is called a prime implicant if there is no other implicant I' such that $I' \subset I$. The *shortest implicant problem* is to find the shortest implicant of a given formula. It is easy to observe that an implicant corresponds to a satisfying partial truth assignment of its formula and the shortest implicant corresponds to a minimum-size satisfying assignment.

4 Related Work

In the Boolean functions community the problem of shortest implicants is studied mostly in the context of Boolean function minimization [28], which is the problem of finding a minimal representation of Boolean functions [13]. The function is often given in form of a CNF formula and the desired output is an equivalent CNF or DNF formula of minimum size. In this context, finding shortest implicants is Σ_2^P – complete for general formulas, [28].

Some papers about enumerating prime implicants also describe methods for finding the shortest implicants. One such paper is by Bieganowky and Karatkevich, which presents a heuristic for Thelen‘s method [5]. Thelen‘s method is an algorithm for enumerating all prime implicants of a CNF formula. The proposed heuristic should lead to a minimal prime implicant, but it is not guaranteed to find an optimal solution.

In [25] a 0-1 programming scheme is used to encode the formula and additional constraints which allow selective enumeration. The constraint can, of course, be the length of the implicant, therefore this method is suitable for our purposes. Considering the efficiency of state-of-the-art pseudo-Boolean optimization (PBO) solvers, this approach appears to be a promising one.

In [23] and [22] the authors describe some methods based on integer linear programming (ILP) and binary decision diagrams (BDD).

Finally, in [8] there is a suggestion, that the problem could be solved by incremental SAT solving. This requires us to encode cardinality constraints into SAT. There are several available methods to do this, a survey of such methods is given in [2].

5 Solving the Shortest Implicant Problem

We shall start this section by describing a technique called *dual rail encoding* [9], which will be used in all of the following methods.

5.1 Dual Rail Encoding

The first step of the dual rail encoding of a CNF formula F is introducing new *dual rail variables* representing possible positive and negative assignments to the original variables of F.

Definition 1 (Dual rail variables). Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. Then the Boolean variables $X_{DR} = \{px_1, nx_1, px_2, nx_2, \ldots, px_n, nx_n\}$ are the dual rail variables for X.

Let ϕ be a partial truth assignment of X. Then we define ϕ_{DR} as a truth assignment of X_{DR} so that $\phi_{DR}(px_i) = 1 \Leftrightarrow \phi(x_i) = 1$ and $\phi_{DR}(nx_i) = 1 \Leftrightarrow \phi(x_i) = 0$.

Notice that px_i and nx_i are both negative under ϕ_{DR} iff x_i is unassigned under ϕ. This implies that the number of assigned variables under ϕ is equal to the number of dual rail variables that are assigned 1 by ϕ_{DR}. Also observe that given ϕ_{DR} we can easily construct ϕ and vice versa.

Definition 2 (Dual rail encoding). Let F be a CNF SAT formula with variables $X = \{x_1, \ldots, x_n\}$ and clauses C. Let C_{DR} be the clauses obtained from the clauses C by replacing all occurrences of the literal x_i by px_i and literal $\overline{x}_i$ by nx_i for all $i \in \{1 \ldots n\}$. The dual rail encoding of F is a CNF formula

$$F_{DR} = C_{DR} \wedge \bigwedge_{i \in \{1 \ldots n\}} (\overline{px}_i \vee \overline{nx}_i)$$

Example 1 (Dual rail encoding). $(x_1 \vee \overline{x}_2) \wedge (x_3 \vee \overline{x}_1) \wedge (\overline{x}_2 \vee \overline{x}_3)$ would be encoded as $(px_1 \vee nx_2) \wedge (px_3 \vee nx_1) \wedge (nx_2 \vee nx_3) \wedge (\overline{px}_1 \vee \overline{nx}_1) \wedge (\overline{px}_2 \vee \overline{nx}_2) \wedge (\overline{px}_3 \vee \overline{nx}_3)$.

