Generalizing Core-Guided Max-SAT

Mark H. Liffiton and Karem A. Sakallah

Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2121
{liffiton,karem}@eecs.umich.edu

Abstract. Recent work has shown the value of using unsatisfiable cores to guide maximum satisfiability algorithms (Max-SAT) running on industrial instances [5,9,10,11]. We take this concept and generalize it, applying it to the problem of finding minimal correction sets (MCSes), which themselves are generalizations of Max-SAT solutions. With the technique's success in Max-SAT for industrial instances, our development of a generalized approach is motivated by the industrial applications of MCSes in circuit debugging [12] and as a precursor to computing minimal unsatisfiable subsets (MUSes) in a hardware verification system [1]. Though the application of the technique to finding MCSes is straightforward, its correctness and completeness are not, and we prove both for our algorithm. Our empirical results show that our modified MCS algorithm performs better, often by orders of magnitude, than the existing algorithm, even in cases where the use of cores has a negative impact on the performance of Max-SAT algorithms.

1 Introduction

In the field of constraint processing, and particularly within the domain of Boolean Satisfiability (SAT), the analysis of infeasible constraint systems has become increasingly important. Following the impressive advancements in the performance of SAT solvers in the past decade, which enable fast answers about the satisfiability of industrially relevant instances, researchers have begun to look at analyses beyond the "unsatisfiable" result returned for overconstrained instances. The work is spurred not only by academic interest but also by novel industrial applications of these analyses. In this paper, we take one of the recent advances in analyzing infeasible instances, namely unsatisfiable-core-guided maximum satisfiability (core-guided Max-SAT), and generalize it to solve a related analysis with direct industrial applications: the identification of minimal correction sets (MCSes).

The concept of core-guided Max-SAT was first developed by Fu & Malik [5] and later enhanced and optimized by Marques-Silva and others [9,10,11]; the algorithms and differences in their approaches are detailed in Section 3. The technique relies on and exploits one of the relationships between satisfiable and unsatisfiable subsets of infeasible systems that have been explored in depth in [3] and [6,7]. Briefly, an unsatisfiable instance will contain one or more *unsatisfiable*

cores. No satisfiable subset of such an instance can contain any complete cores; therefore, any Max-SAT solution must must leave unsatisfied at least one clause from every core. The core-guided Max-SAT approach thus identifies unsatisfiable cores of an instance and only considers clauses within those cores as potential "removals," limiting the search space dramatically.

The use of unsatisfiable cores in solving Max-SAT yields drastically different performance than other current Max-SAT techniques, which are generally based on branch-and-bound. In the 2008 Max-SAT Evaluation [2], core-guided Max-SAT algorithms performed extremely well in the industrial Max-SAT category (one solving 72 of 112 instances within the timeout, when other approaches solved 0-3 and, in one case, 10 instances within the timeout), while performing among the bottom of the pack on random and crafted instances.

The industrial Max-SAT instances in the Max-SAT Evaluation are in fact produced by the circuit debugging system in [12], in which the desired result is actually MCSes of the CNF instances. In that work, an algorithm simply called MCSes [7] is used as a preprocessing step, identifying approximations of MCSes which are then used to boost a complete SAT-based search. In this work, motivated by the success of core-guided Max-SAT on these instances, we generalize the core-guided Max-SAT approach to apply it to the problem of finding MCSes of CNF instances. Our new algorithm, MCSes-U, is described and its correctness proven in Section 4, and we present experimental results showing its improvement over MCSes in Section 5.

2 Preliminaries

Boolean satisfiability (SAT) is a problem domain involving Boolean formulas in *conjunctive normal form* (CNF). Formally, a CNF formula φ is defined as follows:

$$\varphi = \bigwedge_{i=1\ldots n} C_i \qquad\qquad C_i = \bigvee_{j=1\ldots k_i} a_{ij}$$

where each *literal* a_{ij} is either a positive or negative instance of some Boolean variable (e.g., x_3 or $\neg x_3$, where the domain of x_3 is $\{0, 1\}$), the value k_i is the number of literals in the *clause* C_i (a disjunction of literals), and n is the number of clauses in the formula. In more general terms, each clause is a *constraint* of the constraint system φ. We will often treat CNF formulas as sets of clauses (*clause sets*), so equivalently: $\varphi = \bigcup_{i=1\ldots n} C_i$.

