On Using Unsatisfiability for Solving Maximum Satisfiability

Joao Marques-Silva and Jordi Planes

School of Electronics and Computer Science, University of Southampton, UK
{jpms,jp3}@ecs.soton.ac.uk

Abstract. Maximum Satisfiability (MAXSAT) is a well-known optimization problem, with several practical applications. The most widely known MAXSAT algorithms are ineffective at solving hard problems instances from practical application domains. Recent work proposed using efficient Boolean Satisfiability (SAT) solvers for solving the MAXSAT problem, based on identifying and eliminating unsatisfiable subformulas. However, these algorithms do not scale in practice. This paper analyzes existing MAXSAT algorithms based on unsatisfiable subformula identification. Moreover, the paper proposes a number of key optimizations to these MAXSAT algorithms and a new alternative algorithm. The proposed optimizations and the new algorithm provide significant performance improvements on MAXSAT instances from practical applications. Moreover, the efficiency of the new generation of unsatisfiability-based MAXSAT solvers becomes effectively indexed to the ability of modern SAT solvers to proving unsatisfiability and identifying unsatisfiable subformulas.

1 Introduction

The problem of Maximum Satisfiability (MAXSAT) consists of identifying the largest number of clauses in a CNF formula that can be satisfied. Variations of the MAXSAT include partial MAXSAT and weighted MAXSAT. For partial MAXSAT some clauses (i.e. the hard clauses) must be satisfied whereas others (i.e. the soft clauses) may not be satisfied. For weighted MAXSAT, each clause has a given weight, and the objective is to maximize the sum of the weights of satisfied clauses.

The MAXSAT problem and its variations find a number of relevant practical applications, including design debugging of embedded systems [25] and FPGA routing [30]. Unfortunately, the techniques that have proved to be extremely effective in Boolean Satisfiability (SAT) cannot be applied directly to MAXSAT [2, 14]. As a result, most of the existing algorithms [10, 16, 17, 6] implement only a restricted number of techniques, emphasizing bound computation and/or dedicated inference techniques. Despite the extensive research work in this area, existing MAXSAT techniques and algorithms do not scale for large problem instances from practical applications.

Recent work [6] proposed alternative approaches, that build on the existence of effective SAT solvers for identifying unsatisfiable subformulas, and so can indirectly exploit existing effective SAT techniques [21, 22, 4]. However, even though modern SAT solvers are effective at proving unsatisfiability and generating unsatisfiable subformulas, the algorithms described in [6] are in general ineffective for MAXSAT, and so this work focused on partial MAXSAT with a reduced number of soft clauses.

This paper reviews previous MAXSAT algorithms based on identifying unsatisfiable subformulas for MAXSAT, proposes key optimizations to one of these algorithms [6], and develops a new algorithm also based on identifying unsatisfiable subformulas. Experimental results, obtained on a wide range of practical problem instances, show that the new MAXSAT algorithms can be orders of magnitude more efficient than the original algorithms [6], being in general consistently more efficient than previous MAXSAT solvers on instances obtained from practical applications.

The paper is organized as follows. Section 2 briefly introduces the MAXSAT problem and existing algorithms. Afterwards, Section 3 reviews MAXSAT algorithms based on unsatisfiable subformula identification [6]. Section 4 proposes optimizations to these algorithms, and Section 5 proposes a new MAXSAT algorithm. Experimental results on a large sample of problem instances, obtained from a number of practical applications, are analyzed in Section 6. The paper concludes in Section 7.

2 Preliminaries

This section provides definitions and background knowledge for the MAXSAT problem. Due to space constraints, familiarity with SAT and related topics is assumed [21, 4].

The maximum satisfiability (MAXSAT) problem can be stated as follows. Given an instance of SAT represented in Conjunctive Normal Form (CNF), compute an assignment to the variables that maximizes the number of satisfied clauses. Variations of the MAXSAT problem include the partial MAXSAT and the weighted MAXSAT problem. In the partial MaxSAT problem some clauses (i.e. the *hard* clauses) must be satisfied, whereas others (i.e. the *soft* clauses) may not be satisfied. In the weighted MaxSAT problem, each clause has a given weight, and the objective is to maximize the sum of the weights of satisfied clauses.

During the last decade there has been a growing interest on studying MAXSAT, motivated by an increasing number of practical applications, including scheduling, routing, bioinformatics, and design automation [30, 25]. Despite the clear relationship with the SAT problem, most modern SAT techniques cannot be applied directly to the MAXSAT problem [2, 14]. As a result, most MAXSAT algorithms are built on top of the standard DPLL [3] algorithm, and so do not scale for industrial problem instances [10, 16, 17, 6]. The most often used approach for MAXSAT (e.g. most of the solvers in the MAXSAT competition [1]) is based on a Branch and Bound algorithm, emphasizing the computation of a lower bound and the application of inference rules that simplify the instance [10, 16, 17]. Results from the MAXSAT competition [1] indicate that solvers based on Branch and Bound with additional inference rules are currently the most efficient MAXSAT solvers, outperforming all other existing approaches.

One alternative approach for solving the MAXSAT problem is to use Pseudo-Boolean Optimization (PBO) (e.g. [18]). The PBO approach for MAXSAT consists of adding a new (*blocking)* variable to each clause. The blocking variable b_i for clause ω_i allows satisfying clause ω_i independently of other assignments to the problem variables. The resulting PBO formulation includes a cost function, aiming the minimization of the number of blocking variables assigned value 1. Clearly, the solution of the MAXSAT problem is obtained by subtracting from the number of clauses the solution of the PBO problem.

