Illinois Wesleyan University
Digital Commons @ IWU

Honors Projects

Computer Science

Spring 2013

Analyzing and Extending an Infeasibility Analysis Algorithm

Ammar Malik
Illinois Wesleyan University, amalik@iwu.edu

Follow this and additional works at: https://digitalcommons.iwu.edu/cs_honproj

Part of the Computer Sciences Commons

Recommended Citation
Malik, Ammar, "Analyzing and Extending an Infeasibility Analysis Algorithm" (2013).
Honors Projects. 20.
https://digitalcommons.iwu.edu/cs_honproj/20

This Article is protected by copyright and/or related rights. It has been brought to you by Digital Commons @ IWU with permission from the rights-holder(s). You are free to use this material in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This material has been accepted for inclusion by faculty at Illinois Wesleyan University. For more information, please contact digitalcommons@iwu.edu.
©Copyright is owned by the author of this document.

Analyzing and Extending an Infeasibility Analysis Algorithm

Ammar H. Malik

Illinois Wesleyan University, Bloomington IL 61701, USA
amalik@iwu.edu

Abstract. Constraint satisfaction problems (CSPs) involve finding assignments to a set of variables that satisfy some mathematical constraints. Unsatisfiable constraint problems are CSPs with no solution. However, useful characteristic subsets of these problems may be extracted with algorithms such as the MARCO algorithm, which outperforms the best known algorithms in the literature. A heuristic choice in the algorithm affects how it traverses the search space to output these subsets. This work analyzes the effect of this choice and introduces three improvements to the algorithm. The first of these improvements sacrifices completeness in terms of one type of subset in order to improve the output rate of another; the second and third are variations of a local search in between iterations of the algorithm which result in improved *guidance* in the search space. The performance of these improvements is analyzed both individually and in combinations across a variety of benchmarks and they are shown to improve the output rate of MARCO.

1 Introduction

Constraint satisfaction problems (CSPs) are found in a vast array of computer science and math fields from artificial intelligence to operations research to hardware verification. CSPs may be unsatisfiable (have no solution); they are then termed *infeasible* constraint systems. Infeasibility Analysis is a growing research field that deals with analyzing known infeasible CSPs and extracting useful information from them. The two types of information we focus on here are MUSes (Minimal Unsatisfiable Sets) and MCSes (Minimum Correction Sets). Briefly, MUSes provide reasons behind the infeasibility of a CSP while MCSes provide corrections to it, the removal of even one MCS making the rest of the problem satisfiable. We define these and related terms more formally in section 2 and expand upon what we consider the 'search space' for these outputs in section 3.

MARCO is a recently published[9] algorithm that enumerates these outputs. Moreover, it has been shown to outperform the best known algorithms for such purposes, namely Dualize and Advance [2] and CAMUS [10]. We expand upon MARCO in section 4 and analyze and validate a heuristic choice made in the original paper (section 5.1). MARCO's extensibility in allowing modifications to parts of its search procedure while allowing the rest of the algorithm to function properly is one of its defining features. We exploit this feature in section 5 by presenting extensions to its search features that hope to improve the algorithm's performance. The improvements include pruning the search space (section 5.2) and variations of using local search to *guide* the algorithm towards desirable outputs (section 5.3). Section 6 details the extent of these improvements by displaying the results of an empirical analysis of the algorithm and the extensions we suggested. Finally, we conclude by construing the impact of these improvements in section 7 as well as suggesting possible avenues for future research.

2 Preliminaries

Constraint Satisfaction Problems (CSPs) are defined as a combination of mathematical *constraints* on a set of variables. The constraints add restrictions to the values these variables can be assigned. For example: for a variable $x \in \mathbb{R}$, a simple constraint could be a mathematical inequality:

$$x \leq 2$$

A conjunction of constraints make up a constraint satisfaction problem. A solution to a CSP is then an assignment to *all* the variables in the problem that satisfies *all* of its constraints.

Boolean Satisfiability Although MARCO and its extensions presented here can be applied to any generic CSP, the implementation and results shown are done on a branch of CSPs called Boolean Satisfiability Problems (termed SAT). SAT problems are defined on Boolean variables that can be assigned only 1 (true) or 0 (false) values. A SAT constraint is then defined on a subset of the Boolean variables. The constraint is satisfied if a single variable in the constraint satisfies the constraint. A constraint C can either make a truth implication (represented as $x \in C$), where $x = 1$ satisfies the constraint, or a false implication (represented as $\neg x \in C$), where $x = 0$ satisfies the constraint. For example, a constraint over two variables $x, y \in \{0, 1\}$ can be defined as such:

$$(x \vee \neg y)$$

In the above constraint, either $x = 1$ **or** $y = 0$ will satisfy the constraint regardless of the value of the other variable. Therefore, a constraint is essentially a non-exclusive 'or' between the value implication of each variable in the constraint. A collection of such constraints make up a SAT problem. An *assignment* to the problem is an n-tuple representing assignment values for the variables in the problem. A problem is considered **satisfiable** if there exists an assignment that satisfies all of its constraints. It is considered **unsatisfiable** or infeasible if no such assignment exists. Note that a SAT problem can have more than one satisfying assignment.

SAT Solvers The problem of determining the satisfiablity of a SAT instance is fairly well known. Extensive research has yielded impressive results in this field and the culmination of these efforts can be found in software termed 'SAT solvers'. The MARCO algorithm is designed to make use of a SAT solver as an oracle, as a means of testing whether a subset of the problem is satisfiable or not. We used MiniSAT [6] as the particular SAT solver for our implementations and results. While the algorithm makes no assumptions about the particular implementation details of MiniSAT, an analysis into a heuristic leads us into a specific feature of it termed Bias (section 5.1). However, the feature (or some viable alternative) is fairly common among most SAT solvers.

Infeasibility There exist real world problems that, when modeled as SAT problems, are found to be unsatisfiable. Infeasibility Analysis deals with such unsatisfiable constraint systems and seeks to extract useful subsets of the problem. A *subset* of the problem is defined to be any set $S \in \mathcal{P}(C)$ where C is the set of all constraints and $\mathcal{P}(C)$ is its power set. Any subset is then a combination of *some* of the constraints from the CSP. The subset itself may be SAT or UNSAT depending on if there exists a satisfying assignment for only the constraints in the subset. We thus define certain subsets of interest:

1. A subset $M \in \mathcal{P}(C)$ is an **MUS** (Minimal Unsatisfiable Set) iff:
   (a) M is UNSAT, and
   (b) $\forall C_i \in M : \; M \setminus \{C_i\}$ is SAT.
2. A subset $M \in \mathcal{P}(C)$ is an **MSS** (Maximal Satisfiable Set) iff:
   (a) M is SAT, and
   (b) $\forall C_i \in (C \setminus M) : \; M \cup \{C_i\}$ is UNSAT.
3. A subset $M \in \mathcal{P}(C)$ is an **MCS** (Minimum Correction Set) iff:
   (a) the set $C \setminus M$ is an MSS.

MUSes can be seen to be a "core" reason behind an instance's infeasibility. Since they are minimal subsets that are unsatisfiable in and of themselves, they naturally contain a conjunction of constraints that make it impossible for any assignments to satisfy the MUS and consequently the entire SAT problem. MSSes, on the other hand, are large cardinality satisfiable subsets that are maximal in the sense that addition of a single new constraint makes them unsatisfiable. The Maximum Satisfiability Problem (MaxSAT) is the problem of finding the largest satisfiable subset of a Boolean CSP. MaxSAT solutions represent the largest cardinality MSSes, and finding MaxSAT solutions has seen a lot of research in recent times[5,14,8]. However, an MSS may have a lower cardinality than a MaxSAT solution. Finally, an MCS is the complement of an MSS. Therefore, the removal of an MCS from the problem set yields a satisfiable subset, hence the label "correction set." Moreover, it is a minimal correction set in the sense that the removal of an element from an MCS would no longer make it a correction set.

