Enumerating Infeasibility: Finding Multiple MUSes Quickly

Mark H. Liffiton and Ammar Malik

Illinois Wesleyan University, Bloomington IL 61701, USA
{mliffito,amalik}@iwu.edu
http://www.iwu.edu/~mliffito/

Abstract. Methods for analyzing infeasible constraint sets have proliferated in the past decade, commonly focused on finding maximal satisfiable subsets (MSSes) or minimal unsatisfiable subsets (MUSes). Most common are methods for producing a single such subset (one MSS or one MUS), while a few algorithms have been presented for enumerating all of the interesting subsets of a constraint set. In the case of enumerating MUSes, the existing algorithms all fall short of the best methods for producing a single MUS; that is, none come close to the ideals of 1) producing the first output as quickly as a state-of-the-art single-MUS algorithm and 2) finding each successive MUS after a similar delay. In this work, we present a novel algorithm, applicable to any type of constraint system, that enumerates MUSes in this fashion. In fact, it is structured such that one can easily "plug in" any new single-MUS algorithm as a black box to immediately match advances in that area. We perform a detailed experimental analysis of the new algorithm's performance relative to existing MUS enumeration algorithms, and we show that it avoids some severe intractability issues encountered by the others while outperforming them in the task of quickly enumerating MUSes.

1 Introduction

The most common applications of constraint systems (of any type) involve finding satisfying variable assignments for satisfiable constraint sets. As such, a huge range of algorithms exist for finding such assignments and potentially optimizing objective functions over them. Constraint sets for which no satisfying assignments exist, on the other hand, can be processed with the tools of "infeasibility analysis," a smaller but growing field of study.

Broadly, the algorithms of infeasibility analysis can be placed into two categories by the information they seek: 1) how much of an unsatisfiable constraint set can be satisfied, and 2) where in the constraint set the "problem" lies. These two categories and their solutions go by various names in the different fields where constraint systems are studied: Maximum Satisfiability (MaxSAT), Maximum Feasible Subsystem (MaxFS), and MaxCSP for the former, and Minimal[ly] Unsatisfiable Subset (MUS), Irreducible/Irredundant Infeasible/Inconsistent Subsystem (IIS), and Minimal[ly] Unsatisfiable Core (MUC) for the latter.

These two types of information are diametrically opposed ("Max" and "SAT" vs "Min" and "UNSAT"), and yet a strong connection between the two has been known since at least 1987. Specifically, researchers in the field of diagnosis identified a hitting set relationship between the two [11,17]; it is based on the fact that any satisfiable subset of a constraint system cannot fully contain any unsatisfiable subset, and thus the satisfiable set must *exclude* at least one constraint from every unsatisfiable set. This relationship is occasionally exploited in infeasibility analysis by using one type of result to guide searches for the other, such as in algorithms that enumerate MUSes by way of MaxSAT solutions [1,13], solving MaxSAT with the assistance of unsatisfiable cores [6,16], and even finding [non-minimal] unsatisfiable cores to boost MaxSAT to then produce MUSes [14].

In this work, we introduce a new algorithm for infeasibility analysis inspired by this strong connection that fills a gap in the previous work and provides fertile ground for further developments in several directions. The "gap" we address is the lack of algorithms that quickly enumerate MUSes. While several approaches for enumerating MUSes exist, all suffer from severe scalability issues, and none currently match the performance of state-of-the-art algorithms for extracting a *single* MUS from an unsatisfiable constraint set. Ideally, MUS enumeration should produce the first MUS in roughly the same time T_{MUS} taken by the best algorithms for finding a single MUS, and each additional MUS should be produced as quickly as possible, with a reasonable incremental delay being roughly that same time period T_{MUS}. The new algorithm we present here, dubbed MARCO, achieves both of these goals.

In the following sections, we first define terms and describe concepts underlying this work (Section 2), followed by a discussion of past research on enumerating MUSes (Section 3). We then present the MARCO algorithm (Section 4) and an extensive empirical analysis (Section 5) before finally concluding and outlining several research paths that continue from here (Section 6).

2 Preliminaries

In this work, we discuss problems, results, and algorithms in terms of generic sets of constraints for which the constraint type and the variable domain are not specified. Generally, then, we will be discussing an ordered set of n constraints:

$$C = \{C_1, C_2, C_3, \ldots, C_n\}$$

A given constraint C_i places restrictions on assignments to a problem's variables, and C_i is *satisfied* by any assignment that meets its restrictions. If there exists some assignment to C's variables that satisfies every constraint, C is said to be *satisfiable* or *SAT*; otherwise, it is *unsatisfiable*, *infeasible*, or *UNSAT*. Most of the algorithms we describe in this paper, and especially our own algorithm, can be applied to any set of constraints given that there exists a solving method capable of returning SAT or UNSAT for that set of constraints; we call these *constraint-agnostic* algorithms.

