On Finding All Minimally Unsatisfiable Subformulas

Mark H. Liffiton and Karem A. Sakallah

Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2122
{liffiton, karem}@eecs.umich.edu

Abstract. Much attention has been given in recent years to the problem of finding Minimally Unsatisfiable Subformulas (MUSes) of Boolean formulas. In this paper, we present a new view of the problem, strongly linking it to the maximal satisfiability problem. From this relationship, we have developed a novel technique for extracting all MUSes of a CNF formula, tightly integrating our implementation with a modern SAT solver. We also present another algorithm for finding all MUSes, developed independently but based on the same relationship. Experimental comparisons show that our approach is consistently faster than the other, and we discuss ways in which ideas from both could be combined to improve further.

1 Introduction

Many computational problems in a wide range of fields are posed as constraint satisfaction problems, often in the form of Boolean CNF formulas analyzed with satisfiability (SAT) solvers. While SAT solvers can return a short proof in the form of a satisfying assignment when a formula is satisfiable, typically no proof or explanation is given when a formula is found to be unsatisfiable. Explanations of infeasibility are often valuable, and techniques for finding them have been developed for use in these problems. Some techniques have focused on reducing the original set of constraints to produce a minimal, unsatisfiable core representing a cause of infeasibility. In this paper, we present a new approach to finding these cores, focusing on a complete method for finding all unsatisfiable cores of any given formula.

Consider an unsatisfiable CNF formula φ. A Minimally Unsatisfiable Subformula (MUS) of φ is a subset of φ's clauses that is both unsatisfiable and minimal in the sense that all of its proper subsets are satisfiable. An MUS can be seen as an irreducible cause of the infeasibility of the original formula. φ could have multiple reasons for its infeasibility. In this case φ would contain multiple MUSes, and fixing any single MUS may not make φ satisfiable. As long as any MUS is present in the formula, it will remain infeasible. In many applications, it is valuable to find the set of all MUSes, because diagnosing infeasibility is hard, if not impossible, without a complete view of its causes. Additionally, an algorithm that finds all MUSes provides a basis for approximations and techniques that find multiple, though not all, MUSes.

Many methods for finding MUSes have been developed in recent years, both for Boolean satisfiability problems and for other types of constraints. Most techniques find

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 173 – 186, 2005.

a single unsatisfiable subformula (US), often not guaranteeing it to be minimal. For example, AMUSE [10], Bruni & Sassano's algorithm [3,4], and zCore [13] all use information from a SAT solver's resolution procedure to find a single US, but none guarantee its minimality. For these, a "Minimal Unsatisfiability Prover" [7] can be used to minimize the US into an MUS. Chinneck and Dravnieks [5] studied MUSes in the domain of linear and integer programs, calling them Irreducible Infeasible Subsets. Their algorithms return multiple, but not all, MUSes.

Recently, we have developed a sound and complete technique for finding all MUSes of a CNF formula [9], based on a strong relationship between maximal satisfiability and minimal unsatisfiability. The relationship and algorithms derived from it also hold for any other type of constraint with a strict definition of satisfiability (i.e., they do not apply directly to "soft" constraints). Independently, Bailey and Stuckey [1] have also noted this relationship and developed an implementation for a type of constraint used for type-error debugging in software verification.

While both our work and theirs are based on the same underlying concept, the algorithms we developed to exploit it differ greatly. In this paper, we present a comparison of our two approaches, having implemented their algorithm for Boolean constraints. We show that, for the case of Boolean constraints, our algorithm outperforms theirs by one to two orders of magnitude. We further assess the differences between the two approaches and discuss the strengths and weaknesses of both. (In [1], the authors compare their algorithm to the best known previous method for finding all MUSes in [2]. They show that their algorithm is superior to that in [2], so we have not included that method in our comparison.)

The rest of this paper is organized as follows. Section 2 lays the foundation by describing the relationship between maximal satisfiability and minimal unsatisfiability that is the basis for both algorithms. In Section 3, we present our algorithm, and we present Bailey and Stuckey's algorithm in Section 4. Section 5 contains the experimental results comparing the two algorithms, with discussion and analysis in Section 6. Finally, Section 7 contains conclusions and ideas for future work.