We now provide an example of a Boolean SAT problem and its respective MUSes/MCSes in order for the reader to understand the relation between these subsets and the entire set of constraints. Consider a SAT problem defined on three variables $x_1, x_2, x_3 \in \{0, 1\}$ with the following constraints:

C_1	C_2	C_3	C_4	C_5	C_6
(x_1)	$(\neg x_1)$	$(\neg x_1 \vee x_2)$	$(\neg x_2)$	$(\neg x_1 \vee x_3)$	$(\neg x_3)$

The problem will then have the following MUSes, MCSes and MSSes (note how MCSes are complements of MSSes):

MUSes	Explicit Constraints
$\{C_1, C_2\}$	$\{(x_1), (\neg x_1)\}$
$\{C_1, C_3, C_4\}$	$\{(x_1), (\neg x_1 \vee x_2), (\neg x_2)\}$
$\{C_1, C_5, C_6\}$	$\{(x_1), (\neg x_1 \vee x_3), (\neg x_3)\}$

MCSes	MSSes
$\{C_1\}$	$\{C_2, C_3, C_4, C_5, C_6\}$
$\{C_2, C_3, C_5\}$	$\{C_1, C_4, C_6\}$
$\{C_2, C_3, C_6\}$	$\{C_1, C_4, C_5\}$
$\{C_2, C_4, C_5\}$	$\{C_1, C_3, C_6\}$
$\{C_2, C_4, C_6\}$	$\{C_1, C_3, C_5\}$

Note how (x_1) and $(\neg x_1)$ each require an opposing value of x_1 to be satisfied. Therefore, $\{C_1, C_2\}$ is an MUS since it is unsatisfiable but C_1 and C_2 are separately satisfiable. Moreover, this combination of constraints explains one of the 'reasons' for the infeasibility of the instance. This is made more evident by the fact that $\{C_1\}$ is an MCS, which implies that its exclusion renders the rest of the CSP satisfiable. With C_1 excluded, finding a satisfying assignment to the remaining problem is straightforward. The value $x_1 \leftarrow 0$ satisfies constraints C_2, C_3 and C_5. Similarly, the values $x_2 \leftarrow 0$ and $x_3 \leftarrow 0$ satisfy constraints C_4 and C_6 respectively. Therefore, we have found an MUS *explaining* the infeasibility and a constraint C_1, the removal of which *corrects* the remaining CSP or makes it satisfiable. Similar arguments could be made for the remaining MUSes and MCSes.

3 Search Space

Consider a Constraint Satisfaction Problem with C constraints. The set of all possible subsets of the problem is the power set of constraints $\mathcal{P}(C)$. We denote this set the *search space* for the problem. We can define a partial order (based on inclusion) on elements of the search space which allows us to represent it as a lattice figure known as a Hasse Diagram. Figure 1 shows a Hasse diagram of three constraints with each possible subset a 'node' in the diagram. If a subset A is linked to a subset B in the diagram, where B lies above A in the structure, then $A \subset B$. In addition, there exists an element $x \in B$ such that $A = B \setminus \{x\}$. Thus, moving 'up' in the diagram represents the addition of a single element to the subset, and moving 'down' the removal of one.

A subset is either satisfiable (SAT) or unsatisfiable (UNSAT) depending on the constraints within it. This allows us to group subsets into satisfiable and unsatisfiable subsets. Moreover, the search space can then be partitioned into two

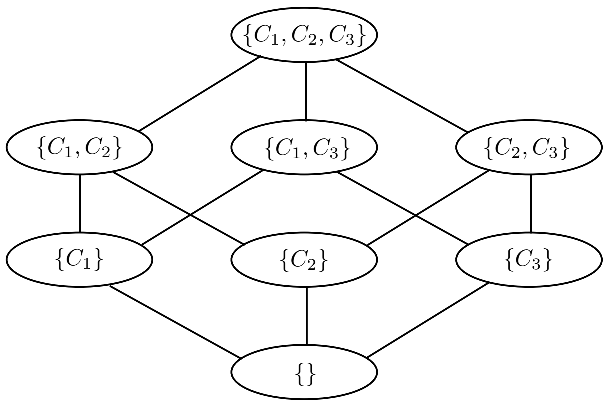


Fig. 1. Hasse diagram of 3 constraints

regions, a satisfiable region containing SAT subsets and an unsatisfiable region containing UNSAT subsets. Since the input problems in infeasibility analysis are unsatisfiable, we know that they necessarily contain at least one satisfiable subset (the subset containing the empty set i.e no constraints) and at least one unsatisfiable subset (the entire input problem). Moreover, note the following properties about subsets:

Property 1. If A is unsatisfiable and $A \subseteq B$, then B is unsatisfiable.
Property 2. If A is satisfiable and $B \subseteq A$, then B is satisfiable.

The removal of a constraint(s) from a satisfiable set cannot make it UNSAT as any satisfying assignment to it will also satisfy all of its subsets. Similarly, if there doesn't exist a satisfying assignment to a set, the addition of more constraints means the lack of a satisfying assignment still holds true. With these properties, we can thus separate the diagram shown in figure 1 into two graphically distinguishable regions; a satisfiable and an unsatisfiable region. For example, consider a problem defined on variables $x_1, x_2 \in \{0, 1\}$ with the following three constraints:

$$\begin{array}{ccc} C_1 & C_2 & C_3 \\ (x_1) & (\neg x_1) & (x_2) \end{array}$$

Then, we can split its respective Hasse diagram into SAT and UNSAT regions as shown in figure 2. Note that while the diagram in figure 1 depends only on the number of constraints as it does not make any assumptions about their satisfiability, figure 2 requires the actual constraints in order to color the regions into satisfiable (yellow) and unsatisfiable parts (orange). Having split the search space in this way, it is easier to 'see' how MUSes and MSSes satisfy their respective minimality in UNSAT regions and maximality in SAT regions. In short, any local high point in the SAT region for which its supersets exist entirely in the UNSAT region is an MSS (its complement being the MCS). In figure 2, subsets

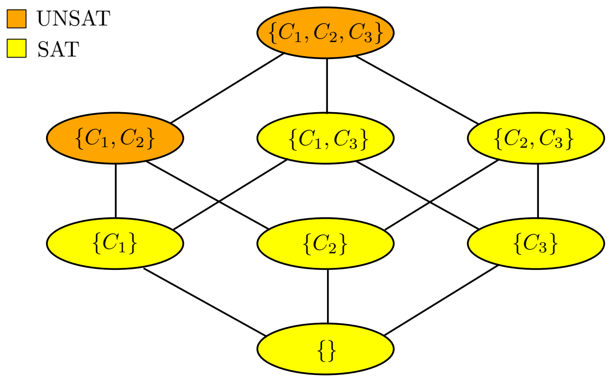


Fig. 2. Hasse diagram with marked SAT/UNSAT regions

$\{C_1, C_3\}$ and $\{C_2, C_3\}$ are MSSes. Similarly, any local low point in the UNSAT region for which its subsets exist only in the SAT region is an MUS. In figure 2, the subset $\{C_1, C_2\}$ of the problem set is an MUS.

4 The MARCO Algorithm

MARCO is an acronym for "**Ma**pping **R**egions of **Co**nstraint sets" [9]. It uses a novel approach to mapping the regions of the search space in order to enumerate MUSes and MCSes. In order to better understand the algorithm, we approach it in parts: extracting a characteristic subset (section 4.1), mapping the search space (section 4.2) and finally putting it all together (section 4.3).

4.1 Extracting a characteristic subset

Given a starting point in the search space, denoted a *seed*, there are two main methods of obtaining a characteristic subset. A characteristic subset is an MUS or an MSS (from which we can obtain an MCS easily). If the seed lies in the SAT region, then it is possible to add elements to it such that it becomes an MSS. This is termed *growing* the seed to an MSS. Likewise, if the seed lies in the UNSAT region, it is possible to remove elements from it such that it becomes an MUS. This is termed *shrinking* the seed to an MUS. A basic implementation of growing and shrinking can be seen in the **grow**$(seed, C)$ and **shrink**$(seed, C)$ functions in figures 3 and 4 respectively.

The **grow** function takes a satisfiable *seed* subset and the entire constraint set C as input parameters. Note that *seed* represents a point in the SAT region of the search space. As a result, there necessarily exists an MSS that is a superset of *seed*. M is used as a local variable that is assigned the value of *seed*. Iterating over each constraint c **not in** M, the **grow** function repeatedly checks if the addition of the constraint c to M alters the satisfiability of M. If the resulting subset $M \cup \{c\}$ remains satisfiable, the constraint c is added to M. This represents

```
grow($seed, C$)
input: A satisfiable subset $seed$
input: The set of all constraints $C$
output: An MSS $M$

1. $M \leftarrow seed$
2. foreach $c \in (C \setminus M)$:
3.     if $M \cup \{c\}$ is satisfiable:
4.         $M \leftarrow M \cup \{c\}$
5. return $M$
```

Fig. 3. **grow** returns an MSS

```
shrink($seed, C$)
input: An unsatisfiable subset $seed$
input: The set of all constraints $C$
output: An MUS $M$

1. $M \leftarrow seed$
2. foreach $c \in M$:
3.     if $M \setminus \{c\}$ is unsatisfiable
4.         $M \leftarrow M \setminus \{c\}$
5. return $M$
```

Fig. 4. **shrink** returns an MUS

moving one node 'up' in a Hasse diagram (hence the term *grow*). Otherwise, if $M \cup \{c\}$ is unsatisfiable, it is discarded, and M remains unchanged for that iteration. This continues until all constraints not in *seed* have been checked in this manner. For the resultant set M, the addition of any further constraints would make it UNSAT, it is therefore an MSS by definition.

