Tunable Online MUS/MSS Enumeration

Jaroslav Bendík[1], Nikola Beneš[2], Ivana Černá[3], and Jiří Barnat[4]

1 **Faculty of Informatics, Masaryk University, Brno, Czech Republic**
`xbendik@fi.muni.cz`

2 **Faculty of Informatics, Masaryk University, Brno, Czech Republic**
`xbenes3@fi.muni.cz`

3 **Faculty of Informatics, Masaryk University, Brno, Czech Republic**
`cerna@fi.muni.cz`

4 **Faculty of Informatics, Masaryk University, Brno, Czech Republic**
`barnat@fi.muni.cz`

Abstract

In various areas of computer science, the problem of dealing with a set of constraints arises. If the set of constraints is unsatisfiable, one may ask for a minimal description of the reason for this unsatisifiability. Minimal unsatisfiable subsets (MUSes) and maximal satisfiable subsets (MSSes) are two kinds of such minimal descriptions. The goal of this work is the enumeration of MUSes and MSSes for a given constraint system. As such full enumeration may be intractable in general, we focus on building an online algorithm, which produces MUSes/MSSes in an on-the-fly manner as soon as they are discovered. The problem has been studied before even in its online version. However, our algorithm uses a novel approach that is able to outperform the current state-of-the art algorithms for online MUS/MSS enumeration. Moreover, the performance of our algorithm can be adjusted using tunable parameters. We evaluate the algorithm on a set of benchmarks.

1998 ACM Subject Classification F.4.1 Logic and constraint programming

Keywords and phrases Minimal unsatisfiable subsets, Maximal satisfiable subsets, Unsatisfiability analysis, Infeasibility analysis

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

In various areas of computer science, such as constraint processing, requirements analysis, and model checking, the following problem often arises. We are given a set of constraints and are asked whether the set of constraints is feasible, i.e. whether all the constraints are satisfiable together. In requirements analysis, the constraints represent the requirements on a given system, usually described as formulae of a suitable logic, and the feasibility question is in fact the question whether all the requirements can actually be implemented at once. In some model checking systems, such as those using the counterexample-guided abstraction refinement (CEGAR) workflow, an infeasible constraint system may arise as a result of the abstraction's overapproximation. In such cases where the set of constraints is infeasible, we might want to explore the reasons of infeasibility. There are basically two approaches that can be used here. One is to try to extract a single piece of information explaining the infeasibility, such as a minimal unsatisfiable subset (MUS) or dually a maximal satisfiable subset (MSS) of the constraints. The other option is to try to enumerate all, or at least as many as possible, of these sets. In this work, we focus on the second approach. Enumerating multiple MUSes is sometimes desirable: in requirements analysis, this gives better insight

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

into the inconsistencies among requirements; in CEGAR-based model checking more MUSes lead to a better refinement that can reduce the complexity of the whole procedure [2].

The enumeration of all MUSes or MSSes is generally intractable due to the potentially exponential number of results. It thus makes sense to study algorithms that are able to provide at least some of those within a given time limit. An even better option is to have an algorithm that produces MUSes or MSSes in an on-the-fly manner as soon as they are discovered. It is the goal of this paper to describe such an algorithm.

1.1 Related Work

The list of existing work that focuses on enumerating multiple MUSes is short as most of the related work only deals with an extraction of a single MUS or even a non-minimal unsatisfiable subset. For example all of [7, 17, 19] use information from a satisfiability solver to obtain an unsatisfiable subset but they do not guarantee its minimality. Moreover, the majority of the algorithms which enumerate all MUSes have been developed for specific constraint domains, mainly for Boolean satisfiability problems.

Explicit Checking The first algorithm for enumerating all MUSes we are aware of was developed by Hou [11] in the field of diagnosis and is built on explicit enumeration of every subset of the unsatisfiable constraint system. It checks every subset for satisfiability, starting from the complete constraint set and branching in a tree-like structure. The authors presented some pruning rules to skip irrelevant branches and avoid unnecessary work. Further improvements to this approach were made by Han and Lee [10] and by de la Banda et. al. [8].

CAMUS A state-of-the-art algorithm for enumerating all MUSes called CAMUS by Liffiton and Sakallah [16] is based on the relationship between MUSes and the so-called minimal correction sets (MCSes), which was independently pointed out by [3, 6, 14]. This relationship states that $M \subseteq C$ is a MUS of C if and only if it is an irreducible hitting set of $\mathrm{MCS}(C)$. CAMUS works in two phases, first it computes all MCSes of the given constraint set, and then it finds all MUSes by computing all the irreducible hitting sets of these MCSes.

A significant shortcoming of CAMUS is that the first phase can be intractable as the number of MCSes may be exponential in the size of the instance and all MCSes must be enumerated before any MUS can be produced. This makes CAMUS unsuitable for many applications which require only a few MUSes but want to get them quickly. Note that CAMUS is able to enumerate MSSes, as they are simply the complements of MCSes.

