Constraints (2016) 21:223–250
DOI 10.1007/s10601-015-9183-0

Fast, flexible MUS enumeration

Mark H. Liffiton · Alessandro Previti · Ammar Malik · Joao Marques-Silva

Published online: 22 March 2015

Abstract The problem of enumerating minimal unsatisfiable subsets (MUSes) of an infeasible constraint system is challenging due first to the complexity of computing even a single MUS and second to the potentially intractable number of MUSes an instance may contain. In the face of the latter issue, when complete enumeration is not feasible, a partial enumeration of MUSes can be valuable, ideally with a time cost for each MUS output no greater than that needed to extract a single MUS. Recently, two papers independently presented a new MUS enumeration algorithm well suited to partial MUS enumeration (Liffiton and Malik, 2013, Previti and Marques-Silva, 2013). The algorithm exhibits good anytime performance, steadily producing MUSes throughout its execution; it is constraint agnostic, applying equally well to any type of constraint system; and its flexible structure allows it to incorporate advances in single MUS extraction algorithms and eases the creation of further improvements and modifications. This paper unifies and expands upon the earlier work, presenting a detailed explanation of the algorithm's operation in a framework that also enables clearer comparisons to previous approaches, and we present a new optimization of the algorithm as well. Expanded experimental results illustrate the algorithm's improvement over past approaches and newly explore some of its variants.

M. H. Liffiton (✉) · A. Malik
Illinois Wesleyan University, Bloomington, IL, USA
e-mail: mliffito@iwu.edu

A. Malik
e-mail: amalik@iwu.edu

A. Previti · J. Marques-Silva
Complex and Adaptive Systems Laboratory, University College Dublin, Dublin, Ireland

A. Previti
e-mail: alessandro.previti@ucdconnect.ie

J. Marques-Silva
e-mail: jpms@tecnico.ulisboa.pt

J. Marques-Silva
INESC-ID, IST, University Lisboa, Lisboa, Portugal

Keywords Minimal unsatisfiable subsets · Minimal correction sets · MUS enumeration · Infeasibility analysis

1 Introduction

Within constraint processing, research into algorithms that analyze unsatisfiable instances, or *infeasibility analysis*, can be split into two areas: algorithms that extract a single result or characteristic from an instance, such as a minimal unsatisfiable subset or maximal satisfiable subset, and those that enumerate multiple or all answers of a given type. Problems of complete enumeration in the latter group are intractable in the general case due to their potentially exponential number of results, but applications can still derive benefit from multiple results produced within some time or memory limit even when the complete set of results is impossibly large. For example, counterexample-guided abstraction refinement (CEGAR) model-checking systems can benefit from the enumeration of minimal unsatisfiable subsets (MUSes). In CEGAR flows, an infeasible constraint system arises when a spurious result is found due to the abstraction's overapproximation, and each MUS of that system provides a refinement of the abstraction that eliminates a different class of spurious results. By enumerating those MUSes and making multiple refinements within a single iteration, a CEGAR flow can reduce both the number of iterations required to converge and the runtime compared to computing a single refinement per iteration [1].

Compared to work on computing a single result for an infeasible instance, there are few published approaches to enumerating all or multiple MUSes. Previous approaches, discussed in Section 4, exhibit poor scaling even when the goal is *partial enumeration*, producing *some* MUSes when the complete set is intractably large. Furthermore, most existing enumeration algorithms were published before recent advances in extracting a single MUS and thus do not make use of the new tools or ideas therein. A reasonable goal for MUS enumeration would be to produce the first MUS output in time similar to that of a state-of-the-art single-MUS algorithm, followed by a roughly equivalent delay between each later MUS produced, but no existing work achieved this. Two recent papers independently presented a new approach to MUS enumeration; the MARCO algorithm by Liffiton and Malik [21] and eMUS by Previti and Marques-Silva [28] both met this goal. These algorithms, nearly identical in implementation, both have the following qualities:

- They are *constraint agnostic*, meaning they apply equally well to any type of constraint system and have no dependencies on any particular constraint feature.
- They can directly employ any state-of-the-art MUS extraction algorithm, immediately benefiting from current and future advances in that area.
- They are effective anytime algorithms, producing MUSes "early and often" and generally at a steady rate.

This paper seeks, first, to present the MARCO/eMUS algorithm in a unified, comprehensive manner. Here, we use the name MARCO[1] throughout to refer to the algorithm. We use a framework for both describing and visualizing infeasibility analysis algorithms in terms of their operation within the *power set lattice* for an infeasible constraint system. Within this framework, we expand the survey of previous work, provide a more complete description and analysis of the MARCO algorithm, and relate the new and old algorithms more directly

[1] From "**Ma**pping **R**egions of **Co**nstraints," which describes a major aspect of the algorithm's operation.

than earlier publications. Furthermore, we present improvements to the MARCO algorithm and its implementation and an expanded empirical analysis of MARCO, its variants, and past approaches. For source code, detailed experimental data, etc., please refer to the project's website: http://www.iwu.edu/~mliffito/marco/.

In the following, we first define terms and describe concepts underlying this work (Section 2) and then describe the power set lattice framework for understanding, analyzing, and comparing infeasibility analysis algorithms (Section 3). We use this framework to discuss past work on MUS enumeration (Section 4), followed by the presentation of the MARCO algorithm (Section 5), first in a basic form, then with optimizations and other variations. We provide comparisons both with previous work and amongst MARCO's variants with an extensive empirical analysis (Section 6), then conclude and describe some avenues for future work (Section 7).

2 Preliminaries

To begin, we will briefly define the terms and concepts used in this paper. The problem of interest in this work is one of analyzing infeasible constraint systems. We're interested in *constraint agnostic* approaches, for which the specific type of constraint is unimportant, hence we will consider a constraint system C as an ordered set of n abstract constraints $\{C_1, C_2, \ldots, C_n\}$ over some set of variables. Each constraint C_i restricts the "allowed" assignments to those variables in some way. The main requirement is that we must have some method of determining, for any subset $S \subseteq C$, whether S is *satisfiable* (SAT), meaning there exists an assignment that is allowed by all $C_i \in S$, or *unsatisfiable* (UNSAT), meaning no such assignment exists.

Boolean Satisfiability (also referred to as SAT)[2] is a simple type of constraint problem that some algorithms use as an auxiliary reasoning device in this work, and it can provide concrete examples of constraint systems throughout. Such constraint systems are presented as Boolean logic formulas in conjunctive normal form (CNF), where C is a conjunction of constraints known as *clauses* $C = \bigwedge_{i=1..n} C_i$; each C_i is a disjunction of literals $C_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik_i}$; and each literal l_{ij} is either a Boolean variable x or its negation $\neg x$. A CNF formula is satisfiable iff there exists an assignment of truth values to its variables such that the formula evaluates to True. The following formula serves as a running example throughout this paper. The clauses are numbered, and they will be referred to by number for brevity.

$$C = \{1 : (a),\ 2 : (\neg a),\ 3 : (\neg a \vee b),\ 4 : (\neg b)\}$$

This example formula is unsatisfiable; this is easily seen by noting that no assignment to a can satisfy constraints 1 and 2 simultaneously.

Some useful analyses of unsatisfiable constraint systems consist in identifying subsets of those systems with particular properties:

- A *minimal unsatisfiable subset* (MUS) M of a constraint system C is a subset $M \subseteq C$ such that M is unsatisfiable and $\forall c \in M : M \setminus \{c\}$ is satisfiable. An MUS can be seen as a minimal explanation of the constraint system's infeasibility.
- A *maximal satisfiable subset* (MSS) M of a constraint system C is a subset $M \subseteq C$ such that M is satisfiable and $\forall c \in C \setminus M : M \cup \{c\}$ is unsatisfiable. The definition of

[2]The intended meaning of SAT, either the adjective or the type of constraint system, will be clear in context.

an MSS is symmetric to that of an MUS, with "satisfiable" and "unsatisfiable" swapped along with maximal for minimal.

– A *minimal correction set* (MCS) M of a constraint system C is a subset $M \subseteq C$ such that $C \setminus M$ is satisfiable and $\forall S \subset M : C \setminus S$ is unsatisfiable. MCSes are so named due to the fact that their removal from C can be seen to "correct" the infeasibility.

It is important to distinguish between subset minimality and maximality as we've used here and minimum or maximum *cardinality* subsets of each type. For example, the Max-SAT problem is concerned with identifying a maximum cardinality satisfiable subset of a CNF formula; this is necessarily an MSS, but there can be MSSes of smaller cardinality as well.

MSSes and MCSes are complementary; any MSS of C is the complement, relative to C, of some MCS and vice versa. They are two sides of the same coin, providing two ways of encoding the same information, and we can thus use the terms somewhat interchangeably. Minimal correction sets are typically more useful in practice than maximal satisfiable subsets, as an MCS is often smaller than its complementary MSS, and MCSes provide information more directly relevant to the "conflict" in an infeasible constraint system. Throughout this work, we will use whichever subset type is most relevant and makes the explanation clearest in a given section. Our running example has two MUSes and three MSS/MCS pairs:

MUSes
$\{1, 2\}$
$\{1, 3, 4\}$

MSSes	MCSes
$\{2, 3, 4\}$	$\{1\}$
$\{1, 3\}$	$\{2, 4\}$
$\{1, 4\}$	$\{2, 3\}$

Finally, we will make use of a duality between MUSes and MCSes defined in terms of *hitting sets* that many MUS enumeration algorithms have exploited in the past. A hitting set of a collection of sets A is a set H such that every set in A is "hit" by H; that is, H contains at least one element from every set in A. Further, a *minimal hitting set* H_{min} is a hitting set that is subset minimal; no element can be removed from H_{min} without removing its hitting set property and "missing" some set in A. Section 3.2 discusses how every MUS of a constraint system is a minimal hitting set of the system's MCSes and every MCS is a minimal hitting set of its MUSes, first noted by Reiter [29] and de Kleer and Williams [7]. This duality can be verified in our running example above.

3 Power set exploration

MARCO and other algorithms in the domain of infeasibility analysis can be understood and compared by viewing them as methodical explorations of power sets. The goal of MUS/MSS enumeration algorithms given a set of constraints C can be seen as the exploration of the power set of the constraints $\mathcal{P}(C)$. We can visualize the power set as a lattice in a Hasse diagram as shown in Fig. 1, in which each level contains subsets of a certain size and edges link sets with their immediate supersets and subsets. Furthermore, we will view the power set $\mathcal{P}(C)$ as a Boolean algebra and reason about it with Boolean functions in the form of propositional formulas.