On the other hand, the **shrink** function takes an unsatisfiable *seed* subset in addition to C as input parameters. Here, *seed* represents a point in the UNSAT region of the search space. Again, M is used as a local variable that takes the value of *seed*. Iterating of each constraint c **in** M in this case, the **shrink** function repeatedly checks if the removal of the constraint c from M results in an unsatisfiable subset. If the resulting subset $M \setminus \{c\}$ remains unsatisfiable, the constraint c is removed from M representing a move 'down' in a Hasse diagram. Otherwise, M remains unchanged. After all constraints $c \in M$ have been checked in this fashion, the algorithm returns the modified (unless *seed* was an MUS to begin with) subset M. Since any further removal of constraints from M would make it satisfiable, M is an MUS by definition.

However, it is important to note that MARCO implements the grow and shrink phases as separate modules. Different algorithms for these functions can then be used in place of **grow** and **shrink** as long as they match input and output parameters. Moreover, this allows MARCO to use any state-of-the-art MUS extraction algorithm as **shrink**, a problem with active research across a variety of fields [4,7,12].

4.2 Mapping the Search Space

The MARCO algorithm seeks to explore the search space, and output all the MUSes and MSSes of the problem it represents. A brute force approach would be to check each subset in the search space in order. However, this approach is grossly inefficient; the elements in the search space are exponential in the number of constraints, 2^n for n constraints to be precise. One potential method of exploring the search space is repeatedly picking random seeds, and successively growing or shrinking to an MSS or an MUS respectively. The drawback to this approach, however, is exploring paths that lead to a previously found outputs.

Moreover, since multiple seeds may lead to the same output, the probability of finding a seed that leads to a new output decreases over time. The MARCO algorithm avoids this problem by keeping track of the previously *explored* and the remaining *unexplored* parts of the search space.

MARCO makes use of a separate constraint system called `map` to keep track of these regions of the search space. Specifically, `map` maintains a Boolean function over the search space: $f : \mathcal{P}(C) \rightarrow \{0, 1\}$. The Boolean variables in `map` have a one-to-one correspondence with the constraints of the original problem. We show this by continuing the simple example from section 3:

Variables of CSP	$\{x_1, x_2\}$
Constraints of CSP	$\{C_1 \leftarrow (x_1), C_2 \leftarrow (\neg x_1), C_3 \leftarrow (x_2)\}$
Variables of `map`	$\{X_1 \leftrightarrow C_1, X_2 \leftrightarrow C_2, X_3 \leftrightarrow C_3\}$

This allows assignments in `map` to represent subsets of the original problem. We consider a true value assignment of a variable of `map` to correspond to an inclusion of the matching constraint in the subset. Similarly, a false value in a particular assignment to a variable of `map` indicates the exclusion of the corresponding constraint from the subset. Since an assignment means every variable of `map` is assigned a Boolean value, each constraint of the CSP is included or excluded from the corresponding subset based on that value. Therefore, every assignment identifies an exact point in the search space. Similarly, each point in $\mathcal{P}(C)$ is mapped to a corresponding assignment of `map`.

Constraints in `map` restrict certain assignments from satisfying this Boolean function. Thus, we further specify the subsets corresponding to assignments that do not satisfy `map` as having been *explored*. Likewise, subsets corresponding to assignments that do satisfy `map` are considered *unexplored*. For example, consider a subset $S \in \mathcal{P}(C)$ and the Boolean function f that `map` maintains. If S satisfies `map`:

$$f(S) = 1$$

then S belongs to the unexplored region of the search space. Likewise, if S does not satisfy `map`:

$$f(S) = 0$$

then S has already been explored. In addition, when we refer to a subset *satisfying* `map`, we will use this to mean that the subset's corresponding assignment (of `map`'s variables) satisfies `map`. Moreover, we refer to the term *blocked* to indicate one or more subsets having been explored, with *blocking* being the addition of the constraint to `map` that actualizes this.

Consider again the example CSP given above. With no constraints in `map`, the entire search space is unexplored (figure 1) as any assignment satisfies `map`. Now, suppose we add the following constraint to `map`:

$$\texttt{map} \leftarrow (X_2)$$

Then, considering the corresponding constraint (C_2), we can split the search space into explored and unexplored regions. This is shown in figure 5. Note that since the constraint implies the inclusion of C_2, any subset containing C_2 is considered unexplored and therefore, its corresponding assignment satisfies `map`.

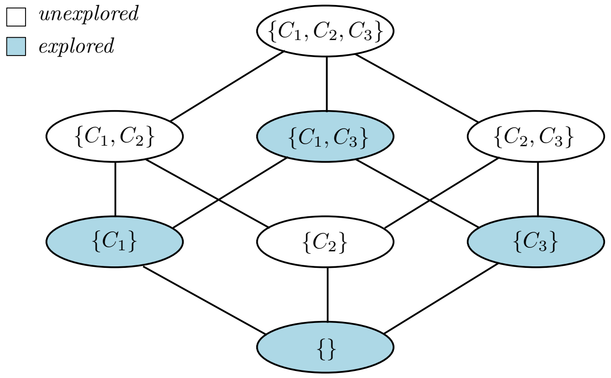


Fig. 5. Hasse diagram with mapped explored/unexplored regions

Notice that `map` will be empty at the start of the algorithm, and becomes successively more constrained as more and more of the search space is explored. To that end, one possible way to keep track of explored regions is to add a restriction to `map` for each subset explored by MARCO. This approach is still inefficient, however, as it requires us to visit each node in the search space in order to fully explore it. Since the goal of the algorithm is the extraction of characteristic subsets, we can block additional subsets that we know not to be a characteristic subsets *without* having explored them. This comes as a direct consequence of the properties defined in section 3. By Property 1, if $M \subseteq C$ is an MUS (and thus unsatisfiable), then all supersets of M are unsatisfiable as well. Since supersets of M are unsatisfiable, they cannot contain an MSS. Moreover, since an MUS cannot be a proper superset of another MUS, it follows that supersets of M do not contain any MUS as well. Therefore, we know supersets of M do not contain any characteristic subsets. Then the following addition to `map` blocks all supersets of the MUS:

$$\texttt{map} \leftarrow \texttt{map} \wedge \left\{ \bigvee_{C_i \in M} \{\neg C_i\} \right\}$$

The constraint implies the mandatory exclusion of at least one constraint from the MUS M from a satisfying assignment's corresponding subset. Since all supersets of M contain the entirety of its constraints, none of them can satisfy this constraint and are thus blocked. Similarly, all subsets of an MSS cannot contain characteristic subsets as they are satisfiable (no MUSes), the MSS itself, or a proper subset of the MSS (no MSSes). Therefore, the subsets of the MSS may be blocked together. Let $M \subseteq C$ be an MSS, then the following addition to `map` blocks all subsets of the MSS:

$$\texttt{map} \leftarrow \texttt{map} \wedge \left\{ \bigvee_{C_i \notin M} \{C_i\} \right\}$$

The above constraint implies that at least one constraint not in MSS M must be included in any satisfying assignment's corresponding subset. Since any subset of M contains only constraints that are contained within M, the restriction implying inclusion of a constraint not in M forces all subsets of M to not satisfy `map`. They are then considered blocked. For example, consider the MSS $\{C_1, C_3\}$ from figure 2. The above rule implies a conjunction of all constraints *not* in the MSS be added to `map`. The addition of $\{C_2\}$ to `map` yields the explored/unexplored regions shown in figure 5 which shows the MSS and all its subsets having been blocked.

4.3 The Algorithm

```
MARCO
input: Unsatisfiable constraint set $C$
output: MUSes and MCSes of $C$
 1. map ← BoolFormula(nvars = $|C|$)
 2. while map is satisfiable:
 3.     $S$ ← map.getModel()
 4.     if $S$ is satisfiable:
 5.         $M$ ← grow($S, C$)                 ◁ S has been grown to an MSS M
 6.         output $C \setminus M$                    ◁ $C \setminus M$ is then an MCS
 7.         map ← map ∧ $\{(C \setminus M)\}$          ◁ Blocks down from an MSS
 8.     else:
 9.         $M$ ← shrink($S, C$)               ◁ S has been shrunk to an MUS M
10.         output $M$
            // Given $M = \{M_1 \vee M_2 \vee ... \vee M_k\}$:
11.         map ← map ∧ $\{\neg M_1 \vee \neg M_2 \vee ... \vee \neg M_k\}$   ◁ Blocks up from MUS
```

Fig. 6. The MARCO algorithm for enumerating MCSes & MUSes of a constraint set.