Despite its simplicity, the PBO formulation does not scale for industrial problems, since the large number of clauses results in a large number of blocking variables, and corresponding larger search space. Observe that, for most instances from practical applications, the number of clauses largely exceeds the number of variables. For the resulting PBO problem, the number of variables equals the sum of the number of variables and clauses in the original SAT problem. Hence, the resulting instance of PBO has a much larger search space than the original instance of SAT.

Besides the PBO model, a number of alternative algorithms exist for MAXSAT. Examples include, `OPT-SAT` [8] and `sub-SAT` [30]. `OPT-SAT` imposes an ordering on the Boolean variables on an existing SAT solver. Experimental results for MAXSAT indicate that this approach is slower than a state-of-art PBO solver, e.g. `minisat+` [5], and so it is unlikely to scale for industrial problems. On the other hand, `sub-SAT` solves a relaxed version of the original problem, hence the exact MAXSAT solution may not be computed. Moreover, the experimental comparison in [6] suggests that `sub-SAT` is not competitive with unsatisfiability-based MAXSAT algorithms. Other approaches have been proposed [9], that are based on the relation of minimally unsatisfiable subformulas and maximally satisfiable subformulas [18, 9]. However, these approaches are based on enumeration of maximally satisfiable subformulas, and so do not scale for instances with a large number of unsatisfiable subformulas. As a result, for most instances only approximate results can be obtained. More recently, an alternative approximate approach to MAXSAT has been proposed [25]. The motivation for this alternative approach is the potential application of MAXSAT in design debugging, and the fact that existing MAXSAT approaches do not scale for industrial problem instances. However, this approach is unable to compute exact solutions to the MAXSAT problem.

The next section addresses MAXSAT algorithms that use the identification of unsatisfiable subformulas. Modern SAT solvers can be instructed to generate a resolution refutation for unsatisfiable formulas [32]. The resolution proof is usually represented as a proof trace, which summarizes the resolution steps used for creating each clause learnt by the SAT solver. Besides resolution refutations, proof traces allow identifying unsatisfiable subformulas, which serve as the source for the resolution refutation. A simple iterative procedure allows generating minimal unsatisfiable subformulas (MUS) from computed unsatisfiable sub-formulas [32]. All modern conflict-driven clause learning (CDCL) SAT solvers can be easily adapted to generate proof traces, and indirectly, unsatisfiable sub-formulas.

3 Unsatisfiability-Based MaxSat Algorithms

As mentioned in the previous section, one of the major drawbacks of the PBO model for MAXSAT is the large number of blocking variables that must be considered. The ability to reduce the number of required blocking variables is expected to improve significantly the ability of SAT/PBO based solvers for tackling instances of MAXSAT. Moreover, any solution to the MAXSAT problem will be unable to satisfy clauses that *must* be part of an unsatisfiable subformula. Consequently, one approach for reducing the number

Algorithm 1 The MAXSAT algorithm of Fu&Malik

MSU1(φ)

1 ▷ Clauses of CNF formula φ are the *initial* clauses
2 ▷ Clauses in φ are tagged non-auxiliary
3 $\varphi_W \leftarrow \varphi$ ▷ Working formula, initially set to φ
4 **while true**
5 **do** $(\text{st}, \varphi_C) \leftarrow \text{SAT}(\varphi_W)$
6 ▷ φ_C is an unsat core if φ_W is unsat
7 **if** st $=$ **UNSAT**
8 **then** $BV \leftarrow \emptyset$
9 **for each** $\omega \in \varphi_C$
10 **do if** ω is not auxiliary
11 **then** b is a new blocking variable
12 $\omega_B \leftarrow \omega \cup \{b\}$ ▷ ω_B is tagged non-auxiliary
13 $\varphi_W \leftarrow \varphi_W - \{\omega\} \cup \{\omega_B\}$
14 $BV \leftarrow BV \cup \{b\}$
15 $\varphi_B \leftarrow \text{CNF}(\sum_{b \in BV} b = 1)$ ▷ *One-Hot* constraint in [6]
16 $\varphi_w \leftarrow \varphi_W \cup \varphi_B$ ▷ Clauses in φ_B are tagged auxiliary
17 **else** ▷ Solution to MAXSAT problem
18 $\nu \leftarrow |$ blocking variables w/ value 1 $|$
19 **return** $|\varphi| - \nu$

of blocking variables is to associate blocking variables only with clauses that are part of unsatisfiable subformulas. However, it is not simple to identify all clauses that are part of unsatisfiable subformulas. One alternative is the identification and relaxation of unsatisfiable subformulas.

This section describes the unsatisfiability-based MAXSAT algorithm described in [6]. In what follows this algorithm is referred to as `msu1` (Fu&Malik's MAXSAT algorithm based on unsatisfiable subformulas). It should be observed that the original algorithm was proposed for partial MAXSAT, but the modifications for the plain MAXSAT problem are straightforward.

Algorithm 1 summarizes Fu&Malik's [6] MAXSAT algorithm. The algorithm iteratively finds unsatisfiable cores (line 5), adds new blocking variables to the non-auxiliary clauses in the unsatisfiable core (line 12), and requires that exactly one of the new blocking variables must be assigned value 1 (line 15). This constraint is referred to as the *One-Hot* constraint in [6]. The algorithm terminates whenever the CNF formula is satisfiable, and the number of assigned blocking variables is used for computing the solution to the MAXSAT problem instance.

The clauses used for implementing the *One-Hot* constraint are declared auxiliary; all other clauses are non-auxiliary. Observe that each non-auxiliary clause may receive more than one blocking variable, and the total number of blocking variables a clause receives corresponds to the number of times the clause is part of an unsatisfiable core. As suggested earlier in this section, by focusing on identification and relaxation (with blocking variables) of unsatisfiable sub-formulas, `msu1` and the other algorithms described later attempt to reduce the number of blocking variables that is necessary to use while solving the MAXSAT problem.