An infeasible constraint set C can be analyzed in a variety of ways, often in terms of producing useful subsets of C. The most common analysis is likely the maximum satisfiability problem (MaxSAT, MaxFS, MaxCSP), which produces a satisfiable subset of C with the greatest possible cardinality. Generalizing MaxSAT by considering maximality instead of maximum cardinality yields the concept of a *Maximal Satisfiable Subset* (MSS):

$$M \subseteq C \text{ is an MSS} \iff M \text{ is SAT and } \forall c \in C \setminus M : M \cup \{c\} \text{ is UNSAT}$$

An MSS is essentially a satisfiable subset of C that cannot be expanded without becoming unsatisfiable. While any solution to the MaxSAT problem is an MSS, some MSSes may be smaller than that maximum size. The complement of an MSS is often more directly useful, and we call such a minimal set (whose removal from C makes it satisfiable or "corrects" it) a *Minimal Correction Set* (MCS):

$$M \subseteq C \text{ is an MCS} \iff C \setminus M \text{ is SAT and } \forall c \in M : (C \setminus M) \cup \{c\} \text{ is UNSAT}$$

Again, the minimality is not in terms of cardinality, but rather it requires that no proper subset of M be capable of "correcting" the infeasibility. A related concept, the focus of this work, is the *Minimal Unsatisfiable Subset* (MUS):

$$M \subseteq C \text{ is an MUS} \iff M \text{ is UNSAT and } \forall c \in M : M \setminus \{c\} \text{ is SAT}$$

MUSes are most commonly considered in terms of minimizing an unsatisfiable constraint set down to a "core" reason for its unsatisfiability. In some work, they are called "unsatisfiable cores," but that is also used to refer to any unsatisfiable subset of a constraint system, regardless of its minimality. Note that the definition of an MUS need not reference the constraint set C of which it is a subset; it is really a free-standing property of any set of constraints, as it does not depend on the existence or the structure of any other constraints. However, as it is most commonly encountered in terms of finding such a minimal subset of some larger constraint set, naming it with "subset" is traditional. In OR, the concept of the *Irreducible Inconsistent Subsystem* (IIS) [15] is equivalent to that of the MUS.

Example 1. Consider the following unsatisfiable set of Boolean clauses:

$$C = \{ \underset{C_1}{(a)} , \underset{C_2}{(\neg a \vee b)} , \underset{C_3}{(\neg b)} , \underset{C_4}{(\neg a)} \}$$

C has two MUSes and three MSS/MCS pairs:

MUSes
$\{C_1, C_2, C_3\}$
$\{C_1, C_4\}$

MSSes	MCSes
$\{C_2, C_3, C_4\}$	$\{C_1\}$
$\{C_1, C_3\}$	$\{C_2, C_4\}$
$\{C_1, C_2\}$	$\{C_3, C_4\}$

Simple constraint-agnostic algorithms for finding MSSes and MUSes of a constraint set C are shown in Figure 1, and their behavior is quite similar. To find

grow$(seed, C)$
input: unsatisfiable constraint set C
input: satisfiable subset $seed \subset C$
output: an MSS of C

1. **for** $c \in C \setminus seed$:
2. **if** $seed \cup \{c\}$ **is satisfiable**:
3. $seed = seed \cup \{c\}$
4. **return** $seed$

shrink$(seed, C)$
input: unsatisfiable constraint set C
input: unsatisfiable subset $seed \subseteq C$
output: an MUS of C

1. **for** $c \in seed$:
2. **if** $seed \setminus \{c\}$ **is unsatisfiable**:
3. $seed = seed \setminus \{c\}$
4. **return** $seed$

Fig. 1. The basic **grow** and **shrink** methods for finding an MSS or an MUS, respectively, of a constraint set.

an MSS (MUS), the **grow** (**shrink**) method starts from some satisfiable (unsatisfiable) subset $seed \subseteq C$ and iteratively attempts to add (remove) constraints, checking each new set for satisfiability and keeping any changes that leave the set satisfiable (unsatisfiable). These algorithms are not novel (for example, **shrink** was described by Dravnieks in 1989 as "deletion filtering" [4]), nor are they particular efficient as shown (many improvements can be made to both), but they serve as simple illustrative examples for the purposes of this work.

Note that the input *seed* can take simple default values if no particular subset is given. The **grow** method can begin its construction with $seed = \emptyset$ (guaranteed to be satisfiable), while **shrink** can start with $seed = C$ (guaranteed UNSAT). Therefore, *seed* can be considered an optional parameter for both, and each method is also a generic method for finding an MSS or MUS of a constraint set C without any additional information. For any given constraint type and solver, both **shrink** and **grow** can be optimized to exploit characteristics of the constraints or features of the solver; most fields have a great deal of research on efficient **shrink** implementations, but **grow** is less often studied.

3 Related Work

The existing work on algorithms for enumerating MUSes is limited, especially when compared to the amount of work on extracting single MUSes and unsatisfiable cores. Some all-MUS algorithms have been developed for specific constraint types. For example, there are many methods for computing all IISes of a linear program such as the original work by van Loon [15], later work by Gleeson and Ryan [8], etc.; however, these approaches are quite specific to linear programming, constructing a polytope and using the simplex method, and they do not generalize well. Additionally, Gasca, et al. developed methods for computing all MUSes of overconstrained numerical CSPs (NCSPs) [7]. Their approach explores all subsets of a constraint system while pruning unnecessary collections of subsets with rules based on structure specific to NCSPs. In the space of *constraint-agnostic* algorithms for enumerating MUSes, three different approaches have been presented. As with the work in this paper, all of the fol-

lowing algorithms are easily applied to any type of constraint system, from CSP to IP to SAT, and none rely on specific features of any constraint type or solving method.