MARCO The desire to enumerate at least some MUSes even in the generally intractable cases led to the development of two independent but nearly identical algorithms: MARCO [13] and eMUS [18]. Both algorithms were later joined and presented in [15] under the name of MARCO. MARCO is able to produce individual MUSes during its execution and it does it in a relatively steady rate. To obtain each single MUS, MARCO first finds a subset U whose satisfiability is not known yet, checks it for satisfiability and if it is unsatisfiable, it is "shrunk" to a MUS. In the case that U is satisfiable, it is in a dual manner expanded into an MSS. The algorithm can be supplied with any appropriate shrink and expansion procedures; this makes MARCO applicable to any constraint satisfaction domain in general.

CAMUS and MARCO were experimentally compared in [15] and the former has shown to be faster in enumerating all MUSes in the tractable cases. However, in the intractable cases, MARCO was able to provide at least some MUSes while CAMUS often provided none. One another algorithm, the Dualize and Advance (DAA) by Bailey and Stuckey [3] was also evaluated in these experiments. DAA is also based on the relationship between MCSes and MUSes and can produce both MUSes and MSSes during its execution; however, it has shown to be substantially slower than CAMUS in the case of complete MUSes enumeration and also slower than MARCO in the partial enumeration.

1.2 Our Contribution

In this paper, we present our own algorithm for online enumeration of MUSes and MSSes in general constraint satisfaction domains that is able to outperform the current state-of-the-art MARCO algorithm. The core of the algorithm is based on a novel concept of *local* MUSes/MSSes. To find these we use a binary-search-based approach. Similarly to MARCO, the algorithm is able to directly employ arbitrary shrinking and expanding procedures. Moreover, our algorithm contains certain parameters that govern in which cases the shrinking and expanding procedures are to be used. We evaluate our algorithm on a variety of benchmarks that show that the algorithm indeed outperforms MARCO.

This paper builds on our previous work [5] where we focused on finding boundary elements in partially ordered sets represented by explicit acyclic graphs. Here we focus on the specific case of powersets represented symbolically. Another difference is that we perform online enumeration here.

Note that there is a constraint solving approach QuickXplain [12] which uses binary search, however it solves a different problem. It uses a linear priority ordering on constraints and extracts a single maximal consistent subset w.r.t. this priority.

Outline of The Paper In Section 2 we state the problem we are solving in a formal way, defining all the necessary notions. In Section 3 we describe the algorithm in an incremental way, starting with the basic schema of MUS/MSS computation and gradually explaining the main ideas of our algorithm. Section 4 provides an experimental evaluation on a variety of benchmarks, comparing our algorithm against MARCO. The paper is concluded in Section 5.

2 Preliminaries

Our goal is to deal with arbitrary constraint satisfaction systems. The input is given as a finite set of constraints $C = \{c_1, c_2, \dots, c_n\}$ with the property that each subset of C is either *satisfiable* or *unsatisfiable*. The definition of satisfiability may vary in different constraint domains, we only assume that if $X \subseteq C$ is satisfiable, then all subsets of X are also satisfiable. The subsets of interest are defined in the following.

▶ **Definition 1** (MSS, MUS). Let C be a finite set of constraints and let $N \subseteq C$. N is a *maximal satisfiable subset* (MSS) of C if N is satisfiable and $\forall c \in C \setminus N : N \cup \{c\}$ is unsatisfiable. N is a *minimal unsatisfiable subset* (MUS) of C if N is unsatisfiable and $\forall c \in N : N \setminus \{c\}$ is satisfiable.

Note that the maximality concept used here is set maximality, not maximum cardinality as in the MaxSAT problem. This means there can be multiple MSSes with different cardinality. We use $MUS(C)$ and $MSS(C)$ to denote the set of all MUSes and MSSes of C, respectively. The formulation of our problem is the following: Given a finite set of constraints C, enumerate (all or at least as many as possible) members of $MUS(C)$ and $MSS(C)$.

To describe the ideas of our algorithm and illustrate its usage, we shall use Boolean satisfiability constraints in the following. In the examples, each of the constraints c_i is going to be a clause (a disjunction of literals). The whole set of constraints can be then seen as a Boolean formula in conjunctive normal form.

▶ **Example 2.** We illustrate the concepts on a small example. Assume that we are given a set C of four Boolean satisfiability constraints $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$. Clearly, the whole set is unsatisfiable as the first two constraints are negations of each other. There are two MUSes: $\{c_1, c_2\}$, $\{c_1, c_3, c_4\}$ and three MSSes: $\{c_1, c_4\}$, $\{c_1, c_3\}$, $\{c_2, c_3, c_4\}$.

The *powerset* of C, i.e. the set of all its subsets, forms a lattice ordered via subset inclusion and denoted by $\mathcal{P}(C)$. In our algorithm we are going to deal with the so-called *chains* of the powerset and deal with local MUSes and MSSes, defined as follows.

▶ **Definition 3.** Let C be a finite set of constraints. The sequence $K = \langle N_1, \dots N_i \rangle$ is a *chain* in $\mathcal{P}(C)$ if $\forall j : N_j \in \mathcal{P}(C)$ and $N_1 \subset N_2 \subset \dots \subset N_i$. We say that N_k is a *local MUS* of K if N_k is unsatisfiable and $\forall j < k : N_j$ is satisfiable. Similarly, we say that N_k is a *local MSS* of K if N_k is satisfiable and $\forall j > k : N_j$ is unsatisfiable.

Note that there is no local MUS if all subsets on the chain are satisfiable, and there is no local MSS if all subsets on the chain are unsatisfiable.