A proof of correctness of algorithm `msu1` is given in [6]. However, [6] does not address important properties of the algorithm, including the number of blocking variables that must be used in the worst case, or the worst-case number of iterations of the algorithm. This section establishes some of these properties. In what follows, n denotes the number of variables and m denotes the number of clauses.

Proposition 1. *During the execution of Algorithm 1, non-auxiliary clauses can have multiple blocking variables.*

Proof: *Consider the following example CNF formula:*

$$\begin{array}{l}(x_1) \wedge (\neg x_1 \vee \neg y_1) \wedge (y_1) \wedge (\neg x_1 \vee \neg z_1) \wedge (\neg y_1 \vee \neg z_1) \\ (x_2) \wedge (\neg x_2 \vee \neg y_2) \wedge (y_2) \wedge (\neg x_2 \vee \neg z_1) \wedge (\neg y_2 \vee \neg z_1) \\ (z_1 \vee z_2) \wedge (z_1 \vee \neg z_2)\end{array}$$

One possible execution of the algorithm follows. Identify core $(x_1) \wedge (\neg x_1 \vee \neg y_1) \wedge (y_1)$*. Add blocking clauses, respectively* b_1, b_2, b_3*, and require* $b_1 + b_2 + b_3 = 1$*. Identify core* $(x_2) \wedge (\neg x_2 \vee \neg y_2) \wedge (y_2)$*. Add blocking clauses, respectively* b_4, b_5, b_6*, and require* $b_4 + b_5 + b_6 = 1$*. Identify core* $(x_1 \vee b_1) \wedge (y_1 \vee b_3) \wedge (\neg x_1 \vee \neg z_1) \wedge (\neg y_1 \vee \neg z_1) \wedge (z_1 \vee z_2) \wedge (z_1 \vee \neg z_2) \wedge \varphi_e$*, where* φ_e *denotes clauses from encoding* $b_1 + b_2 + b_3 = 1$ *in CNF. Add blocking clauses to non-auxiliary clauses, respectively* $b_7, b_8, b_9, b_{10}, b_{11}, b_{12}$*, and require* $b_7 + b_8 + b_9 + b_{10} + b_{11} + b_{12} = 1$*. At this stage, some of the non-auxiliary clauses have two blocking variables, e.g.* b_1 *and* b_7 *are associated with* (x_1)*.* ■

Proposition 2. *During the execution of Algorithm 1, for iteration* j*, exactly* $j-1$ *blocking variables must be assigned value 1, or the formula is unsatisfiable.*

Proof: *Observe that each iteration adds a constraint requiring the sum of a set of new blocking variables to be equal to 1. Hence, at iteration* j*, either* $j-1$ *blocking variables are assigned value 1, or the formula is unsatisfiable.* ■

Proposition 3. *During the execution of Algorithm 1, if* φ_W *is satisfiable, at most 1 of the blocking variables associated with a given clause can be assigned value 1.*

Proof: *Each blocking variable is associated with a clause as the result of identifying an unsatisfiable core. Consider clause* ω_i *that is part of two cores* c_1 *and* c_2*, each adding to* ω_i *a blocking variable, respectively* $b_{i,1}$ *and* $b_{i,2}$*. Assume that the formula could be satisfied such that* ω_i *would have the two blocking variables* $b_{i,1}$ *and* $b_{i,2}$ *assigned value 1. This would imply that both cores* c_1 *and* c_2 *could be deactivated by blocking clause* ω_i*. But this would also imply that the second core* c_2 *could not have been identified, since assigning* $b_{i,1}$ *would deactivate core* c_2*; a contradiction.* ■

This result allows deriving an upper bound on the number of iterations of Algorithm 1.

Proposition 4. *The number of iterations of Algorithm 1 is* $\mathcal{O}(m)$*.*

Proof: *Immediate from Propositions 2 and 3. At each iteration* j*,* $j-1$ *blocking variables must be assigned value 1. Moreover, none of these blocking variables can be from* *the same clause. Hence, at iteration $m + 1$ all clauses must be satisfied by assigning a blocking variable to 1. Hence, the number of iterations of Algorithm 1 is $\mathcal{O}(m)$.* ■

It should be observed that the algorithm will *never* execute $m + 1$ steps. Indeed, for arbitrary CNF formulas, at least half of the clauses can be trivially satisfied [12], and so the number of iterations never exceeds $\frac{m}{2} + 1$. Moreover, the upper bound on the number of iterations serves for computing an upper bound on the total number of blocking variables.

Proposition 5. *During the execution of Algorithm 1, the number of blocking variables is $\mathcal{O}(m^2)$ in the worst case.*

Proof: *From Proposition 4 the number of iterations is $\mathcal{O}(m)$. In each iteration, each unsatisfiable core can have at most m clauses (i.e. the number of original clauses). Hence the result follows.* ■

The previous result provides an upper bound on the number of blocking variables. A tight lower bound is not known, even though a trivial lower bound is $\Omega(m)$.

4 Optimizing Unsatisfiability-Based MAXSAT Algorithms

This section proposes improvements to Fu&Malik's MAXSAT algorithm [6] described in the previous Section. The resulting algorithm is referred to as `msu2`.