2 Maximal Satisfiability and Minimal Unsatisfiability

Both our technique and Bailey and Stuckey's algorithm are based on a strong relationship between maximal satisfiability and minimal unsatisfiability. The Maximum Satisfiability problem (Max-SAT) is an optimization problem on a CNF formula φ in which the goal is to find an assignment to the variables of φ that maximizes the number of satisfied clauses. In other words, Max-SAT yields a satisfiable subset of φ's clauses with maximum cardinality. For example, the formula

$$\varphi = (x_1) \wedge (\neg x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2)$$

has a Max-SAT solution with three satisfied clauses:

$$\{(\neg x_1), (\neg x_1 \vee x_2), (\neg x_2)\}.$$

Whereas the Max-SAT problem has been defined with the cardinality of a subset of clauses as the optimization goal, the problem can be relaxed to have *inaugmentability*

as the goal instead. For this, we define a new problem, *Maximally Satisfiable Subset* (MSS). Each MSS of a formula φ is a subset of the clauses in φ that is satisfiable and inaugmentable; adding any of the other clauses in φ to an MSS will render it unsatisfiable. The definition of the set of MSSes of a formula φ follows, with the set of MUSes defined similarly for comparison:

$$\mathrm{MSSes}(\varphi) = \left\{ \begin{array}{l} m \subseteq \varphi : \ m \text{ is satisfiable, and} \\ \forall c \in (\varphi - m), \ m \cup \{c\} \text{ is unsatisfiable} \end{array} \right\}$$

$$\mathrm{MUSes}(\varphi) = \left\{ \begin{array}{l} m \subseteq \varphi, \ m \text{ is unsatisfiable, and} \\ \forall c \in m, \ m - \{c\} \text{ is satisfiable} \end{array} \right\}$$

Notice that MSSes(φ) and MUSes(φ) are essentially duals of one another! An MSS is satisfiable and cannot be made larger, and an MUS is *un*satisfiable and cannot be made *smaller*. Their relationship is more than cosmetic; the complete set of one type (i.e., all MSSes or all MUSes) is actually an implicit encoding of the other.

The distinction between Max-SAT and MSS is subtle. Any Max-SAT solution is also an MSS; the maximal cardinality of the Max-SAT solution entails the inaugmentability required to be an MSS. MSSes, however, may be of different sizes, and not all will necessarily have maximum cardinality, as shown by the earlier example formula. One MSS of that formula is $\{(\neg x_1), (\neg x_1 \vee x_2), (\neg x_2)\}$, which corresponds to the Max-SAT solution. $\{(x_1), (\neg x_2)\}$ is another MSS (adding either of the other two constraints would make it unsatisfiable), though it is smaller.

Now consider (x_1), the clause not included in the Max-SAT solution to our earlier example. Removing this clause from φ makes it satisfiable. In general, given any MSS of an infeasible formula, the clauses *not* in that MSS describe an irreducible "fix" to the formula in the sense that removing them will correct the infeasibility. Therefore, we define a "CoMSS" as the complement of an MSS, and the set of CoMSSes is simply:

$$\mathrm{CoMSSes}(\varphi) = \{ m \subseteq \varphi : \ (\varphi - m) \in \mathrm{MSSes}(\varphi) \}$$

This complementary view of MSSes provides the real link to MUSes. Note that the presence of any MUS in a CNF formula makes that formula unsatisfiable. Therefore, to correct the infeasibility by removing constraints, at least one constraint from every MUS must be removed in order to "neutralize" all of the MUSes. A CoMSS of a formula φ is a set of constraints whose removal renders φ satisfiable, thus every CoMSS must contain at least one clause from each MUS of φ.

This relationship between a CoMSS and the set of MUSes of a formula can be described as a solution to a set covering problem. Specifically, a CoMSS is a *hitting set* of the set of MUSes. A hitting set of a collection of sets is a set that contains at least one element from each set in the collection. In this case, the collection is the set of MUSes, and the CoMSS is a set of clauses that contains at least one clause from every MUS. A CoMSS is a hitting set of the MUSes with the additional restriction that it cannot be any smaller without losing its defining property: it is an *irreducible hitting set*. Figure 1 shows the example formula from above along with its MSSes, CoMSSes, and MUSes to illustrate the relationship. Notice how each CoMSS is a hitting set of the MUSes.

$$\varphi = \underset{A}{(x_1)} \wedge \underset{B}{(\neg x_1)} \wedge \underset{C}{(\neg x_1 \vee x_2)} \wedge \underset{D}{(\neg x_2)}$$

MSSes(φ)	CoMSSes(φ)
{B,C,D}	{A}
{A,D}	{B,C}
{A,C}	{B,D}

MUSes(φ)
{A,B}
{A,C,D}

Fig. 1. Example formula with its MSSes, CoMSSes, and MUSes

In line with the duality noted earlier, every MUS of a formula is an irreducible hitting set of the CoMSSes of that formula (which also can be seen in Figure 1). The intersection of any CoMSS of a formula with any MUS of that formula must be non-empty. The duality between CoMSSes and MUSes comes from this commutative relationship between them. The relationship also gives us a way to compute all MUSes of a formula, and it is the basis for the two algorithms compared in this paper.