A CNF instance is said to be *satisfiable* if there exists some assignment to its variables that makes the formula evaluate to 1 or TRUE; otherwise, it is *unsatisfiable*. We will use the following unsatisfiable CNF instance φ as an example.

$$\varphi = (a)(\neg a)(\neg a \vee b)(\neg b)$$

Maximum Satisfiability (Max-SAT) is the problem of, given an unsatisfiable CNF formula, identifying a satisfiable subset of its clauses with maximum cardinality. Alternatively, we can say it is the problem of finding an assignment of the formula's variables that satisfies a maximum cardinality subset of the clauses. The example formula φ has a single Max-SAT solution: $\{(\neg a), (\neg a \vee b), (\neg b)\}$ are satisfied by $a = \mathrm{F}, b = \mathrm{F}$.

Minimal Correction Sets (MCSes) can be understood as generalizations of Max-SAT solutions. Given any Max-SAT solution in the form of a satisfiable subset of clauses, we can look at those clauses left unsatisfied as a *correction set* (CS), because removing them from the formula *corrects* it, making it satisfiable. Due to the maximum cardinality of a Max-SAT solution, its corresponding correction set has minimum cardinality; no smaller correction sets exist. We generalize this to the concept of minim**al** correction sets: An MCS is a correction set such that all of its proper subsets are *not* correction sets. MCSes are minimal, or irreducible, but not necessarily minim**um**. Every Max-SAT solution indicates an MCS, but there can be more MCSes than those that are complements of a Max-SAT solution. The clause (a), not satisfied in φ's Max-SAT solution, is an MCS, as are $\{(\neg a), (\neg a \vee b)\}$ and $\{(\neg a), (\neg b)\}$.

Unsatisfiable Cores / MUSes: Given an unsatisfiable CNF formula, an *unsatisfiable core* of the formula is any subset of its clauses that is unsatisfiable. A *Minimal Unsatisfiable Subset* (MUS) is then an unsatisfiable core that is minimal in the same sense that an MCS is minimal: every proper subset of an MUS is satisfiable. MUSes are thus minimal/irreducible, but, again, not necessarily minim**um**. There are two MUSes in φ: $\{(a), (\neg a)\}$ and $\{(a), (\neg a \vee b), (\neg b)\}$.

Resolution Proofs: In the process of solving unsatisfiable instances, some SAT solvers produce *resolution proofs* (or *resolution refutations*), directed acyclic graphs containing the resolution steps used to prove unsatisfiability. As a solver progresses, it learns new clauses arising from applying resolution to combinations of existing clauses (e.g., $(x_2 \vee x_5) \wedge (\neg x_5 \vee x_7) \rightarrow (x_2 \vee x_7)$), and the "parent" clauses of each new clause can be stored in a graph structure. With this structure, the provenance of any learned clause can be traced back to a subset of the original clauses from which the learned clause can be derived. When a solver "learns" the empty clause, the instance must be unsatisfiable, and tracing the empty clause's parents back to the original clauses identifies an unsatisfiable core of the instance [17] (the identified core must be unsatisfiable because those clauses can be used to derive the empty clause).

AtMost Constraints are a type of counting or cardinality constraint. Given a set of n literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, s.t. $k < n$, an AtMost constraint is defined as

$$\mathrm{AtMost}(\{l_1, l_2, \ldots, l_n\}, k) \equiv \sum_{i=1}^{n} \mathrm{val}(l_i) \leq k$$

where val(l_i) is 1 if l_i is assigned TRUE and 0 otherwise. This constraint places an upper bound on the number of literals in the set assigned TRUE.

Clause-Selector Variables can be used to augment a CNF formula in such a way that standard SAT solvers can manipulate and, in effect, reason about the formula's clauses without any modification to the solver itself.

Every clause C_i in a CNF formula φ is augmented with a negated clause-selector variable y_i to give $C'_i = (\neg y_i \vee C_i)$ in a new formula φ'. Notice that each C'_i is an implication, $C'_i = (y_i \rightarrow C_i)$. Assigning a particular y_i the value TRUE implies the original clause, essentially enabling it. Conversely, assigning y_i FALSE has the effect of disabling or removing C_i from the set of constraints, as the augmented clause C'_i is satisfied by the assignment to y_i. This change gives a SAT solver the ability to enable and disable constraints as part of its normal search, checking the satisfiability of different subsets of constraints within a single backtracking search tree.

3 Use of Cores in Max-SAT

As described in Section 1, unsatisfiable cores can be used to guide Max-SAT algorithms by limiting the number of clauses they must consider for "removal" or leaving unsatisfied. Researchers have developed a number of algorithms exploiting this, all using the inexpensive resolution proof method for generating unsatisfiable cores.