4.1 Encoding Cardinality Constraints

The *one-hot* constraint used in `msu1` [6] corresponds to the well-known pairwise encoding for Equals 1 constraints [7], i.e. cardinality constraints of the form $\sum_{i=1}^{r} b_i = 1$. Usually, Equals 1 constraints are encoded with two constraints, one AtMost 1 constraint (i.e. $\sum_{i=1}^{r} b_i \leq 1$) and one AtLeast 1 constraint (i.e. $\sum_{i=1}^{r} b_i \geq 1$). It is also well-known that the pairwise encoding requires $\frac{r\,(r-1)}{2} + 1$ clauses, one clause for the AtLeast 1 constraint, and $\frac{r\,(r-1)}{2}$ binary clauses for the AtMost 1 constraint. Hence, the quadratic number of clauses results from encoding the AtMost 1 constraint. For large r, as is often the case for the MAXSAT problem, the pairwise encoding can require a prohibitively large number of clauses. For example, for an unsatisfiable core with 10,000 clauses, the resulting AtMost 1 constraint is encoded with 49,995,000 binary clauses. For practical applications, unsatisfiable cores are likely to exceed 10,000 clauses. As shown in Section 6, in many cases, the pairwise encoding of an AtMost 1 constraint exhausts the available physical memory resources.

There are a number of alternatives to the pairwise encoding of AtMost 1 constraints [29, 7, 26, 5], all of which are linear in the number of variables in the constraint. These encodings can either use sequential counters, sorters, or binary decision diagrams (BDDs). One simple alternative is to use BDDs for encoding a cardinality constraint. A Boolean circuit is extracted from the BDD representation, which can then be converted to CNF using Tseitin's encoding [28]. In most cases, the encoding takes into account the polarity optimizations of Plaisted and Greenbaum [24, 5] when generating the CNF formula. For the AtMost 1 constraint, the BDD-based encoding of a cardinality constraint is linear in n [5]. For the results in Section 6, the most significant performance gains are obtained from using a BDD-based encoding for AtMost 1 constraints, using Tseitin's encoding and Plaisted&Greenbaum's polarity optimizations.

One final remark is that Fu&Malik's algorithm will also work if only AtMost 1 constraints are used instead of Equals 1 constraints. This allows saving one (possibly quite large) clause in each iteration of the algorithm.

4.2 Blocking Variables Associated with each Clause

Another potential drawback of Fu&Malik's algorithm is that there can be several blocking variables associated with a given clause (see the analysis of Algorithm 1, including Propositions 1 and 5). Each time a clause ω is part of a computed unsatisfiable core, a *new* blocking variable is added to ω. Observe that correctness of the algorithm requires that more than one blocking variable may be associated with each clause. On the other hand, despite the potentially large (but at most linear in m) number of blocking variables associated with each clause, *at most* one of these additional blocking variables can be used for actually preventing the clause from participating in an unsatisfiable core (see Proposition 3).

A simple improvement for pruning the search space associated with blocking variables is to require that *at most one* of the blocking variables associated with a given clause ω can be assigned value 1. For a given clause ω_i, let $b_{i,j}$ be the blocking variables associated with ω_i. As a result, the condition that at most 1 of the blocking variables of ω_i is assigned value 1 is given by:

$$\sum_j b_{i,j} \leq 1 \tag{1}$$

In general, the previous condition is useful when Algorithm 1 must execute a large number of iterations, and many clauses are involved in a significant number of unsatisfiable cores.

Example 1. Consider the example given in the proof of Proposition 1. In the third iteration of the algorithm, the first clause (x_1) has been modified to $(x_1 \vee b_1 \vee b_7)$. As a result, the CNF encoding of the additional constraint $b_1 + b_7 \leq 1$ can be added to the CNF formula. Since this is an AtMost 1 constraint, the encoding proposed in the previous section can also be used.

5 A New Unsatisfiability-Based MAXSAT Algorithm

This section proposes a new alternative algorithm for MAXSAT. Compared to the algorithms described in the previous sections, `msu1` and `msu2`, the new algorithm guarantees that *at most 1* blocking variable is associated with each clause. As a result, the worst case number of blocking variables that can be used is m. Moreover, during a first phase, the new algorithm extracts identified cores, whereas in a second phase the algorithm

Algorithm 2 A new MAXSAT algorithm

```
MSU3($\varphi$)
1  $\varphi_W \leftarrow \varphi$                              ▷ Working formula, initially set to $\varphi$
2  UC $\leftarrow \emptyset$
3  while true                                                  ▷ Phase 1: Identify disjoint cores
4      do $(\text{st}, \varphi_C) \leftarrow \text{SAT}(\varphi_W)$   ▷ $\varphi_C$ is an unsat core if $\varphi_W$ is unsat
5         if st $=$ UNSAT
6           then $\varphi_W \leftarrow \varphi_W - \varphi_C$
7                UC $\leftarrow$ UC $\cup \{\varphi_C\}$
8           else break                                         ▷ Move to 2nd loop
9  BV $\leftarrow \emptyset$
10 for each $\varphi_C \in$ UC                                 ▷ Add blocking variables
11     do for each $\omega \in \varphi_C$
12            do $b$ is a new blocking variable
13               $BV \leftarrow BV \cup \{b\}$
14               $\varphi_W \leftarrow \varphi_W \cup \{\omega \cup \{b\}\}$
15 $\lambda = |\text{UC}|$                                     ▷ Lower bound on true blocking variables
16 $\varphi_B \leftarrow \text{CNF}(\sum_{b \in BV} b = \lambda)$
17 $\varphi_W \leftarrow \varphi_W \cup \varphi_B$             ▷ Current cardinality constraint
18 while true                                                  ▷ Phase 2: Increment lower bound $\lambda$
19     do $(\text{st}, \varphi_C) \leftarrow \text{SAT}(\varphi_W)$   ▷ $\varphi_C$ is an unsat core if $\varphi_W$ is unsat
20        if st $=$ UNSAT
21          then $\lambda \leftarrow \lambda + 1$
22               for each $\omega \in \varphi_C$
23                   do if $\omega$ has no blocking variable
24                        then $b$ is new blocking variable
25                             $\omega_B \leftarrow \omega \cup \{b\}$          ▷ $\omega_B$ is tagged non-auxiliary
26                             $\varphi_W \leftarrow \varphi_W - \{\omega\} \cup \{\omega_B\}$
27                             $BV \leftarrow BV \cup \{b\}$
28               $\varphi_W \leftarrow \varphi_W - \varphi_B$
29               $\varphi_B \leftarrow \text{CNF}(\sum_{b \in BV} b = \lambda)$   ▷ New cardinality constraint
30               $\varphi_W \leftarrow \varphi_W \cup \varphi_B$      ▷ Clauses in $\varphi_B$ are tagged auxiliary
31          else return $|\varphi| - \lambda$                  ▷ Solution to MAXSAT problem
```