3 Finding All MUSes

In our algorithm for finding all MUSes of a formula φ, we decompose the process into two steps: 1) Finding the complete set of CoMSSes of φ, and 2) Extracting MUSes from the set of CoMSSes. This decomposition follows naturally from the relationship described in the previous section. And it has a nice property in that each step is independent of the other, which allows different algorithms to be used for each step without affecting the other.

3.1 Finding CoMSSes(φ)

Every CoMSS is the set of clauses *not* included in some maximally satisfiable subformula. To find MSSes, we incrementally solve a Max-SAT problem, removing solutions as they are found to continue the search until all have been found. Figure 2 provides a pseudocode outline of the procedure.

Our algorithm is tightly integrated with and takes advantage of a modern SAT solver. We find maximally satisfiable subsets of constraints by giving the solver the ability to enable and disable constraints and check for the satisfiability of the enabled constraints all within a single search tree. Every clause $C_i = x_i^1 \vee \ldots \vee x_i^m$ in φ is augmented with a negated *clause selector variable* y_i to give $C_i' = \neg y_i \vee x_i^1 \vee \ldots \vee x_i^m$ in a new formula φ'. This is accomplished by **AddYVars** in the pseudocode. While solving φ', assigning a certain y_i FALSE has the effect of disabling or removing C_i from the set of constraints, as the augmented clause is satisfied by the assignment to y_i. Conversely, assigning y_i TRUE enables the original clause. An MSS can be obtained by finding a satisfying assignment with a minimal number of y_i variables assigned FALSE, which ensures that as few constraints as possible are disabled. The clauses left unsatisfied, which are a CoMSS, are indicated by the set of y_i variables assigned FALSE.

```
CoMSSes(formula)
1.  // formula is a CNF instance
2.  formula ⟵ AddYVars(formula)
3.  bound ⟵ 1
4.  CoMSSes ⟵ ∅
5.  While (Sat(formula))
6.      boundedFormula ⟵ AddAtMost(formula, bound)
7.      Repeat
8.          newCoMSS ⟵ IncrementalSat(boundedFormula)
9.          If (newCoMSS = ∅)
10.             End Repeat
11.         CoMSSes ⟵ CoMSSes ∪ {newCoMSS}
12.         boundedFormula ⟵ AddBlocking(boundedFormula, newCoMSS)
13.         formula ⟵ AddBlocking(formula, newCoMSS)
14.     bound ⟵ bound+1
15. Return CoMSSes
```

Fig. 2. Pseudocode for finding all CoMSSes of a formula

Instead of solving an optimization problem for every solution, however, we utilize a sliding objective approach that allows us to use a more efficient incremental search. We set a bound on the number of y_i that may be assigned FALSE using an AtMost bound. Given a set of literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, an AtMost bound is defined as

$$\text{AtMost}\left(\{l_1, l_2, \ldots, l_n\}, k\right) \equiv \sum_{i=1}^{n} \text{assign}\left(l_i\right) \leq k$$

where assign(l_i) is 1 if l_i is assigned TRUE and 0 otherwise. In practice, an efficient implementation of this constraint propagates the negation of each of the remaining literals in its set once k of them have been assigned TRUE. Because we only use one such constraint at a time, the same effect could easily be created by modifying any standard SAT solver to "watch" the assignments to the variables involved in the constraint and to force the remaining assignments as necessary once the bound has been reached.

In this application, we add a constraint of the form AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k$) to place a bound on the number of disabled constraints. For each value of the bound, starting at 1 and incrementing by 1, we exhaustively search for all satisfiable assignments to the augmented formula φ', which will find all CoMSSes the size of the bound. Within this search, the incremental SAT solver can utilize learned clauses and dynamic variable ordering heuristics to full effect.

When one solution is found, the corresponding CoMSS is recorded, and the search continues incrementally after adding a blocking clause that forces out that solution. The blocking clause is a disjunction of the y variables for the clauses in that CoMSS. For example, if the solution contains $y_2 = y_4 = y_7 = F$, indicating that $\{C_2, C_4, C_7\}$ is a CoMSS, then we add a blocking clause: $y_2 \vee y_4 \vee y_7$. This excludes the CoMSS, and any supersets of it, from future solutions. Finding CoMSSes in order of increasing size

```
ExtractMUS(CoMSSes)
1.  MUS ⟵ ∅
2.  While (CoMSSes ≠ ∅)
3.      curCoMSS ⟵ Pop(CoMSSes)
4.      newClause ⟵ Pop(curCoMSS)
5.      MUS ⟵ MUS ∪ {newClause}
6.      // Remove all clauses in curCoMSS from all remaining CoMSSes
7.      For Each clause ∈ curCoMSS
8.          For Each testCoMSS ∈ CoMSSes
9.              If (clause ∈ testCoMSS)
10.                 testCoMSS ⟵ testCoMSS - {clause}
11.     // Remove any CoMSSes containing newClause
12.     For Each testCoMSS ∈ CoMSSes
13.         If (newClause ∈ testCoMSS)
14.             CoMSSes ⟵ CoMSSes - {testCoMSS}
15. Return MUS
```