Lemma 1. *Let F be a* CNF *formula. Then ϕ is a satisfying assignment of F iff ϕ_{DR} is a satisfying assignments of F_{DR}.*

Proof. Let ϕ satisfy F. Let C be an arbitrary clause of F, then there is a literal x (or $\overline{x}$) in C that is satisfied under ϕ. It implies by definition that px (or nx) is $True$ under ϕ_{DR}. Hence the clause corresponding to C in F_{DR} is satisfied by px (or nx). The clauses $(\overline{nx}_i \vee \overline{px}_i)$ of F_{DR} are satisfied under any ϕ_{DR} since a Boolean variable cannot be assigned both values $True$ and $False$.

On the other hand, let ϕ_{DR} satisfy F_{DR}. The $(\overline{nx}_i \vee \overline{px}_i)$ clauses ensure that either px_i or nx_i is $False$ under ϕ_{DR} and thus ϕ is a valid partial truth assignment. Let C be an arbitrary clause in F. The corresponding clause to C in F_{DR} is satisfied by a literal px (or nx), surely is then C satisfied by x (or $\overline{x}$) under ϕ.

5.2 Solving via Pseudo-Boolean Optimization

In this section we describe a method for solving the shortest implicant problem by reducing it to the pseudo-Boolean optimization problem [7]. We start by its definition.

A *PB-constraint* is an inequality $C_0 \times x_0 + C_1 \times x_1 + \cdots + C_{k-1} \times x_{k-1} \geq C_k$, where C_i are integer coefficients and x_i are literals. The integer value of a Boolean variable is defined as 1 (0) if it is *True* (*False*). Positive (negative) literals of a variable x are expressed as x ($(1-x)$) in the inequality. A (partial) truth assignment ϕ *satisfies a PB-constraint* if the inequality holds. An *objective function* is a sum $C_0 \times x_0 + C_1 \times x_1 + \cdots + C_l \times x_l$, where C_i are integer coefficients and x_i are literals. The pseudo-Boolean optimization problem is to find a satisfying assignment to a set of PB-constraints that minimizes a given objective function.

Now, we describe how a CNF formula F can be reduced into a PB optimization problem. For a clause $C = (l_1 \vee l_2 \vee \cdots \vee l_k)$ we define its PB-constraint $\mathrm{PB}(C) = (1 \times l_1 + 1 \times l_2 + \cdots + 1 \times l_k \geq 1)$. It is easy to see that a partial assignment ϕ satisfies the clause C iff it satisfies its PB-constraint $\mathrm{PB}(C)$. For a CNF formula F we denote its PB-constraints $\mathrm{PB}(F) = \{\mathrm{PB}(C) \mid C \in F\}$.

Example 2 (Reducing a clause into a PB-constraint). $(x_1 \vee \bar{x}_2 \vee x_3)$ would yield $1 \times x_1 + 1 \times (1 - x_2) + 1 \times x_3 \geq 1$.

For a given CNF formula F we encode the instance of the shortest implicant problem as the pseudo-Boolean optimization problem $\mathrm{PBO}(F)$ as follows. First we construct the dual rail encoding F_{DR} of F. Then we translate it into its PB-constraints $\mathrm{PB}(F_{DR})$. Finally, we define the objective function $\mathrm{O}(F_{DR})$ as $\mathrm{O}(F_{DR}) = \sum_{i=1}^{n} (1 \times px_i + 1 \times nx_i)$. Let us denote $\mathrm{PBO}(F) = (\mathrm{PB}(F_{DR}), \mathrm{O}(F_{DR}))$ the pseudo-Boolean optimization problem with the constraints $\mathrm{PB}(F_{DR})$ and the objective function $\mathrm{O}(F_{DR})$.