Fu & Malik first introduced the idea of using unsatisfiable cores to assist in solving Max-SAT in [5]. They described an algorithm based on "diagnosis" that repeatedly finds a core by the resolution proof method, adds clause-selector variables to the clauses in that core, places a one-hot constraint on those clause-selector variables, and searches for a satisfying solution to the modified problem. Essentially, the algorithm identifies a core in each iteration that must be neutralized (by the removal of a clause) in any Max-SAT solution.

Marques-Silva, Planes, and Manquinho [9,10,11] improved upon Fu & Malik's algorithm, which they dubbed MSU1, with several refinements and optimizations. In [10] and [11], Marques-Silva and Planes introduce algorithms MSU1.1, MSU3, and MSU4[1]. The MSU1.1 algorithm is a variant of MSU1 with three important modifications. First, it uses a better encoding for the one-hot constraints, namely a BDD representation of a counter converted to CNF with several optimizations. Second, MSU1.1 exploits the authors' observation that the one-hot constraints can actually be AtMost(1) constraints, because the identified cores are unsatisfiable if no clauses are removed. Third, an AtMost(1) constraint is placed on the clause-selector variables for each clause that has more than one.

The authors also describe MSU3, which avoids some of the size explosion of the additional variables and clauses created by MSU1 by using a single clause-selector variable per clause and a single AtMost constraint over all of them. In

[1] We have adopted the algorithm naming from the most recent paper [9], which is slightly changed from earlier papers.

[11], the authors further introduce MSU4, essentially a modification of MSU3 that exploits relationships between unsatisfiable cores and bounds on Max-SAT solutions. Finally, Marques-Silva and Manquinho introduce MSU1.2 and MSU2 in [9]. MSU1.2 improves on MSU1.1 using a bitwise encoding, with a logarithmic number of auxiliary variables, for each cardinality constraint, and MSU2 takes that a step further, employing a bitwise one-hot encoding on the clause-selector variables themselves.

Algorithm	Cardinality Constraints	Cardinality Encoding
MSU1	Per-core One-Hot	Adder tree
MSU1.1	Per-core AtMost	BDD to CNF
MSU1.2	Per-core AtMost	Bitwise on variables
MSU2	Per-core One-Hot	Bitwise on clauses
MSU3	Single AtMost	BDD to CNF
MSU4-v1	Single AtMost	BDD to CNF
MSU4-v2	Single AtMost	Sorting networks

Table 1. Comparison of all MSU* algorithms

A parallel development of the concept of using unsatisfiable cores for Max-SAT was done in the domain of logic circuit debugging/diagnosis by Sülflow, et al. [15]. Without explicitly noting the connection to Max-SAT, they developed a new SAT-based debugging framework that exploits unsatisfiable core extraction. Though the terminology is different and the theories and algorithms are often described in terms of gates instead of constraints or clauses, SAT-based debugging is essentially the process of solving Max-SAT for circuit-derived CNF instances [12].

The primary difference between the work of Sülflow, et al. and the MSU* algorithms is that the debugging framework produces *all* Max-SAT results (equivalent to finding all minimum-cardinality MCSes) by an iterative solving procedure. Their use of cores is closest to MSU3, with a single cardinality constraint covering all identified cores; however, non-overlapping cores are given separate cardinality constraints, as this limits the size of the search space with little overhead. They do mention alternative approaches for producing cardinality constraints, including one which creates a separate constraint for every intersection of any subset of the cores, but they dismiss these as not outperforming their chosen approach in most of their experiments.

4 Using Cores to Find MCSes

Our algorithm is a synthesis of 1) the MCSes algorithm for finding all MCSes of an infeasible constraint system from [7] and 2) the application of unsatisfiable cores to the Max-SAT problem as first shown by Fu & Malik [5] and refined by

MCSes-U(φ)

1. $k \leftarrow 1$ ◁ iteration counter
2. $\texttt{MCSes} \leftarrow \emptyset$ ◁ growing set of results
3. $\texttt{Core}_k \leftarrow \textbf{Core}(\varphi)$ ◁ any unsatisfiable core (preferably small) of φ
4. **while** ($\textbf{InstrumentAll}(\varphi) + \textbf{Blocking}(\texttt{MCSes})$) **is satisfiable**
   ▽ clauses contained in $\texttt{Core}_k$ are instrumented with clause-selector variables
5. $\varphi_k \leftarrow \textbf{Instrument}(\varphi, \texttt{Core}_k) + \textbf{AtMost}(\texttt{Core}_k, k)$
   ▽ **AllSAT** finds all models of φ_k corresponding to MCSes of size k
6. $\texttt{MCSes} \leftarrow \texttt{MCSes} + \textbf{AllSAT}(\varphi_k)$
   ▽ the **Core** function projects instrumented clauses onto clauses of φ
7. $\texttt{Core}_{k+1} \leftarrow \texttt{Core}_k + \textbf{Core}(\varphi_k + \textbf{Blocking}(\texttt{MCSes}))$
8. $k \leftarrow k + 1$
9. **return** MCSes

Fig. 1. The MCSes-U algorithm finds all MCSes of an unsatisfiable formula φ using unsatisfiable cores.