Fig. 3. Pseudocode for extracting a single MUS from a set of CoMSSes

(i.e., MSSes in order of decreasing size) and excluding supersets from future solutions ensures that only irreducible CoMSSes are found.

As long as we are *adding* constraints (the blocking clauses), we can use an incremental search. For each search with a particular AtMost bound, every new solution is removed with a blocking clause and the search continues until no further solutions exist for the current bound. Incrementing the bound, however, relaxes a constraint on the system, so the search must start over with a new copy of the formula, augmented with all blocking clauses created thus far.

Before beginning the search with the next bound, the algorithm checks that φ' augmented with all collected blocking clauses is still satisfiable without any bound on the y_i variables. If there is no satisfying assignment, even with no restrictions on the y_i variables, the entire set of CoMSSes has been found, and the algorithm terminates.

3.2 Obtaining MUS(φ)

The set of CoMSSes implicitly encodes the entire set of MUSes of a formula, and information can be extracted from it in a variety of ways. Here, we focus on methods for extracting MUSes, though it is likely that other useful data can be obtained by analyzing the set as well.

Extracting a Single MUS in Polynomial Time

Every MUS of a formula φ is an irreducible hitting set of the CoMSSes of φ. Although MINIMAL-HITTING-SET is an NP-Hard problem [8], irreducibility is a less strict requirement than minimal cardinality. Along with the fact that no CoMSS is a subset of any other, this allows us to find an irreducible hitting set for the set of CoMSSes in polynomial time. We can generate an MUS by a greedy, iterative construction, with no search necessary. Figure 3 outlines the construction in pseudocode.

Intuitively, we want to generate a set of clauses with at least one clause from each CoMSS, such that every clause is an essential element of the set. By "essential" we mean that removing a clause will leave at least one CoMSS unrepresented in the generated MUS; this enforces the irreducibility requirement.

The algorithm works by sequentially adding clauses to a forming MUS. In the main loop, a clause is selected for inclusion in the MUS, the working set of CoMSSes is altered to force that clause to be essential, and the process iterates with the altered set of CoMSSes. The first two lines in the loop choose a CoMSS and a clause from that CoMSS (the choices can be arbitrary). The clause is added to the MUS. Then, all of the other clauses in the chosen CoMSS are removed from the remaining problem. This prevents any of those clauses from being added to the MUS in later iterations, which could make the chosen clause non-essential. Next, any CoMSSes containing the chosen clause are removed, because they are now represented in the MUS. After these modifications are made, the algorithm iterates with the resulting set of CoMSSes. When no more CoMSSes remain, the constructed set of clauses is a complete MUS.

Extracting All MUSes

Finding all MUSes involves searching for all irreducible hitting sets of the set of CoMSSes. In general, this may be impractical due to the possibly exponential number of MUSes, but in many cases the result is tractable.

Our algorithm for extracting the complete set of MUSes from the CoMSSes uses the general form of the algorithm in Figure 3. The order in which CoMSSes and clauses are selected (the choices made in the first two lines of the while loop) determines the particular MUS created by the algorithm; therefore, by branching on these two decisions, all possible MUSes can be generated. We implemented this with a recursive algorithm that takes as input the remaining set of CoMSSes (initially the entire set) and the MUS under construction in the given branch of the recursion (initially the empty set). The branching is not ideal, and many duplicate branches are encountered in practice. We employ ordering heuristics to prune as many duplicate branches as possible without missing any MUSes.

4 Dualize and Advance

The algorithm developed by Bailey and Stuckey in [1] was implemented to find all minimally unsatisfiable subsets of systems of Herbrand constraints, used for type-error debugging of Haskell programs. It exploits the same relationship between maximal satisfiability and minimal unsatisfiability as ours, however, and so it can be applied to any type of constraint as well. We present an overview of their algorithm here.