The optimal solution of $\mathrm{PBO}(F)$ is a truth assignment that satisfies all the constraints and minimizes the objective function. Now, we can use a PB solver to find an optimal solution of $\mathrm{PBO}(F)$ and from the optimal solution we can extract the shortest implicant in the following way.

Definition 3. For a truth assignment ψ of the dual rail variables we define the term I_ψ as

$$I_\psi = \bigwedge_{i:\ \psi(px_i)=1} x_i \wedge \bigwedge_{i:\ \psi(nx_i)=1} \bar{x}_i$$

Theorem 1. *Let F be a* CNF *formula and ψ the optimal solution of* $\mathrm{PBO}(F)$. *Then I_ψ is the shortest implicant of F.*

Proof. From Lemma 1 and the correspondence of satisfying assignments and implicants we get that I_ψ is an implicant of F. By contradiction we show that there is no shorter implicant. Let I' be a shorter implicant than I_ψ. Then I' defines a satisfying assignment ϕ of F. Realize the fact that the length of the implicant is exactly the

number of the variables assigned by ϕ which equals the value of the objective function $\mathrm{O}(F_{DR})$ for ϕ_{DR}. Thus ϕ allows us to construct a better solution for $\mathrm{PBO}(F)$ than ψ. That is contradictory with ψ being an optimal solution of $\mathrm{PB}(F)$.

5.3 Solving via Partial Maximum Satisfiability

In this section we describe a reduction of shortest implicant problem into a partial maximum satisfiability (PMAXSAT) problem [7]. The reduction is again based on dual rail encoding, therefore it is very similar to the PB optimization approach. First we define the PMAXSAT problem.

A PMAXSAT *formula* is a tuple of two sets of clauses called *soft clauses* and *hard clauses*. A *solution* of a PMAXSAT problem is a truth assignment that satisfies all hard clauses and some soft clauses. An *optimal solution* of a PMAXSAT problem is a solution ϕ that there is no other solution that satisfies more soft clauses than ϕ.

To reduce shortest implicant problem given by a CNF formula F to a PMAXSAT problem $\mathrm{PMAX}(F)$ we first apply dual rail encoding on F. The clauses of F_{DR} are the hard clauses of $\mathrm{PMAX}(F)$. The soft clauses of $\mathrm{PMAX}(F)$ are defined as the unit clauses $\overline{px}_i$ and $\overline{nx}_i$ for each i. The shortest implicant from the optimal solution of $\mathrm{PMAX}(F)$ is extracted in the same way as in the case of PB optimization. A precise formulation and proof follows.

Theorem 2. *Let F be a* CNF *formula and ψ an optimal solution of* $\mathrm{PMAX}(F)$. *Then I_ψ is a shortest implicant of F.*

Proof. Let ψ be an optimal solution of $\mathrm{PMAX}(F)$. All hard clauses of $\mathrm{PMAX}(F)$ are satisfied under ψ and thus by Lemma 1 I_ψ is an implicant of F. The implicant I_ψ is also the shortest possible. The existence of a shorter one would allow a partial truth assignment ϕ of F such that ϕ_{DR} satisfies more soft clauses than ψ. Indeed, the number of unsatisfied soft clauses is equal to the number dual rail variables assigned the value True.

5.4 Solving via Partial Minimum Satisfiability

The partial minimum satisfiability (PMINSAT) problem [7] is analogous to the PMAXSAT problem with the only difference being, that the goal is to minimize the number of satisfied soft clauses. The reduction of the shortest implicant problem to PMINSAT is a straightforward modification of the PMAXSAT reduction. Instead of using the unit soft clauses $\overline{px}_i$ and $\overline{nx}_i$ for each i we use px_i and nx_i (e.g. the soft clauses of $\mathrm{PMAX}(F)$ are negated).