3 Algorithm

In this section, we gradually present an online MUS/MSS enumeration algorithm. Consider first a naive enumeration algorithm that would explicitly check each subset of C for satisfiability, split the subsets of C into satisfiable and unsatisfiable subsets, and choose the maximal and minimal subsets of the two groups, respectively. The main disadvantage of this approach is the large number of satisfiability checks. Checking a given subset of C for satisfiability is usually an expensive task and the naive solution makes an exponential number of these checks which makes it unusable.

Note that the problem of MUS enumeration contains the solution to the problem of satisfiability of all subsets of C as each unsatisfiable subset of C is a superset of some MUS. This means that every algorithm that solves the problem of MUS enumeration has to make several satisfiability checks during its execution. These checks are usually done employing an external satisfiability solver. Clearly, the number of such external calls corresponds with the efficiency of the algorithm. Therefore, we want to minimise the number of calls to the solver.

3.1 Basic Schema

Recall that the elements of $\mathcal{P}(C)$ are partially ordered via subset inclusion and each element is either satisfiable or unsatisfiable. The key assumption on the constraint domain, as declared above, is that the partial ordering of subsets is preserved by the satisfiability of these subsets. If we thus find an unsatisfiable subset N_u of C then all supersets of N_u are also unsatisfiable; dually, if we find a satisfiable subset N_s of C then all subsets of N_s are also satisfiable. Moreover, none of the supersets of N_u can be a MUS and none of the subsets of N_s can be an MSS. In the following text we refer to this property as to the *monotonicity* of $\mathcal{P}(C)$ and to the elements of $\mathcal{P}(C)$ as to *nodes*.

Our basic algorithm is described in pseudocode as Algorithm 1. The algorithm consists of two phases. In the first phase it determines the satisfiability of all nodes and extracts from $\mathcal{P}(C)$ a set of MSS *candidates* MSS_{can} and a set of MUS candidates MUS_{can} ensuring that $MSS(C) \subseteq MSS_{can}$ and $MUS(C) \subseteq MUS_{can}$. In the second phase it reduces MSS_{can} to $MSS(C)$ and MUS_{can} to $MUS(C)$.

During the execution of the first phase the algorithm maintains a classification of nodes; each node can be either *unexplored* or *explored* and some of the explored nodes can belong

Algorithm 1: The basic schema of our algorithm

```
Unex ← P(C)
MSS_can, MUS_can ← ∅
while Unex is not empty do
    Nodes ← some unexplored nodes
    for each N ∈ Nodes do
        if N is satisfiable then
            MSS_can ← MSS_can ∪ {N}
            Unex ← Unex \ Sub(N)
        else
            MUS_can ← MUS_can ∪ {N}
            Unex ← Unex \ Sup(N)
extract MSSes from MSS_can
extract MUSes from MUS_can
```

to MSS_{can} or to MUS_{can}. The *explored* nodes are those whose satisfiability the algorithm already knows and the *unexplored* nodes are the remaining ones. The algorithm stores the unexplored nodes in the set *Unex* which initially contains all nodes from $\mathcal{P}(C)$. The first phase is iterative; the algorithm in each iteration selects some unexplored nodes *Nodes*, determines their satisfiability using an external satisfiability solver, and exploits the monotonicity of $\mathcal{P}(C)$ to deduce satisfiability of some other unexplored nodes. At the end of each iteration the algorithm updates the set *Unex* by removing from it the nodes whose satisfiability was decided in this iteration. Based on its satisfiability, every node from the set *Nodes* is added either into MSS_{can} or MUS_{can}.

In the pseudocode, we use $Sup(N)$ to denote the set of all unexplored supersets of N including N and $Sub(N)$ to denote the the set of all unexplored subsets of N including N.

Clearly, the schema converges as the set of unexplored nodes decreases its size in every iteration. The schema also ensures that after the last iteration it holds that $MUS(C) \subseteq MUS_{can}$ and $MSS(C) \subseteq MSS_{can}$. This is directly implied by the monotonicity of $\mathcal{P}(C)$ as no node whose satisfiability was deduced can be an MSS and dually no node whose unsatisfiability was deduced can be a MUS.

In the second phase our algorithm extracts all MUSes and MSSes from MUS_{can} and MSS_{can}. Both these extractions can be done by any algorithm that extracts the highest and the lowest elements from any partially ordered set. A trivial algorithm can just test each pair of elements for the subset inclusion and remove the undesirable elements, which can be done in time polynomial to the number of constraints in C and the size of the sets of candidates. We assume that this part of our algorithm is not as expensive as the rest of it, especially when each check for a satisfiability of a set of constraints may require solving an NP-hard problem. We therefore omit the discussion of the second phase in the following and focus solely on the way the set *Nodes* is chosen in each iteration and the way the unexplored nodes are managed.

3.2 Symbolic Representation of Nodes

Our algorithm highly depends on an efficient management of nodes. In particular it needs to reclassify some nodes from unexplored to explored and build chains from the unexplored nodes.

Probably the simplest way of managing nodes would be their explicit enumeration; however, there are exponentially many subsets of $C = \{c_1, \cdots, c_n\}$ and their explicit enumeration is thus intractable for large instances. We thus use a symbolic representation of nodes instead.