3.1 Maps

In exploring this power set, an algorithm will implicitly or explicitly be determining the feasibility of various subsets of C. A map (in the mathematical sense) from subsets to their feasibility,

$$f : X \subseteq C \rightarrow \{\text{SAT}, \text{UNSAT}\}$$

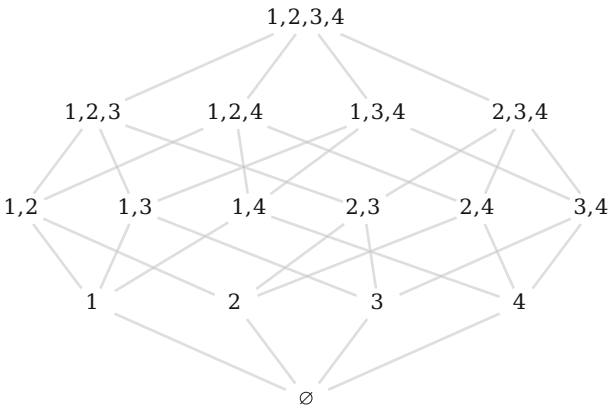


Fig. 1 Hasse diagram of the power set lattice for a generic set of four constraints $C = \{1, 2, 3, 4\}$

then can be represented by a map (in the cartographic sense) made by coloring each node in the power set lattice with a color representing its satisfiability. For our running example from Section 2, the fully colored map of C is shown in Fig. 2. The UNSAT and SAT regions are marked, with the MUSes ($\{1, 2\}$, $\{1, 3, 4\}$) and MSSes ($\{1, 3\}$, $\{1, 4\}$, $\{2, 3, 4\}$) circled in each. We can state a few simple facts about infeasible constraint sets along with the interpretation of each in the context of the Hasse diagram.

- Every subset of $\mathcal{P}(C)$ is either SAT or UNSAT — In the diagrams, we will color SAT nodes green and UNSAT nodes red.[3]
- If a given subset is SAT (UNSAT), then all of its subsets (supersets) are SAT (UNSAT) as well — Given a green (red) node, all successors (predecessors) will be the same color.
- From the definition of MSS and MUS, MSSes and MUSes are maximal green or minimal red nodes within their respective regions of the same color — These nodes will be outlined in the diagrams.

A fully colored power set lattice, with every subset marked as either SAT or UNSAT, fully describes what we will call the *feasibility* of the constraint set C. Many problems of infeasibility analysis, such as the Max-SAT problem, extracting an UNSAT core, or enumerating all MUSes, can be seen as problems of filling in some or all of the map with the correct colors either to identify a point with certain characteristics (e.g. Max-SAT, UNSAT core) or to fully determine every point in it (e.g. MUS/MSS enumeration).

3.2 Hitting set dualization

Complete enumeration can also be considered as a process of identifying the *frontier* between the SAT and the UNSAT regions. The frontier can be fully specified with either the set of "low" UNSAT points (the MUSes) or the set of "high" SAT points (the MSSes), as these are duals of one another. In the domain of model-based diagnosis, closely related

[3]For the color blind and those reading black and white copies, the SAT and UNSAT regions can be differentiated by the edges *between* SAT and UNSAT nodes, which are lighter gray than the others. Then, the region that includes the bottom $\emptyset$ node is SAT and the other is UNSAT.

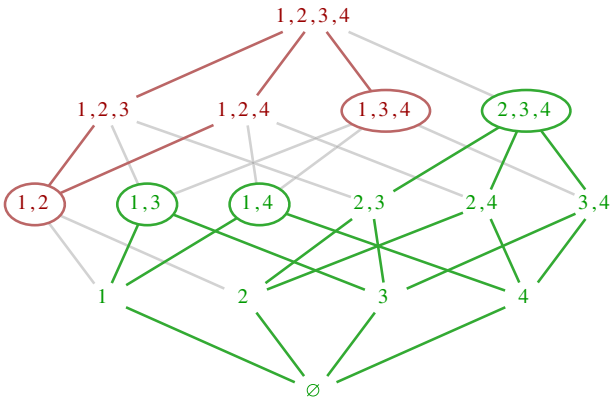


Fig. 2 Hasse diagram / map of our running example, where $C = \{1 : (a),\ 2 : (\neg a),\ 3 : (\neg a \vee b),\ 4 : (\neg b)\}$

to infeasibility analysis, Reiter [29] and de Kleer and Williams [7] both noted this duality, and it has since proven useful in infeasibility analysis. Diagnosis deals with analyzing a *system*, comprised of multiple components, that has failed some test, producing observations that do not match the expected outcomes for that system. For a given system, one can find *diagnoses*, minimal subsets of the components whose failure could explain the incorrect observations, and *conflicts*, minimal subsets of the components that, if all operating correctly, could not have produced those observations. Mapped into the domain of constraints, diagnoses are equivalent to minimal correction sets, and conflicts are minimal unsatisfiable subsets; many characteristics of diagnoses and conflicts apply to MCSes and MUSes as well.

The duality can be stated briefly: any MCS of an instance is a minimal hitting set of the collection of MUSes for that instance, and any MUS is a minimal hitting set of the MCSes (Theorem 4.4 and Corollary 4.5 in [29]). One ramification of this is that the collection of all MUSes ("low" UNSAT points) is an implicit encoding of the set of MCSes (complements of the "high" SAT points) and vice-versa; one can translate from one to the other by enumerating minimal hitting sets of abstract elements, i.e. constraint indexes, without dealing with the underlying constraints at all. Many infeasibility analysis algorithms have exploited this duality, either explicitly or implicitly computing minimal hitting sets, since Reiter's work [29]; we describe existing MUS enumeration algorithms that do so in Section 4.3.

3.3 Exploring infeasibility

Considering the problem of enumerating MUSes and MSSes as one of exploring and mapping the infeasibility of a constraint set can provide helpful insight into the operation of many algorithms. First, we will clarify that in the course of any such exploration of a constraint set C, there are in fact *two* functions of interest:

1. The constraint set's "ground truth," G, which describes the feasibility (SAT or UNSAT) of every subset of C:

$$G : \mathcal{P}(C) \rightarrow \{\text{False (UNSAT)}, \text{True (SAT)}\}$$

   This is not initially known, but it is implicit in the constraints of the instance. Figure 2 is thus a representation of G for the running example.

2. A map of what has been explored, Map, which maps from subsets of C to whether or not their feasibility has yet been determined. Here, we will use the interpretation that $Map(X \subseteq C)$ is True if X is **un**explored and its feasibility is **un**determined and False otherwise:

$$Map : \mathcal{P}(C) \rightarrow \{\text{False (explored), True (unexplored)}\}$$

A given algorithm seeks to determine the function G, either in whole or in part, while Map keeps track of which parts of G have been determined.

We can then formally define the *unexplored subset problem* for a constraint set C, a set of known-unsatisfiable subsets U, and a set of known-satisfiable subsets S as follows:

- We define a dominance relation over subsets in U and S that indicates "explored" points. A subset $X \subseteq C$ is dominated by a subset $Y \in U \iff X \supseteq Y$, and a subset $X \subseteq C$ is dominated by a subset $Z \in S \iff X \subseteq Z$. In other words, a subset is dominated by an element of U or S iff that subset's feasibility is known and thus it is "explored."
- The solution to the unexplored subset problem is a subset $X \subseteq C$ such that X is not dominated by any element of U or S if such a subset X exists or NULL otherwise.

Note that subsets in U and S need not be minimal or maximal, respectively, and that U and S can both be empty.

The unexplored subset problem is equivalent to finding a subset X for which $Map(X) =$ True. Note that any function $f : \mathcal{P}(C) \rightarrow \{\text{False, True}\}$ can be represented by a propositional formula over $|C|$ variables. The unexplored subset problem can be solved by formulating Map as a Boolean CNF formula `map` as follows:

- Every constraint $C_i \in C$ is assigned a Boolean variable x_i.
- Any complete assignment indicates a subset $X \subseteq C$ by the variables assigned True:

$$x_i = \text{True} \iff C_i \in X$$

- For every known-unsatisfiable constraint set $Y \in U$, `map` contains a clause:

$$\bigvee_{i:C_i \in Y} \neg x_i$$

  This marks every superset of Y as explored; their infeasibility is known as they all must be unsatisfiable.
- Similarly, for every known-satisfiable constraint set $Z \in S$, `map` contains a clause using the complement of Z with respect to C:

$$\bigvee_{i:C_i \notin Z} x_i$$

  This marks every subset of Z as explored, as all such subsets are known to be satisfiable.

Lemma 1 (Correctness) *Given the above formulation of* `map` *for the unexplored subset problem for a constraint set C, every model of* `map` *indicates a subset of C that is not dominated by any subset in U or S (i.e., it is unexplored).*

Proof by contradiction in cases We prove separately that it can not be dominated by a subset in U nor by a subset in S.

Case 1: Assume there exists some model of `map` such that the indicated subset is dominated by a known-unsatisfiable subset $Y \in U$. Thus, $\forall C_i \in Y, x_i$ is assigned True. However,

map contains a clause $\bigvee_{C_i \in Y} \neg x_i$. The model violates this clause, and it follows that the assumption must be false, proving Case 1 impossible.

Case 2: Assume there exists some model of map such that the indicated subset is dominated by a known-satisfiable subset $Z \in S$. Thus, $\forall C_i \notin Z$, x_i is assigned False. However, map contains a clause $\bigvee_{C_i \notin Z} x_i$. The model violates this clause, and it follows that the assumption must be false, proving Case 2 impossible.

Case 1 and Case 2 together prove that no model of map indicates a subset of C that is dominated by any constraint set in U nor by any in S, and thus the lemma holds. □

Lemma 2 (Completeness) *Given the above formulation of* map *for the unexplored subset problem for a constraint set C,* map *is unsatisfiable if and only if U contains all MUSes of C and S contains all MSSes of C (and thus every subset of C is explored).*

Proof The formula map is unsatisfiable iff every complete assignment falsifies at least one clause in map. Additionally, every clause in map is falsified by only the assignments that indicate a subset of C dominated by the constraint set from which that clause was generated. Therefore, every complete assignment falsifies at least one clause in map iff every subset of C is dominated by some unsatisfiable subset in U or some satisfiable subset in S. Every subset of C is dominated by some set in U or some set in S iff U contains all MUSes of C and S contains all MSSes of C. By the transitive property of iff, the lemma holds. □

Theorem 1 The above formulation of map provides a complete, correct solution for the unexplored subset problem.