Marques-Silva, et al. [9,10,11]. Because finding MCSes is a generalization of the Max-SAT problem (cf. Section 2), this combination is a natural one. In fact, the MCSes algorithm is very similar to the MSU3 algorithm described in [10].

Briefly, the overall approach of both MCSes and MSU3 is to instrument clauses in an unsatisfiable clause set with clause-selector variables, then to use cardinality constraints on those clause-selector variables to search for small subsets of clauses whose removal leaves the remaining set satisfiable. For Max-SAT, the goal is to find such a set of the smallest cardinality; finding MCSes requires finding all such sets that are minimal or irreducible. Therefore, it is reasonable to assume that an approach used to solve Max-SAT, especially one that has been paired with a basic algorithm so similar to that used for finding MCSes, could be applied to an algorithm for finding MCSes.

4.1 Algorithm

Figure 1 contains pseudocode for our algorithm, dubbed MCSes-U (the -U signifies its use of **u**nsatisfiable cores). Two persistent variables, k and MCSes, keep track of the current iteration and the set of results, respectively. In any particular iteration of the **do/while** loop, $\texttt{Core}_k$ contains the set of clauses that will be considered for removal, and thus potentially included in an MCS, in that iteration. The input formula is instrumented with clause-selector variables on those clauses contained within $\texttt{Core}_k$, and an AtMost constraint is added on those selector variables with the current bound k. The **AllSAT** function in MCSes-U behaves exactly like the incremental solving employed in MCSes: find a solution,

record the MCS, block that MCS from future solutions with a blocking clause formed from its clause-selector variables, and continue until no solutions remain.

The core extraction in line 7 produces an unsatisfiable core of the combination of the instrumented formula φ_k with the blocking clauses produced from the set of MCSes found thus far (φ_k itself is satisfiable). This core is mapped back to clauses in the original clause set φ and added to $\texttt{Core}_k$ to make $\texttt{Core}_{k+1}$ for the following iteration. The process repeats as long as further MCSes remain, which can be determined by checking whether there is *any* way to make φ satisfiable by removing clauses *without* removing any MCS identified thus far.

For comparison purposes, consider that the previous algorithm MCSes is equivalent to MCSes-U under the condition that **Core** always returns the complete set of clauses in φ. In this situation, the entire formula will be instrumented with clause-selector variables in each iteration, and the AtMost bound will always apply to all of the clause-selector variables as well. The primary difference between MCSes and MCSes-U is that here we are using unsatisfiable cores to identify subsets of the clause set in which we know the MCSes must be found, or, conversely, we determine subsets that we know must *not* contain any MCSes. In the following subsection, we prove that this use of unsatisfiable cores is correct.

4.2 Completeness/Correctness Proof

Fu and Malik proved that their use of unsatisfiable cores in Max-SAT is correct in [5]; however, that proof does not carry over to our algorithm other than to prove that the first result returned will be a Max-SAT solution. We must further prove both 1) that every result returned by MCSes-U is an MCS (correctness) and 2) that all MCSes are found by the algorithm (completeness). These two points are interrelated:

Theorem 1. *Given an unsatisfiable clause set φ and a positive integer k:*
If all MCSes of φ of size less *than k are found, then every result of size k returned by MCSes-U(φ) is an MCS of φ.*

This is stated without a formal proof, but it follows from the correctness of the underlying algorithm for finding MCSes, described fully in [7], that we have adapted in this work. Briefly, the algorithm finds MCSes in increasing order of size; as every MCS of a size less than k is found, it is blocked from future solutions, and any correction set of size k that is found then must be minimal. With this theorem, we see that the algorithm's correctness hinges on its completeness. We shall prove that MCSes-U is complete in the following.

We wish to prove that the algorithm produces all MCSes of an instance. We will presuppose the completeness of the base algorithm as described in [7] and focus on the effect of our use of unsatisfiable cores. The base algorithm is equivalent to that presented in Figure 1 if we take $\texttt{Core}_k$ to be the complete formula φ in every iteration of the **while** loop (i.e., with no limitation on the clauses considered for finding MCSes). Therefore, we will prove here that the MCSes-U algorithm is complete in that it does not miss any MCSes due to restricting the search for MCSes to the clauses in $\texttt{Core}_k$.