We exploit the well-known isomorphism between finite powersets and Boolean algebras. That is, we encode the set of constraints $C = \{c_1, \ldots, c_n\}$ using a set of Boolean variables $X = \{x_1, \ldots, x_n\}$. Each subset of C (i.e. each node in our algorithm) is then represented by a valuation of the variables of X. This allows us to represent sets of nodes using Boolean formulae over X. We use $f(\mathit{Nodes})$ to denote the Boolean formula representing the set *Nodes* in the following.

As an example, consider a set of constraints $C = \{c_1, c_2, c_3\}$ and let $\mathit{Nodes} = \{\{c_1\}, \{c_1, c_2\}, \{c_1, c_3\}\}$ be a set of three nodes. Using the Boolean variables representation of C, we can encode the set *Nodes* using the Boolean formula $f(\mathit{Nodes}) = x_1 \wedge (\neg x_2 \vee \neg x_3)$.

The advantage of this representation is that we can efficiently perform set operations over sets of nodes. The union of two sets of nodes *NodesA*, *NodesB* is carried out as a disjunction and their intersection as a conjunction. To get an arbitrary node from a given set, say *Unex*, we use an external SAT solver (more details in the next subsection). Note that this means that our algorithm employs two external solvers: One is the constraint satisfaction solver that decides satisfiability of the nodes, one is the SAT solver that works with our Boolean description of the constraint set and is employed to produce unexplored nodes. To clearly distinguish between these two we shall in the following use the phrases "constraint solver" and "SAT solver" rigorously.

3.3 Unexplored Nodes Selection

Let us henceforth denote one specific call to the constraint solver as a *check*. We now clarify which nodes our algorithm chooses in each of its iterations to be *checked* and which nodes it adds into the sets of candidates on MUSes and MSSes. We also extend the basic schema which was presented as Algorithm 1. We want to minimise the ratio of performed checks to the number of nodes in $\mathcal{P}(C)$. Every algorithm for solving the problem of MUSes enumeration has to perform at least as many checks as there are MUSes, so this ratio can never be zero. Also, it is impossible to achieve the ratio with a minimal value without knowing which nodes are satisfiable and which are not and this information is not a part of the input of our algorithm. Instead of minimising this overall ratio, our algorithm tends to minimise this ratio locally in each of its iterations.

In order to select the nodes which are checked in one specific iteration, our algorithm at first constructs an *unexplored chain*. An *unexplored chain* is a chain $K = \langle N_1, \ldots, N_k \rangle$ that contains only unexplored nodes and that cannot be extended by adding another unexplored nodes to its ends, i.e. N_1 has no unexplored subset and N_k has no unexplored superset. The monotonicity of $\mathcal{P}(C)$ implies that either (i) all nodes of K are satisfiable, (ii) all nodes of K are unsatisfiable, or (iii) K has a local MSS and a local MUS, i.e. there is some j such that $\forall 0 \leq i \leq j : N_i$ is satisfiable and $\forall k \geq l > j : N_l$ is unsatisfiable. This allows us to employ binary search to find such j performing only logarithmically many checks in the length of the chain. Let us analyse the three possible cases:

(i) all nodes of K are satisfiable, hence our algorithm deduces that all proper subsets of N_k are satisfiable and none of them can be an MSS, and it marks N_k as an MSS candidate;
(ii) all nodes of K are unsatisfiable, hence our algorithm deduces that all proper supersets of N_1 are unsatisfiable and none of them can be a MUS, and it marks N_1 as a MUS candidate; or

Algorithm 2: The modification of the basic schema of our algorithm

...
while *Unex is not empty* **do**
  $K \leftarrow$ some unexplored chain `// this line is added`
  $Nodes \leftarrow$ `processChain`(K) `// this line is modified`
  for *each* $N \in Nodes$ **do**
    ...

(iii) N_j is the local MSS of K and N_{j+1} is its local MUS, hence our algorithm deduces that all proper subsets of N_j are satisfiable, all proper supersets of N_{j+1} are unsatisfiable, and it marks N_j as an MSS candidate and N_{j+1} as a MUS candidate.

Algorithm 2 shows the modification of the basic schema of our algorithm (see Algorithm 1) which incorporates the above method for choosing nodes to be checked. At the beginning of each iteration the algorithm finds an unexplored chain K which is subsequently processed by the *processChain* method. This method finds the local MUS and local MSS of K (possibly only one of those) using binary search and returns them.

To construct an unexplored chain, our algorithm first finds a pair of unexplored nodes (N_1, N_k) such that $N_1 \subseteq N_k$ and then builds a chain $\langle N_1, N_2, \dots, N_{k-1}, N_k \rangle$ by connecting these two nodes. The intermediate nodes $N_2, \dots, N_{k-1}$ are obtained by adding one by one the constraints from $N_k \setminus N_1$ to the node N_1. We refer to each such pair of unexplored nodes (N_1, N_k) that are the end nodes of some unexplored chain as to an *unexplored couple*.