5.5 *Solving via Iterative* SAT *Solving*

The method described in this section is in a way similar to planning as satisfiability [19]. For a given CNF formula F we construct another CNF formula $G(F,k)$ which will be satisfiable iff F has an implicant of size k or shorter. We construct and test $G(F,k)$ for various k iteratively until we find the smallest k such that $G(F,k)$ is satisfiable. From the satisfying assignment of $G(F,k)$ we extract a shortest implicant of size k.

To construct $G(F,k)$ we again start by dual rail encoding F into F_{DR} and then we add a cardinality constraint $\leq_k (px_1, nx_1, \ldots, px_n, nx_n)$ meaning $(\sum_{i=1}^{n} px_i + nx_i) \leq k$. There are several methods of encoding cardinality constraints into SAT. A survey on these methods is given in [2]. Many of these encodings are polynomial (relative to n and k) in size and time required to construct them. There is even a linear encoding [14]. The resulting formula $G(F,k)$ is a conjunction of the cardinality constraint and the dual rail encoding of the original formula.

The reduction can be improved by adding a set of n new variables sx_i which encode if the variable x_i is assigned: $(px_i \vee nx_i) \rightarrow sx_i$. Then we encode the cardinality constraint over sx_i instead of px_i and nx_i. The improved reduction $G_s(F,k)$ is then

$$\begin{aligned} G_s(F,k) = & F_{DR} \wedge \bigwedge_{i=1}^{n} [(\overline{px}_i \vee sx_i) \wedge (\overline{nx}_i \vee sx_i)] \\ & \wedge \leq_k (sx_1, sx_2, \ldots, sx_n) \end{aligned}$$

Why is this an improvement? In fact most encodings of cardinality constraints add a lot of new variables and clauses to the formula. Therefore it is good to use the cardinality constraint on fewer variables. Overall, $G_s(F,k)$ has fewer variables than $G(F,k)$ for almost every known cardinality encoding. Also, in our experiments $G_s(F,k)$ vastly outperformed $G(F,k)$ in terms of time required to solve them by a SAT solver. A theorem of this approach's validity follows.

Theorem 3. *Let F be a* CNF *formula, k be the smallest integer such that $G_s(F,k)$ is satisfiable. If ψ is a partial truth assignment satisfying $G_s(F,k)$, then the term $I_{\psi|\{px_1,nx_1,\ldots,px_n,nx_n\}}$* [1] *is the shortest implicant of F.*

Proof. Lemma 1 implies that $I = I_{\psi|\{px_1,nx_1,\ldots,px_n,nx_n\}}$ is an implicant of F. Observe that I has length exactly k. For the sake of contradiction assume that there is a shorter implicant I' with length $k' < k$. Then $G_s(F,k')$ must be satisfiable which is contradictory with the choice of k.

The proper k can be found for example by iteratively solving $G_s(F,k-1)$ for $k = n, n-1, \ldots, 1$ until $G_s(F,k-1)$ is unsatisfiable. A better way is to use binary search to find the proper k, which we used in our experiments.

What we described above is actually a polynomial reduction of the decision version of the shortest implicant problem into SAT. Since SAT is in NP, the shortest implicant problem for CNF formulas is also in NP.

[1] By $\psi|\{px_1, nx_1, \ldots, px_n, nx_n\}$ we mean the restriction of ψ to the variables $px_1, nx_1, \ldots, px_n, nx_n$.

Table 1 Results for the tested algorithms and instances

Benchmark Set		maxsat	minsat	iter. sat	pbo	local	sat
Random	no. solved (opt. no./approx.)	79	80	80	**81**	79 (75/1.0003)	81
	total time [s]	8767	6516	5887	**5037**	**4005**	20.15
SPL	no. solved (opt. no./approx.)	24	35	**98**	19	**98** (20/1.0045)	98
	total time [s]	139835	133047	**4680**	143827	18447	1.51
BMC	no. solved (opt. no./approx.)	0	3	**9**	3	0 (0/∞)	13
	total time [s]	23400	18302	**12096**	18502	11714	14

The bold value indicates the best result. In the case of local search we also give the number of optimal solutions and the average approximation ratio.