addresses the problem of computing the number of blocking variables that must be assigned value 1. The objective of the first phase is to simplify identification of disjoint unsatisfiable cores.

Algorithm 2 shows the new MAXSAT algorithm. The first phase of the algorithm is shown in lines 3 to 8. During this phase disjoint cores are identified and removed from the formula. The first set of blocking variables are associated with each clause in an unsatisfiable core in lines 9 to 14. The second phase of the algorithm is shown in lines 18 to 31. During this phase the lower bound on the number of blocking variables assigned value 1 is iteratively increased until the CNF formula becomes satisfiable. For each identified unsatisfied core, a unique blocking variable is associated with non-auxiliary clauses that do not have a blocking variable. The cardinality constraint $\sum b_i = k$ is encoded with one AtLeast k ($\sum b_i \leq k$) and one AtMost k ($\sum b_i \geq k$) constraints. As with `msu2`, these constraints are represented with BDDs and converted to CNF using

Tseitin's transformation [28] and including the polarity optimizations of Plaisted and Greenbaum [24, 5]. In this case the size of the representation if $\mathcal{O}(r \cdot k)$, where r is the number of variables [5] and k is the cardinality constraint bound.

Despite msu3 guaranteeing that the number of blocking variables never exceeds m, there are a few potential drawbacks. The AtLeast k and AtMost k cardinality constraints used by msu3 are significantly more complex to encode than the simple AtMost 1 constraint used by msu1 and msu2. As a result, msu3 is expected to perform better when the MAXSAT solution is not far from the total number of clauses.

As mentioned earlier for msu1, Algorithm 2 can use AtMost k cardinality constraints instead of Equals k constraints. Finally, algorithm msu2 also allows evaluating whether two phases can be useful for solving MAXSAT. Clearly, the algorithm could easily be modified to also use only one phase.

6 Experimental Evaluation

This section evaluates a number of MAXSAT solvers on industrial test cases. Most instances are obtained from unsatisfiable industrial instances used in past SAT competitions [15] or available from SATLIB [11]. The classes of instances considered were the following:

1. Bounded model checking sintances from IBM [31]. The problem instances were restricted to unsatisfiable instances, up to 35 computation steps, for a total of 252.
2. Instances from the parametrized pipelined-processor verification problem [19]. The problem instances were restricted to the smallest 58 instances.
3. Verification of out-of-order microprocessors, from UCLID [13]. 31 unsatisfiable instances were considered.
4. Circuit testing instances [11]. 228 unsatisfiable instances were considered.
5. Automotive product configuration [27]. 84 unsatisfiable instances were considered.

In addition, instances from design debugging [25] (29 unsatisfiable instances) and FPGA routing [30] (16 unsatisfiable instances) were also considered. These MAXSAT instances are known to be difficult, and most have no known MAXSAT solutions. As a result, the total number of problem instances used in the experiments was 698.

The MAXSAT solvers considered were the following: the best performing solver in the MAXSAT 2007 evaluation [1], maxsatz [16, 17], a PBO formulation of the MAXSAT problem solved with minisat+, one of the best performing PBO solvers [5, 20], an implementation of the algorithm based on identification of unsatisfiable cores (msu1) [6], msu1 with the improvements proposed in Section 4 (msu2), and the new MAXSAT algorithm described in Section 5 (msu3). msu1, msu2 and msu3 are built on top of the same unsatisfiable core extractor, implemented with minisat 1.14 [4].

Other alternative MAXSAT algorithms were not considered [8, 30, 9, 18]. Existing results for OPT-SAT indicate that it is not competitive with the PBO model solved with minisat+. On the other hand, both sub-SAT [30] and HYCAM [9] only compute approximate solutions. Moreover, results from [9] also show that existing approaches based on enumerating all minimally unsatisfiable subformulas [18] are not competitive.

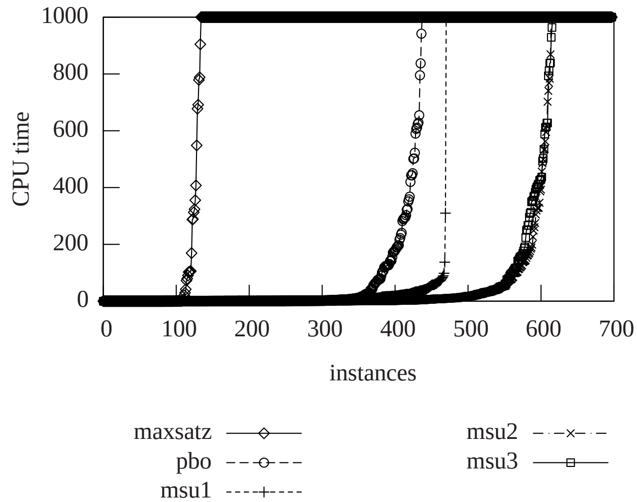

Fig. 1. Run times on all instances

Table 1. Number of aborted instances, out of a total of 698 instances

MAXSAT solver	maxsatz	PBO	msu1	msu2	msu3
Aborted instances	564	261	228	84	82

With respect to the PBO model, `minisat+` was configured to use sorters for the cost function. The reason for using sorters is that for many problem instances the use of BDDs would exhaust the available physical memory.