They call the algorithm "Dualize and Advance" (DAA), as it interleaves the use of the hitting-set duality with the search for what we call CoMSSes. Whereas our technique finds all CoMSSes before finding hitting sets of them, DAA computes hitting sets on a partial set of CoMSSes after finding each CoMSS — it outputs any MUSes found at that stage and also uses the results to direct the search for the next CoMSS. The full DAA algorithm is presented in pseudocode in Figure 4.

```
DAA(Constraints)
1.  MUSes ⟵ ∅
2.  CoMSSes ⟵ ∅
3.  Seed ⟵ ∅
4.  Repeat
5.      MSS ⟵ Grow(Seed, Constraints)
6.      CoMSSes ⟵ CoMSSes ∪ {Constraints - MSS}
7.      PotentialMUSes ⟵ ExpandHittingSets(MUSes, {Constraints - MSS})
8.      Seed ⟵ ∅
9.      For Each S ∈ (PotentialMUSes - MUSes)
10.         If (Sat(S))
11.             Seed ⟵ S
12.             Break
13.         Else
14.             MUSes ⟵ MUSes ∪ {S}
15. Until (Seed = ∅)
16. Return MUSes

Grow(S, Constraints)
1.  For Each c ∈ (Constraints - S)
2.      If (IncrementalSat(S ∪ {c}))
3.          S ⟵ S ∪ {c}
4.  Return S

ExpandHittingSets(MUSes, CoMSS)
1.  If MUSes = ∅
2.      For Each c ∈ CoMSS
3.          MUSes ⟵ MUSes ∪ {c}
4.  Else
5.      // Compute cross product
6.      newMUSes ⟵ ∅
7.      For Each MUS ∈ MUSes
8.          For Each c ∈ CoMSS
9.              newMUSes ⟵ newMUSes ∪ {(MUS ∪ {c})}
10.     // Perform minimization
11.     newMUSes ⟵ Sort(newMUSes)// in order of increasing cardinality
12.     MUSes ⟵ ∅
13.     For Each testMUS ∈ newMUSes
14.         If (∀ m ∈ MUSes. m ⊄ testMUS)
15.             MUSes ⟵ MUSes ∪ {testMUS}
16. Return MUSes
```

Fig. 4. Dualize and Advance pseudocode

DAA finds MSSes in a straightforward manner by "growing" them. Given a seed in the form of a satisfiable set of constraints (which is empty in the first iteration), an MSS is constructed by attempting to add each of the problem's remaining constraints to the seed, only keeping those which do not create a conflict. After going through all possible constraints, this process will have collected a maximally satisfiable subset of the constraints, because those excluded were left out specifically because they would make it unsatisfiable. This is all contained in the **Grow** subroutine.

When an MSS is found in this manner, the complement is added to the growing set of CoMSSes. At this point, the hitting sets of the CoMSSes are computed to potentially output MUSes and/or create a new seed. Each minimal hitting set that is unsatisfiable is an MUS (they are all guaranteed to be unsatisfiable only if the complete set of CoMSSes is used), and if one is found that is satisfiable, then this set is used as a seed for the next iteration. The seed created in this way is guaranteed to not intersect with any of the MSSes found thus far because it was created from the CoMSSes. For every MSS found previously, the seed will contain at least one constraint not in that MSS. Thus, the MSS grown from the seed is guaranteed to be new.

Bailey and Stuckey mention that there are many ways to compute minimal hitting sets, noting that the problem is equivalent to the hypergraph transversal problem. For their implementation, they chose a simple method that "is simple to implement and behaves reasonably efficiently." They compute the hitting sets of a set G by ordering the sets in G, then computing partial cross products of those sets, minimizing the results at each step. In the case of the DAA algorithm, the process can be made incremental. At each iteration, only a single set is added to the set of CoMSSes; by remembering the hitting sets computed in the last iteration, the hitting sets for the current iteration can be computed by taking the cross products of the new CoMSS with each of the hitting sets from the previous iteration. Their paper described the process, including the minimization, in mathematical terms, which we interpreted and implemented algorithmically as shown in Figure 4.

Bailey and Stuckey also discuss an optimization to their algorithm in the form of a heuristic for the order of adding constraints in the **Grow** subroutine. They aim to collect CoMSSes in increasing order of size to optimize the partial hitting set calculations. Using the constraint interaction graph, they estimate which constraints are most likely to cause unsatisfiability, ordering the constraints to choose these later in the **Grow** subroutine. In their results, the heuristic only decreased the runtime by at most half, and in some cases its use resulted in longer runtimes. Due to this and the lack of details describing it in the paper, we did not implement this heuristic in our version of DAA.