5.6 Solving via Incomplete Methods

In the previous sections we incorporated various complete methods which find the optimal solution for certain optimization problems. However it is often the case that there are also incomplete methods based on local search which solve the same problems. The general advantage of incomplete solvers is that they run fast and are able to quickly produce a first but rough estimate of the objective function. The unpleasant price is that they are not guaranteed to find the optimal solution.

Several incomplete methods have been already designed for the PMAXSAT problem. They can, of course, be used for solving the shortest implicant instances using the same encoding. Thus the only difference is that the produced implicant cannot be proven to be of minimum size. When using incomplete methods we have to consider the quality of the solutions together with the running time of the algorithms in order to compare the algorithms correctly.

6 Experiment Setup

To compare the practical usability of the above described methods, we conducted experiments on various benchmark problems. We implemented the reductions in Java, particularly, to encode cardinality constraints for iterative SAT we employed the BoolVar/PB Java library [3]. BoolVar/PB implements several methods; we concretely used the "linear" encoding, which implements a sorter based encoding introduced by Eén and Sörensson [14].

For PMAXSAT solving we used Akmaxsat by Adrian Kügel [20] and for PMINSAT minsat [15]. For PB-optimization we selected bsolo [21]. The SAT solver used for iterative SAT solving was PrecoSAT by Armin Biere [6]. As for the incomplete solver we chose UBCSAT [27] particularly the g2wsat algorithm.

Our focus was on our own new benchmark set – SPL – but we also used benchmark formulas from SATLIB [17].

As for the SPL benchmark, we have exploited several SPL use-cases [10, 11], As a simple example, consider the following: if a method M uses two implementations A and B of a library function, we may want to express that performance of M depends on performance of the fastest one of A and B. In the propositional skeleton of the corresponding SPL formula, this could be expressed (after simplification) by the following conjunction:

$$\begin{array}{rrcl} & (V_{A\_is\_faster\_than\_B} & \Longrightarrow & V_{M\_depends\_on\_A}) \\ \wedge & (V_{B\_is\_faster\_than\_A} & \Longrightarrow & V_{M\_depends\_on\_B}) \\ \wedge & (V_{A\_is\_faster\_than\_B} & \vee & V_{B\_is\_faster\_than\_A}) \\ \wedge & (\neg V_{A\_is\_faster\_than\_B} & \vee & \neg V_{B\_is\_faster\_than\_A}) \end{array}$$

Note, that in SPL such a formula corresponds to particular values of performance parameters (e.g., size of an input array). A scenario typically covers several such values (e.g., array sizes 100, 200, and 500). We have always considered several scenarios to generate each benchmark CNF formula. In particular, the formula comprises a conjunction of sub-formulas encoding the individual scenarios. Since the scenarios are independent, all the sub-formulas use disjoint sets of variables. Basically, the scenarios cover different forms of selection of a suitable variant of a function implementation, based on the relative performance of the implementation for the given performance parameters (e.g. size of an input array). In general, the main parameters determining the produced sub-formula for each scenario are: (i) the number of alternative implementation variants, and (ii) the range of performance parameter values to be covered. The former case increases the size of the clauses of the generated sub-formulas, while the latter increases the number of the sub-formulas. Overall, the SPL benchmark uses randomization while generating the sub-formulas. The final formula is produced by repeating the randomized generation process until reaching the required number of clauses and/or variables. For our experiments, we have generated formulas in a range of sizes, starting with hundreds of clauses and variables, and ending with tens of thousands. As an aside, in SPL, the relation "faster than" has a slightly different semantics to "slower or equal", therefore there are two different variables in the example – $V_{A\_is\_faster\_than\_B}$ and $V_{B\_is\_faster\_than\_A}$ – rather the just one and its negation. Moreover, because of SPL semantics, the variable $V_{M\_depends\_on\_A}$ has to be in the benchmark formula actually represented as a conjunction of variables "*M is at most c1% slower than A*" and "*M is at most c2% faster than A*", where c1 and c2 express the level of dependency of M on A.