Proof By Lemmas 1 and 2. □

3.4 Implementation

The unexplored subset problem can form the basis of algorithms for exploring the infeasibility of a constraint set. In any algorithm that explores $\mathcal{P}(C)$, the power set is initially unexplored, hence map $= \top$. Any time a region of $\mathcal{P}(C)$ is explored and its feasibility is determined, it can be marked as such by adding one or more clauses to map to block or remove the corresponding models. For example, an algorithm could check the satisfiability of C itself and determine it to be UNSAT. Then, it could block the model of map corresponding to C, $[x_1 = x_2 = x_3 = x_4 =$ True] (i.e., all constraints of C are included), by adding a single clause to map: $(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4)$. The map formula would then represent the fact that C is explored and all proper subsets of C remain unexplored.

With the view of C in terms of its power set lattice, the two functions G and Map, and the formulation of the unexplored subset problem into the Boolean CNF formula map, we can describe useful "actions" that can be taken in the context of exploring a power set. They are presented here as subroutines that we will use in later algorithm pseudocode. Note that these are all constraint-agnostic, and they can be implemented on top of existing solvers for any constraint type.

– **GetUnexplored**(map) → subset X
  Assuming map is satisfiable (indicating there is at least one subset still unexplored), we can obtain an unexplored subset of C by getting a model of map using a Boolean satisfiability solver.

- **SAT**(subset X) $\rightarrow$ {T, F}
  We can check whether any subset $X \subseteq C$ is feasible simply by sending it to a constraint solver. This determines one point in G, the underlying ground truth of the feasibility of C.
- **Shrink**(unsatisfiable subset X) $\rightarrow$ MUS M
  Given a known-unsatisfiable subset X, we can extract an MUS $M \subseteq X$ via any single-MUS extraction algorithm. One of the simplest such algorithms involves a series of **SAT** calls to explore proper subsets of X, descending through the lattice on any UNSAT subsets found until one is reached whose children are all SAT.
- **Grow**(satisfiable subset X) $\rightarrow$ MSS M
  As the dual of **Shrink**, we can find an MSS with any known-satisfiable subset X as a starting point. A method that "walks" up through the lattice by identifying SAT supersets can be employed here. Note that this is *not* equivalent to solving the Max-SAT problem, as an MSS need not have the largest possible cardinality.
- **BlockUp**(unsatisfiable subset X) $\rightarrow$ clause B
  We can mark a region of the lattice as explored by adding a new clause to `map` following the formulation above:

$$B = \bigvee_{i:C_i \in X} \neg x_i$$

- **BlockDown**(satisfiable subset X) $\rightarrow$ clause B
  Likewise, a satisfiable subset and all of its subsets can be marked as explored with a similar clause:

$$B = \bigvee_{i:C_i \notin X} x_i$$

3.5 Power set exploration in other work

These ideas are present in a range of infeasibility analysis algorithms—sometimes explicitly acknowledged in similar terms, though often just implicit in the operation of an algorithm—including both MUS enumeration algorithms as detailed in Section 4 and the optimization problems Max-SAT and Max-SMT (Max-SMT is Max-SAT raised to the more general constraint types in SAT Modulo Theories, or SMT). For example, some "core-guided" Max-SAT algorithms [27] operate by finding unsatisfiable cores, removing their supersets from a search space with blocking clauses like **BlockUp**, and implicitly searching for the smallest hitting set of the cores found thus far using a constraint solver. In other words, they find the highest unexplored point under a known-UNSAT region; if that point is found to be UNSAT, a core is extracted from it, the known-UNSAT region is expanded, and the process repeats.

Davies and Bacchus more directly exploit the hitting set duality to solve Max-SAT [6] while Cimatti, et al. follow a similar approach for Max-SMT [5]. Additionally, both explore and "map" the power set of a set of constraints with a secondary abstraction like the `map` formula discussed above. With the use of these techniques, both of these approaches are strongly related to the MARCO algorithm, despite solving a much different problem. By considering an infeasibility analysis algorithm in terms of its exploration of a power set, these connections become clearer. We describe previous MUS enumeration work in similar terms to aid comparison and draw further connections.

4 Enumerating MUSes: Past approaches

The existing work on enumerating MUSes of infeasible constraint systems is somewhat sparse, especially in relation to the amount of research done on extracting single MUSes and non-minimal UNSAT cores. We separate the work into algorithms that are tightly coupled to specific constraint types and those that are more generalizable and constraint-agnostic, and we focus on the latter as most relevant to this paper.

4.1 Specific constraint types

Several MUS enumeration algorithms have been presented whose operation relies on features of a particular type of constraint. In operations research (OR), for example, and particularly for linear programs (LP) and integer linear programs (ILP), MUSes are known as *Irreducible Inconsistent Subsystems* (IISes). The OR literature contains several methods for computing all IISes of an LP, such as the original work by van Loon [31], later work by Gleeson and Ryan [13], etc. However, these approaches are specific to linear programming, relying on techniques like constructing polytopes and the simplex method, and they do not generalize well.

In the constraint programming domain, Gasca, et al. developed methods for computing all MUSes of overconstrained numerical CSPs (NCSPs) [12]. NCSPs consist of numeric variables defined over the reals and constraints in the form of inequalities or equalities between linear or polynomial combinations of the variables. Their approach explores all subsets of a constraint system while pruning unnecessary collections of subsets with rules based on structure specific to NCSPs that also do not generalize.

4.2 Constraint-agnostic: Subset enumeration

In the space of constraint-agnostic algorithms for enumerating MUSes, a few different classes of algorithms have been presented. As with the work in this paper, all of the following 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.

Early constraint-agnostic algorithms relied on explicit subset enumeration, exploring the power set element-by-element. This technique was first explored in the field of diagnosis by Hou [17], 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. This can be visualized as depth-first search through the power set lattice (as in Fig. 1), calling **SAT** on each node, backtracking when a satisfiable subset is reached, and noting which unsatisfiable nodes are "low points" and thus MUSes. Han and Lee corrected an error in one of the pruning rules and presented additional improvements to the technique [16], and further optimizations and enhancements were made by García de la Banda et al. [8].

In work primarily focused on extracting "preferred explanations" (MUSes w.r.t. preferences on constraints), Junker briefly describes a method for enumerating MUSes that also enumerates subsets, though it is not presented in detail [18]. Junker's algorithm operates by branching on each constraint C_i, in one direction removing C_i and recursively enumerating MUSes if the remaining subsystem is unsatisfiable, and in the other direction removing

other constraints and finding MUSes that contain C_i by an unspecified mechanism. Effectively, this walks through the power set subset-by-subset as well. The primary drawback of the general subset enumeration approach is the large number of **SAT** calls required as the infeasible region of the power set and its frontier is mapped subset-by-subset. Later approaches provide better efficiency and scaling by reducing the number of subsets that are checked explicitly.

4.3 Constraint-agnostic: Hitting set dualization

CAMUS The CAMUS algorithm presented by Liffiton and Sakallah [22, 23] uses the hitting set MUS/MCS duality discussed in Section 3.2 to produce MUSes by first computing all MCSes of an instance, then computing minimal hitting sets of them; as noted, each such hitting set is an MUS. It computes MCSes with what can be considered a "top-down" search through the power set, searching a level (subsets of a particular size) for satisfiable subsets that are not subsumed by some larger satisfiable subset found in a higher level.

Specifically, every constraint C_i is augmented with a selector variable y_i to become $C'_i = \neg y_i \vee C_i$ such that an assignment of True to y_i "enables" C_i and False "disables" it. Any satisfiable model of the augmented system $C' = \bigwedge C'_i$ thus indicates a satisfiable subset of C by the y_i variables assigned True in that model; conversely, the corresponding correction set is indicated by the y_i variables assigned False. Cardinality constraints with bounds on the number of y_i variables assigned False restrict the search to a particular level or size of MCS, and using a linear progression of bounds, CAMUS searches for MCSes of increasing size / MSSes of decreasing size. Subsets of any MSS found are blocked with a **BlockDown** clause before continuing the search. A variant of CAMUS described in [23] uses an equivalent of the **Grow** subroutine to compute MSSes (and thus MCSes) more efficiently in some cases than the strict linear search, but its performance has not been studied in depth.

Figure 3 illustrates the execution of the first phase of CAMUS on our running example. The SAT and UNSAT regions are not known but are implicitly present in the augmented formulation of C', whose models correspond to the satisfiable subsets of C. Therefore, the UNSAT region is implicitly "marked" as explored in C'. Then, with each bound k, any model found indicates an MCS of size k, whose corresponding MSS is blocked down to mark it and its subsets as explored. The combination of the implicitly unsatisfiable region

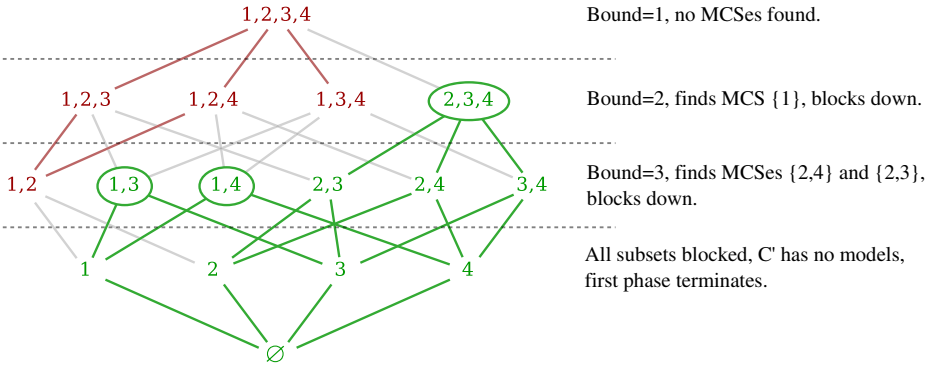


Fig. 3 Illustrating the first phase of CAMUS on the running example. Each iteration explores a lower level of the power set

with the explicitly blocked subsets of MSSes eventually covers the entire power set, making C' unsatisfiable, and the first phase terminates. The result of this enumeration is the complete set of MCSes, and a minimal hitting set enumeration algorithm (equivalent to the hypergraph transversal problem) is applied to produce MUSes.