Subset Enumeration The technique of explicitly enumerating and checking every subset of the unsatisfiable constraint system was first explored in the field of diagnosis by Hou [10], who presented a technique for enumerating subsets in a tree structure along with pruning rules to reduce its size and avoid unnecessary work. Starting from the complete constraint set C, the algorithm searches the power set $\mathcal{P}(C)$, branching to explore all subsets. Each subset is checked for satisfiability, and any subset found to be unsatisfiable and whose children (proper subsets) are all satisfiable is an MUS. Han and Lee corrected an error in one of the pruning rules and presented additional improvements [9], and further optimizations and enhancements were made by de la Banda et al. [2].

CAMUS A later algorithm for enumerating MUSes by Liffiton and Sakallah [12,13,14] avoids an explicit search of the power set of C by exploiting the relationship between MCSes and MUSes [11,17]. CAMUS works in two phases, first computing all MCSes of the constraint set, then finding all MUSes by computing the minimal hitting sets of those MCSes. The two-phase method can be applied with any technique for enumerating MCSes and any minimal hitting set algorithm. The authors provide an algorithm for the first phase that gives a constraint solver the ability to search for satisfiable subsets of constraints without having to feed each subset to the solver individually. With this ability, the algorithm then searches for satisfiable subsets in decreasing order of size, blocking any solutions found before continuing its search, thus guaranteeing it finds only maximal satisfiable sets whose complements are the MCSes it seeks. The second phase of CAMUS, as a purely set theoretic problem, operates independently of any constraint solver.

Due to the complexity and potential intractability of the first phase (the number of MCSes may be exponential in the size of the instance), CAMUS is unsuitable for enumerating MUSes in many applications that require multiple MUSes quickly. Variations on the core algorithm can relax its completeness and adapt it to such situations [13], but the control they provide, essentially a tradeoff between time and completeness, is crude. In any case, CAMUS is not able to be run in an *incremental* fashion, with short, consistent delays between each MUS, such that one can make a decision about the time/completeness tradeoff dynamically *while* the algorithm runs.

DAA Closer to the goal of this work, providing a much more incremental approach than CAMUS, is the Dualize and Advance algorithm (DAA) by Bailey and Stuckey [1]. It exploits the same relationship between MCSes and MUSes, but it discovers both types of sets throughout its execution. Therefore, like our algorithm and unlike CAMUS, it is capable of producing MUSes "early" in its execution. Pseudocode for DAA is shown in Figure 2. It repeatedly computes MCSes by growing MSSes from seeds with the **grow** method and taking their

DAA
input: unsatisfiable constraint set C
output: MSSes and MUSes of C as they are discovered

```
1.  MCSes, MUSes, seed ← ∅
2.  haveSeed ← True
3.  while haveSeed:
4.      MSS ← grow(seed, C)
5.      yield MSS
6.      MCSes ← MCSes ∪ {C \ MSS}        ◁ the complement of an MSS is an MCS
7.      haveSeed ← False
8.      for candidate ∈ (hittingSets(MCSes) \ MUSes):
9.          if candidate is satisfiable:
10.             seed ← candidate              ◁ if SAT, candidate is a new MSS seed
11.             haveSeed ← True
12.             break
13.         else:
14.             yield candidate               ◁ if UNSAT, candidate is an MUS
15.             MUSes ← MUSes ∪ {candidate}
```

Fig. 2. The DAA algorithm for enumerating MSSes & MUSes of a constraint set.

complements. The initial seed is the empty set. It then computes the minimal hitting sets of the MCSes found thus far, as CAMUS does once it has the *complete* set of MCSes. With an incomplete set of MCSes, some of the hitting sets may be unsatisfiable, and these are guaranteed to be MUSes. DAA therefore checks each for satisfiability, reporting every unsatisfiable set as an MUS, and the first set found to be satisfiable is taken as the next seed for the algorithm to repeat.

Comparisons Bailey and Stuckey found that DAA performed much better than the subset enumeration algorithm as presented by de la Banda, et al. in their experimental evaluation [1], while somewhat limited experiments in [13] indicated that CAMUS outperformed DAA for finding all MUSes of a constraint system. However, the incremental nature of DAA is not matched by CAMUS, and so comparisons to both are warranted here. We contrast the features of DAA and CAMUS with our new algorithm following its description in Section 4, and the experimental results in Section 5 further illustrate the differences.

4 Exploring Infeasibility with the MARCO Algorithm

Here, we present a novel algorithm for enumerating all MUSes of an unsatisfiable constraint set C. (As with CAMUS and DAA, it also enumerates all MSSes of C, but they are not the focus of this work.) It efficiently explores the power set $\mathcal{P}(C)$ by exploiting the idea that any power set can be analyzed and manipulated as a Boolean algebra; that is, one can perform set operations within $\mathcal{P}(C)$ by

manipulating Boolean functions as propositional formulas. Specifically, we note that any function $f : \mathcal{P}(C) \to \{0, 1\}$ can be represented by a propositional formula over $|C|$ variables.