The results for all MAXSAT solvers on all problem instances were obtained on a Linux server running RHE Linux, with a Xeon 5160 3.0 GHz dual-core processor. For the experiments, the available physical memory of the server was 2 GByte. The time limit was set to 1000 seconds per instance.

Figure 1 plots the run times of each solver sorted by increasing run times. As can be observed, the performance difference for the MAXSAT solvers considered is significant. `msu2` and `msu3` solve many more problem instances than any of the other solvers. As can also be observed in Figure 1, `msu1` exhibits a sharp transition between instances it can solve and instances it is unable to solve. The reason is due to the size of the computed unsatisfiable cores. For the more complex instances, the size of the cores is significant, and so `msu1` most often aborts due to excessive memory requirements.

A summary of the number of aborted instances is shown in Table 1. Over all instances, `msu2` aborts 2 more instances than `msu3` (respectively 84 vs. 82), and both abort significantly less instances than any of the other solvers. msu1 aborts 146 more instances than `msu3` and 144 more instances than `msu2`. Somewhat surprisingly, the PBO model performs reasonably well when compared with `msu1`. As might be expected, `maxsatz` aborts most industrial instances.

Figure 2 compares `msu1` with `maxsatz` and PBO, respectively. As can be seen, with the exception of a few outliers, always taking negligible CPU time, `msu1` performs

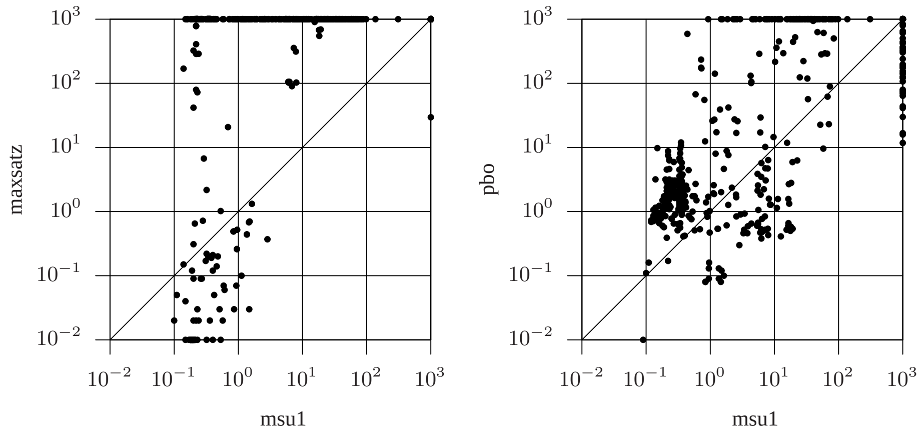

Fig. 2. Scatter plots on all instances maxsatz and PBO vs. `msu1`

significantly better than `maxsatz`. In contrast, `msu1` does not clearly outperform PBO. This is in part explained by the quadratic encoding of AtMost 1 constraints in `msu1`, which causes a significant number of instances to abort. In addition, the PBO model uses the most recent version of `minisat+`, whereas the unsatisfiable core extractor used in all versions of the `msu` algorithm is based on `minisat` 1.14.

Figure 3 compares `msu2` against PBO and `msu1`. The performance difference is clear. `msu2` outperforms `msu1` in almost all problem instances. For a very small number of examples, `msu1` can outperform `msu2`, but the differences are essentially negligible, never exceeding a small percentage of the total run time. `msu2` also clearly outperforms PBO, aborting a fraction (close to 20%) of the instances aborted by PBO. However, a few outliers exist, and these are explained by the fact that PBO uses the most recent version of `minisat+`, and `msu2` uses an unsatisfiable core extractor based on `minisat` 1.14.

Figure 4 compares `msu3` against `msu1` and `msu2`. `msu1` performs significantly worse than `msu3`, with a few outliers, only one of which is relevant. In contrast, `msu2` and `msu3` perform similarly, even though `msu2` usually exhibits smaller run times. Nevertheless, the results also suggest that `msu3` can be an interesting alternative to `msu2` for a reasonable number of problem instances.

The previous results provide clear evidence that unsatisfiability-based MAXSAT algorithms are effective for solving problem instances obtained from industrial settings. However, several of the problem instances considered, albeit unsatisfiable, do not represent problems originally formulated as MAXSAT problems. Recent work has shown that MAXSAT has practical application in FPGA routing [30] and system debugging [25]. However, and motivated by the limitations of existing MAXSAT solvers, these MAXSAT instances were solved with approximate algorithms. Our results indicate that unsatisfiability-based MAXSAT algorithms are very efficient at solving problem instances from design debugging, but less effective at solving FPGA routing instances. The class FPGA contains 16 unsatisfiable instances. Of these 16, `msu2` solves 3, `msu1`