Note that the search for MCSes in CAMUS operates on the constraints directly, but it can be implemented in any domain that allows augmenting constraints with selector variables or otherwise efficiently searching for satisfiable subsets. The second phase, computing minimal hitting sets, operates on sets of abstract elements (constraints can be represented by integers, e.g.) and thus is independent of the implementation and constraint solver used in the first phase. Overall, the approach is constraint-agnostic with only modest requirements on the constraint solver used.

Beyond the basic CAMUS algorithm, Grégoire, et al. boosted the search for MCSes using an incomplete local search oracle to identify candidate MCSes [14], which is more efficient than the complete MCS enumeration but still relies on it for completeness and correctness. Another optimization using unsatisfiable cores to boost the search for MCSes improved the algorithm's performance substantially [24] with no major drawbacks.

A significant shortcoming of CAMUS, in all of these variants, is that the first phase can be intractable. A constraint system may have an exponential number of MCSes, all of which must be enumerated in the first phase of the algorithm before any MUSes are produced. To combat this intractability, another variant in [23] introduces the concept of the *partial correction set* (PCS), a subset of some MCS. The variant is a modification to the first phase of CAMUS that computes PCSes by selectively removing constraints from the problem as it finds MCSes. This reduces the number of results found in that phase and reduces the search space as it progresses, completing more quickly overall. Several techniques for choosing constraints to remove are described, and one simple method is to truncate each MCS found to a set size k, removing an arbitrary set of constraints from each MCS to reach that size. The minimal hitting sets of the PCSes found in this way are still MUSes, and so the second phase is unaltered. Thus, using PCSes instead of MCSes sacrifices completeness for speed in a controllable fashion.

DAA Another algorithm that exploits the hitting set duality between MCSes and MUSes is Dualize and Advance (DAA) by Bailey and Stuckey [2], based on the algorithm of the same name by Gunopulos, et al., for discovering patterns in data mining [15]. DAA uses the **Grow** subroutine on known-satisfiable subsets of C (initially the empty set) to produce MSSes and their complementary MCSes, then computes minimal hitting sets (hypergraph transversals) of the MCSes found thus far. Each hitting set is tested for satisfiability: unsatisfiable subsets are MUSes and can be immediately output, while any satisfiable subset found is taken as the next starting point for **Grow** to find another MSS/MCS. The computation of hitting sets can be seen as an implementation of **GetUnexplored** in this context. The MCSes implicitly encode the known-satisfiable region of the power set at any point in time, and a minimal hitting set of the MCSes is a "low point" in the region above that. Therefore, each hitting set is either a known MUS (easily filtered out), a new unexplored MUS to be output, or a new unexplored SAT subset.

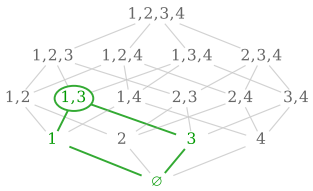
The following example illustrates the execution of DAA on our running example. The diagrams show the known-satisfiable region implicitly encoded by the MCSes at each iteration.

Example execution of DAA

$C = \{1 : (a),\ 2 : (\neg a),\ 3 : (\neg a \vee b),\ 4 : (\neg b)\}$

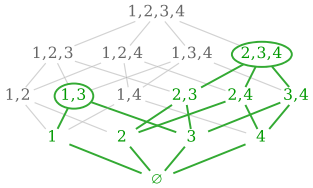
Initially, no MCSes have been found, so the first known-satisfiable subset is the empty set. This is given to **Grow** to obtain an MSS, and its complementary MCS is added to the set of MCSes from which hitting sets are computed.

- **Grow**$(\emptyset) \rightarrow \{1, 3\}$ – MCS: $\{2, 4\}$
- $\texttt{MCSes} = \{\{2, 4\}\}$
- **HittingSets**$(\texttt{MCSes}) \rightarrow \{\{2\}, \{4\}\}$



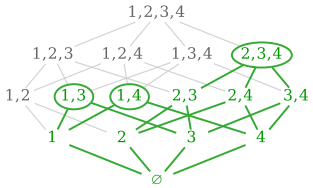
Both of these hitting sets (minimal points in the region above the known-SAT region) are satisfiable subsets, so one is taken as the next starting point for **Grow**.

- **Grow**$(\{2\}) \rightarrow \{2, 3, 4\}$ – MCS: $\{1\}$
- $\texttt{MCSes} = \{\{2, 4\}, \{1\}\}$
- **HittingSets**$(\texttt{MCSes}) \rightarrow \{\{1, 2\}, \{1, 4\}\}$



The first of these hitting sets, $\{1, 2\}$ is unsatisfiable, and it is output as an MUS. The other is given to **Grow** for the next iteration.

- **Grow**$(\{1, 4\}) \rightarrow \{1, 4\}$ – MCS: $\{2, 3\}$
- $\texttt{MCSes} = \{\{2, 4\}, \{1\}, \{2, 3\}\}$
- **HittingSets**$(\texttt{MCSes}) \rightarrow \{\{1, 2\}, \{1, 3, 4\}\}$



The first hitting set has been found already and is skipped, while the other, $\{1, 3, 4\}$ is found to be unsatisfiable and output as an MUS. With no satisfiable subsets found in the hitting sets, the algorithm has found all MUSes, and it terminates.

While originally presented for systems of Herbrand constraints, the DAA algorithm is constraint-agnostic. It relies on the **Grow** and **SAT** subroutines, which are easily adapted to any constraint type, and it otherwise operates independently of the constraints. Additionally, the authors note that the computation of minimal hitting sets (hypergraph transversals) can be accomplished with any suitable algorithm; they use an implementation of a simple algorithm by Berge [4] for testing, though they and Gunopulos, et al. note that with the incremental algorithm presented by Fredman and Khachiyan [11] the runtime is worst-case subexponential in the size of the output. In contrast with CAMUS, DAA outputs MUSes interleaved with the MCSes it finds, meaning it can produce MUSes before all MCSes have been found, avoiding the intractability of the first phase of CAMUS. However, as presented, it suffers from a different intractability in that the set of hitting sets computed in each iteration can become exponentially large, exhausting memory limits early in its execution.

PDDS An approach tightly related to DAA was later proposed in the diagnosis domain by Stern, et al., who referred to it as Primal-Dual Diagnostic Search (PDDS) [30]. If translated into the terminology of analyzing infeasible constraint systems, PDDS is given as an

algorithm for computing MCSes (subset-minimal diagnoses in their terms) from MUSes (subset-minimal conflicts), but it is noted that the algorithm can operate in either direction due to the symmetry of the duality between the two types. Briefly, the algorithm repeatedly computes a new minimal hitting set of the MUSes found thus far, keeps it as a new MCS if its complement is satisfiable, and minimizes it to another MUS otherwise. Alternatively, the symmetry of the relationship between MUS and MCSes means that PDDS can operate by computing hitting sets of MCSes, keeping unsatisfiable sets as MUSes, and growing satisfiable sets to MSSes whose complements are kept as new MCSes. In this direction, it is similar to DAA. The main differences are that PDDS is presented as taking an initial set of either MUSes or MCSes as input, though it is a minor modification for this set to be empty as it is in DAA, and PDDS does not necessarily compute all hitting sets of the MCSes at each iteration, avoiding the memory scaling issues of DAA.

Stern, et al. also present the Switching Diagnostic Engine (SDE), an enumeration algorithm which repeatedly applies one iteration of PDDS in alternating directions. In terms of MCSes and MUSes, it first computes one hitting set of the known MCSes to either produce an MUS or another satisfiable subset for **Grow**, then it computes a hitting set of the known MUSes to produce either an MCS or another unsatisfiable subset for **Shrink**, etc. The authors only present results for SDE, and it is not clear how its performance compares to running PDDS in a single direction (as in DAA). The difference is primarily in the order in which each algorithm will explore the power set, and it isn't clear that the interleaved order of SDE is superior to using PDDS in a single direction.

5 Enumerating MUSes: The MARCO Algorithm

The MARCO algorithm enumerates MUSes (and, as a necessary side-effect, MSSes) by means of the hitting set dualization discussed in Section 3 and employed in various ways by the algorithms presented in Section 4.3. We first describe a basic version of the MARCO algorithm to illustrate the underlying concepts, and we then present an optimized version geared toward enumerating MUSes.

Recall that this algorithm enumerates both the MUSes and the MSSes of a constraint set C. In the general case, this complete enumeration is intractable, as a constraint set with n constraints may have a number of MUSes and MSSes exponential in n. Sperner's Theorem provides a loose upper bound on the number of either as $\binom{n}{\lfloor n/2 \rfloor}$ (intuitively, this is the number of elements in the "widest" row of the Hasse diagram for the power set lattice), and many real-world benchmarks exhibit exponentially many MUSes and/or MSSes.

In the face of this intractability, the MARCO algorithm has the desirable property of yielding results both "quickly" and "early," compared to other enumeration approaches. Section 6 presents empirical data to support those claims, and we justify them in the discussion of the algorithm here. Notably, we show that the first MUS output by the MARCO algorithm is produced with a delay equivalent to that of the best-known single-MUS extraction algorithm (in fact, our algorithm can directly employ whatever implementation is currently state-of-the-art), and each successive result can be returned with a similar delay, potentially even faster. Following the basic version of the algorithm, we will present a variety of optimizations that improve the performance of MARCO at the task of enumerating MUSes. The

dual nature of MUSes and MSSes is reflected in the fact that these optimizations often have clear duals that instead optimize performance for enumerating MSSes, but these are not the focus of this work and are only mentioned briefly here.

5.1 Basic MARCO Algorithm

The basic version of the algorithm is presented as pseudocode in Algorithm 1 written using the subroutines presented in Section 3. Fundamentally, the MARCO algorithm operates by repeatedly:

1. Selecting an unexplored point in the power set lattice, a subset of C that we call a *seed*,
2. Checking the satisfiability of the seed,
3. Growing or shrinking it to an MSS or an MUS as appropriate, and
4. Marking a corresponding region of the lattice as explored.

Each iteration identifies one new MUS or MSS, and the process terminates when all subsets have been explored (characterized as SAT or UNSAT) and all MUSes and MSSes have been found.