Our algorithm maintains a particular function $f : \mathcal{P}(C) \to \{0, 1\}$ that tracks "unexplored" subsets $C' \subseteq C$ such that $f(C') = 1$ iff the satisfiability of C' is unknown and it remains to be checked. This function, stored as a propositional formula, can be viewed as a "map" of $\mathcal{P}(C)$ showing which "regions" have been explored and which have not. Named after the Venetian explorer Marco Polo, we have dubbed the algorithm MARCO (**Ma**pping **R**egions of **Co**nstraint sets) and the general technique of maintaining a power set map as a propositional logic formula POLO (**Po**wer set **Lo**gic). Overall, MARCO enumerates MUSes by repeatedly selecting an unexplored subset $C' \in \mathcal{P}(C)$ from the map, checking whether C' is satisfiable, minimizing or maximizing it into an MUS or an MSS, and marking a region of the map as explored based on that result.

MARCO
input: unsatisfiable constraint set $C = \{C_1, C_2, C_3, \ldots, C_n\}$
output: MSSes and MUSes of C as they are discovered

```
1.  Map ← BoolFormula(nvars = |C|)   ◁ Empty formula over |C| Boolean variables
2.  while Map is satisfiable:
3.      m ← getModel(Map)
4.      seed ← {C_i ∈ C : m[x_i] = True}        ◁ Project the assignment m onto C
5.      if seed is satisfiable:
6.          MSS ← grow(seed, C)
7.          yield MSS
8.          Map ← Map ∧ blockDown(MSS)
9.      else:
10.         MUS ← shrink(seed, C)
11.         yield MUS
12.         Map ← Map ∧ blockUp(MUS)
```

Fig. 3. The MARCO algorithm for enumerating MSSes & MUSes of a constraint set.

Figure 3 contains pseudocode for the MARCO algorithm. The formula Map is created to represent the "mapping" function described above, with a variable x_i for every constraint C_i in C. Initially, the formula is a tautology, true in every model, meaning every subset of C is still unexplored. Given its semantics, any model of Map can be projected onto C (lines 3 and 4) to identify a yet-unexplored element of C's power set whose satisfiability is currently unknown. If this subset, *seed*, is satisfiable, then it must be a subset of some MSS, and it can be "grown" into an MSS. Likewise, if it is unsatisfiable, *seed* must contain at least one MUS, and it can be "shrunk" to produce one. In either case, the result

is reported (via **yield** in the pseudocode, indicating that the result is returned but the algorithm may continue).

Each result provides information about some region of $\mathcal{P}(C)$ that is either satisfiable or unsatisfiable, and so a clause is added to Map to represent that region as "explored." For an MSS M, all subsets of M are now known to be satisfiable, and so models corresponding to any subset of M are eliminated by requiring that later models of Map include at least one constraint not in M:

$$\textbf{blockDown}(M) \equiv \bigvee_{i \,:\, C_i \notin M} x_i$$

Similarly, all supersets of any MUS M are known to be unsatisfiable; supersets of M are blocked by requiring models to exclude at least one of its constraints:

$$\textbf{blockUp}(M) \equiv \bigvee_{i \,:\, C_i \in M} \neg x_i$$

Eventually, all MSSes and MUSes are enumerated, the satisfiability of every element in $\mathcal{P}(C)$ is known, and MARCO terminates when Map has no further models. We discuss implementation details after an example.

Example 2. Suppose we run MARCO on the constraint set from Example 1:

$$C = \{\underset{C_1}{(a)}\,,\ \underset{C_2}{(\neg a \vee b)}\,,\ \underset{C_3}{(\neg b)}\,,\ \underset{C_4}{(\neg a)}\ \}$$

Initialization:	$Map \leftarrow$ [empty formula over $\{x_1, x_2, x_3, x_4\}$]
Iteration 1:	$Map = \top$: **SAT**
	getModel $\rightarrow [x_1, x_2, x_3, x_4]$
	$seed \leftarrow \{C_1, C_2, C_3, C_4\}$: **UNSAT**
	shrink $\rightarrow$ MUS: $\{C_1, C_2, C_3\}$
	$Map \leftarrow Map \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$
Iteration 2:	$Map = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$: **SAT**
	getModel $\rightarrow [\neg x_1, x_2, x_3, x_4]$
	$seed \leftarrow \{C_2, C_3, C_4\}$: **SAT**
	grow $\rightarrow$ MSS: $\{C_2, C_3, C_4\}$ — equiv. MCS: $\{C_1\}$
	$Map \leftarrow Map \wedge (x_1)$
Iteration 3:	$Map = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1)$: **SAT**
	getModel $\rightarrow [x_1, \neg x_2, x_3, x_4]$
	$seed \leftarrow \{C_1, C_3, C_4\}$: **UNSAT**
	shrink $\rightarrow$ MUS: $\{C_1, C_4\}$
	$Map \leftarrow Map \wedge (\neg x_1 \vee \neg x_4)$

At this point, MARCO has found all MUSes. It must ensure completeness, however, and so it exhaustively explores all remaining subsets.

Iteration 4:	Map =	$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1)$
		$\wedge\, (\neg x_1 \vee \neg x_4)$: **SAT**
	getModel $\rightarrow$	$[x_1, \neg x_2, x_3, \neg x_4]$
	$seed \leftarrow$	$\{C_1, C_3\}$: **SAT**
	grow $\rightarrow$	MSS: $\{C_1, C_3\}$ — equiv. MCS: $\{C_2, C_4\}$
	$Map \leftarrow$	$Map \wedge (x_2 \vee x_4)$
Iteration 5:	Map =	$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1)$
		$\wedge\, (\neg x_1 \vee \neg x_4) \wedge (x_2 \vee x_4)$: **SAT**
	getModel $\rightarrow$	$[x_1, x_2, \neg x_3, \neg x_4]$
	$seed \leftarrow$	$\{C_1, C_2\}$: **SAT**
	grow $\rightarrow$	MSS: $\{C_1, C_2\}$ — equiv. MCS: $\{C_3, C_4\}$
	$Map \leftarrow$	$Map \wedge (x_3 \vee x_4)$
Iteration 6:	Map =	$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1)$
		$\wedge\, (\neg x_1 \vee \neg x_4) \wedge (x_2 \vee x_4) \wedge (x_3 \vee x_4)$: **UNSAT**

In the final iteration, all MSSes and all MUSes have been found; therefore, every model of Map is blocked, Map is UNSAT, and the algorithm terminates.