The complete pseudocode for MARCO is shown in Figure 6. The algorithm starts with the creation of the `map` constraint set (line 1). The `map` instance starts out with variables corresponding to constraints of the original problem set C and no constraints of its own. The algorithm then repeatedly loops until `map` is no longer satisfiable. Within the loop, since `map` is satisfiable, a *model* is generated from the solver instance (line 3). The model is the satisfying assignment that the solver used to obtain the fact that the problem was satisfiable. This model (like any other assignment) has a corresponding subset that is then tested against the original constraint set which returns whether it is satisfiable or unsatisfiable. If it is satisfiable, then the model's corresponding subset belongs to the SAT region of the search space. As a result, it is grown to an MSS (line 5). Since the output specification includes MCSes, the algorithm converts this MSS to an MCS by taking a complement against the entire constraint set (line 6). All subsets of the MSS are subsequently blocked (line 7). On the other hand, if the model's corresponding subset ends up being unsatisfiable, then it belongs to the UNSAT region of the search space. It is thus shrink-ed to an MUS (line 9) which is output. In addition, all supersets of the MUS are blocked (line 11). All possible conditional flows within the loop lead to `map` being modified by the addition of more constraints. As the loop returns for the next iteration, `map` is checked again for satisfiability. The program exits when `map` no longer remains satisfiable. At this point, all elements of the search space have been explored, thus all MUSes and MCSes have been output by MARCO.

5 Extending and Improving MARCO

With an idea of how the algorithm works, we now present variants and extensions that can improve the performance of MARCO. Most of these improvements find one of MUSes or MCSes faster at the expense of the other. In addition, the improvements also have dual implementations in the sense that we can swap which of MUSes or MCSes we want to find faster at the expense of the other. These improvements can be applied to MARCO individually or all together, albeit some may work to the detriment of others. We explore the potential combinations in the experiments in Section 6.

5.1 Bias

Before moving on to extensions, a closer look at SAT solvers provides greater insight into the search mechanics of MARCO. When initializing the solver (MiniSAT [6] in our implementation), each variable is added separately with the option to introduce a 'bias'. The bias acts as a default value for that variable. As a result, in the inner decision making processes of the SAT solver, variables tend towards their bias values although not restrictively so. Therefore, in terms of the solver, the bias is not an assumption about the value of the variable, and rather more of a 'suggestion' that the solver will use if no other constraints restrict it from doing so. Since we are dealing with Boolean variables, the bias is restricted to Boolean values as well.

We focus on bias in the `map` instance of the solver. This is because bias here affects the search mechanism and can thus act as a heuristic choice that directs how the search space is explored based on biased values of constraints of C (variables of `map`). While each variable may have an independent bias, we focus on all variables having the same bias: true or false. In terms of `map`, a true bias implies the inclusion of the constraint while a false bias implies exclusion when `map` seeks to find a satisfying solution. We also use the terms high and low bias to refer to true and false biases respectively as it links to the visual imagery of a distributive lattice structure of the search space (figure 2). Note in the figure that high cardinality subsets, with more 'included' constraints, appear near the top of the figure. A high bias `map` is more likely to find unexplored subsets higher up in the diagram. Similarly, low cardinality subsets, with more 'excluded' constraints, appear near the bottom of the figure. A low bias `map` is more likely to find unexplored subsets lower down in the diagram.

To illustrate in detail, consider the first iteration of the algorithm. `map` is unconstrained and thus any bias results in an assignment matching the bias values as any assignment is a satisfying assignment. A high bias will then tend towards the inclusion of all constraints, and an assignment to `map` would be produced matching this bias. The resulting subset would be the entire problem set C. Similarly, a low bias will tend towards the exclusion of constraints, and the resulting subset would be the empty set $\varnothing$. Moreover, the effect of bias pervades the first iteration as subsets found in later iterations will be affected as well. While it is not possible to predict the exact subset found, we can make some general claims about how the search will be directed. Since we are dealing with necessarily UNSAT instances, note that large cardinality subsets are more inclined to be UNSAT due to their higher likelihood of containing an MUS, while smaller cardinality subsets are likely under constrained and thus SAT. Therefore, having a bias of elements towards inclusion or exclusion will make it more likely for subsets found in each iteration to be UNSAT or SAT respectively.

MARCO was introduced in the previously published work with a true bias so as to make it more inclined towards an output favoring more MUSes. In Section 6, we validate this choice with experiments that test MARCO and its variants with both high and low biases. However, the bias mechanism is not limited to merely all low- or all high-biased variables. The ability to give different variables different biases, and to do so dynamically, could be exploited in future work to further enhance the heuristic and potentially to adapt MARCO to specific applications.

5.2 MUSOnly

An improvement over the MARCO algorithm that we will call **MUSOnly** involves blocking *down* upon finding an MUS, similarly to how it already blocks down upon finding an MCS (see figure 5). This extra step excludes all subsets of an MUS from the search space. Since all subsets of an MUS are, by definition, satisfiable, blocking these subsets does not block any other MUSes. However, an MSS *may* be a subset of an MUS, and therefore be blocked without having been explored. Therefore, while the further reduction of the search space can speed up the search for unique MUSes, we sacrifice completeness in finding MSSes/MCSes as a consequence. Since applications tend to require only one of MUSes or MCSes (see dual MCSOnly), the lack of completeness in finding the other does not impinge upon the usefulness of this extension.

In terms of implementation, a small modification to MARCO can yield this improvement. The pseudocode in figure 6 can be modified to reflect this: MARCO blocks down(subsets) from an MSS on line 7, and up(supersets) from an MUS on line 11. The addition of a line in the same scope after line 11 that is exactly the same as line 7 will result in blocking subsets of an MUS, completing the variant.

MCS exclusion We use a simple example to illustrate the lack of completeness in finding MSSes (and subsequently MCSes). This approach could exclude an MCS/MSS from being found when an MSS of the problem is a subset of some MUS. Consider a Boolean SAT instance with the following constraints:

$$\overset{C_1}{(x_1)} \wedge \overset{C_2}{(x_2)} \wedge \overset{C_3}{(\neg x_1)} \wedge \overset{C_4}{(\neg x_2)} \wedge \overset{C_5}{(\neg x_1 \vee \neg x_2)}$$

Then, we have the following MUSes:

MUSes
$\{C_1, C_3\}$
$\{C_2, C_4\}$
$\{C_1, C_2, C_5\}$

and the following MSSes:

MSSes
$\{C_1, C_2\}$
$\{C_1, C_4, C_5\}$
$\{C_2, C_3, C_5\}$
$\{C_3, C_4, C_5\}$

Note that $\{C_1, C_2\} \subseteq \{C_1, C_2, C_5\}$. Therefore, if the MUSOnly version of MARCO were to find the MUS $\{C_1, C_2, C_5\}$ before finding the MSS $\{C_1, C_2\}$, it would block all subsets of $\{C_1, C_2, C_5\}$ and never find that MSS.

MCSOnly is a dual version of the MUSOnly extension. Geared towards a primary purpose of obtaining MCSes, we block *up* whenever we find an MSS similarly to how it already blocks up upon finding an MUS. This step will then exclude all supersets of the MSS from the search space. Again, the focal point of this extension is the fact that an MSS cannot be a superset of another MSS. However, it is possible that an MUS is a superset of the MSS, and therefore we sacrifice completeness in obtaining MUSes as a result of this extension. We will see the impact of this extension in enumerating MCSes in the results section.

5.3 MCSGuided and MCSGuided++

The nature of the MARCO algorithm ensures that we cannot guarantee that it only produce MUSes. Therefore, when it happens to produce an MCS and we are primarily concerned with finding MUSes, currently, it blocks subsets of the corresponding MSS and continues in the next iteration from a different point in the search space. However, the MSS provides potentially useful information that can be exploited; for example, all supersets of the MSS are necessarily UNSAT. We make use of this fact in improvements that we dub **MCSGuided** and **MCSGuided++**.

Both MCSGuided and MCSGuided++ influence the next iteration of the algorithm by making use of the MCS found to provide potential UNSAT seeds for the newer iterations. Since we know by definition that addition of any element to an MSS makes it UNSAT, an MCS(MSS) being found means we can add elements to it and use the modified subset as a seed for the next iteration. Note that this is simply using a superset (of which there could be many) as a seed for the next iteration. However, we must also ensure that these supersets of the MSS have not previously been blocked (explored) and relevant checks against `map` ensure that. Notice that, provided at least one superset of the MSS has not yet been explored, this extension ensures an MUS output in the very next iteration after an MSS/MCS has been found.