First, we must define a new term, "k-correction," and present a useful lemma linking k-corrections to MCSes.

Definition 1. A k-correction of a set of clauses C is a set of k or fewer clauses whose removal makes C satisfiable.

Lemma 1. *Given an unsatisfiable subset C of a clause set φ and an integer k:*
If every $(k-1)$-correction of C contains some MCS of φ, then C contains all MCSes of φ with size k.

(Proofs of this and the following lemmas are included in Appendix A.)

With this lemma, we can prove the completeness of our algorithm by induction. We wish to prove that the MCSes-U algorithm finds all MCSes of size k in the k^{th} iteration of its loop. We will first prove by induction that every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ. Then, using Lemma 1, we can directly show that $\texttt{Core}_k$ contains all MCSes of size k, for all k. First, we will prove the base case of the inductive portion of our proof, for $k = 1$.

Lemma 2. *In MCSes-U, every 0-correction of $\texttt{Core}_1$ contains an MCS of φ.*

With Lemmas 1 and 2, we see that the algorithm is complete for $k = 1$. $\texttt{Core}_1$ contains all single-clause MCSes of φ, and the algorithm produces all MCSes of size 1. This can be seen from a different perspective by noting that an MCS of size 1 is a single clause, c, contained in every MUS of a formula, and thus $\texttt{Core}_k$, which is some unsatisfiable core of φ, must contain every MCS of size 1.

With our base case proven in Lemma 2, we now prove the inductive step.

Lemma 3. *Given some positive integer k:*
In MCSes-U, if every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ, then every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.

With these lemmas, we can prove the completeness of MCSes-U in Theorem 2, which, with Theorem 1, proves that it is correct as well.

Theorem 2. *For any positive integer k: the MCSes-U algorithm finds all MCSes of size k in the k^{th} iteration of its loop.*

Proof. By Lemmas 2 and 3, we have that every $(k-1)$-correction of $\texttt{Core}_k$ contains an MCS of φ, for all k. With Lemma 1, then, $\texttt{Core}_k$ contains every MCS of φ of size k for all k.
□

5 Experimental Results

Our primary experimental goal was to determine the value of using unsatisfiable cores to guide the search for MCSes in practice; specifically, we wished to compare the performance of MCSes and MCSes-U on industrial instances. In the course of running these experiments, we noticed an interesting situation in which using cores was in fact detrimental to the performance of Max-SAT algorithms but the MCSes-U algorithm still benefited, and we explore this case here as well.

Experimental Setup: All experiments were run in Linux (Fedora 9) on a 3.0GHz Intel Core 2 Duo E6850 with 3GB of physical RAM. The MCSes and MCSes-U algorithms were implemented in C++ using MiniSAT version 1.12b [4], which allows "native" AtMost constraints (instead of CNF encodings thereof). We added unsatisfiable core extraction to this version of MiniSAT using the resolution-graph method [17], storing the parents of each learned clause in memory. Binaries for MSU1.1 and MSU1.2 were supplied by João Marques-Silva.

Benchmark Families: We selected four sets of unsatisfiable industrial CNF benchmarks for these experiments:

- **Diagnosis:** These 108 instances, from the Max-SAT 2008 Evaluation [2], are generated in a process that diagnoses potential error locations in a physical circuit that is producing incorrect output(s) [12]. In this application, the MCSes of each instance directly identify the candidate error locations. (The set used in the Max-SAT Evaluation has 112 instances, and we removed 4 that are satisfiable.)
- **Reveal:** Reveal is a system for logic circuit verification that operates on Verilog code with a counterexample-guided abstraction refinement flow [1]. These 62 instances were generated in the abstraction refinement phase of Reveal when run on three different designs.
- **FVP-UNSAT.2.0:** 21 instances, used in previous SAT competitions, obtained from [16], and "generated in the formal verification of correct superscalar microprocessors."
- **DC:** This is a set of 84 instances from an automotive product configuration domain [13,14] that have previously been shown to have a wide range of characteristics with respect to each instance's MCSes and MUSes. They are of interest mainly because their diversity of results (from small sets found in less than a second to intractably large sets) exercises algorithms broadly.

The value of using cores is evident when we look at the results for finding multiple MCSes across all of these instances. Because the complete set of MCSes can be intractably large, we look at the *velocity* of finding MCSes: the number of MCSes found per second until all have been found or until a set timeout (600 seconds, here) has been reached. Many applications do not require the complete set of MCSes: the diagnosis task in [12] finds MCSes up to a certain cardinality, and the CAMUS algorithm used in Reveal can use a subset of the MCSes to find a subset of the MUSes of an instance [7]. Figure 2 compares the velocity of MCSes (w/o cores) to that of MCSes-U (w/ cores) on these instances. Points above the diagonal are instances where MCSes-U finds MCSes more quickly. MCSes-U outperforms MCSes in nearly all cases. With the Diagnosis instances in particular, we see several benchmarks for which MCSes finds no MCSes within the timeout, while MCSes-U outputs up to several hundred per second.