4.1 Implementation and Efficiency

An implementation of MARCO requires solvers for both C and Map. It is constraint-agnostic, as it only needs a solver that can take a set of constraints (some subset of C) and return a SAT/UNSAT result. The solver for Map is separate, and any engine for obtaining a model of a Boolean formula can be used; an incremental interface such as provided by modern SAT solvers or Binary Decision Diagram (BDD) engines will be most efficient.

MARCO's efficiency depends primarily on the implementation of the **grow** and **shrink** subroutines, as they are the most expensive steps. Constraint-agnostic methods for both are described in Section 2, but algorithms specific to a particular constraint type will be able to leverage details of those constraints for better performance. Due to the difficulty and broad applicability of extracting MUSes (the **shrink** method), much research has been done on the problem, and efficient algorithms for **shrink** exist for many constraint types. The **grow** method is less studied, and far more work is done on MaxSAT, MaxFS, etc. than on the easier problem of finding an MSS. Note that the solvers for C and Map and the methods for **grow** and **shrink** are black boxes as far as MARCO is concerned; an advance in the state-of-the-art for any one of the four can be immediately "plugged in" to boost the algorithm's performance.

4.2 Impact of the Map Solver

Another important factor for performance is the behavior of the solver for Map. The particular model returned by **getModel** cannot affect correctness, but it can impact the work done by **grow** and **shrink**. For example, imagine a simple

constraint set that is an MUS itself. If the first model found for Map corresponds to the empty set, **grow** will be called, and it will have to proceed through several steps to reach an MSS, which in this case must contain all but one constraint of C. If the models of Map continue to represent very small subsets of C, **grow** will continue to require a large number of steps in each call. On the other hand, if the first model of Map corresponds to C itself, with all constraints included, **shrink** will be called on an "easy" seed (an MUS that will not be shrunk farther). And now, if models of Map are generally large subsets of C, the MSSes will be found by much faster calls to **grow**, as it will have much less "distance" to cover to reach an MSS in each case.

Returning to the earlier stated goal of producing the first MUS as fast as a state-of-the-art single-MUS algorithm, we see that we can achieve this simply by ensuring that the first model found for Map sets all x_i variables to True, resulting in $seed = C$, the entire constraint set. This takes negligible time, and the algorithm will immediately call **shrink** on C to produce the first MUS. Given that **shrink** can be any state-of-the-art MUS extraction algorithm, MARCO can thus find the first MUS as quickly as any other algorithm.

While we cannot then guarantee that each successive model of Map will also correspond to an unsatisfiable subset of C, which would trigger further calls to **shrink** immediately, it *is* possible to bias a solver in that direction. If the solver for Map favors assigning variables to True, then it will be more likely to produce models corresponding to large, nearly-complete subsets of C, which are the subsets most likely to be unsatisfiable. In Example 2, the model m found in each iteration is biased in this way, and the first $seed$ is thus C itself. The next model, even if maximizing the number of variables assigned True, will not necessarily correspond to an unsatisfiable $seed$, as illustrated in Iteration 2, but it is still likely to locate other unsatisfiable subsets of C quickly.

4.3 Comparison to CAMUS and DAA

With regards to tractability, CAMUS suffers from the fact that its first phase may produce an intractably large set of MCSes, with no good way to make progress on MUSes until the MCSes are all found. DAA also faces a severe tractability issue in the intermediate collections of hitting sets it computes; these collections can be exponential in size even if the number of MUSes is not [1]. MARCO, on the other hand, faces no such issues; the only information stored outside of its black box solvers is the formula it maintains in Map, which grows linearly with the number of results found.

The intractability of the first phase of CAMUS also impacts the time until its first MUS output, which can be effectively infinite even for small problems. DAA fares better, but it still must find at least k MCSes before it might output an MUS of size k, meaning it may face a lengthy delay before outputting its first MUS. The very first step of MARCO, however, finds an MUS directly using an efficient MUS algorithm, and each subsequent MUS can be found in roughly the same amount of time. At an algorithmic level, MARCO is better suited to finding multiple MUSes quickly than either CAMUS or DAA.

5 Empirical Analysis

To evaluate the MARCO algorithm and to compare it to the previous approaches for MUS enumeration, CAMUS and DAA, we ran all three on a set of 300 benchmarks from the Boolean satisfiability domain. Compared to analyzing decision or optimization problems, addressing the indefinite nature of enumerating potentially intractable sets requires a more detailed analysis, and so we present a variety of analyses to illustrate each algorithm's strengths and weaknesses.