MARCO (additions marked with ✪, unchanged lines have original line numbers)

```
✪  seeds ← Queue()
1. map ← BoolFormula(nvars = $|C|$)
2. while map is satisfiable:
✪      if seeds is empty:
3.         $S$ ← map.getModel()
✪      else:
✪          $S$ ← seeds.pop()
4.     if $S$ is satisfiable:
5.         $M$ ← grow($S, C$)
✪          seeds.push( mcsguided($M$, $C$, map) )
   ⋮
```

Fig. 7. Modifications to MARCO for the MCSGuided/MCSGuided++ extensions

Figure 7 shows the relevant portions of MARCO with additions marked by ✪ that add the MCSGuided feature. The algorithm maintains a queue to store `seeds` being generated by the guidance from MCSes. More specifically, after the algorithm finds itself in the growing phase and outputs an MSS, the very next step involves calling the **mcsguided** (figure 8) function on line 5. The **mcsguided** function returns a single unsatisfiable seed, which is then added to the `seeds` queue. In the subsequent iteration, the `seeds` queue will override asking the solver for a seed, and thus the seed is later found to be unsatisfiable. In addition, when we know we are using a seed from the `seeds` queue, we can skip checking for its satisfiability and skip straight to the shrink phase. (This change is not shown in the pseudocode, but it is included in our implementation.)

MCSGuided++ extends this concept by using *all* supersets of the MSS that are not blocked as seeds. The modifications to the pseudocode would be nearly identical to figure 7, the only difference being a call to **mcsguided++** in place of calling **mcsguided** on line 5. The **mcsguided++** function may return multiple seeds, and thus it affects more than just the next iteration.

```
mcsguided($M$, $C$, map)
input: unsatisfiable constraint set $C$
input: Boolean formula map
input: an MSS $M$ of $C$
output: one new seed subset

1. for $c \in C \setminus M$:
2.     if $M \cup \{c\}$ is SAT in map:
3.         return $M \cup \{c\}$
```

```
mcsguided++($M$, $C$, map)
input: unsatisfiable constraint set $C$
input: Boolean formula map
input: an MSS $M$ of $C$
output: a set of new seed subsets

1. seeds $\leftarrow \{\}$
2. for $c \in C \setminus M$:
3.     if $M \cup \{c\}$ is SAT in map:
4.         seeds $\leftarrow$ seeds $\cup \{M \cup \{c\}\}$
5. return seeds
```

Fig. 8. Helper functions for MCSGuided and MCSGuided++

Pseudocode for the **mcsguided** and **mcsguided++** helper functions is shown in Figure 8. They are similar in terms of implementation. Both continually check if the result of adding new elements to the MSS is constrained (blocked) by map. The difference lies in where they return. While **mcsguided** returns at the first UNSAT seed found that is yet to be explored, **mcsguided++** seeks all unexplored supersets of the MSS, and adds them to the seeds queue before returning.

MUSGuided and MUSGuided++

A dual approach to this extension could generate new known-SAT seeds from any MUS found. Like with the MSS, we know that any subset of an MUS is necessarily SAT and therefore can act as a seed to be eventually grown towards an MSS. This would also require a check to ensure the subset of the MUS had not previously been explored in the search space. We show in Section 6 how MCSGuided helps improve MUS output rate, and how MUSGuided helps improve the MCS output rate similarly.

We extend MUSGuided similarly to how MCSGuided is extended to MCSGuided++ by enabling the use of a queue and placing all unexplored 'subsets' of the MUS in the queue as potential seeds. MUSGuided++ is then the dual of the MCSGuided++ variant. It is important to note that we use the same queue for both variants to enable the use of local search to exhaustively explore all close regions in both SAT and UNSAT portions of the search space. This allows us to create new variants in various combinations of mcsguided, musguided, mcsguided++, and musguided++. Again, we see how each of these combinations affect output enumeration rates in the results section.

6 Empirical Analysis

The experiments were conducted on a collection of over 300 Boolean SAT instances used in the MUS track of the 2011 SAT Competition. [1]. SAT competitions contain a wide variety of problem instances modeling real world problems to test the latest improvements in the field. The instances, which are infeasible, were collected from existing problems to test single-MUS extraction algorithms. We used a C++ implementation for MARCO and all its variants (section 5) for these experiments. In addition, the implementation makes use of the MUSer2 MUS extraction software [3] for the **shrink** function, made possible by MARCO's extensibility as explained in section 4.1. Underlying the algorithm is a SAT solver that acts as an oracle for SAT checks and generating models for satisfiable subsets; our solver of choice is MiniSAT v2.2 [6]. The experiments were run in Linux on 3.4GHz AMD Phenom II CPUs with a 3600 second timeout and a 1.8 GB memory limit. Of the 300 instances, the MARCO algorithm is unable to find a single MUS within the time limit in 56 cases or a single MCS in 45 cases. Therefore, the output results below show measure performance on the remaining 244 and 255 instances for MUSes and MCSes respectively.

Unsatisfiable constraint sets may be highly complex, with the number of MUSes and MCSes reaching exponential sizes in proportion to the constraint set [10]. As a result, extracting these characteristic subsets is a generally intractable problem that cannot be expected to complete under any realistic time restrictions. In fact, during our experiments, over 90% exceed the time limit. Therefore, total runtime to completion is not a useful metric when extracting information from highly intractable problems. 'Information' here is the generation of any MUS or MCS regardless of whether the algorithm runs to completion. As a result, the metrics we measure are rates of MUS and MCS enumeration. Dealing with such intractable output sizes was one of the weaknesses of earlier algorithms: CAMUS required enumerating all MCSes before a single MUS was generated, and thus a large number of MCSes acted as a bottleneck when seeking to output MUSes; for DAA, the size of a set of intermediate results quickly exhausted memory limits. MARCO is free from such restrictive measures and can enumerate both MUSes and MCSes in intractably large problem sets. We thus measure performance as the rate at which it generates these outputs irrespective of the total output size.

We compare the performance of variants of MARCO by comparing their respective enumeration rates. Since either of the variants might improve one of the output metrics (MUSes or MCSes) at the expense of the other, we split our experiments into those seeking to optimize MUS output rates (section 6.1) and those for MCS output rates (section 6.2). The graphs we present in this section are log-log scatter-plots comparing outputs of two variants, each represented along one of the axes. Each instance is tested on both the x-axis variant

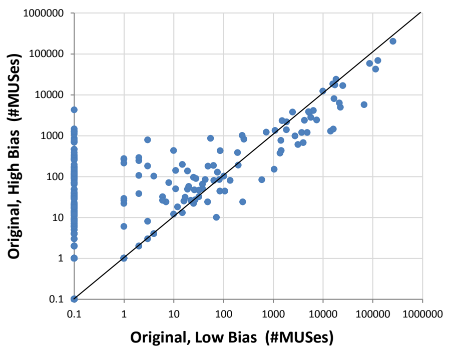


Fig. 9. #MUSes: High vs Low Bias

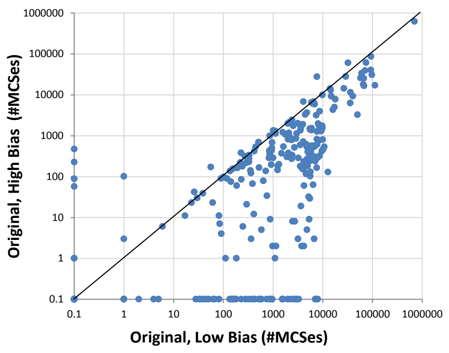


Fig. 10. #MCSes: High vs Low Bias

and the y-axis variant and the number of outputs for each variant represents its x-offset and y-offset respectively. A point on the diagonal thus represents an equal number of outputs across both variants. A point below the diagonal represents a relatively greater x-offset (compared to the y-offset) indicated more outputs for the x-axis variant. Likewise, a point above the diagonal represents better performance by the y-axis variant. Moreover, we use logarithmic scales in order to account for the highly variable number of outputs across different instances. The logarithmic scales necessitate mapping the 0 value to 0.1 in order to represent an output of 0 on the graph (i.e., when no MUS/MCS is found). While individual performances on instances might be split in terms of which variant performs better, we aggregate the extent of these improvements over all instances in order to determine the 'better' variant near the culmination of each of sections 6.1 and 6.2.

Search Biases We start by comparing the output rates of MARCO based on the 'bias' heuristic choice we defined in section 5.1. The original presentation used a high bias for MUS enumeration and we corroborate its effectiveness here. A high biased version (y-axis variant) of the original algorithm is compared, in terms of MUS enumeration, against a low biased version (x-axis variant) in figure 9. A cursory look reveals a majority of the instances represented above the diagonal indicating that a high bias outperforms a low bias in the number of MUS output. Moreover, a majority (75%) of the instances lie *on* the y-axis indicating cases where the high bias variant found at least one MUS while the low bias variant found none. This dominance is likely due to the high bias variant having favored UNSAT seeds which led to a greater number of **shrink** phases which, in turn, resulted in more MUSes. However, there do exist some cases where the low bias variant outperforms the high bias variant in obtaining MUSes. In fact, these cases are more prevalent in instances with a larger number of MUSes. Future work could explore the reasons behind this occurrence, however, it is obvious from the graph that the high bias variant outperforms the low bias variant in the general case.