In order to find an unexplored couple our algorithm asks for a member of *Unex* by employing the SAT solver (by asking for a model of the formula $f(\mathit{Unex})$). Besides the capability of finding an arbitrary member of *Unex*, we require the following capability: For a given member $N_p \in \mathit{Unex}$, the SAT solver should be able to produce a *minimal* $N_q \in \mathit{Unex}$ such that $N_q \subseteq N_p$, where *minimal* means that there is no other N_r with $N_r \subset N_q$. Similarly, we require the SAT solver to be able to produce *maximal* such N_q. One of the SAT solvers that satisfies our requirements is miniSAT [9] that allows the user to fix values of some variables and to select a default polarity of variables at decision points during solving. To obtain a minimal N_q which is a subset of N_p, we set the default polarity of variables to False and fix the truth assignment to the variables that have been assigned False in N_p. Similarly for the maximal case.

We now describe two approaches of obtaining unexplored couples, assuming that we employ a SAT solver satisfying the above requirements.

Basic approach The *Basic approach* consists of two calls to the SAT solver. The first call asks the SAT solver for an arbitrary minimal member of *Unex*. If nothing is returned then there are no more unexplored nodes. Otherwise we obtain a node N_k which is minimal in *Unex*. We then ask the SAT solver for a maximal node $N_l \in \mathit{Unex}$ such that N_l is a superset of N_k. The pair (N_k, N_l) is then the new unexplored couple.

Pivot based approach Supposing that the SAT solver works deterministically, a series of calls for maximal (minimal) nodes of *Unex* may return nodes from some local part of the search space that may lead to construction of unnecessarily short chains. Therefore, we propose to first choose a *pivot* N_p, an unexplored node which may be neither maximal nor minimal and which should be chosen somehow randomly. As the next step this approach asks the SAT solver for a minimal node N_k such that $N_k \subseteq N_p$ and for a maximal node

Algorithm 3: processChain($C, K = \langle N_1, \ldots, N_k \rangle$)

find local MSS N_s and MUS N_u of K using binary search
if $u < S(|K|)$ **then**
  $N_u \leftarrow shrink(N_u)$
  `yieldMUS`(N_u) // Output MUS
if $s > |K| - G(|K|)$ **then**
  $N_s \leftarrow grow(N_s)$
  `yieldMSS`(N_s) // Output MSS
return $\{N_u, N_s\}$ // Note that N_u or N_s may not exist

N_l such that $N_p \subseteq N_l$. The new unexplored couple is then (N_k, N_l). The randomness in choosing the node N_p is expected to ensure that we hit a part of *Unex* with large chains.

To get the pivot, we can set the SAT to assign a random polarity to variables at the decision points during solving.

3.4 Online MUS/MSS Enumeration

The algorithm as presented until now is only able to provide MUSes and MSSes in the second phase, after it finishes exploring all the nodes. We now describe the last piece of our final algorithm, namely the way of producing MUSes and MSSes during the execution of the first phase. To do so, we need to employ two procedures: The *shrink* procedure is an arbitrary method that can turn an unsatisfiable node N_u into a MUS. Dually, the *grow* procedure is a method that can turn a satisfiable node N_s into MSS. A simple shrink (grow) method iteratively attempts to remove (add) constraints from N_u (N_s), checking each new set for satisfiability and keeping any changes that leave the set unsatisfiable (satisfiable). These simple variants serve just as illustrations, there are known efficient implementations of both shrink and grow for specific constraint domains; as an example see MUSer2 [4] which implements the shrink method for Boolean constraints systems.

Recall that as a result of processing a single chain K, our algorithm finds either a local MUS N_u, or a local MSS N_s, or both of them. To get a MUS (MSS) we propose to employ the shrink (grow) method on this local MUS (MSS). However, performing shrink (grow) on each local MUS (MSS) can be quite expensive and can significantly slow down our algorithm. The amount of time needed for performing one specific shrink (grow) of N_u (N_s) correlates with the position of N_u (N_s) on K; the closer N_u (N_s) is to the start (end) of K the bigger amount of time needed for the shrink (grow) can be expected.

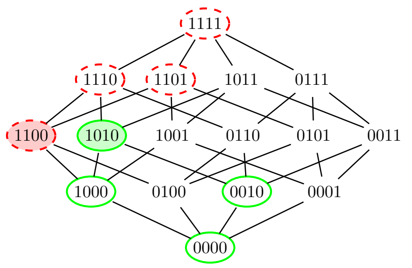
Therefore, we propose to shrink (grow) only some of the local MUSes (MSSes) based on their position on K. Let $|K|$ be the length of K, u the index of N_u in K, and $S : \mathbb{N} \rightarrow \mathbb{N}$ be an arbitrary user defined function. Our algorithm shrinks N_u into a MUS if and only if $u < S(|K|)$. As an example, consider $S(x) = \frac{x}{2}$; in such case N_u is shrunk only if it is contained in the first half of K. Similarly, let s be the index of local MSS N_s of chain K and $G : \mathbb{N} \rightarrow \mathbb{N}$. The local MSS N_s is grown only if $s > |K| - G(|K|)$, which for example for $G(x) = \frac{x}{2}$ means that N_s is grown only if it is contained in the second half of K. The complexity of performing shrinks also depends on the type of constrained system that is being processed, therefore the concrete choice of S and G is left as a parameter of our algorithm. Algorithm 3 shows an extended version of the method *processChain* which is able to produces MUSes and MSSes during its execution based on the above mechanism.