Algorithm 1 The most basic version of the MARCO algorithm

input: Unsatisfiable constraint set C
output: MUSes and MSSes of C

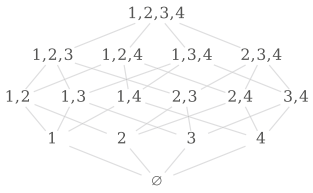
```
    // The "map" is initially an empty formula with |C| Boolean variables
 1. map ← BoolFormula(nvars = |C|)
 2. while map is satisfiable:
 3.     seed ← GetUnexplored(map)
 4.     if SAT(seed):
 5.         MSS ← Grow(seed)
 6.         output MSS
            // Block all subsets of MSS
 7.         map ← map ∧ BlockDown(MSS)
 8.     else:
 9.         MUS ← Shrink(seed)
10.         output MUS
            // Block all supersets of MUS
11.         map ← map ∧ BlockUp(MUS)
```

The following example illustrates the execution of the basic algorithm on our running example.

Example execution of MARCO (basic)

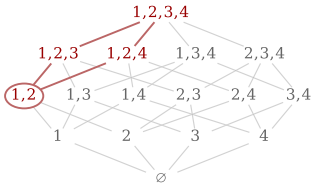
$C = \{1 : (a),\ 2 : (\neg a),\ 3 : (\neg a \vee b),\ 4 : (\neg b)\}$

The algorithm begins with map $= \top$, and nothing is known about the infeasibility of the instance. The map is empty, and so **GetUnexplored** can return any subset.



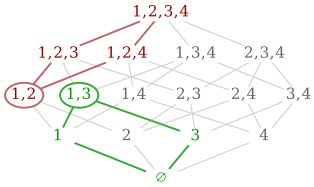
In this example, **GetUnexplored** returns an initial seed of {1, 2, 4}, the seed is found to be UNSAT, and an MUS is produced:

- **GetUnexplored**(map) → {1, 2, 4}
- **SAT**({1, 2, 4}) → False (UNSAT)
- **Shrink**({1, 2, 4}) → {1, 2}
  {1, 2} is output as an MUS.
- map ← map ∧ **BlockUp**({1, 2})
  The map is updated as shown to the right.



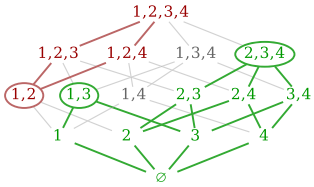
The next seed is SAT, producing an MSS:

- **GetUnexplored**(map) → {3}
- **SAT**({3}) → True
- **Grow**({3}) → {1, 3}
- map ← map ∧ **BlockDown**({1, 3})

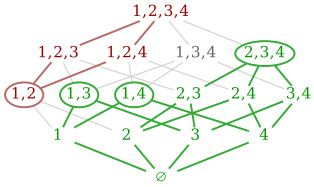


And so on...

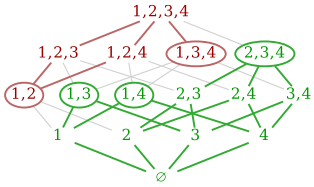
- **GetUnexplored**(map) → {2}
- **SAT**({2}) → True
- **Grow**({2}) → {2, 3, 4}
- map ← map ∧ **BlockDown**({2, 3, 4})



- **GetUnexplored**(map) → {1, 4}
- **SAT**({1, 4}) → True
- **Grow**({1, 4}) → {1, 4}
- map ← map ∧ **BlockDown**({1, 4})



- **GetUnexplored**(map) → {1, 3, 4}
- **SAT**({1, 3, 4}) → False
- **Shrink**({1, 3, 4}) → {1, 3, 4}
- map ← map ∧ **BlockUp**({1, 3, 4})



At this point, every subset in the lattice is explored, map is unsatisfiable, and the **while** loop terminates, ending the algorithm.

The correctness of the MARCO algorithm, that it produces only MUSes and MSSes, follows directly from that of the **Shrink** and **Grow** subroutines used. MARCO does not prescribe any particular implementation for these two subroutines, it merely requires that they be correct.

Theorem 2 (Correctness) Every output of Algorithm 1 is either an MUS or an MSS.

Proof The only MUS (MSS) outputs of MARCO are outputs of the **Shrink** (**Grow**) subroutine. If **Shrink** and **Grow** are correct, they can only output MUSes and MSSes, respectively. □

The completeness of the algorithm can be understood intuitively by seeing that MARCO never repeats an output, due to the blocking in `map`, and it never misses an output. Therefore, every iteration of the algorithm produces a new MUS or MSS, and all will eventually be found. More formally:

Lemma 3 (No duplicates) *If Algorithm 1 outputs a given MUS or MSS once, that MUS or MSS will never be output again.*

Proof If Algorithm 1 outputs an MUS (MSS) M, then models corresponding to M and all of its supersets (subsets) are blocked in `map` by the clause obtained from **BlockUp** (**BlockDown**). Therefore, by Lemma 1 in Section 3.3, later calls to **GetUnexplored**(`map`) will never return M nor any of its supersets (subsets). For **Shrink** (**Grow**) to output M, it must be given some input constraint set that is a superset (subset) of M. Therefore, following the first output of M, no subsequent iteration of the **while** loop in Algorithm 1 can produce M again. □

Theorem 3 (Completeness) Algorithm 1 outputs all MUSes and MSSes of an infeasible constraint system C.

Proof In each iteration of its **while** loop, Algorithm 1 outputs a "new" MUS or MSS that has not been output previously (by Lemma 3). As long as some MUS or MSS remains to be found, `map` will be satisfiable, because the model corresponding to that MUS or MSS will not yet be blocked (by the contrapositive of Lemma 2 in Section 3.3). Therefore, the **while** loop will repeat as long as there is any MUS or MSS not yet output, and because each iteration outputs a new result, every MUS and MSS will be output in some iteration of the loop. □

Theorem 4 (Termination) Algorithm 1 will terminate.

Proof Every MUS and MSS of C will be output (by Theorem 3). If every MUS and MSS has been output, then every superset of an MUS and every subset of an MSS has been blocked in `map` by either line 7 or line 11. Every subset of C is either unsatisfiable, hence a superset of an MUS, or satisfiable, hence a subset of an MSS. Therefore, if every MUS and MSS has been output, then every subset of C is explored, every model is blocked in `map`, and the **while** loop will terminate. □

5.2 Analyzing performance

Analyzing this basic version of the algorithm can inform and support the optimizations described in Section 5.3. First, we can show that MARCO can return the first MUS as quickly as the best known single-MUS extraction algorithm at any point in time, and in so doing we identify potential optimizations for the algorithm as a whole. If Algorithm 1's first seed is C itself, then the algorithm will immediately call **Shrink**(C) and produce its first MUS. Because **Shrink** can be any MUS extraction algorithm, the algorithm can use the best known implementation for any given type of constraint. As a corollary, no enumeration algorithm can return its first MUS faster than MARCO.

For each successive MUS to be output with a similar delay requires that each successive seed be UNSAT, so that each will be passed to the same **Shrink** subroutine to produce a new MUS. This cannot be guaranteed, but the algorithm can be biased to favor UNSAT seeds

early in its execution. Furthermore, successive MUSes *may* be output with an even *shorter* delay than the first. Each later call to **Shrink** will necessarily be operating on some proper subset of C, hence a smaller instance, and information in addition to the constraints themselves may be given to the subroutine to boost its performance in later calls. Optimizations discussed in Section 5.3 address both the biasing toward UNSAT seeds and the boosting of **Shrink** with extra information.

The overall runtime of Algorithm 1 depends primarily on the performance of the **Shrink** and **Grow** subroutines; the other steps operate on the simple clause set `map` and take negligible time, while **Shrink** and **Grow** operate on subsets of C itself. Empirically, in our experiments on Algorithm 1, **Grow** and **Shrink** combined take over 80 % of the runtime on average. Both of these subroutines can be black-box oracles as far as MARCO is concerned, and so their implementations can be optimized independently.

5.3 Optimized MARCO

Due to the preceding analyses, our optimizations work towards three different goals:

1. Bias the algorithm to favor UNSAT seeds over SAT seeds early in the execution
2. Eliminate some or all of the calls to **Shrink** or **Grow**
3. Boost the performance of individual calls to **Shrink**

Maximal Models The first two goals were addressed in both [21] and [28] similar fashions: generate seeds by computing *large* or *maximal models* of `map`. A maximal model of `map` is one in which no False assignment can be made True without violating a clause; therefore, a maximal model will represent a subset of C for which all supersets have been explored. It is not guaranteed to produce an unsatisfiable subset of C, but the larger a subset is, the more likely it is to be unsatisfiable. For example, if the MUS $\{1, 2\}$ has been found in our running example and the map is as shown in Fig. 4, the maximal models of `map` will correspond to constraint sets $\{1, 3, 4\}$ and $\{2, 3, 4\}$; the former is unsatisfiable, and the latter is satisfiable.

Furthermore, Previti and Marques-Silva proved (in Lemma 2 of their paper [28]) that if Algorithm 1 uses a maximal model for every seed, then any such seeds found to be satisfiable will necessarily be MSSes and **Grow** is unnecessary. This means that using maximal models to compute seeds will both bias the algorithm toward finding MUSes

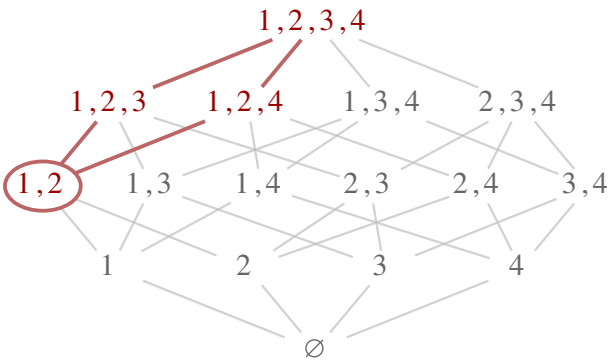


Fig. 4 An example intermediate map state with two maximal models: $\{1, 3, 4\}$ and $\{2, 3, 4\}$

early *and* remove the need to ever call **Grow**, potentially resulting in greater efficiency. This optimization is shown in Algorithm 2, in which the **GetUnexploredMax** function is a modification of **GetUnexplored** that finds and uses maximal models. As using maximal models merely adds a restriction to the **GetUnexplored** function, this optimization does not affect any of the theorems in Section 5.1, and those results hold for Algorithm 2 as well.