Each algorithm was implemented in C++ using the MiniSAT solver [5]. We used the most recent release of CAMUS for Boolean SAT, which is built on MiniSAT 1.12b, while we implemented MARCO and DAA using MiniSAT 2.2[1]. Both MARCO and DAA were written in the same framework so that each would share as much code as possible, including the implementation of the **grow** method. For the **shrink** method in MARCO, we used the MUSer2 algorithm [3], a state-of-the-art MUS extraction algorithm for Boolean SAT. The solver for Map in MARCO was biased toward models representing larger subsets of C, as described in Section 4.2. All experiments were run on 3.4GHz AMD Phenom II CPUs with a 3600 second timeout and a 1.8 GB memory limit.

We used the 300 benchmarks selected for the MUS track of the recent 2011 SAT Competition[2]. These benchmarks were drawn from a wide variety of applications and cover a range of sizes, from 26 clauses (constraints) up to 4.4 million. Of the 300 instances, our experiments found that 219 contained more than one MUS, 17 had exactly one MUS, and the remaining 64 were indeterminate (i.e., on these instances, every algorithm ran out of time or memory and output only zero or one MUS before it was terminated).

Table 1. Number of instances in which each algorithm found all, multiple, or at least one MUS.

	n	CAMUS	DAA	MARCO
All instances	300			
Found all MUSes		**41**	24	25
Found ≥ 1 MUS		113	51	**244**
Instances w/ >1 MUS	219			
Found all MUSes		**26**	8	11
Found > 1 MUS		98	32	**215**
Found ≥ 1 MUS		98	35	**217**

An overview of the number of instances for which each algorithm reached certain thresholds of enumerating MUSes is shown in Table 1. The results are broken out for the complete set of 300 benchmarks and for the set of 219 benchmarks that are known to contain more than one MUS. For the goal of enumerating

[1] While this may disadvantage CAMUS, our experiments have shown that it does not perform substantially better when built on top of MiniSAT 2.2.

[2] http://www.satcompetition.org/2011/

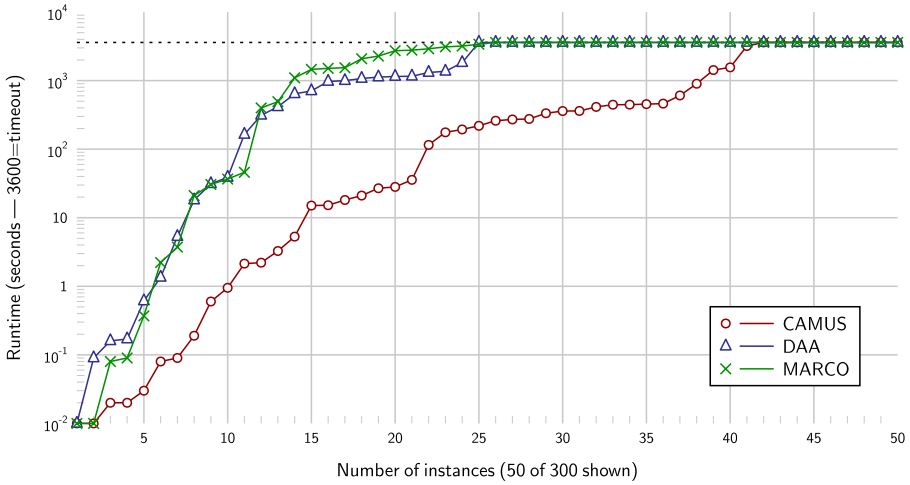

Fig. 4. Logarithmic cactus plot of each algorithm's runtime (to completion; i.e., enumerating *all* MUSes).

all MUSes, CAMUS outperforms the other two algorithms by a wide margin, completing within the time and memory limits for 41 benchmarks, compared to 24 for DAA and 25 for MARCO. Figure 4 shows a cactus plot[3] of the runtimes of the three algorithms, further supporting this point; CAMUS appears to be the best option for enumerating the *complete* set of MUSes.

It is also clear, however, that enumerating the complete set of MUSes is generally intractable (in fact, the set's cardinality may be exponential in the size of the instance), and CAMUS is outperformed by MARCO in the task emphasized by this work: enumerating *some*, but not all, MUSes. The number of instances for which MARCO can find a single MUS or multiple MUSes within the resource limits is more than twice that of CAMUS. This is consistent with the fact that the first phase of CAMUS is potentially intractable, and it often times out before reaching the second phase and producing even a single MUS. The DAA algorithm is outperformed by CAMUS in enumerating all MUSes (which agrees with earlier, more limited results [13]), and DAA produces no output at all in far more instances than either algorithm, especially MARCO. DAA most commonly exhausts its memory limit, due primarily to the number of intermediate hitting sets it generates in every iteration, and in many cases memory is exhausted before a single MUS has been found. Therefore, the remainder of the analysis will focus primarily on comparing MARCO and CAMUS.

[3] Cactus plots are created by sorting and plotting values in order within each series, showing distributions of values within a series, but not allowing pairwise comparisons between them. Each point (x, y) can be read as, "x instances have a value [e.g., runtime] of y or less."