An interesting situation is displayed in Figure 3, which compares the runtime of the MCSes algorithm solving Max-SAT against three Max-SAT algorithms that use unsatisfiable cores: MCSes-U in the same Max-SAT mode, MSU1.1, and MSU1.2 (these were the two MSU* algorithms with implementations available to

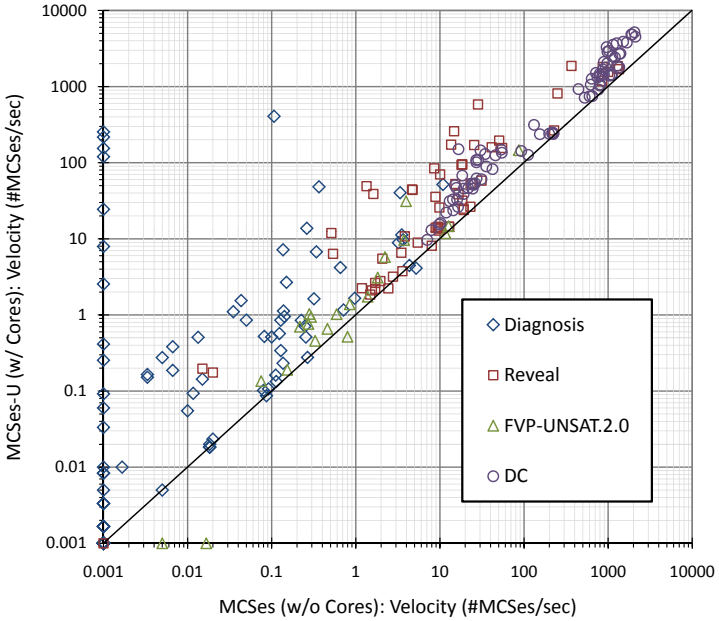

Fig. 2. Comparing the performance of MCSes and MCSes-U on industrial benchmarks. (600 second timeout, 0 velocity mapped to 0.001.)

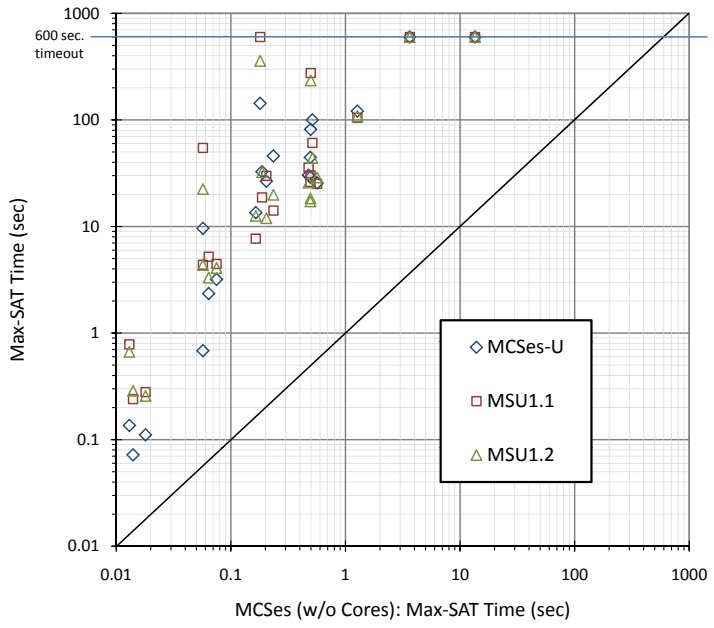

Fig. 3. Comparing the performance of MCSes solving Max-SAT against MCSes-U MSU1.1, and MSU1.2 on industrial benchmarks. (FVP-UNSAT.2.0 benchmarks.)

us). For MCSes and MCSes-U to solve Max-SAT, they can both return the first MCS found and stop. Because the algorithms generate MCSes in increasing order of size, the first result is guaranteed to be an optimal Max-SAT solution; the complement of the MCS will be a maximum cardinality satisfiable subset of the constraints. Note that the MSU* binaries had a different implementation, using a different solver and CNF-encoded cardinality constraints, than MCSes[-U], so the results are not *directly* comparing the underlying algorithms.