The other input data are chosen from the SATLIB benchmarks [16]. For our experiments we selected the "bmc" and "Uniform Random-3-SAT" formulas. The BMC formulas arise from bounded model checking problem instances which are modelled as SAT. And the random formulas are from phase transition region with number of variables ranging from 50 to 250. For further explanation of the formulas consult the SATLIB benchmark site [16].

The experiments were run for each input type on a computer with Intel i7 920 CPU @ 2.67 GHz processor and 6 GB of memory. The timelimit for a single in-

stance was 1800 seconds. The instances were sorted by the number of variables and if the solver timed out eight times in a row we stopped running it on that input set.

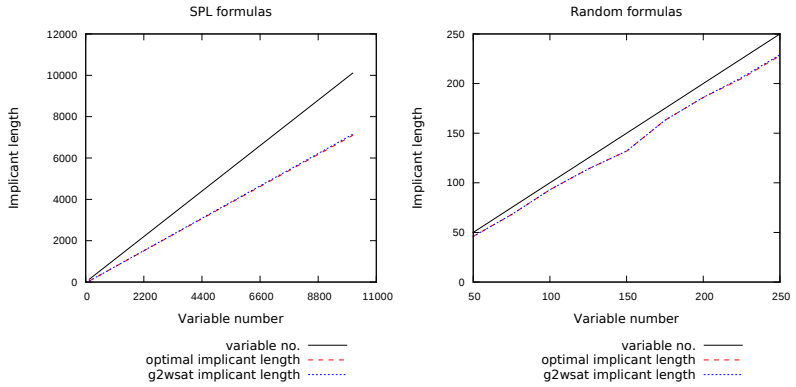

Fig. 2 Comparison of the length of the shortest implicant to the number of variables for SPL and BMC formulas.

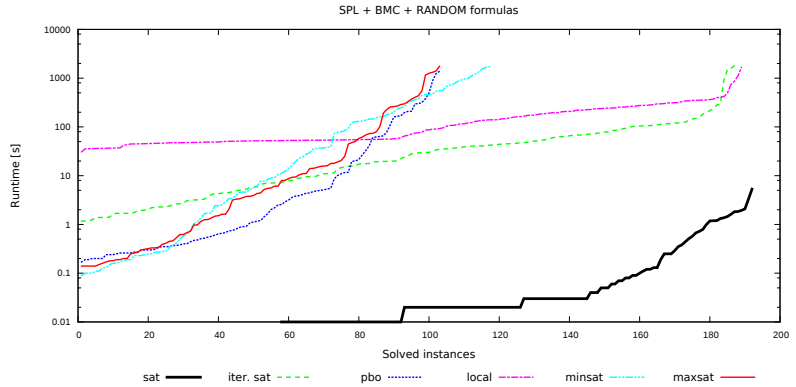

Fig. 3 Number of solved instances and runtimes of all the algorithms for all the input instances.

7 Experiment Results

In Table 1 and Figure 3 we compared the running times required to solve the SAT formulas (using Precosat [6]) with the running times required to find the shortest

implicants by the described methods. In the case of the local search, g2wsat algorithm, (named **local**) we also provide the number of optimal solutions found and the average approximation ratio – the length of the found implicant divided by the optimal length. The *total time* is the sum of the running times on all the instances. If the solver did not terminate within the time limit we used the time limit as the running time. Let us note that the SAT solver was able to solve all the instances.

Verifying if the input formula is satisfiable turns out to be by orders of magnitude faster than finding the minimum satisfying assignment. We think that this is also the reason why the iterative SAT is the fastest method.