3.5 Example Execution of Our Algorithm

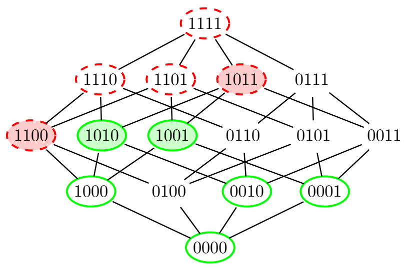
The following example explains the execution of our algorithm on a simple set of constraints. The example is illustrated in Fig. 1. Let $C = \{c_1 = a, c_2 = \neg a, c_3 = b, c_4 = \neg a \vee \neg b\}$, $S(x) = x$ and $G(x) = x$.

Initially $MSS_{can} = \emptyset$, $MUS_{can} = \emptyset$ and all nodes are unexplored, i.e. $f(Unex) = True$. Figure 1 shows the values of control variables in each iteration and also illustrates the current states of $\mathcal{P}(C)$. In order to save space we encode nodes as bitvectors, for example the node $\{c_1, c_3, c_4\}$ is written as 1011.

I. iteration
– Unex. couple $\langle 0000, 1111\rangle$
– Unex. chain $\langle 0000, 1000, 1100, 1110, 1111\rangle$
– Local MSS 1000 and local MUS 1100 are found and grown/shrunk to MSS 1010 and MUS 1100
– $MSS_{can} = \emptyset$ is updated to $\{1010\}$
– $MUS_{can} = \emptyset$ is updated to $\{1100\}$
– $f(Unex)$ is set to $(x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2)$



II. iteration
– Unexplored couple $\langle 0001, 1011\rangle$
– Unexplored chain $\langle 0001, 1001, 1011\rangle$
– Local MSS 1001 is grown to the MSS 1001
– local MUS 1011 is shrunk to the MUS 1011
– $MSS_{can} \leftarrow MSS_{can} \cup \{1001\}$
– $MUS_{can} \leftarrow MUS_{can} \cup \{1011\}$
– $f(Unex) \equiv (x_2 \vee x_4) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$



III. iteration
– Unexplored couple $\langle 0011, 0111\rangle$
– Unexplored chain $\langle 0011, 0111\rangle$
– Local MSS 0111 is grown to the MSS 0111
– local MUS *undefined*
– $MSS_{can} \leftarrow MSS_{can} \cup \{0111\}$
– $f(Unex) \equiv (x_2 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

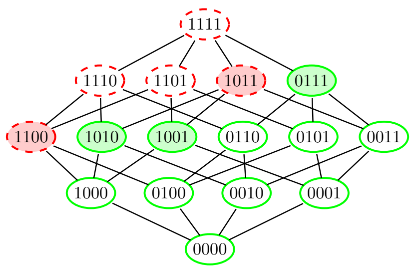


Figure 1 An example execution of our algorithm

After the last iteration of the first phase of our algorithm there is no model of $f(Unex)$ (this means that $Unex$ is empty), $MSS_{can} = \{1010, 1001, 0111\}$ and $MUS_{can} = \{1100, 1011\}$. Because functions S and G were stated in this example as $S(x) = x, G(x) = x$, each candidate on MUS or MSS has been already shrunk or grown to MUS or MSS, respectively, therefore $MSS(C) = MSS_{can}, MUS(C) = MUS_{can}$ and the second phase of our algorithm can be omitted.

Note that in the first iteration the node 1010 was found to be a MSS, which means that all its supersets are unsatisfiable. One could use this fact to mark all supersets of 1010 as explored, however our algorithm does not do this because some of these subsets can be MUSes (1011 in this example). If we were interested only in MSS enumeration we could mark all supersets of each MSS as explored; dually in the case of only MUS enumeration.

Table 1 The number of instances in which the algorithms output at least one MSS (the first number in each cell) or MUS (the second number).

	$G(x)$ \ $S(x)$	x	$0.8x$	$0.6x$	$0.4x$	$0.2x$	$0x$
Basic approach	x	56 | 56	151 | 40	150 | 33	144 | 12	149 | 16	151 | 0
	$0.8x$	56 | **60**	149 | 44	151 | 37	144 | 16	150 | 20	**152** | 0
	$0.6x$	56 | **60**	149 | 44	144 | 35	144 | 18	151 | 22	151 | 0
	$0.4x$	54 | **60**	149 | 45	140 | 36	143 | 32	150 | 30	151 | 0
	$0.2x$	53 | **60**	148 | 45	138 | 43	138 | 40	144 | 35	145 | 0
	$0x$	0 | **60**	0 | 47	0 | 46	0 | 44	0 | 37	0 | 0
Pivot based app.	x	56 | 56	151 | 40	151 | 32	151 | 14	151 | 12	144 | 0
	$0.8x$	56 | 60	151 | 43	151 | 36	150 | 18	149 | 16	145 | 0
	$0.6x$	56 | 60	151 | 43	151 | 35	151 | 18	**152** | 16	144 | 0
	$0.4x$	54 | 60	150 | 43	147 | 35	151 | 14	150 | 13	144 | 0
	$0.2x$	51 | 60	146 | 45	145 | 31	148 | 12	148 | 12	143 | 0
	0	0 | **61**	0 | 33	0 | 22	0 | 11	0 | 9	0 | 0
	MARCO	**51** | **51**					

4 Experimental Results

We now demonstrate the performance of several variants of our algorithm on a variety of Boolean CNF benchmarks. In particular, we implemented in C++ both the Basic and the Pivot Based approach for constructing chains and we evaluated both these approaches using several variants of the functions S and G. We also give a comparison with the MARCO algorithm [15].