These results are for the FVP-UNSAT.2.0 benchmarks. For these instances, we see that all of the algorithms that use cores take about two orders of magnitude longer than the vanilla MCSes algorithm. The number of Max-SAT solutions in each benchmark is large. For example, for each of the three 2pipe* instances in this set, approximately one quarter of the clauses are single-clause MCSes; removing any one of them makes the instance satisfiable. Therefore, solving Max-SAT for these instances is simple, as there are so many solutions, and they will be found in the first iteration of MCSes. Interestingly, the time taken to run a solver on each benchmark (in order to extract a core) far outweighs that taken to identify a single clause whose removal yields satisfiability.

We see that the time used to find a core can outweigh that needed to find a single MCS or Max-SAT solution in instances like the FVP-UNSAT.2.0 set with very many, single-clause MCSes. However, when finding MCSes, the overhead of finding cores is amortized over the large number of results and is outweighed by the increase in velocity gained from limiting the search space, even in those instances that appear to be worst-case scenarios for exploiting cores. Therefore, core extraction appears to be a safe addition to MCS algorithms with potentially large performance gains in industrial instances.

6 Conclusion

We have presented a generalization of the core-guided Max-SAT approach, applying it to the more general problem of identifying minimal correction sets (MCSes). By using unsatisfiable cores to guide the search for MCSes in a similar manner to their use in Max-SAT [5,9,10,11], we have realized significant performance gains in MCSes-U, an enhancement of the algorithm for finding MCSes in [7]. Experimental results show the value of this approach on a variety of industrial instances; it is particularly effective on instances generated by a circuit diagnosis application in which MCSes have a direct application.

Looking forward, we see that there are further ideas from the Max-SAT domain that can be applied to MCS algorithms. Notably, the use of one AtMost constraint per identified core, as in the MSU1.* algorithms, may be applicable to MCSes-U. For Max-SAT, the MSU1.* approach has shown better performance than the approach used in MSU3 and MSU4 of creating a single monolithic AtMost constraint over all extracted cores, and it may be beneficial for MCSes-U as well. As with the proofs in this paper, determining and proving the correct application of the concept to the generalized problem of finding MCSes may require non-trivial work. We are also interested in investigating the combination and

interplay of the core-guidance technique with autarky pruning, another method for reducing the search space of the MCS search [8].

Further, the results here motivate applying MCSes-U in circuit debugging / diagnosis, as MCSes was applied in [12]. While MCSes was used as an approximating preprocessor for an exact search in that work, the improved performance of MCSes-U may make it suitable for solving problems directly. A comparison to the algorithm in [15] could be instructive as well; though it is algorithmically very similar to MCSes-U, any substantial performance differences would indicate important implementation details that would aid in engineering future implementations. Further, [15] is restricted to only find minimum-cardinality solutions, and the more complete view of examining the set of *all* MCSes in such instances, which MCSes-U enables, could be beneficial.

Acknowledgments

We thank João Marques-Silva for providing binaries for his MSU* algorithms. This material is based upon work supported by the National Science Foundation under ITR Grant No. 0205288. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation (NSF). This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center.

References

1. Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of the 2006 conference on Asia South Pacific design automation (ASP-DAC'06)*, pages 19–24, 2006.
2. J. Argelich, C. M. Li, F. Manyà, and J. Planes. Max-SAT evaluation 2008. Website. http://www.maxsat.udl.es/08/.
3. E. Birnbaum and E. L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 15:25–46, 2003.
4. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, volume 2919 of *LNCS*, pages 502–518, 2003.
5. Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006)*, volume 4121 of *LNCS*, pages 252–265, 2006.
6. M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, volume 3569 of *LNCS*, pages 173–186, 2005.
7. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, January 2008.
8. M. H. Liffiton and K. A. Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT-2008)*, volume 4996 of *LNCS*, pages 182–195, May 2008.

9. J. Marques-Silva and V. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT-2008)*, volume 4996 of *LNCS*, pages 225–230, May 2008.
10. J. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *Computing Research Repository*, abs/0712.1097, December 2007.
11. J. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'08)*, March 2008.
12. S. Safarpour, M. Liffiton, H. Mangassarian, A. Veneris, and K. Sakallah. Improved design debugging using maximum satisfiability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 13–19, November 2007.
13. C. Sinz. SAT benchmarks from automotive product configuration. Website. http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/.
14. 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.
15. A. Sülflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI, 2008*, pages 77–82, 2008.
16. M. Velev. Miroslav Velev's SAT Benchmarks. Website. http://www.miroslav-velev.com/sat_benchmarks.html.
17. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *The 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003)*, 2003.