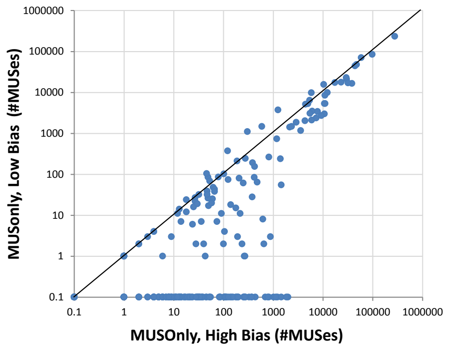


Fig. 11. MUSOnly: High vs Low Bias

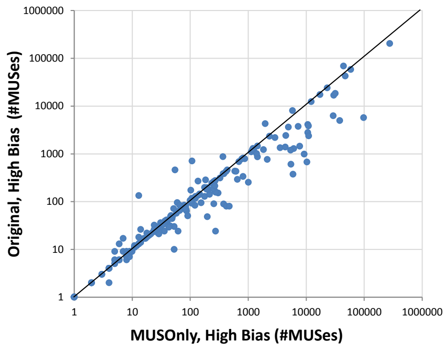


Fig. 12. MUSOnly vs Original

Similarly, the converse holds for the effect of low bias on finding MCSes which is shown in figure 10. A large number of instances, in this case, lie on the x-axis indicating non-zero MCS outputs for the low bias variant and no MCSes for the high biased variant. A low bias favors smaller cardinality seeds, which are more likely to exist within the SAT region (see section 5.1) and this resulted in more **grow** phases for the low bias variant, which in turn led to more MSS (and MCS) outputs. However, unlike in figure 9, the dominance here is more absolute as only a handful of instances exist above the diagonal which can be explained as likely edge cases.

Having seen how bias affects output rates, we move on to analyzing performance of the extensions to the original algorithm. We take into account the result of the bias heuristic choice when comparing variants. We favor high bias variants for MUS enumeration and low bias variants for MCS enumeration. However, this does not guarantee that a high or low bias will always outperform the other in terms of its favored output as modification may affect the high-bias MUS or low-bias MCS dominance. Therefore, we include 'high vs low' comparisons in order to justify using a particular bias in an extension variant when comparing it against the original algorithm. Since the variants often improve one of MUSes of MCSes at the expense of one another, we analyze performance for each of these outputs separately. We seek to optimize performance in terms of MUS enumeration in section 6.1 and MCS enumeration in section 6.2.

6.1 MUS Output

MUSOnly The MUSOnly extension (section 5.2) blocks particular SAT regions of the search space whenever it finds an MUS. Blocking more SAT regions makes it more likely for MARCO to generate a seed in the UNSAT region. More UNSAT seeds lead to a greater number of **shrink** phases and thus more MUS outputs. Both high and low bias versions of the MUSOnly extension are compared against each other in figure 11. The high bias variant is, expectantly, the dominant

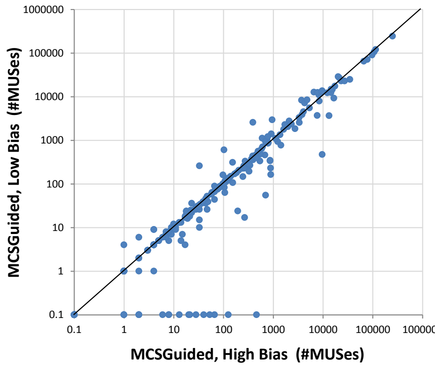


Fig. 13. MCSGuided: High vs Low Bias

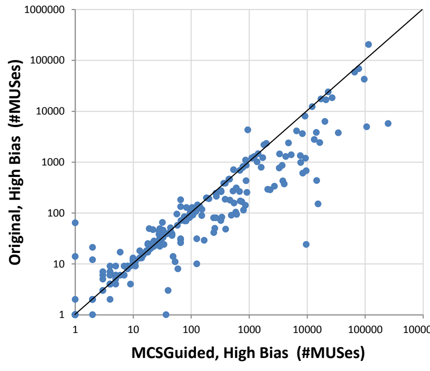
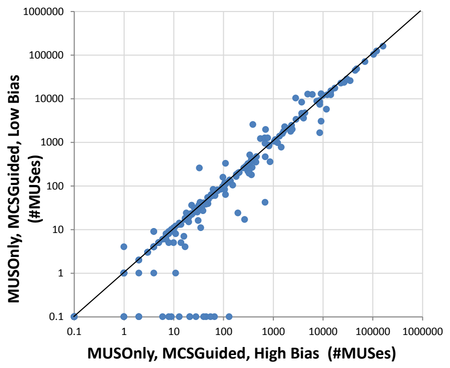


Fig. 14. MCSGuided vs Original

version, outperforming the low bias variant in terms of MUS enumeration. We thus compare the high biased variant of MUSOnly against the high biased variant of the original algorithm (which we have shown to be better than the low bias variant in figure 9). The comparison results are shown in figure 12 and show a definite improvement of the extension variant over the original as more instances clearly lie below the diagonal. One interesting observation is that instances with a higher number of MUSes show MUSOnly to be more dominant. This is likely due to the benefits of trimming the search space more decidedly outweighing the cost of the additional constraints that they require. Moreover, note (figure 11) that unlike the original (figure 9), the MUSOnly low bias variant is not favored for high MUS instances and this contributes to a better performance by MUSOnly for these high MUS instances. The net effect of the improvement is analyzed later in this section when we aggregate results for all variants.

MCSGuided The MCSGuided extension (section 5.3) makes use of MSSes(MCSes) to generate UNSAT seeds in order to better facilitate MUS enumeration. As a result of the improvement, finding an MSS directly leads to an MUS in the very next iteration if a superset of the MSS consists of an unconstrained UNSAT seed. This affects the high vs low bias comparison a great deal as low bias variants favoring MSSes are potentially linked to finding MUSes, in a sense mitigating the problem of finding MUSes in low biased variants. The high and low biased variants of MCSGuided are compared and the result is shown in figure 13. The instances seem to be closely placed around the diagonal except for some cases where the benefits of MCSGuided fail to have any effect and they fail to find *any* MUSes. These are instances where a low bias variant acts against the arrangement of the search space and the intractability of the problem is more realized. For instances where both versions find an MUS, their respective points seem fairly evenly distributed across the diagonal. However, since a high bias performs better in the general case, we compare this variant against the original algorithm in figure 14. The effect of the improvement seems far more pronounced



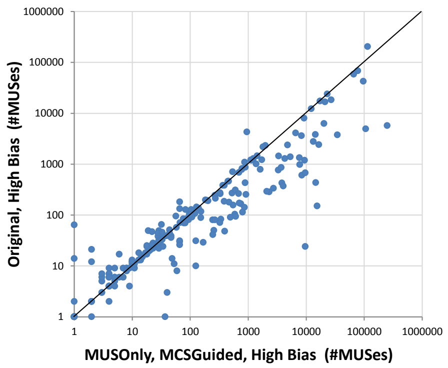


Fig. 15. MCSGuided+MUSOnly: High vs Low Bias

Fig. 16. MCSGuided+MUSOnly vs Original

here than in MUSOnly with a greater number of points lying below the diagonal. Again, we see the overall effect later in the section when we aggregate the relative performance across all instances.

MCSGuided + MUSOnly The improvements we have presented so far are not mutually exclusive, allowing us to add both to the original algorithm. Like in the previous extensions, we start with a comparison of the high and low biased variants. This is shown in figure 15 and, almost exactly, matches the corresponding comparison for MCSGuided (figure 13). As the high biased variant performs better on average, we compare it against the high biased variant of the original algorithm in figure 16. The scatter of instances in the graph matches closely with the performance comparison of MCSGuided (figure 14) suggesting that MCSGuided is the more effective extension in this variant and dominates the search procedure. There are subtle differences, however, and their effect is shown when we aggregate results later in the section.

MCSGuided++ The final improvement we suggest for MUS enumeration is MCSGuided++ (section 5.3). This variant extends the MCSGuided variant by exhausting any UNSAT seed options provided by every single MSS. However, the extra steps and constraint checking used to exhaust these options often ends up being more costly than just using one UNSAT seed per MSS like in MCSGuided. The difference is very obvious in the comparison of high and low biased variants in figure 17 where a high bias variant completely outperforms low bias variant in stark contrast to the respective graph for MCSGuided (figure 13). As a result, we use the high biased variant in a comparison against the original algorithm. The comparison against the original algorithm is shown in figure 18 and it likewise shows a less extensive improvement over the original than the previous variants; however, an interesting characteristic in this figure is that an instance *rarely* performs worse than the original algorithm unlike previous variants which show some degree of variance in this regard.