The MARCO algorithm was presented by its authors in two variants, the basic variant and the optimised variant which is tailored for MUS enumeration. Both variants are iterative. The basic variant finds in each iteration an unexplored node, checks its satisfiability and based on the result the node is either shrunk into a MUS or grown into an MSS. Subsequently, MARCO uses the monotonicity of $\mathcal{P}(C)$ to deduce satisfiability of other nodes in the same way our algorithm does. The optimised variant differs from the basic variant in the selection of the unexplored node; it always selects a maximal unexplored node. If the node is unsatisfiable it is shrunk into a MUS, otherwise it is guaranteed to be an MSS. We used the optimised variant in our experiments. The pseudocodes of both variants can be found in [15]. The key difference between our algorithm and MARCO is the usage of local MUSes and MSSes which are much easier to find and can be used to prune the powerset in the same way as global MUSes/MSSes.

Note that both compared algorithms (MARCO and our algorithm) employ several external tools during their execution, namely a SAT solver for finding the unexplored nodes, a constraint solver to decide the satisfiability of constraint sets, and the two procedures *shrink* and *grow* mentioned above. The list of external tools coincides for both algorithms. Therefore, we reimplemented MARCO in C++ to ensure that the two algorithms use the same implementations of the shrink and grow methods and the same solvers. As both the SAT solver and constraint solver we used the miniSAT tool [9] and we used the simple implementation of the shrink and grow methods as described earlier. Note that there are some efficient implementations of the shrink and grow methods for Boolean constraints, however, in general there might be no effective implementation of these methods. That is why we used the simple implementations.

As experimental data we used a collection of 294 unsatisfiable Boolean CNF Benchmarks that were taken from the MUS track of the 2011 SAT competition [1]. The benchmarks

Table 2 The 5% trimmed sum of outputted MSSes and MUSes (summed over all 294 instances). The first number in each cell is the number of outputted MSSes, the second is the number of outputted MUSes.

	$G(x)$ \ $S(x)$	x	$0.8x$	$0.6x$	$0.4x$	$0.2x$	$0x$
Basic approach	x	1744 | 339	9798 | 212	9936 | 87	6942 | 0	9726 | 2	10216 | 0
	$0.8x$	1741 | 344	9908 | 217	9756 | 94	6787 | 2	9684 | 6	9378 | 0
	$0.6x$	1740 | 348	9859 | 224	6969 | 40	6999 | 4	9696 | 8	9436 | 0
	$0.4x$	1877 | 436	10013 | 252	7218 | 67	7694 | 50	10420 | 39	10114 | 0
	$0.2x$	1757 | **635**	10161 | 527	7925 | 262	8196 | 101	**10853** | 66	10111 | 0
	0	0 | 632	0 | 554	0 | 356	0 | 107	0 | 68	0 | 0
Pivot based app.	x	2535 | 349	8330 | 208	7775 | 71	6705 | 0	6725 | 0	5089 | 0
	$0.8x$	2660 | 492	8336 | 255	7680 | 85	6961 | 4	6889 | 2	5061 | 0
	$0.6x$	2771 | 567	8481 | 290	7779 | 92	7066 | 4	6830 | 2	5067 | 0
	$0.4x$	2814 | 597	8418 | 388	7975 | 145	6814 | 0	6950 | 0	5302 | 0
	$0.2x$	2763 | 837	**8633** | 697	7220 | 41	6563 | 0	6409 | 0	4910 | 0
	0	0 | **839**	0 | 404	0 | 10	0 | 0	0 | 0	0 | 0
	MARCO	**749** | **215**					

range in their size from 70 to 16 million constraints and from 26 to 4.4 million variables and were drawn from a variety of domains and applications. All experiments were run with a time limit of 60 seconds.

Due to the potentially exponentially many MUSes and/or MSSes in each instance, the complete MUS and MSS enumeration is generally intractable. Moreover, even outputting a single MUS/MSS can be intractable for larger instances as it naturally includes solving the satisfiability problem, which is for Boolean instances NP-complete. Table 1 shows in how many instances the variants of our algorithm were able to output at least one MUS or MSS. MARCO was able to output at least one MUS and one MSS in 51 instances whereas several variants of our algorithm were able to output some MSSes in about 150 instances and some MUSes in up to 60 instances. Some of the 296 instances are just intractable for the solver which is not able to perform even a single consistency check within the used time limit. The other significant factor that affected the results is the complexity of the shrink method. MARCO in every iteration either "hits" a satisfiable node and directly outputs it as an MSS or waits till the shrink method shrinks the unsatisfiable node into a MUS. Therefore, each call of the shrink method can suspend the execution for a nontrivial time.

One can see that our algorithm also suffers from the possibly very expensive shrink calls and performs very poorly when the S function is set to $S(x) = x$. On the other hand, the variants that perform only the "easier" shrinks by setting S to be $S(x) < x$ achieved better results. The grow method is generally cheaper to perform than the shrink method as checking whether an addition of a constraint to a satisfiable set of constraints makes this set unsatisfiable is usually cheaper than the dual task. No significant difference between the Basic and the Pivot based approach was captured in this comparison.