A Proofs

Lemma 1. *Given an unsatisfiable subset C of a clause set φ and an integer k:*
If every $(k-1)$-correction of C contains some MCS of φ, then C contains all MCSes of φ with size k.

Proof. By contradiction: Assume that there exists some MCS M of φ with size k that is *not* contained entirely within C. We will denote the subset of M contained within C by $M' = M \cap C$. Thus, our assumption requires $|M'| \leq k-1$.

Because M is an MCS of φ and C is a subset of φ, M' must be a correction set of C. Formally, if $\varphi - M$ is satisfiable, then $C \cap (\varphi - M)$ must be as well. This can be transformed:

$$C \cap (\varphi - M) \;=\; (\varphi \cap C) - (M \cap C) \;=\; C - M'$$

And so M' is a correction set of C, because $C - M'$ is satisfiable.

Furthermore, M' is a $(k-1)$-correction of C, because $|M'| \leq k-1$. By the antecedent of this lemma, we know that M' must contain some MCS of φ. Because M is a proper superset of M', which contains an MCS, M can not be a *minimal* correction set of φ. This is a contradiction, and therefore we have proven that any MCS M of φ with size k must be contained entirely within C. □

Lemma 2. *In MCSes-U, every 0-correction of* $\texttt{Core}_1$ *contains an MCS of* φ.

Proof. Unsatisfiable clause sets have no 0-corrections, as removing 0 clauses can not make them satisfiable. $\texttt{Core}_1$ is an unsatisfiable clause set; therefore, $\texttt{Core}_1$ has no 0-corrections, and the lemma is trivially true.
□

Lemma 3. *Given some positive integer* k*:*
In MCSes-U, if every $(k-1)$*-correction of* $\texttt{Core}_k$ *contains an MCS of* φ*, then every* k*-correction of* $\texttt{Core}_{k+1}$ *contains an MCS of* φ.

Proof. Proof by cases, depending on the k-corrections of $\texttt{Core}_k$:

Case 1: $\texttt{Core}_k$ has no k-corrections.
The algorithm includes $\texttt{Core}_k$ in $\texttt{Core}_{k+1}$. Therefore, in this case, $\texttt{Core}_{k+1}$ will have no k-corrections, as it is a superset of $\texttt{Core}_k$. Thus, trivially, every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.

Case 2: Every k-correction of $\texttt{Core}_k$ contains an MCS of φ.
Again, due to the fact that $\texttt{Core}_k \subseteq \texttt{Core}_{k+1}$, every k-correction of $\texttt{Core}_{k+1}$ is also a k-correction of $\texttt{Core}_k$, and thus every k-correction of $\texttt{Core}_{k+1}$ must contain some MCS of φ.

Case 3: At least one k-correction, δ, of $\texttt{Core}_k$ contains no MCSes of φ.
Because δ does not contain any MCSes of φ, the blocking clauses added to φ_k based on the MCSes of φ will all allow the relaxation of the clauses in δ. We will say that δ is thus an *unblocked* k-correction. At line 6 of MCSes-U, there exists a complete assignment for φ_k that relaxes all MUSes contained within $\texttt{Core}_k$ without violating the AtMost bound on relaxed clauses; such an assignment can relax the clauses in any unblocked k-correction.
However, φ_k is unsatisfiable at this point, after the addition of all blocking clauses for the MCSes found thus far (up to size k). Therefore, for any complete assignment that satisfies the blocking clauses and relaxes all MUSes contained in $\texttt{Core}_k$, there must be some MUS of φ that is not relaxed by that assignment. Any unsatisfiable core of φ_k will necessarily include one MUS of φ that is not relaxed for every such assignment. That is, any unblocked k-correction δ of $\texttt{Core}_k$ must be "counteracted" by including in $\texttt{Core}_{k+1}$ an MUS of φ untouched by δ.
Any k-correction of $\texttt{Core}_{k+1}$ must contain a k-correction of $\texttt{Core}_k$, because $\texttt{Core}_k \subseteq \texttt{Core}_{k+1}$. Any unblocked k-correction of $\texttt{Core}_k$ necessarily leaves at least one MUS in $\texttt{Core}_{k+1}$ untouched (by the construction of $\texttt{Core}_{k+1}$ described above). Thus, unblocked k-corrections of $\texttt{Core}_k$ cannot be k-corrections of $\texttt{Core}_{k+1}$. Therefore, the only k-corrections of $\texttt{Core}_{k+1}$ must be "unblocked," containing an MCS of φ.

These cases cover all possibilities, and, in every case, every k-correction of $\texttt{Core}_{k+1}$ contains an MCS of φ.
□