Algorithm 2 An optimized version of MARCO using maximal models

input: Unsatisfiable constraint set C
output: MUSes and MSSes of C

```
1.  map ← BoolFormula(nvars = |C|)
2.  while map is satisfiable:
3.      seed ← GetUnexploredMax(map)
4.      if SAT(seed):
            // seed is an MSS
5.          output seed
6.          map ← map ∧ BlockDown(seed)
7.      else:
8.          MUS ← Shrink(seed)
9.          output MUS
10.         map ← map ∧ BlockUp(MUS)
```

Liffiton and Malik's implementation did not guarantee finding maximal models, but rather it implemented a heuristic "bias" toward larger models using an unmodified MiniSat solver [21]. Their implementation assigned default polarities to every variable in `map` so that the initial assignment for any newly-assigned variable would be True (aiming to include as many constraints in the corresponding subset as possible). Previti and Marques-Silva's implementation eMUS, on the other hand, computes maximal models using SAT&PREF, a MiniSat-based solver for satisfiability problems with qualitative preferences [9]. To compute a maximal model, every variable is given a preference of being assigned True, with all preferences given equal weight.

The implementation of MARCO studied empirically in Section 6 guarantees maximal models with an approach similar to that in Liffiton and Malik's implementation: using a more recent version of MiniSat,[4] every variable in `map` is assigned a *user polarity* of True, which overrides any other heuristics for choosing a variable's polarity at a decision point during solving. This guarantees maximality because it matches the behavior of the SAT&PREF algorithm when given an empty ordering over preferences (equal weights) as we have when computing a maximal model.

Note that this optimization has a dual: if every seed is a *minimal* model, then any unsatisfiable seed found is guaranteed to be an MUS, and all calls to **Shrink** can be removed. This is attractive, due to the high cost of the **Shrink** subroutine, but it is balanced by the fact that it will bias the algorithm toward satisfiable subsets (hence, MSSes) and away from MUSes early in its execution.

*Boosting **Shrink*** If MARCO is to include calls to **Shrink**, a single-MUS extraction algorithm, then another route of optimization is to boost that algorithm by providing it more information than just the constraints in `seed`. During its execution, MARCO gathers

[4] Specifically, commit cd3a2d653f on GitHub.

information about C that may be useful. For example, certain constraints in C may be found to be *necessary* (following Kullmann, et al.'s terminology [20]), included in every MUS, and these can be given to **Shrink** as hard constraints, reducing its search space.

Specifically, any constraint found to be a singleton MCS (i.e., with cardinality one) is a necessary constraint and can be used in this way. Note that a singleton MCS will result in a single-literal **BlockDown** clause (it is the one constraint not included in its complementary MSS). More generally, then, any literal that is implied True by the `map` formula corresponds to a constraint that is included in every MUS of C. These implications can be easily extracted from the `map` solver and passed to **Shrink** as hard constraints during the course of MARCO's execution. As with using maximal models, this optimization does not affect any of the proofs in Section 5.1.

Note that this has a dual as well: any literal implied to be False by the `map` formula corresponds to a constraint that is included in every MCS (i.e., that is excluded from every MSS). However, this indicates a constraint that induces a conflict by itself — impossible in Boolean CNF and unlikely in other constraint systems. Therefore, we do not explore it further, but it is an illustration of the highly dual nature of the MARCO algorithm and underlying concepts.

5.4 Comparison to other Algorithms

In comparison to direct subset enumeration algorithms, MARCO has far fewer **SAT** checks, as large regions of the search space are eliminated by the blocking clauses added after every output. MARCO also compares favorably at an algorithmic level to the algorithms based on hitting set dualization, which have already been shown to outperform the subset enumeration approach. First, it entirely avoids the potential intractability of the first phase of CAMUS. While CAMUS is sensitive to the number of MSSes/MCSes in an instance due to its first phase, MARCO outputs MUSes "early" without having to compute all or even any given fraction of the MSSes. Likewise, MARCO avoids the memory scaling problem of DAA, which can compute an exponentially large number of hitting sets in each iteration.

However, minimal or maximal models of `map` correspond to minimal hitting sets of the MUSes or MCSes, respectively, found thus far [19], which draws a clear connection between MARCO and both DAA and PDDS. If implemented to find *minimal* models of `map`, MARCO is similar to DAA and the primary presentation of PDDS; with maximal models as in the MUS-optimized version, MARCO is like the alternative version of PDDS. The primary differences, then, arise from: 1) MARCO's unified representation of discovered MUSes and MSSes, maintaining `map` as a Boolean formula, and 2) MARCO's implicit hitting set computation via a SAT solver that can provide maximal models.

First, MARCO computes a single hitting set at a time (as in PDDS), avoiding the scaling issue faced by DAA. Secondly, previously-seen sets will be automatically avoided due to the blocking clauses in `map` for both MUSes and MSSes; DAA and PDDS need to augment the hitting set computation or implement a secondary check to detect and skip sets that have been explored previously. Third, the use of maximal models both biases MARCO toward MUSes early in its execution and provides MSSes for "free." And finally, the `map` formula enables constraint-based reasoning such as the use of implications from `map` to boost **Shrink**, above, flexibility in the implementation of **getUnexplored**, and the potential for further improvements along those lines as well. Therefore, MARCO with minimal models explores the power set in the same fashion as DAA and the primary version of PDDS,

while MUS-optimized MARCO matches the ordering of the alternate PDDS direction, but its internal constraint-based reasoning provides advantages over both.

The DAA and PDDS algorithms have the benefit that they are decoupled from the choice of hitting set (hypergraph transversal) algorithm. As noted by Gunopulos, et al., the choice of the Fredman Khachiyan algorithm for computing hitting sets results in a version of the DAA algorithm with worst case runtime that is subexponential in the size of the output [15]. MARCO, on the other hand, relies on a SAT solver for generating new seeds, as it uses additional information and solves a problem more complex than a pure hypergraph transversal problem. This gains MARCO the benefits outlined above but at the cost of a worse complexity bound, given that each new seed and thus each output arises from a search with worst-case exponential runtime.

6 Results

We experimentally compared variants of the MARCO algorithm with each other and with the best known existing MUS enumeration algorithms. Earlier work has shown that CAMUS typically outperforms DAA for complete MUS enumeration [23], and DAA outperforms the subset enumeration approaches [2]. The incremental nature of DAA makes it more suitable to the problem of interest here, namely finding *some* MUSes quickly, and so it warrants further attention in this context alongside CAMUS. To match the focus of this work on enumerating *some* MUSes even when complete enumeration is intractable, we also include the variant of CAMUS that produces PCSes for better scalability, specifically with MCSes truncated to two clauses each: "CAMUS (2PCSes)". Finally, we have included three variants of the MARCO algorithm based on a new implementation:

1. MARCO+ – the optimized version of the MARCO algorithm.[5]
2. MARCO (basic) – the basic variant of the algorithm (Algorithm 1).
3. MARCO (MSS bias) – a variant biased toward finding MSSes (using minimal models instead of maximal) – both to show the value of maximal models and as a stand-in for an optimized version of DAA without the scaling issues caused by enumerating all minimal hitting sets after each output.

Every algorithm was run on a collection of 300 unsatisfiable Boolean CNF benchmarks that were selected for the MUS track of the 2011 SAT competition (available: http://www.cril.univ-artois.fr/SAT11/). The instances were drawn from a variety of domains and applications, and they range in size from 26 to 4.4 million variables and from 70 to 16.0 million constraints. All experiments were run on Amazon Elastic Compute Cloud (EC2) "cc2.8xlarge" instances with Intel Xeon E5-2670 processors and 60.5GiB of RAM. Every execution ran with a limit of 3600 seconds (1 hour) and 3000MB of RAM. We used the following implementations of the algorithms tested:

– MARCO v1.0 (available: http://www.iwu.edu/~mliffito/marco/) – a Python implementation using MUSer2 [3] for the **Shrink** subroutine and MiniSat [10] (commit cd3a2d653f on GitHub) for **Grow** and **GetUnexplored[Max]**.

[5] We will refer to this version as MARCO+ to differentiate it from the implementation tested in [21].

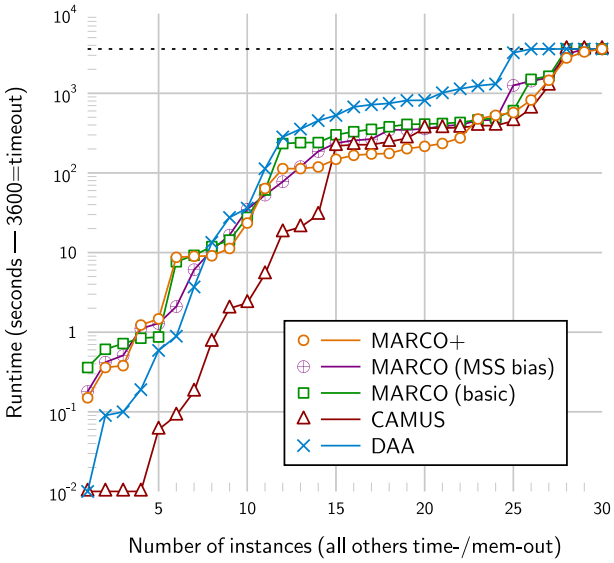


Fig. 5 Cactus plot of the total runtime to complete MSS/MUS enumeration for each algorithm

- CAMUS v1.05 (available: http://www.iwu.edu/~mliffito/camus/) – written in C++ using MiniSat v1.12.
- DAA – a custom implementation of DAA for Boolean CNF written in C++ using MiniSat v2.2.

While Python is generally slower than C++, the MARCO implementation spends the great majority of its runtime in MUSer2 and MiniSat, which are compiled C++. Therefore, the use of Python for the high-level algorithm employing those solvers has a negligible impact on the overall runtime. As noted earlier, DAA can be implemented with any hitting set algorithm; our implementation here uses the algorithm developed as part of CAMUS, which performed well compared to existing implementations of other algorithms in [23]. There are no other known implementations of DAA for Boolean CNF, and while using the Fredman Khachiyan hitting set algorithm would result in a better runtime bound, the (MSS bias) variant of MARCO stands in for an optimized version of DAA, providing a comparison of DAA's bias towards satisfiable seeds with the bias used in the optimized version of MARCO.