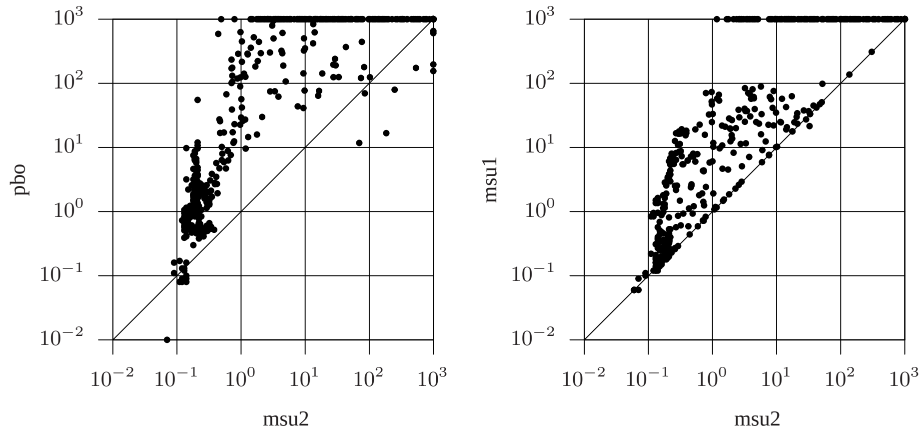

Fig. 3. Scatter plots on all instances, PBO and msu1 vs. msu2

solves 2, `msu3` solves 1, and `maxsatz` solves 1. It is well-known that SAT instances from FPGA routing have specific structure, that makes them quite difficult for SAT solvers [23, 30]. This in part explains the results of all MAXSAT solvers on these instances.

For the design debugging instances the results are quite different. Table 6 shows the CPU times for all MAXSAT solvers on all design debugging instances [25]. As before, the unsatisfiability-based MAXSAT algorithms perform remarkably better than the other algorithms, `maxsatz` and the PBO model. In addition, and also as before, `msu2` is the best performing algorithm, and aborts only one instance. `msu3` also aborts only one instance, but in general performs worse than `msu2`. Finally, `msu1` aborts 2 instances, and performs worse than `msu2` for almost all instances. For instances with large unsatisfiable cores (e.g. b15-bug-onevec-gate-0) the linear encoding used in `msu2` ensures manageable size representations of the AtMost 1 constraints. The same holds true to `msu3`, for small values of k. In contrast, `msu1` uses a quadratic encoding and so it often aborts instances with large unsatisfiable cores. Moreover, for instances requiring the identification of several unsatisfiable cores sharing common clauses, the additional constraints proposed in Section 4.2 are useful for `msu2`. It should be observed that the only design debugging instance that is aborted by both `msu2` and `msu3` is also aborted by the unsatisfiable core extractor, again suggesting that performance of unsatisfiability-based MAXSAT solvers is indexed to the efficiency of SAT solvers.

7 Conclusions

Recent work has shown that MAXSAT has a number of significant practical applications [25]. However, current state of the art MAXSAT solvers are ineffective on most problem instances obtained from practical applications.

This paper focus on solving MAXSAT problem instances obtained form practical applications, and conducts a detailed analysis of MAXSAT algorithms based on unsat-

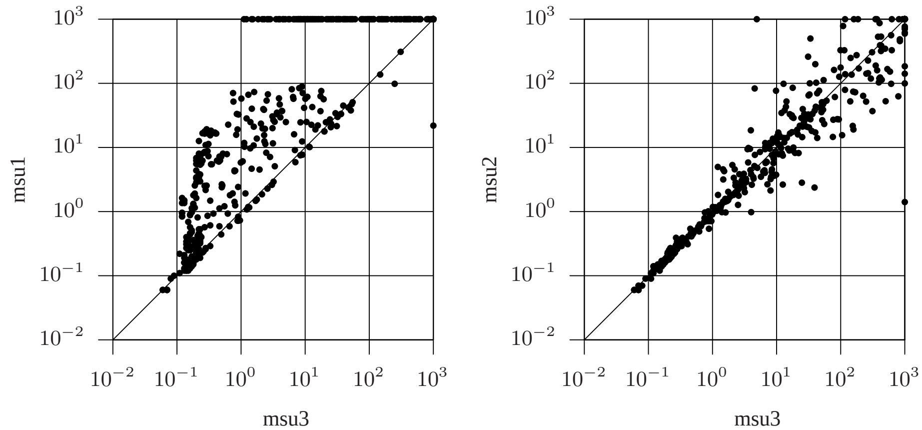

Fig. 4. Scatter plots on all instances, msu1 and msu2 vs. msu3

isfiable subformula identification. Moreover, the paper develops improvements to existing algorithms and proposes a new MAXSAT algorithm. The proposed improvements (`msu2`) and new algorithm (`msu3`) provide significant performance improvements, and allow indexing the hardness of solving practical instances of MAXSAT to the ability of modern SAT solvers for proving unsatisfiability and identifying unsatisfiable subformulas. The algorithms described in this paper are by far the most effective for instances obtained from practical applications, clearly outperforming existing state of the art MAXSAT solvers, and further improvements are to be expected.

Despite the promising results of the new MAXSAT algorithms, a number of research directions can be envisioned. As the experimental results show, the role of encoding cardinality constraints is significant, and an extensive evaluation of alternative encodings should be considered. The unsatisfiable core extractor is based on `minisat` 1.14. A core extractor based a more recent SAT solver is expected to improve the efficiency of `msu1`, `msu2` and `msu3`. Finally, the problem instances for the FPGA routing problem are still challenging, even though the new MAXSAT algorithms can solve some of these instances, and so motivate the development of further improvements to MAXSAT algorithms.

Acknowledgement. This work is partially supported by EU project IST/033709 and by EPSRC grant EP/E012973/1.