To support this idea we also experimented with various modifications of iterative SAT. First we chose different initial lower and upper bounds on the the length of the shortest implicant. We obtained them using the Akmaxsat solver or set them to 1 and n respecitvely. The other modification is just a simple linear search, i.e., we used the iterative SAT reduction with the limit u, then $u-1$ and so on until the optimum length was found.

Out of all the possible modifications binary search with the initial bounds set to 1 and n performed the best when considering the number of solved instances. It also always ran faster and solved more instances than the binary search with the bounds initialized by Akmxasat solver. Thus we think that Akmaxsat spends a nonnegligible amount of time by deriving the bounds, especially the upper one. This fact is based on the observation that the linear search starting from the lower bound achieves comparable results to the binary search.

All the iterative methods solve more instances than the Akmaxsat solver, especially the method approaching the optimum from below. We also observed that the iterative methods based on the linear search are less stable than the binary search with bounds 1 and n, i.e. they never solved more instances. However on some inputs the linear search approaching the optimum from below performed faster.

The performance of linear search does not substantially depend on the fact if the bound is derived by akmaxsat – the methods have roughly the same performance. For the methods starting with lower bounds, which seems to be easy to obtain, we think that sat solving dominates the running time. For the methods starting with the upper bound the number of iterations is certainly lower see Figure 2. On the other hand sat solving is harder since the upper limit on the implicant length prunes less of the search space than the lower bound. For the lower bound methods the fact that solving unsatisfiable formulas does seem to have a detrimal effect.

The other complete methods (PMAXSAT, PMINSAT, and PBO) give very similar results relative to each other but are considerably weaker than iterative SAT. It is interesting that there is a relatively big gap between PMAXSAT (worst of the 3) and PMINSAT (best of the 3) since these problems and our encodings for them are very similar. The difference is probably caused by the different heuristics and implementation of the solvers.

Let us note that the performance of incomplete methods, especially the quality of the solution, crucially depends on a proper choice of the parameters of the algorithm such as the number of steps, number or restarts and overall iteration count. When

these parameters are well chosen the quality of the solution is comparable to the optimal solution as observed in Figure 2.

Altogether we can conclude that the best strategy is iterative SAT followed by iterative SAT using simple linear search. For the hard formulas incomplete methods could also be useful but one has to tweak their parameters.

8 Conclusion

In this paper we have shown that finding minimum-size satisfying assignments is both useful and can be computed relatively efficiently for many relevant formulas. The usefulness was demonstrated by describing a class of SMT problems with a costly decision procedure and an application of this kind – the SPL framework.

We described five possible methods to solve this problem from which the reductions to PMINSAT and PMAXSAT are novel to our best knowledge. Although the other three already appeared in the literature, there is no published comparison of these methods.

We did exhaustive experiments using modern state-of-the-art solvers and relevant benchmark problems to measure the performance of the methods we described. One of the benchmark sets was generated according to ideas of the SPL framework. Unfortunately, we were unable to do direct experiments to measure the usefulness of the methods for the SPL framework, since it is still under development and the number of its large-scale case studies is limited.

As for future work, we plan to improve the methods with support for assignment costs. Finding optimal short assignments with respect to a given assignment cost function would be beneficial in the cases presented in the motivation section, SPL in particular. Here, the cost of a SAT assignment could be determined by the execution times of the measurements to be performed by the SMT decision procedure in order to decide the feasibility of the skeleton assignment. In consequence, this would allow preferring the fast measurements to the slower ones while solving the SPL formulas.

Acknowledgements This research was partially supported by the SVV project number 267314, the Grant agency of the Charles University under contracts no. 266111 and 600112, and the Charles University institutional funding SVV-2013-267312. This work was also partially supported by the Grant Agency of the Czech Republic project GACR P202/10/J042.