Complete MUS enumeration is generally an intractable problem, due to the potentially exponential number of MUSes, and no algorithm was able to complete within the timeout for more than 30 of the 300 instances. The runtimes for MARCO (both the optimized version and the basic version), CAMUS, and DAA are compared in a logarithmic cactus plot[6] in Fig. 5. CAMUS is the fastest overall algorithm in the majority of the instances, but MARCO+ is close or better in many. Note that there is not a substantial difference here between the three variants of the MARCO algorithm, though the MARCO+ variant biased toward MUSes does have a slight edge overall. For complete enumeration, CAMUS may be a more efficient approach than MARCO+, but of course the aim of this work

[6] 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."

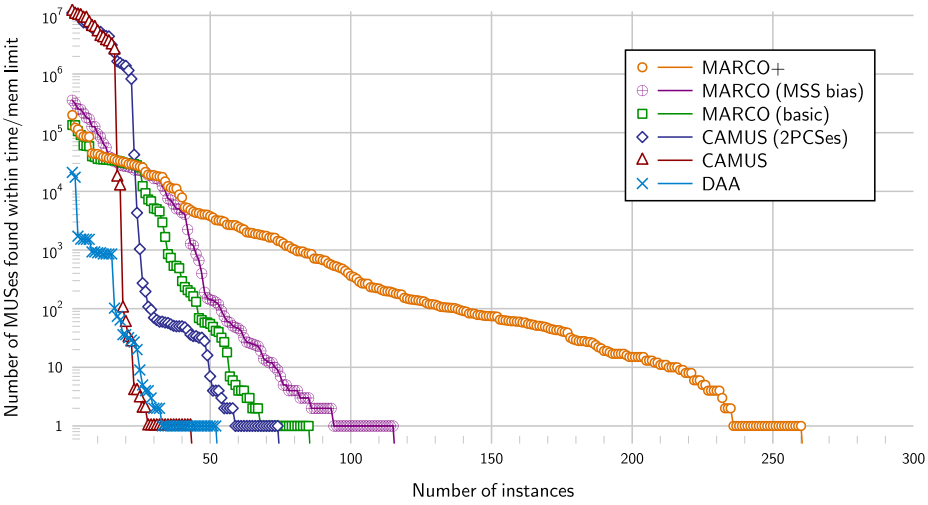


Fig. 6 Reverse cactus plot of the number of MUSes output within the time and memory limits by each algorithm

is to produce some MUSes quickly in the large number of instances for which CAMUS produces *none*.

Therefore, these algorithms are best analyzed in terms of how many results they produce and when they produce them, as opposed to simply measuring runtime to completion, which will most often be impossibly long. Figure 6 contains a reverse cactus plot[7] of the numbers of MUSes produced by each algorithm within the 3600 second time- and 3000 MB memory-limit. We can see that MARCO+ far outperforms the previous approaches. It finds two or more MUSes within the resource limits in 235 of the 300 benchmarks, while the next closest, CAMUS using 2-constraint PCSes, finds multiple MUSes in only 58 instances. Likewise, MARCO+ finds over 100 MUSes in 137 instances, compared to 28 instances for CAMUS (2PCSes). For an idea of how quickly MARCO+ outputs MUSes, Fig. 7 shows similar plots of the number of MUSes found within three shorter time limits: 600, 60, and 10 seconds. In each of these cases, MARCO+ still finds multiple MUSes in far more instances than the existing algorithms, and it finds more MUSes within the time limit for all but a few benchmarks.

In the full 3600 second experiment, there are roughly 20 instances in which CAMUS (either variant) outputs more MUSes than MARCO+ within the resource limits; in these cases, CAMUS can find the complete set of MCSes or PCSes relatively quickly, and its efficient hitting set algorithm can then enumerate MUSes much more rapidly than MARCO+'s call to **Shrink** for each. In some of these cases, CAMUS outputs millions of MUSes before MARCO+ even completes its first call to **Shrink** to output a single MUS. Thus, while MARCO+ does not outperform existing algorithms in every case, it does so in a large majority of instances.

Comparing the variants of the MARCO algorithm, we see that the optimized version, biased toward finding MUSes early, greatly outperforms both the basic version and the variant biased toward finding MSSes early, as expected. Interestingly, the MSS-biased variant

[7] In this plot, a point (x, y) can be read as, "x instances have a value of y or more."

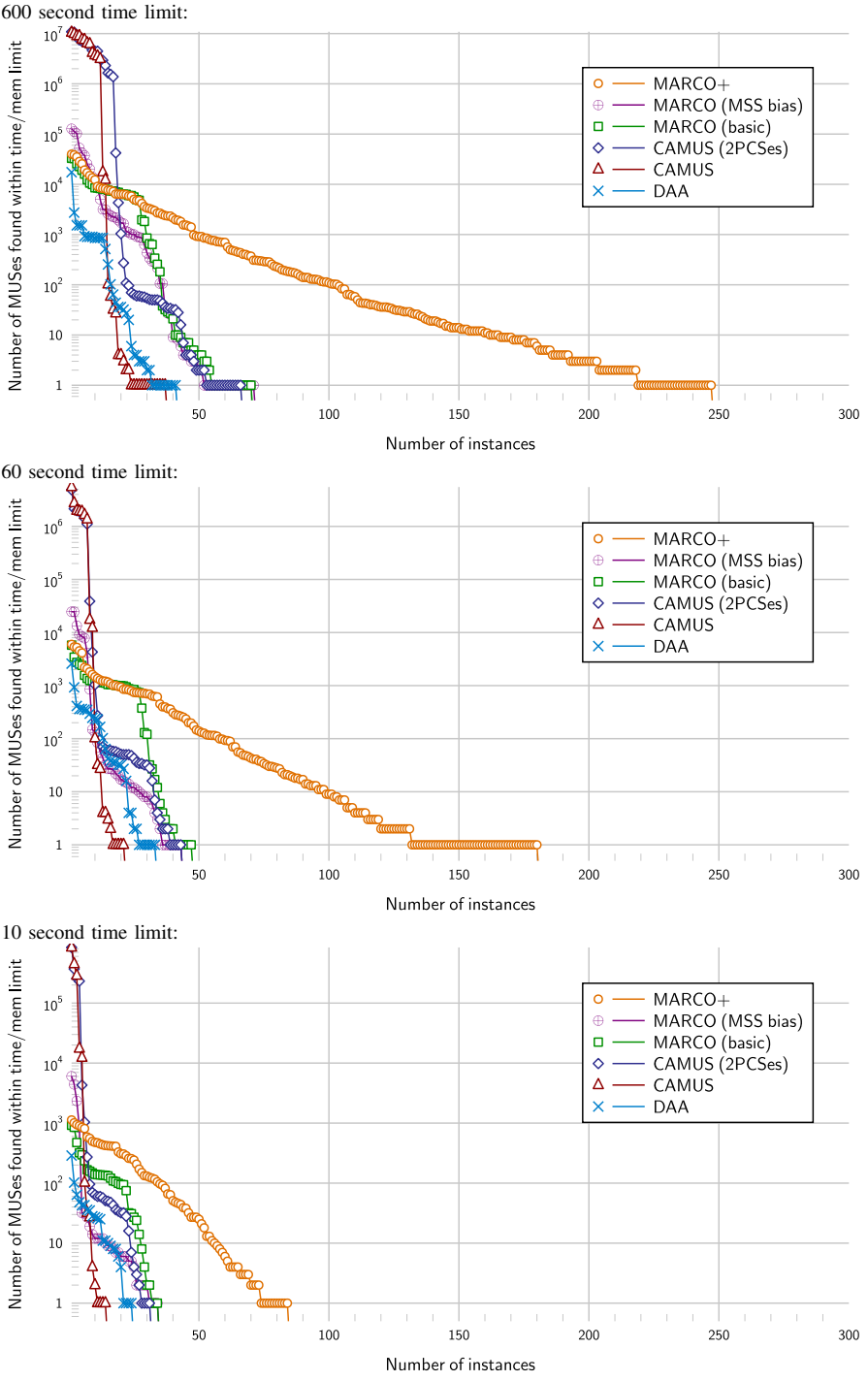


Fig. 7 Reverse cactus plots of MUSes output within three shorter time limits (600, 60, and 10 seconds, top to bottom)

outperforms the basic algorithm at enumerating MUSes. This suggests that the cost to that variant of hitting more satisfiable subsets early on is outweighed by the benefit of removing all calls to the **Shrink** subroutine. In about 30 instances, the MSS-biased variant outputs more MUSes than the MUS-biased version, indicating that in these instances, eliminating the calls to **Shrink** is more beneficial than eliminating **Grow**, even *with* the additional cost of hitting satisfiable subsets early. As noted earlier, the bulk of MARCO+'s runtime is spent in one or both of these two subroutines, and it is worth exploring their performance further.

For a more precise view of when during its execution each algorithm outputs MUSes, Figure 8 contains anytime plots for MARCO in its optimized configuration and with the MSS bias, for DAA, and for CAMUS using 2PCSes. Each line traces the cumulative percentage of MUSes output over time during an algorithm's run on each benchmark, with the time normalized such that the total runtime of an instance (usually the 3600 second timeout) is scaled to 1.0. A line directly along the diagonal would thus represent an instance for which an algorithm produced MUSes steadily throughout its execution,