5 Results

We evaluated both our own technique and our implementation of Bailey and Stuckey's DAA algorithm using a large set of unsatisfiable CNF benchmarks from automotive product configuration [11,12]. Each benchmark encodes a set of available configurations for a product, along with constraints enforcing a specific property to be checked. We observed that the encodings were not "tight," in that they contained numerous duplicate clauses. Duplicate clauses can yield a combinatorial explosion of MUSes, so

they were removed before gathering data. There are a total of 84 benchmarks in the set, each with around 1500-1800 variables and 4000-8000 clauses. This set of benchmarks was chosen because of the range of results it provides. Though all of the instances were generated in the same manner and have the same general size, the number and size of CoMSSes and MUSes in each instance vary widely. Some have a single MUS, while others have millions; runtimes can range from less than a millisecond to days or longer.

All of the algorithms were implemented in C++. Both our algorithm for finding CoMSSes and our implementation of DAA used MiniSAT [6] as a framework for constraint solving. MiniSAT is primarily a SAT solver, but it can be extended to handle other types of constraints as well. This made it possible to integrate AtMost constraints alongside the standard Boolean CNF clauses for our CoMSSes algorithm. The data were collected in Linux on a PC with a 2.2GHz Opteron processor and 8GB of RAM.

We ran every instance against both algorithms, with a 600 second timeout for both. Of the 84 benchmarks, all MUSes were found for 31 of them within the timeout by at least one of the two algorithms. The results for these 31 are presented in Table 1. The first column lists the benchmark name, and columns 2 and 3 give the size of each benchmark with the number of variables and clauses, respectively. The following three columns list the time in seconds our algorithm spent in finding the set of CoMSSes, the time spent on the set of MUSes, and the total time as the sum of both. The seventh column lists the runtime in seconds of the DAA algorithm for finding the complete set of MUSes. The "Ratio" column provides the ratio of the runtime of DAA to that of our algorithm (column 7 divided by column 6). Finally, the last two columns list the number of CoMSSes and the number of MUSes in each benchmark.

Of the 31 benchmarks for which at least one algorithm completed, ours finished all 31 while DAA reached the 600 second timeout for 6. Additionally, our algorithm is consistently faster than DAA, usually by about one to two orders of magnitude. The benchmarks which DAA was not able to complete all had more than 10,000 MUSes, indicating that calculating sets of potential MUSes at each stage was taking most of the time. Our algorithm is likewise affected by benchmarks with large numbers of MUSes, but because the set of MUSes is only calculated once, the impact is much smaller.

We should also note that of the 53 benchmarks that neither algorithm completed, ours was able to find the complete set of CoMSSes for 18. These instances all timed out in the MUS extraction stage. For each of the 18 instances, many MUSes were generated (up to 4.5 million) before the 600 second timeout was reached. This illustrates how the problem can be made intractable by the sheer number of MUSes. In one instance, C170_FR_SZ_95, our algorithm found the complete set of CoMSSes in just 0.34 seconds, and it had generated more than 1.5 million MUSes by the time the 600 second timeout was reached.

6 Analysis

The performance numbers paint a clear picture that our algorithm is faster than DAA for Boolean constraints. However, the performance of each algorithm is dependent on