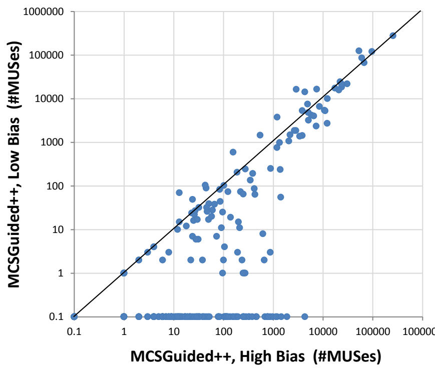


Fig. 17. MCSGuided++: High vs Low Bias

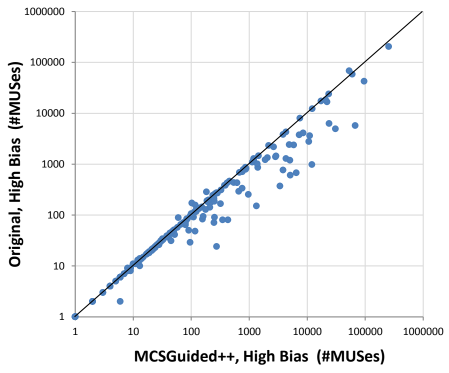


Fig. 18. MCSGuided++ vs Original

Aggregate Results While a scatter of values on a graph can indicate general trends on which variant performs better in a one-on-one comparison, comparisons between more than two variants are harder to display. As a result, we aggregate performance over all instances under a commonly used metric. For each instance, we can represent *performance* as a ratio of the number of outputs of the variant to the number of outputs of the original algorithm. A value of 1 would then represent the same number of outputs for both and would be a point on the diagonal in the corresponding scatter-plot. Additionally, this makes the ratio independent of the specific number of outputs for that instance. We use a *geometric* mean to aggregate such ratios across all instances for each of the variants. Table 1 shows these ratios averaged over 244 instances for which at least one MUS was found. By this metric, the MCSGuided variant performs the best with a 42% increase in MUSes output over the original algorithm.

Table 1. Geometric mean of ratios: #MUSes output by each variant vs original

Variant	#MUSes Ratio: Variant / Original
MUSOnly	1.195
MCSGuided	**1.423**
MCSGuided+MUSOnly	1.342
MCSGuided++	1.228

Beyond the overall increase in performance, it is helpful to note how *many* instances actually experienced an increased or decreased output. Thus using the same ratios from the previous aggregation, we use counts of particular interest to us instead of calculating a geometric mean. In this case, we count the number of instances over which output increased, decreased, doubled, halved, etc. This

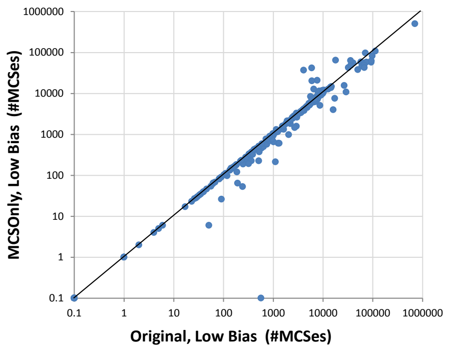


Fig. 19. Original (Low Bias) vs MCSOnly (Low Bias)

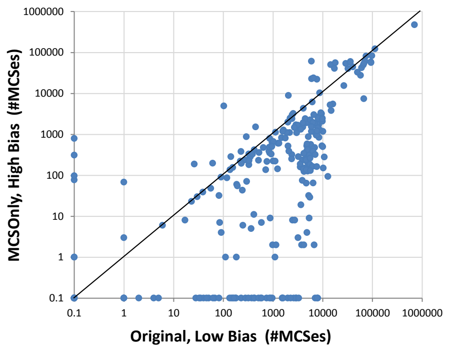


Fig. 20. Original (Low Bias) vs MCSOnly (High Bias)

is shown in table 2 for each of the extensions. The bolded values indicate the most increases (or fewest decreases) for each particular count. The results show that MCSGuided has the greatest number of instances with a general increase in output while MCSGuided++ has the fewest instances with a general decrease in output.

Table 2. Number of instances with increases, decreases in #MUSes vs original

	#MUSes Ratio:					
Variant	<¼	<½	<1	>1	>2	>4
MUSOnly	3	7	56	99	29	17
MCSGuided	5	13	72	**104**	**61**	**39**
MCSGuided+MUSOnly	7	14	76	**105**	53	32
MCSGuided++	**0**	**0**	**31**	96	30	12

6.2 MCS Output

MCSOnly The MCSOnly extension (Section 5.2) acts as 'dual' to the MUSOnly extension and blocks particular UNSAT regions of the search space whenever it finds an MSS. Blocking more UNSAT regions makes it more likely for MARCO to generate a seed in the SAT region which would lead to more **grow** phases and MCSes. Figure 19 shows the comparison of a low biased MCSOnly variant against the low biased original version. Any improvements are scarce, as most points lie on the diagonal. In fact, when we aggregate results later in the section, we find MCSOnly actually *decreases* performance over the original. For comparison's sake, we include a graph (figure 20) showing the comparison of a high biased MCSOnly variant against the original (low biased). The unique changes from

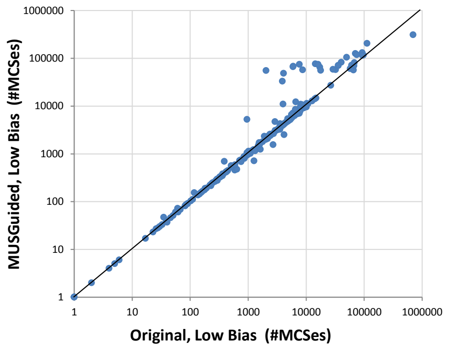


Fig. 21. MUSGuided vs Original (both low biased)

MCSOnly do not fail to affect the dominance of low bias in heavily outperforming the higher biased version in terms of MCS enumeration. The few instances where the high biased variant manages to outperform the low biased one are similarly matched in the original high vs low comparison (see figure 10) and thus are unrelated to any changes in MCSOnly.

MUSGuided The MUSGuided extension similarly follows as a dual of the MCSGuided extension (section 5.3). It uses MUSes to generate SAT seeds in order to improve the likelihood of obtaining MSSes from **grow** operations. In fact, given that it can find an unconstrained SAT seed as a subset of the MUS, MUSGuided will find an MSS (and thus MCS) in the very next iteration of MARCO. The MUSGuided with a low bias outperforms the high biased variant and therefore we compare it against the original in figure 21. A majority of the points lie on the diagonal, however, in case of instances with a large number of MCSes, the MUSGuided variant seems to outperform the original. While the extent of the increased performance is unclear, it seems definite that there is an increase in performance over the original. We analyze the aggregated impact of the extension later in this section.

MUSGuided+MCSGuided This is a particularly interesting combination of variants that resulted from the following thought experiment:

1. An MUS is found.
2. Subsets of the MUS are satisfiable.
3. An unconstrained SAT subset is picked as a seed. (MUSGuided)
4. The seed is put through a **grow** function.
5. An MSS is found.
6. Supersets of the MSS are unsatisfiable.
7. An unconstrained UNSAT superset is picked as a seed. (MCSGuided)
8. The seed is put through a **shrink** function.
9. Repeat from 1.

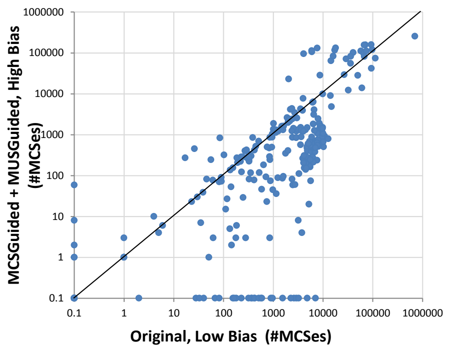


Fig. 22. Original (Low Bias) vs MUSGuided+MCSGuided (Low Bias)

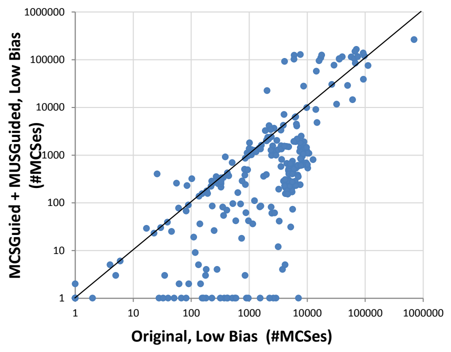


Fig. 23. Original (Low Bias) vs MUSGuided+MCSGuided (Low Bias)