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using smt solvers instead of sat solvers. International Journal on Software Tools for Technology Transfer (STTT) **11**(1), 69–83 (2009)

2. Bailleux, O.: On the cnf encoding of cardinality constraints and beyond. CoRR **abs/1012.3853** (2010)
3. Bailleux, O.: Boolvar/pb v1.0, a java library for translating pseudo-boolean constraints into cnf formulae. CoRR **abs/1103.3954** (2011)
4. Barrett, C., Dill, D., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to sat. In: Computer Aided Verification, *LNCS*, vol. 2404. Springer (2002)
5. Bieganowski, J., Karatkevich, A.: Heuristics for thelen's prime implicant method. Schedae Informaticae **14**, 125–125 (2005)
6. Biere, A.: Precosat home page. `http://fmv.jku.at/precosat/` (2013)
7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185. IOS Press (2009)
8. Brauer, J., King, A., Kriener, J.: Existential quantification as incremental sat. In: CAV, pp. 191–207 (2011)
9. Bryant, R.E.: Boolean analysis of mos circuits. IEEE Trans. on CAD of Integrated Circuits and Systems **6**(4), 634–649 (1987)
10. Bulej, L., Bures, T., Horky, V., Keznikl, J., Tuma, P.: Performance Awareness in Component Systems: Vision Paper. In: Proceedings of COMPSAC 2012, COMPSAC'12 (2012)
11. Bulej, L., Bures, T., Keznikl, J., Koubkova, A., Podzimek, A., Tuma, P.: Capturing performance assumptions using stochastic performance logic. In: Proceedings of ICPE'12 (2012)
12. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC, pp. 151–158 (1971)
13. Crama, Y., Hammer, P.L.: Boolean Functions - Theory, Algorithms, and Applications, *Encyclopedia of mathematics and its applications*, vol. 142. Cambridge University Press (2011)
14. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into sat. JSAT **2**(1-4), 1–26 (2006)
15. Heras, F., Morgado, A., Planes, J., Silva, J.P.M.: Iterative sat solving for minimum satisfiability. In: ICTAI, pp. 922–927 (2012)
16. Hoos, H., Stutzle, T.: Satlib benchmark site. `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html` (2013)
17. Hoos, H.H., Stutzle, T.: Satlib: An online resource for research on sat. pp. 283–292. IOS Press (2000)
18. Horky, V.: Stochastic Performance Logic (SPL) Home Page. `http://d3s.mff.cuni.cz/projects/performance_evaluation/spl/` (2013)
19. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI 92: Tenth European Conference on Artificial Intelligence, pp. 359–363. Vienna, Austria (1992)
20. Kügel, A.: Homepage of Adrian Kügel. `http://www.uni-ulm.de/en/in/institute-of-theoretical-computer-science/m/kuegel.html` (2012)
21. Manquinho, V.: bsolo home page. `http://sat.inesc-id.pt/~vmm/research/index.html` (2012)
22. Manquinho, V., Oliveira, A., Marques-Silva, J.: Models and algorithms for computing minimum-size prime implicants. In: Proceedings of the International Workshop on Boolean Problems (1998)
23. Manquinho, V.M., Flores, P.F., Silva, J.P.M., Oliveira, A.L.: Prime implicant computation using satisfiability algorithms. In: ICTAI, pp. 232–239 (1997)
24. de Moura, L., Bjrner, N.: Satisfiability modulo theories: An appetizer. In: Formal Methods: Foundations and Applications, *LNCS*, vol. 5902. Springer (2009)
25. Palopoli, L., Pirri, F., Pizzuti, C.: Algorithms for selective enumeration of prime implicants. Artificial Intelligence **111**(1), 41 – 72 (1999)
26. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier and MIT Press (2001)
27. Tompkins, D.: Ubcsat home page. `http://www.satlib.org/ubcsat/` (2012)
28. Umans, C.: The minimum equivalent dnf problem and shortest implicants. In: FOCS, pp. 556–563 (1998)
29. Velev, M.N., Bryant, R.E.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. J. Symb. Comput. **35**(2), 73–106 (2003)