Table 1. Performance Results

Benchmark			**Our Algorithm**			**DAA**	**Ratio**	**Solutions**	
Name	#V	#C	CoMSSes (sec)	MUSes (sec)	Sum (sec)	DAA (sec)	$\frac{\text{DAA}}{\text{Sum}}$	# CoMSSes	# MUSes
C208_FC_SZ_128	1513	4469	0.06	0.00	0.06	11.1	201.8	32	1
C208_FC_SZ_127	1513	4469	0.06	0.00	0.06	12.0	206.9	34	1
C208_FA_SZ_121	1516	4247	0.07	0.00	0.07	9.9	135.3	32	2
C208_FA_SZ_120	1516	4247	0.07	0.00	0.07	10.5	141.9	34	2
C202_FS_SZ_122	1556	5385	0.09	0.00	0.09	16.5	189.7	33	1
C170_FR_SZ_92	1528	4195	0.13	0.00	0.13	38.2	293.8	131	1
C202_FW_SZ_124	1561	7435	0.15	0.00	0.15	27.6	190.3	33	1
C210_FS_SZ_130	1607	4894	0.18	0.00	0.18	11.1	61.0	31	1
C210_FS_SZ_129	1607	4894	0.19	0.00	0.19	12.1	63.4	33	1
C210_FW_SZ_136	1628	6384	0.25	0.00	0.25	17.0	67.2	31	1
C202_FW_SZ_123	1561	7437	0.26	0.00	0.26	25.0	96.9	38	4
C210_FW_SZ_135	1628	6384	0.26	0.00	0.26	18.3	70.7	33	1
C168_FW_UT_852	1804	6756	0.45	0.00	0.45	15.1	33.5	30	102
C168_FW_UT_851	1804	6758	0.45	0.00	0.45	15.2	33.6	30	102
C168_FW_UT_854	1804	6753	0.45	0.00	0.45	15.2	33.6	30	102
C168_FW_UT_855	1804	6752	0.47	0.00	0.47	15.3	32.3	30	102
C220_FV_RZ_14	1530	4013	0.57	0.00	0.57	4.3	7.7	20	80
C208_FA_RZ_64	1516	4246	0.61	0.00	0.61	48.0	78.8	212	1
C220_FV_SZ_121	1530	4035	0.65	0.00	0.65	25.3	38.9	102	9
C208_FC_RZ_70	1513	4468	0.67	0.00	0.67	53.9	80.9	212	1
C202_FS_SZ_121	1556	5387	0.83	0.00	0.83	9.5	11.4	24	4
C202_FW_RZ_57	1561	7434	1.11	0.00	1.11	137.0	123.3	213	1
C208_FA_SZ_87	1516	4255	0.46	1.41	1.87	>600	>320.5	139	12884
C170_FR_RZ_32	1528	4067	0.64	1.96	2.60	>600	>230.4	242	32768
C220_FV_RZ_13	1530	4014	1.22	2.34	3.56	47.7	13.4	76	6772
C208_FA_UT_3254	1805	6153	1.63	8.94	10.57	>600	>56.8	155	17408
C208_FA_UT_3255	1805	6156	1.68	18.40	20.08	>600	>29.9	155	52736
C210_FS_RZ_40	1607	4891	0.44	30.10	30.54	73.7	2.4	212	15
C210_FW_RZ_59	1628	6381	0.56	30.40	30.96	114.0	3.7	212	15
C220_FV_RZ_12	1530	4017	1.23	65.20	66.43	>600	>9.0	150	80272
C220_FV_SZ_65	1530	4014	2.05	65.80	67.85	>600	>8.8	198	103442

a number of factors, and in this section we discuss the details behind the performance, comparing the strengths and weaknesses of each algorithm.

One difference contributing greatly to the performance of our algorithm is its tight integration with a modern SAT solver. By formulating the problem with clause-selector variables, we let the SAT solver handle the search for MSSes itself. Additionally, by finding multiple MSSes (of a single size) within a single search tree, we immediately take advantage of all of the features of modern SAT solvers, especially learned clauses. While DAA can use an incremental search within the **Grow** subroutine, it must restart the search after any added constraint makes the growing MSS unsatisfiable. It also restarts with a new search tree for every MSS, as compared to our algorithm which only restarts the search after all MSSes of a particular size have been found.

Note that while our approach is more heavily integrated with a SAT solver, it is still fairly independent of the particular solver itself. It can be implemented with any SAT solver that provides an incremental solving interface, allows the addition of constraints mid-search, and supports the AtMost constraint. (While the last requirement is not standard, its implementation in MiniSAT is quite simple, and as noted earlier, the effect can be obtained by modifying other SAT solvers with little difficulty.) The DAA algorithm simply calls a standard solver as a subroutine, making it even simpler to implement with different solvers.

The strength of our algorithm's integration with the solver is also a drawback, in that it makes it less immediately applicable to other constraint types. We rely on the ability to encode constraint enabling and disabling within the syntax of the constraints themselves. This is easy to do with Boolean disjunctions, but other types of constraints may not be as suitable. DAA, on the other hand, can immediately be implemented using a solver of any type of constraint.

Another large difference between our algorithm and DAA is the distinction between our serial, two-phase algorithm and DAA's interleaved approach. Obtaining MUSes before computing the entire set of CoMSSes is beneficial in applications that do not require the complete set of CoMSSes nor all MUSes because it can provide results sooner. The interleaved approach could easily be adopted in our algorithm. Hitting sets of the partial set of CoMSSes could be calculated after every CoMSS is found, between stages of the incremental search (when incrementing the bound on CoMSS size), or at any desired interval. This could add a great deal of overhead, however, especially if every potential MUS had to be checked for unsatisfiability (as opposed to aborting after one set is found to be satisfiable, as DAA does to use that set as the next seed). This seems to be the case for DAA, as the instance for which it had its fastest runtime also had the fewest CoMSSes, and thus the fewest incremental hitting-set calculations. Though our algorithm could be interleaved to potentially gain efficiency, DAA could not be "de-interleaved," as it depends on the set of potential MUSes to provide the seed for the next iteration and to determine when it has found all CoMSSes.