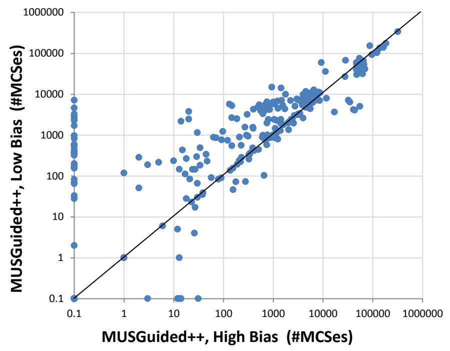


Fig. 24. MUSGuided++ Low vs High Bias

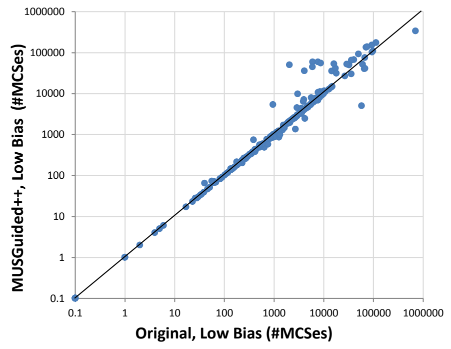


Fig. 25. Original (Low Bias) vs MUSGuided++ (Low Bias)

The starting point of this variant may be from 1. or 5. depending on whether an MSS or MUS is found first. Although the probability of so many seeds being unconstrained is low, it has the potential to make use of both, the MCSGuided and MUSGuided extensions. We run this 'extension' with both high and low biased variants and compare the output against the original algorithm. The high biased variant against the original (with a low bias to favor MCS output) is shown in figure 22 while a low biased variant is compared to the original in figure 23. Unfortunately, the method does not perform well against the original and in fact, does *worse* than the original algorithm in finding MCSes or MUSes with either bias. However, on the few instances where it outperforms the original in MCS enumeration, it also outperforms other variants with a relatively greater increase in the number of outputs. This is indicated when we consider the number of instances over which the variant performs twice as well as the original later in this section.

MUSGuided++ MUSGuided++ acts similarly to its counterpart in MUS enumeration. It exhausts all subsets of an MUS in the search for SAT seeds to increase MCS enumeration. Low and high biased variants of the extension are compared in figure 24. The low biased variant outperforms the high biased variant and therefore, we compare it against the original in figure 25. Its performance against the original is similar to MUSGuided's performance with a majority of the instances lying on the diagonal and a few large MCS instances showing a performance gain.

Aggregate Results We use similar aggregation metrics for MCSes that we used when comparing MUS enumeration rates. Table 3 shows the geometric mean of the ratios averaged over the 255 instances for which at least one MCS was found. All variants shown are low biased (as they tend to favor MCS enumeration) except where explicitly mentioned. In this metric, the MUSGuided variant performs the best, with an 11% increase in MCS output rate over the original algorithm.

Table 3. Average ratio of #MCSes output by each variant vs original

Variant	#MCSes Ratio: Variant / Original
MCSOnly	0.925
MUSGuided	**1.106**
MUSGuided+MCSGuided	0.200
MUSGuided+MCSGuided (high bias)	0.218
MUSGuided++	1.101

Again, we tabulate information to show how many instances actually experienced an increase or decrease in output rate compared to the original. Table 4 shows this information. We similarly group instances where performance increased, decreased, doubled, halved, etc. The bolded values indicate the most increases (or fewest decreases) for each particular performance count. In particular, MUSGuided appears to be the best performing with the most number of instances outperforming the original and the least having performed worse than the original. An intriguing thing to note is how MUSGuided+MCSGuided (true bias) has the greatest number (although only about 10% of the instances) of instances on which it performed over 200% to over 400% better than the original algorithm. However, overall, it decidedly performs worse than the original which is plainly evident from table 3.

7 Conclusion

The extensible nature of MARCO makes it convenient to make changes in specific parts of the search mechanism without affecting how the rest of the algorithm

Table 4. Number of instances with increases, decreases in #MCSes vs original (low bias)

Variant	#MCSes Ratio: <¼	<½	<1	>1	>2	>4
MCSOnly	5	13	108	88	5	2
MUSGuided	**1**	**2**	**70**	**122**	16	10
MUSGuided+MCSGuided	124	147	178	50	21	10
MUSGuided+MCSGuided (high bias)	125	147	175	61	**28**	**18**
MUSGuided++	**1**	**2**	**68**	117	11	7

operates. This allows us to optimize the algorithm in favor of certain outputs, namely MCSes or MUSes. In this paper, we demonstrate precisely this. In addition to these optimizing improvements, we validate a heuristic choice made about the search bias in the original implementation of the algorithm. We show how a high bias leads to an MUS favorable output while a low bias leads to an MCS favorable one.

The MARCO algorithm is targeted towards large constraint systems where finding all MUSes/MCSes is not viable under any realistic time constraint. Therefore, the enumeration rate of MUSes/MCSes determines the performance of the algorithm in relation to the improvements we suggest. The first improvement, MUSOnly, makes use of the fact that for any particular application, it is only important to obtain one of MUSes/MCSes and therefore, if we block all known SAT regions of the search space as soon as we can make such a distinction, the algorithm spends more time exploring the UNSAT region thus finding more MUSes. The same would appear to hold theoretically for UNSAT regions and MCSes, but the experiments show no improvement and we explain why that is so. MCSGuided, the second improvement, makes use of the fact that having found an MCS/MUS, using a random seed in the next iteration ignores valuable information about the satisfiability of its subsets/supersets. We use this information to generate a seed for the next iteration of the algorithm depending on our output specification. The third improvement, MCSGuided++, naturally extends from MCSGuided as we exhaust all possible 'local' seeds before asking the solver to generate a random one.

The improvements that we outline outperform the original algorithm with MCSGuided performing the best of the lot in terms of MUS enumeration with a mean improvement of 42%.For MCS enumeration, the results are not as pronounced; however, having explored the theoretical duals of the improvements for MUS enumeration is significant in itself. The fact that MCS enumeration rates remain fairly constant while MUS enumeration rates are affected with similar changes belies the duality that exists between them.

7.1 Future Work

To conclude, we have shown improvements to MARCO and outlined its extensibility. As a result, several avenues of research in improving MARCO still remain open. Specific directions for future work might include:

1. Looking into possible combinations of biases rather than configuring them all to be high or low. This could produce variability and add more specificity into where the search 'tends' to based on certain types of problems. For example, for a problem where you expect the MUSes to be of a certain cardinality, dynamically allocating bias so as to induce seeds of (or near) that cardinality would help improve enumeration rates by cutting down on grow/shrink times.
2. More tightly integrating and sharing information between the `map` solver and the CSP.
3. Comparing the performance of MARCO's MCS enumeration with that of existing algorithms [11,13].
4. Investigating the differences between MCS and MUS enumeration in MARCO and why they respond differently to similar modifications. Exploring in greater depth the duality that exists between them and using *all* the previous MUSes/MCSes found by MARCO to improve 'guessing' at future seeds.
5. Putting forth additional completeness relaxations and analyzing respective performance trade-offs. For example: combinations of MUSOnly and MCSOnly, or blocking from non-characteristic subsets, i.e., greedily blocking to reduce the size of the unexplored search space in order to speed up enumeration rates.

References

1. SAT 2011 Competition website, http://www.cril.univ-artois.fr/SAT11/
2. 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)
3. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. Journal on Satisfiability, Boolean Modeling and Computation 8, 123–128 (2012)
4. Chinneck, J.W.: Isolating infeasibility. In: Feasibility and Infeasibility in Optimization, International Series in Operations Research and Management Science, vol. 118, pp. 93–157. Springer US (2008)
5. De Givry, S., Larrosa, J., Meseguer, P., Schiex, T.: Solving max-sat as weighted csp. In: Principles and Practice of Constraint Programming–CP 2003. pp. 363–376. Springer (2003)
6. 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)
7. Kleine Büning, H., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 11, pp. 339–401. IOS Press (Feb 2009)
8. Li, C.M., Manya, F.: Maxsat, hard and soft constraints. Handbook of satisfiability 185, 613–631 (2009)
9. Liffiton, M.H., Malik, A.: 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) (May 2013), [to appear, available online]
10. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (Jan 2008)
11. 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)
12. Marques-Silva, J.: Minimal unsatisfiability: Models, algorithms and applications (invited paper). In: 40th IEEE International Symposium on Multiple-Valued Logic (ISMVL-2010). pp. 9–14 (May 2010)
13. Morgado, A., Liffiton, M.H., , Marques-Silva, J.: MaxSAT-based MCS enumeration. In: Proceedings of the 8th International Haifa Verification Conference (HVC-2012) (2012), [to appear]
14. Zhang, H., Shen, H., Manya, F.: Exact algorithms for max-sat. Electronic Notes in Theoretical Computer Science 86(1), 190–203 (2003)