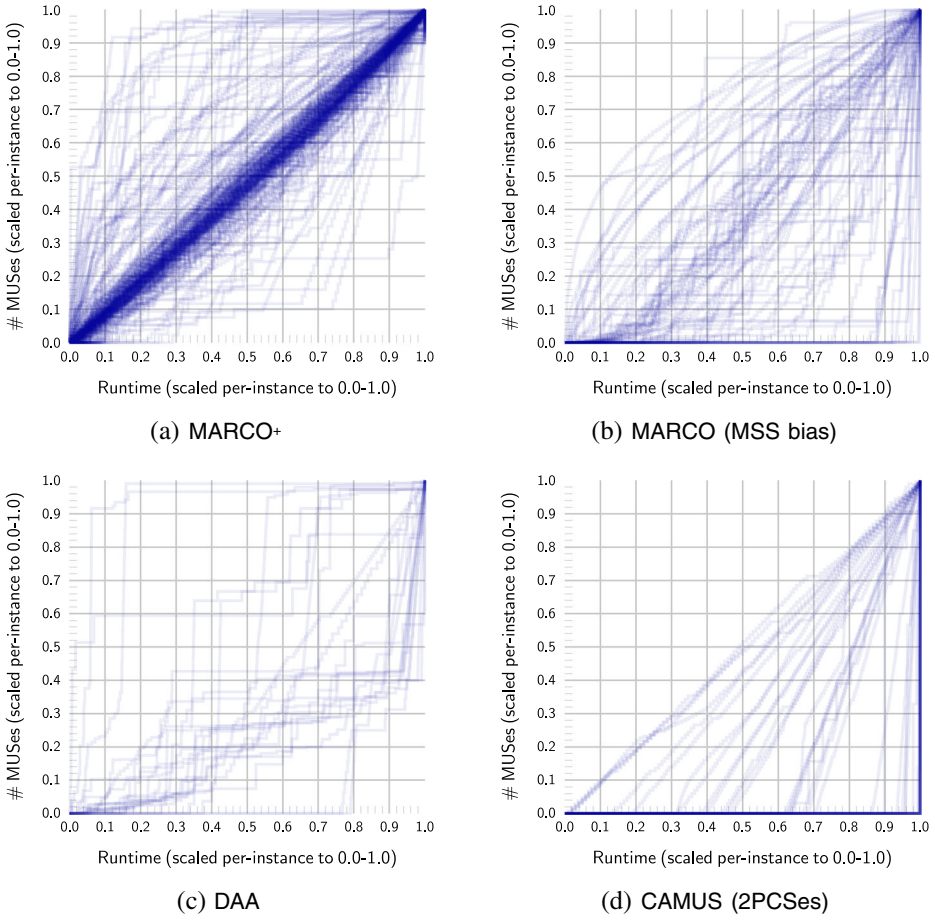


Fig. 8 Anytime plots showing the percent of MUSes found at any time during each run, normalized to compare rates and trajectories across instances

with equal delay between each. Lines curving above the diagonal indicate instances for which MUSes were output "early" in an algorithm's execution, while lines below represent instances with MUSes produced more often later in the execution. Each of these plots contains traces for any benchmark for which the given algorithm produced 10 or more MUSes.

The anytime plots illustrate several important characteristics of the algorithms. Figure 6 has already shown that the optimized version MARCO+ typically produces more MUSes within a given timeout than the MSS-biased version, and the anytime plots further indicate that the optimized version most frequently has a steady rate of MUS output, while the MSS-biased variant often has a longer delay before any MUSes are produced and generally outputs them later. DAA has fewer traces due to producing multiple MUSes in fewer instances, but in the traces we have, it exhibits an overall trend of producing MUSes later in its execution, with the curves typically under the diagonal. Finally, the anytime plot for CAMUS reflects its two-phase nature. Instances exhibit a delay, often long, while the first phase executes, followed by a fairly even rate of MUS output once the second phase begins. Overall, these plots show that MARCO+ is best for enumerating *some* MUSes quickly, given its short delay to the first output, generally steady output of MUSes from the beginning, and frequent front-loading of MUS outputs (with more curves above the diagonal than below).

7 Conclusions & future work

We have presented the MARCO / eMUS algorithm for enumerating MUSes of an infeasible constraint system, unifying and expanding upon the earlier independent work in which it was first presented [21, 28]. Through the lens of power set exploration, we have more fully described the algorithm's operation, including some new optimizations, and we drew stronger comparisons back to earlier work in the field. We demonstrated several positive properties of the MARCO algorithm:

- It is constraint agnostic, meaning it can easily be implemented for any type of constraint.
- It has good anytime performance, producing outputs quickly and early in its execution, especially compared to previous MUS enumeration algorithms.
- It can immediately benefit from any future advances in single-MUS extraction algorithms, as it can use any such algorithm as a black box oracle for one of its more expensive operations.

Experiments showed that MARCO outperforms existing algorithms for the task of *partial* MUS enumeration, especially when complete enumeration is impossible within reasonable time limits, though it is not necessarily an improvement in the state-of-the-art for *complete* enumeration.

The structure of the basic MARCO variant (Algorithm 1) provides a great deal of flexibility in its implementation. We have presented an optimized version (Algorithm 2) well-suited to enumerating MUSes, but there is potential for further improvements. For example, the use of implications from the `map` formula to boost **Shrink** is one step into sharing information between the solvers for the `map` abstraction and the original constraints C, and additional improvements may be found in that direction. The **GetUnexplored** function also allows

for a wide range of implementations, and it is worth exploring methods beyond the maximal, minimal, and random models tried here; for example, the alternation between minimal and maximal models implicitly used in Stern, et al.'s SDE [30] provides one alternative. Additionally, while a large body of work on single-MUS extraction algorithms has provided efficient implementations for **Shrink**, the problem of finding a single MSS (**Grow**) is much less studied. The **Grow** subroutine is unneeded when using maximal models as in Algorithm 2, but alternative approaches for **GetUnexplored** could benefit from improved algorithms for finding a single MSS such as the recent work in [25].

In some settings there is interest in computing preferred MUSes and MCSes [18]. Computation of lexicographic preferred MUSes and MCSes is hard for the 2nd level of the polynomial hierarchy [26], and so it is often impractical in practice. In contrast, the partial enumeration of anti-lexicographic MUSes and MCSes is computationally easier [26]. The MARCO / eMUS algorithm can compute maximal models that will represent anti-lexicographically preferred MCSes/MUSes. For this, the `map` solver needs to be able to compute an anti-lexicographically preferred MCS [26], and the MUS extractor must be able to compute an anti-lexicographically preferred MUS.

And finally, the general approach of exploring a constraint system's power set by "mapping" regions as an algorithm progresses may benefit infeasibility analysis problems beyond MUS enumeration and those described in Section 3.5. While it is implicit in many existing algorithms, directly reasoning in this way and visualizing the power set lattice may illuminate further advances in the field.

Acknowledgments We would like to thank the anonymous reviewers for their helpful comments and suggestions for improving this article. This work is partially supported by an Amazon AWS in Education Research Grant award, by SFI PI grant BEACON (09/IN.1/I2618), and by FCT grants ATTEST (CMU-PT/ELE/0009/2009), POLARIS (PTDC/EIA-CCO/123051/2010), and by national funds through FCT with reference UID/CEC/50021/2013.

References

1. Andraus, Z.S., Liffiton, M.H., & Sakallah, K.A. (2008). Reveal: A formal verification tool for Verilog designs. In *Proceedings of 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-2008)* (pp. 343–352).
2. Bailey, J., & Stuckey, P.J. (2005). 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) of LNCS,* Vol. 3350 (pp. 174–186).
3. Belov, A., & Marques-Silva, J. (2012). MUSer2: An efficient MUS extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, *8*, 123–128.
4. Berge, C. (1989). *Hypergraphs of North-Holland Mathematical Library.* Vol. 45. Amsterdam: North-Holland Publishing Co.
5. Cimatti, A., Griggio, A., Schaafsma, B.J., & Sebastiani, R. (2013). A modular approach to MaxSAT modulo theories. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT-2013)* (pp. 150–165).
6. Davies, J., & Bacchus, F. (2011). Solving MAXSAT by solving a sequence of simpler SAT instances. In *Principles and Practice of Constraint Programming (CP 2011)* (pp. 225–239): Springer.
7. de Kleer, J., & Williams, B.C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, *32*(1), 97–130.
8. García de la Banda, M.J., Stuckey, P.J., & Wazny, J. (2003). 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).
9. Di Rosa, E., & Giunchiglia, E. (2013). Combining approaches for solving satisfiability problems with qualitative preferences. *AI Communications*, *26*(4), 395–408.

10. Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-2003) of LNCS,* Vol. 2919 (pp. 502–518).
11. Fredman, M.L., & Khachiyan, L. (1996). On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, *21*(3), 618–628.
12. Gasca, R.M., Valle, C.D., López, M.T.G., & Ceballos, R. (2007). 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) of LNCS,* Vol. 4788 (pp. 160–169).
13. Gleeson, J., & Ryan, J. (1990). Identifying minimally infeasible subsytems. *ORSA Journal on Computing*, *2*(1), 61–67.
14. Grégoire, É., Mazure, B., & Piette, C. (2007). Boosting a complete technique to find MSSes and MUSes thanks to a local search oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)* Vol. 2 (pp. 2300–2305).
15. Gunopulos, D., Khardon, R., Mannila, H., Saluja, S., Toivonen, H., & Sharma, R.S. (2003). Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, *28*(2), 140–174.
16. Han, B., & Lee, S.-J. (1999). 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.
17. Hou, A. (1994). A theory of measurement in diagnosis from first principles. *Artificial Intelligence*, *65*(2), 281–328.
18. Junker, U. (2004). QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th AAAI Conference on Artificial Intelligence (AAAI 2004)* (pp. 167–172).
19. Kavvadias, D.J., Sideri, M., & Stavropoulos, E.C. (2000). Generating all maximal models of a Boolean expression. *Information Processing Letters*, *74*(3), 157–162.
20. Kullmann, O., Lynce, I., & Marques-Silva, J. (2006). Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006) of LNCS,* Vol. 4121 (pp. 22–35).
21. Liffiton, M.H., & Malik, A. (2013). Enumerating infeasibility: Finding multiple MUSes quickly. In *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2013)* (pp. 160–175).
22. Liffiton, M.H., & Sakallah, K.A. (2005). On finding all minimally unsatisfiable subformulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005) of LNCS,* Vol. 3569 (pp. 173–186).
23. Liffiton, M.H., & Sakallah, K.A. (2008). Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, *40*(1), 1–33.
24. Liffiton, M.H., & Sakallah, K.A. (2009). Generalizing core-guided Max-SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT-2009) of LNCS,* Vol. 5584 (pp. 481–494).
25. Marques-Silva, J., Heras, F., Janota, M., Previti, A., & Belov, A. (2013). On computing minimal correction subsets. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-2013)* (pp. 615–622): AAAI Press.
26. Marques-Silva, J., & Previti, A. (2014). On computing preferred MUSes and MCSes. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT-2014)* (pp. 58–74): Springer.
27. Morgado, A., Heras, F., Liffiton, M., Planes, J., & Marques-Silva, J. (2013). Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, *18*(4), 478–534.
28. Previti, A., & Marques-Silva, J. (2013). Partial MUS enumeration. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI-2013)* (pp. 818–825).
29. Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, *32*(1), 57–95.
30. Stern, R.T., Kalech, M., Feldman, A., & Provan, G.M. (2012). Exploring the duality in conflict-directed model-based diagnosis. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI-2012)*.
31. van Loon, J. (1981). Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, *8*(3), 283–288.