The process of "growing" an MSS from a seed has potential application within our algorithm as well. Recall that we restart the search for MSSes after exhausting the search space for each bound on the CoMSS size. Each restart throws away valuable learned clauses. We could relax the AtMost constraints to search for CoMSSes of 2, 3,

or more sizes at one time. For example, by starting with an AtMost bound of 3 on the clause selector variables, the algorithm would find CoMSSes of size 1, 2, and 3 within one search tree. There would be no guarantee that those of size 2 or 3 are irreducible, however, so the **Grow** subroutine could be used to maximize the corresponding satisfiable subset (thus minimizing the CoMSS). When that search tree is exhausted, the bound could be increased by 3, finding CoMSSes of size 4, 5, and 6 in the next iteration. In general, the range of sizes covered by each iteration could be extended at the cost of increased overhead from calls to **Grow**. The impact of this tradeoff is unclear, and it should be investigated to determine an optimal range.

Finally, the constraint-graph heuristic used in DAA to guide it towards smaller CoMSSes could be adapted for our algorithm. For example, the ranking of constraints generated by the heuristic could be implemented as an influence on the variable ordering of the clause-selector variables. In general, heuristics specific to finding CoMSSes and relevant to the constraints themselves (as opposed to the formula's variables) could be effective for both algorithms.

7 Conclusion and Future Work

We have presented a relationship between maximal satisfiability and minimal unsatisfiability that can be used to find all Minimally Unsatisfiable Subformulas of a Boolean CNF formula. We experimentally compared two methods that exploit this relationship, our own algorithm and DAA, developed by Bailey and Stuckey [1]. The results show that ours is about one to two orders of magnitude faster. We discussed the relative strengths and weaknesses of each approach, examining the causes of the performance difference as well as practical implementation details.

There are many possible directions for future improvement. Though ours is the fastest known algorithm for finding all MUSes of a Boolean CNF formula, it does not scale nearly as well as modern SAT solvers. While this is unavoidable due to the much greater complexity of the problem, work should be done to make it more efficient and practically useful. We discussed some ways in which our algorithm could be combined with ideas from Bailey and Stuckey's approach which might lead to increased performance. Additionally, performance may be increased by relaxing optimality constraints.

Finally, we believe that the relationship on which both of the algorithms in this paper are based should be explored further. The relationship between MUSes, CoMSSes, and the general idea of constraint conflicts could yield further algorithms and practical applications.

Acknowledgement

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).

References

[1] J. Bailey and P. J. Stuckey. "Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization." In *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages* (PADL05), volume 3350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[2] M. de la Banda, P. Stuckey, and J. Wazny. "Finding All Minimal Unsatisfiable Subsets." In *Proc. of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP 2003), pages 32-43, 2003.

[3] R. Bruni and A. Sassano. "Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae." *Electronic Notes in Discrete Mathematics*, vol. 9, 2001.

[4] R. Bruni. "Approximating Minimal Unsatisfiable Subformulae by Means of Adaptive Core Search." *Discrete Applied Mathematics*, vol. 130(2), pages 85–100, 2003.

[5] J.W. Chinneck and E.W. Dravnieks, "Locating Minimal Infeasible Constraint Sets in Linear Programs." *ORSA Journal on Computing*, Vol. 3, No. 2, pp. 157-168, 1991.

[6] N. Eén and N. Sörensson. "An Extensible SAT-solver." In *Sixth International Conference on Theory and Applications of Satisfiability Testing* (SAT03), 2003.

[7] J. Huang. "MUP: A Minimal Unsatisfiability Prover." In *Proc. of the Tenth Asia and South Pacific Design Automation Conference* (ASP-DAC), January 2005.

[8] R. M. Karp. "Reducibility Among Combinatorial Problems." In *Proc. of a Symposium on the Complexity of Computer Computations*, pages 85-103, 1972.

[9] M. Liffiton, Z. Andraus, and K. Sakallah. "From Max-SAT to Min-UNSAT: Insights and Applications." *Technical Report CSE-TR-506-05*, University of Michigan, 2005.

[10] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. L. Markov. "AMUSE: A Minimally-Unsatisfiable Subformula Extractor." In *Proc. of the 41st Annual Conference on Design Automation*, pages 518–523, ACM Press, 2004.

[11] SAT benchmarks from Automotive Product Configuration, http://www-sr.informatik.unituebingen.de/~sinz/DC/

[12] C. Sinz, A. Kaiser, and W. Küchlin. "Formal Methods for the Validation of Automotive roduct Configuration Data." In *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 17 no. 1, pages 75-97, 2003.

[13] L. Zhang and S. Malik. "Extracting small unsatisfiable cores from unsatisfiable Boolean formula." Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003.