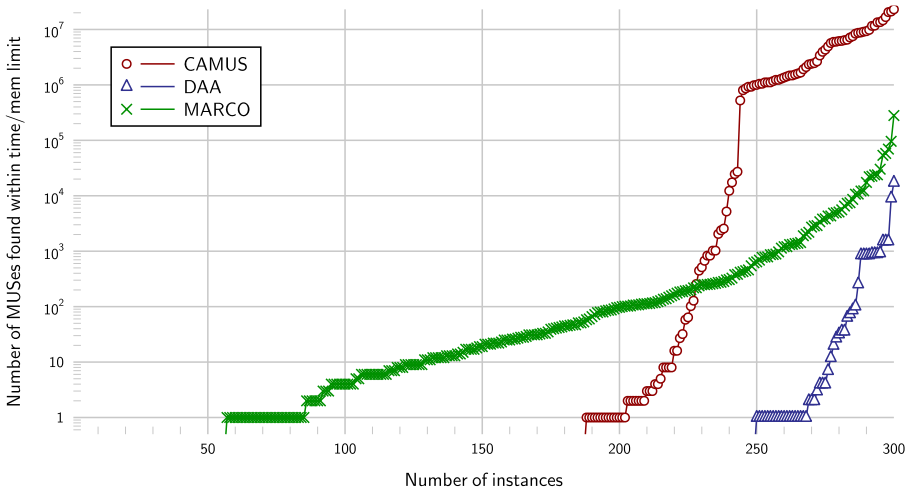

Fig. 5. Logarithmic cactus plot of the number of MUSes found by each algorithm within time / memory limits.

Another view of the difference between MARCO and CAMUS is shown in a cactus plot of the numbers of MUSes found by each algorithm in Figure 5. This chart echoes some of the information in Table 1, showing the number of instances in which each algorithm is able to find one or more than one MUS, but it adds additional information about other output counts as well. For example, we can see that MARCO produces 10 or more MUSes in more than 170 instances, while CAMUS does so in only about 80 instances. However, CAMUS finds much larger sets of MUSes within the timeout in many instances, returning over 10^6 results in more than 50 instances, while MARCO only reaches above 10^5 results in one instance. This suggests that CAMUS will produce many more MUSes than MARCO, *when* it produces any, but MARCO is more robust in terms of scaling to produce some MUSes for more instances overall.

Figure 6 explores this further with pairwise comparisons of the number of MUSes found. DAA never produces more MUSes than MARCO. CAMUS, on the other hand, often produces orders of magnitude more MUSes. However, the chart also shows the large set of instances for which CAMUS outputs nothing and MARCO produces multiple MUSes; the reverse is true in only two instances.

Finally, to further contrast the performance of CAMUS and MARCO, we can look at anytime charts of their output over time, showing how many MUSes will be produced if execution is stopped at any particular time. The anytime charts in Figure 7 contain one trace for each instance that had 10 or more outputs, plotting the number of MUSes produced on the y-axis against time on the x-axis. For the sake of comparison, the data have been normalized to a scale of 0.0 to 1.0 such that 1.0 represents 100% of each algorithm's runtime on each instance (on the x-axis) or 100% of the MUSes it found in that time (y-axis). On

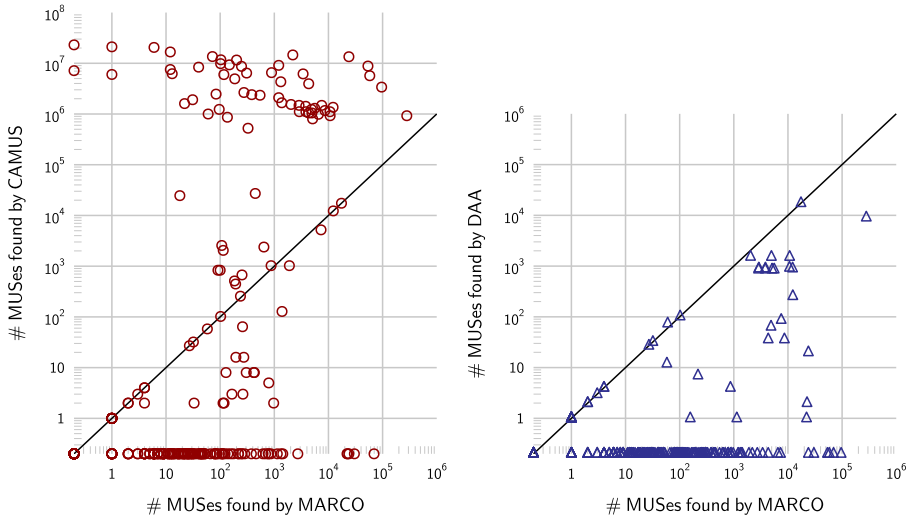

Fig. 6. Comparing MARCO to CAMUS (left) and DAA (right): number of MUSes found within time / memory limits (counts of 0 remapped to 0.2 to lie on the axis).