Another comparison can be found in Table 2 that shows the 5% trimmed sums of outputted MSSes and MUSes (summed over all of the 294 instances), i.e. 5% of the instances with the least outputted MSSes (MSSes) and 5% of the instances with the most outputted MSSes (MSSes) were discarded. The trimmed sum is based on a trimmed median which is useful estimator in statistics because it discards the most extreme observations.

All variants of our algorithm were noticeably better in MSS enumeration than MARCO. In the case of MUS enumeration MARCO outperformed these variants of our algorithm that shrink only some of the local MUSes, i.e. variants where $S(x) = 0.6x$ and $S(x) = 0.4x$.

Table 3 The results of the experiments with a time limit of 1800 seconds.

	MSS enumeration		MUS enumeration	
	at least one MSS	5% trimmed sum	at least one MUS	5% trimmed sum
MARCO	112	50855	112	7337
BA $S(x) = 0.2x$, $G(x) = 0.2x$	167	80921	52	159
BA $S(x) = x$, $G(x) = 0.2x$	106	61010	**114**	**19059**
PBA $S(x) = 0.8x$, $G(x) = 0.2x$	**170**	**118151**	76	14565
PBA $S(x) = x$, $G(x) = 0.2x$	104	61537	112	19030

However, the variants with $S(x) = x$ and $S(x) = 0.8x$ performed better, especially the variant with $G(x) = 0.2x$, $S(x) = x$ outputted about three times more MUSes than MARCO. As the Pivot based approach is randomised its performance may vary if it is run repeatedly on the same instances; the result of a single run may be misleading. Therefore, we ran all tests of the Pivot based approach repeatedly and the tables show the average values.

The time limit of 60 seconds is quite short and the results of such experiments may be misleading. Therefore, we also evaluated MARCO and both the Basic approach (BA) and the Pivot based approach (PBA) on the same set of benchmarks with a time limit of 1800 seconds. The results of these experiments are shown in Table 3. We used two different settings for BA and two different settings for PBA which were chosen based on the results of the experiments with the time limit of 60 seconds. MARCO was able to output at least one MSS in 112 instances whereas PBA with $S(x) = 0.8x$ and $G(x) = 0.2x$ was able to output at least one MSS in 170 instances. Also, the 5% trimmed sum of outputted MSSes by PBA is more than 2 times higher the 5% trimmed sum of outputted MSSes by MARCO.

In the case of MUS enumeration the number of instances in which MARCO was able to output at least one MUS is almost the same as the number achieved by BA and PBA with $S(x) = x, G(x) = 0.2x$. However, the 5% trimmed sum of outputted MUSes by MARCO is significantly lower. We believe that this is caused by the relative complexity of performing shrinks. Our algorithm performs easier shrinks because it shrinks local MUSes which are usually “closer” to (global) MUSes whereas MARCO shrinks random nodes. Therefore, MARCO may be able to perform some shrinks within the given time limit but it is able to perform significantly fewer shrinks than our algorithm.

5 Conclusion

In this paper, we have presented a novel algorithm for online enumeration of MUSes and MSSes which is applicable to any type of constraint system. The core of the algorithm is based on a novel approach utilising the so-called local MUSes/MSSes found using binary search. This approach allows the algorithm to efficiently explore the space of all subsets of a given set of constraints. We have made an experimental comparison with MARCO, the state-of-the-art algorithm for online MUS and MSS enumeration. The results show that our algorithm outperforms MARCO. Our algorithm can be built on top of any consistency solver and can employ any implementation of the *shrink* and *grow* methods, therefore any future advance in this areas can be reflected in the performance of our algorithm.

One direction of future research is to aim at parallel processing of the search space in order to improve the performance of our approach; there are usually many disjoint unexplored chains that can be processed concurrently. Another possible direction is to focus on some specific types of constraint systems and customise our algorithm to be more efficient for these systems.

References

1 MUS track of the 2011 SAT competition. `http://www.cril.univ-artois.fr/SAT11/`.

2 Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 343–352. Springer, 2008.

3 James Bailey and Peter J Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Practical Aspects of Declarative Languages*, pages 174–186. Springer, 2005.

4 Anton Belov and Joao Marques-Silva. MUSer2: An efficient MUS extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:123–128, 2012.

5 Jaroslav Bendík, Nikola Beneš, Jiří Barnat, and Ivana Černá. Finding boundary elements in ordered sets with application to safety and requirements analysis. In *SEFM*, volume 9763 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2016.

6 Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1):25–46, 2003.

7 Renato Bruni and Antonio Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. *Electronic Notes in Discrete Mathematics*, 9:162–173, 2001.

8 Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 32–43. ACM, 2003.

9 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

10 Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.

11 Aimin Hou. A theory of measurement in diagnosis from first principles. *Artif. Intell.*, 65(2):281–328, 1994.

12 Ulrich Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

13 Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple muses quickly. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2013.

14 Mark H. Liffiton, Michael D. Moffitt, Martha E. Pollack, and Karem A. Sakallah. Identifying conflicts in overconstrained temporal problems. In *IJCAI*, pages 205–211. Professional Book Center, 2005.

15 Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, pages 1–28, 2015.

16 Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

17 Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. In *DAC*, pages 518–523. ACM, 2004.

18 Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013.

19 Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. *SAT*, 3, 2003.