References

1. J. Argelich, C. M. Li, F. Manyà, and J. Planes. MaxSAT evaluation. www.maxsat07.udl.es.
2. M. L. Bonet, J. Levy, and F. Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.
3. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

Table 2. Results for design debugging instances

Instance	#V	#C	solution	maxsatz	pbo	msu1	msu2	msu3
b14_opt_bug2_vec1-gt-0	130328	402707	402706	–	–	10.12	**9.79**	11.74
b15-bug-4vec-gt-0	581064	1712690	–	–	–	–	–	–
b15-bug-1vec-gt-0	121836	359040	359039	–	–	–	**27.43**	93.47
c1_DD_s3_f1_e2_v1-bug-4vec-gt-0	391897	989885	989881	–	–	**41.90**	**41.90**	47.29
c1_DD_s3_f1_e2_v1-bug-1vec-gt-0	102234	258294	258293	–	–	**5.89**	5.92	7.04
c2_DD_s3_f1_e2_v1-bug-4vec-gt-0	400085	1121810	1121806	–	–	98.25	**51.89**	249.42
c2_DD_s3_f1_e2_v1-bug-1vec-gt-0	84525	236942	236941	–	–	16.03	**5.72**	6.72
c3_DD_s3_f1_e1_v1-bug-4vec-gt-0	33540	86944	86940	–	–	2.92	**2.84**	3.21
c3_DD_s3_f1_e1_v1-bug-1vec-gt-0	8385	21736	21735	–	590.12	**0.44**	**0.44**	0.49
c4_DD_s3_f1_e1_v1-bug-gt-0	797728	2011216	2011208	–	–	137.19	**136.72**	148.02
c4_DD_s3_f1_e2_v1-bug-4vec-gt-0	448465	1130672	1130668	–	–	47.23	**47.22**	53.20
c4_DD_s3_f1_e2_v1-bug-1vec-gt-0	131584	331754	331753	–	–	7.70	**7.62**	9.06
c5315-bug-gt-0	1880	5049	5048	169.44	3.18	**0.14**	**0.14**	0.15
c5_DD_s3_f1_e1_v1-bug-4vec-gt-0	100472	270492	270488	–	–	10.26	**10.20**	11.49
c5_DD_s3_f1_e1_v1-bug-gt-0	200944	540984	540976	–	–	**33.26**	33.32	36.25
c5_DD_s3_f1_e1_v1-bug-1vec-gt-0	25118	67623	67622	–	–	1.47	**1.46**	1.69
c5_DD_s3_f1_e1_v2-bug-gt-0	200944	540984	540976	–	–	**33.22**	33.32	36.17
c6288-bug-gt-0	3462	9285	9284	–	17.12	6.01	0.54	**0.45**
c6_DD_s3_f1_e1_v1-bug-4vec-gt-0	170019	454050	454046	–	–	17.79	**17.63**	19.91
c6_DD_s3_f1_e1_v1-bug-gt-0	298058	795900	795892	–	–	50.71	**50.69**	54.98
c6_DD_s3_f1_e1_v1-bug-1vec-gt-0	44079	117720	117719	–	–	2.61	**2.57**	3.03
c6_DD_s3_f1_e2_v1-bug-4vec-gt-0	170019	454050	454046	–	–	17.81	**17.62**	19.95
c7552-bug-gt-0	2640	7008	7007	–	55.06	0.81	**0.21**	**0.21**
mot_comb1._red-gt-0	2159	5326	5325	0.15	1.08	**0.14**	**0.14**	0.16
mot_comb2._red-gt-0	5484	13894	13893	6.71	2.58	**0.29**	**0.29**	0.33
mot_comb3._red-gt-0	11265	29520	29519	–	67.43	**0.59**	**0.59**	0.66
s15850-bug-4vec-gt-0	88544	206252	206248	–	–	**7.55**	7.63	8.42
s15850-bug-1vec-gt-0	22136	51563	51562	–	26.01	1.08	**1.07**	1.23
s38584-bug-1vec-gt-0	314272	819830	819829	–	–	23.43	**21.12**	25.41

4. N. Een and N. Sörensson. An extensible SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, May 2003.
5. N. Een and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, March 2006.
6. Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265, 2006.
7. I. Gent. Arc consistency in SAT. In *European Conference on Artificial Intelligence*, pages 121–125, July 2002.
8. E. Giunchiglia and M. Maratea. Solving optimization problems with DLL. In *European Conference on Artificial Intelligence*, pages 377–381, 2006.
9. É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *International Joint Conference on Artificial Intelligence*, pages 2300–2305, 2007.
10. F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSat: a new weighted Max-SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 41–55,

2007.
11. H. Hoos and T. Stützle. SATLib. http://www.satlib.org/.
12. D. S. Johnson. Approximation algorithms for combinatorial problems. In *ACM Symposium on Theory of Computing*, pages 1–9, 1973.
13. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in CAD*, pages 142–159, 2002.
14. J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, 2007. In press.
15. D. Le Berre, L. Simon, and O. Roussel. SAT competition. www.satcompetition.org.
16. C. M. Li, F. Manyà, and J. Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *National Conference on Artificial Intelligence*, 2006.
17. C. M. Li, F. Manyà, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 2007. In press.
18. M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 173–186, June 2005.
19. P. Manolios and S. K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 363–366, October 2005.
20. V. Manquinho and O. Roussel. Pseudo-Boolean evaluation. www.cril.univ-artois.fr/PB07.
21. J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.
22. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.
23. G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based boolean SAT. In *International Symposium on Field-Programmable Gate Arrays*, February 1999.
24. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
25. S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer-Aided Design*, November 2007.
26. C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 827–831, October 2005.
27. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, 2003.
28. G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.
29. J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
30. H. Xu, R. A. Rutenbar, and K. A. Sakallah. sub-SAT: a formulation for relaxed boolean satisfiability with applications in routing. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(6):814–820, 2003.
31. E. Zarpas. Benchmarking SAT solvers for bounded model checking. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 340–354, June 2005.
32. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Testing in Europe Conference*, March 2003.