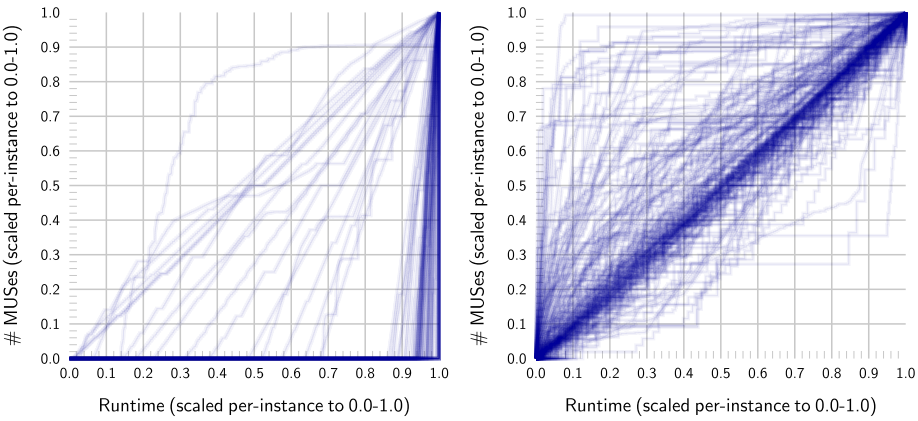

Fig. 7. Normalized anytime charts for CAMUS (left) and MARCO (right).

these charts, we can see that CAMUS typically outputs the great majority of an instance's MUSes "late," in the last 10% or less of its runtime (the dark, nearly vertical band on the right of the chart). This is consistent with its operation in two phases, where MUSes are only output in the second phase, and it shows that there is typically a long delay before any output is produced. In contrast, MARCO most often produces MUSes in a fairly steady progression (seen in the darkest band of traces along the diagonal) with a regular delay between each output, and it produces MUSes "early" (traces above the diagonal) more often than "late" (traces below). Overall, CAMUS can produce MUSes more quickly than MARCO, *if* it produces any at all (i.e., if it is given enough time), while MARCO outputs them at a much more steady pace from the beginning, making it more suitable for computing *some* MUSes quickly.

6 Conclusion

We have presented MARCO (**Ma**pping **R**egions of **Co**nstraint sets), a novel algorithm for enumerating MUSes in any type of constraint system, with the focus on producing *multiple* MUSes *quickly*, and empirical results show that MARCO outperforms other MUS enumeration algorithms at this task. It operates within the POLO framework (**Po**wer set **Lo**gic), maintaining a "map" of a constraint set's power set in a propositional logic formula, marking "explored" areas of the power set as it progresses, and using the map to find new subsets to check. Experimental results show that the CAMUS algorithm can enumerate the *complete* set of MUSes faster than MARCO, but the faster *early* results of MARCO are preferable in any application for which one wants multiple MUSes within some time limit and for which all MUSes are not needed, especially if they number in the millions or higher. MARCO can be implemented on top of any existing constraint solver, and in fact its critical **shrink** method can be implemented by "plugging in" any state-of-the art single-MUS extraction algorithm; therefore, it can always mirror the performance of any advances in MUS extraction.

Future research directions include exploring the effects of biasing and other heuristics in the solver for the "map" formula, as well as changes to the algorithm that focus it on enumerating MUSes alone, at the expense of missing some MSSes, or vice versa. Additionally, there are many opportunities to relax its completeness and/or optimality to be able to produce results in cases that remain too difficult for finding minimal unsatisfiable subsets as opposed to just small unsatisfiable cores. And finally, the general POLO technique provides a foundation from which new infeasibility analyses may be developed, such as new algorithms for finding a smallest MUS (SMUS) of a constraint set or for solving the MaxSAT/MaxFS/MaxCSP problem.

Acknowledgements

Many thanks to Anton Belov for providing assistance with the MUSer2 source code and to the anonymous reviewers for their helpful comments.

References

1. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'05). LNCS, vol. 3350, pp. 174–186 (2005)
2. de la Banda, M.J.G., Stuckey, P.J., Wazny, J.: Finding all minimal unsatisfiable subsets. In: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming (PPDP'03). pp. 32–43 (2003)
3. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. Journal on Satisfiability, Boolean Modeling and Computation (2012), in press
4. Dravnieks, E.W.: Identifying minimal sets of inconsistent constraints in linear programs: deletion, squeeze and sensitivity filtering. Master's thesis, Carleton University (1989), https://curve.carleton.ca/theses/22864
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003). LNCS, vol. 2919, pp. 502–518 (2003)
6. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006). LNCS, vol. 4121, pp. 252–265 (2006)
7. Gasca, R.M., Valle, C.D., López, M.T.G., Ceballos, R.: NMUS: Structural analysis for improving the derivation of all MUSes in overconstrained numeric CSPs. In: Current Topics in Artificial Intelligence, 12th Conference of the Spanish Association for Artificial Intelligence (CAEPIA 2007). LNCS, vol. 4788, pp. 160–169 (2007)
8. Gleeson, J., Ryan, J.: Identifying minimally infeasible subsytems. ORSA Journal on Computing 2(1), 61–67 (1990)
9. Han, B., Lee, S.J.: Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. IEEE Transactions on Systems, Man, and Cybernetics, Part B 29(2), 281–286 (Apr 1999)
10. Hou, A.: A theory of measurement in diagnosis from first principles. Artificial Intelligence 65(2), 281–328 (1994)
11. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial Intelligence 32(1), 97–130 (1987)
12. Liffiton, M.H., Sakallah, K.A.: On finding all minimally unsatisfiable subformulas. In: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005). LNCS, vol. 3569, pp. 173–186 (2005)
13. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (Jan 2008)
14. Liffiton, M.H., Sakallah, K.A.: Generalizing core-guided Max-SAT. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT-2009). LNCS, vol. 5584, pp. 481–494 (2009)
15. van Loon, J.: Irreducibly inconsistent systems of linear inequalities. European Journal of Operational Research 8(3), 283–288 (Nov 1981)
16. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'08) (Mar 2008)
17. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)