Masaryk University, Faculty of Informatics

Algorithms for Finding Maximal Satisfiable Sets of Constraints

Master's thesis

Jaroslav Bendík

Brno, 2016

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

Jaroslav Bendík

Advisor: prof. RNDr. Ivana Černá, CSc.

Abstract

The motivation for this diploma thesis comes from various sources such as parametric formal verification, requirements engineering and safety analysis. In these areas, there are often situations in which we are given a set of configurations and a property of interest with the goal of computing all the configurations for which the property is valid. Checking the validity of each single configuration may be a costly process. We are thus interested in reducing the number of such validity queries. In this work, we assume that the configuration space is equipped with a partial ordering that is preserved by the property to be checked. In such a case, the set of all valid configurations can be effectively represented by the set of all maximal valid (or minimal invalid) configurations w.r.t. the ordering. We show an algorithm to compute such boundary elements. We explain how this general setting applies to consistency and redundancy checking of requirements and to finding minimal cut-sets for safety analysis.

We also itroduce an algorithm tailored for more specific case where the configuration space has a particular shape of hypercube. This situation, for example, arises in the area of constraints processing where the configuration space corresponds to the set of all subsets of a given set of constraints and the property of interest is satisfiability of these subsets. If an unsatisfiable set of constraints is given, one may ask for a minimal description of the reason for this unsatisfiability, i.e. for the minimal unsatisfiable subsets (MUSes) and/or maximal satisfiable subsets (MSSes).

Keywords

Minimal unsatisfiable subsets, Maximal satisfiable subsets, Unsatisfiability analysis, Infeasibility analysis, Requirements analysis, Safety analysis, Formal verification

Acknowledgements

My sincere thanks goes primarily to my thesis advisor Ivana Černá for all the motivation she gave me and for all the time she devoted to me during my work on this thesis. I would also like to thank Nikola Beneš and Jiří Barnat for our numerous discussions on the topic. Finally, I am grateful to all members of the ParaDiSe laboratory, both present and past, for creating very friendly and inventive atmosphere.

Contents

1 Introduction

The motivation for this work comes from various source areas, such as parametric formal verification, requirements engineering, safety analysis, or software product lines. In these areas, the following situation often arises: We are given, as an input, a set of configurations and a property of interest. The goal is to compute the set of all the configurations that satisfy the given property. We call such configurations valid. As a short example, one may imagine a system with tunable parameters that is to be verified for correctness. The set of configurations, in that case, is a set of all possible parameter values and the goal is to find all such values that ensure the correctness of the given system. If we are given a method to ascertain the validity of a single configuration, we could try running the method repeatedly for each configuration to obtain the desired result. In the case of an infinite set of configurations, this approach does not terminate, and we get at most a partial answer. However, even if the configuration space is finite, checking configurations one by one may be too costly. We are thus interested in reducing the number of validity checks in the finite case.

Although such reduction might be impossible in general, we focus on problems whose configuration space is equipped with a certain structure that is preserved by the property of interest. This may then be exploited in order to check a smaller number of configurations and still obtain the full answer. The desired structure is a set of dependencies of the form: "If configuration A violates the property then configuration B does too." Mathematically, we can either view such structure as a directed acyclic graph of those dependencies, or as a partial ordering on the set of all configurations induced by this graph. Viewed as an ordered set, the set of all the valid configurations can be effectively represented by the set of all the maximal valid (alternatively, minimal invalid) configurations.

We are interested in finding this boundary between valid and invalid configurations while minimising the number of validity queries, i.e. the potentially costly checks whether a given configuration satisfies the property.

We are not aware of any previous work which deals with exactly the same problem as we do. The most related problems can be found among the Constraint Satisfaction Problems (CSPs) where a satisfiability of a set of constraints is examined. When a set of constraints C is infeasible (unsatisfiable) the most common analysis is the maximum satisfiability problem (MaxSAT, MaxCSP), which asks for a satisfiable subset of C with the greatest possible cardinality. Our problem is different from MaxSAT and more related to the maximal satisfiable subset problem (MSS) that considers maximality in the ordering sense instead of maximum cardinality. The goal of MSS is to find a subset of C that is satisfiable, and that becomes unsatisfiable if any other constraint is added to this subset. Similarly, one can define the minimal unsatisfiable subset problem (MUS).

Both MSSes and MUSes describe the boundary between the satisfiable and unsatisfiable subsets of C and both these problems were recently addressed in works

[2,5,7,25,26]. To solve the problem, the papers use different approaches like the duality that exists between MUSes and MSSes [2,26] or parallel enumeration from bottom and top [5]. In [25] authors unify and expand upon the earlier work, presenting a detailed explanation of the algorithm's operation in a framework that also enables clear comparisons. Paper [7] describes an MUS extractor tool MUSer2 which implements a number of MUS extraction algorithms.

Subsets of a set of constraints are naturally ordered by the subset relation, thus our approach can be also used to solve these problems. We deal with a more general problem as we consider arbitrary directed acyclic graphs instead of the hypercube graphs representing subsets of constraints. Our approach is thus more general and has a wider area of potential usage. Futhermore, we show that our approach can be competitive with the state-of-the-art tool MARCO in the case of analysing hypercube graphs.

On the other hand, the generality of our approach disallows it to fully exploit the specific properties of hypercube graphs. Therefore, we also present a modification of the general approach which is designed specially for enumeration of MUSes and MSSes of infeasible set of constraints. 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.

1.1 Outline of The Thesis

The thesis is divided into two parts. The first part is designated to the problem of finding maximal valid (minimal invalid) configurations in systems that can be modeled by arbitrary directed acyclic graphs. First, we motivate the need for solving this problem on two particular examples from the areas of safety and requirements analysis. Subsequently, the problem is formally defined and our approach for solving this problem is presented. The first part is concluded with results of an experimental evaluation of our approach.

In the second part we focus on the problem of online enumeration of MUSes and MSSes of infeasible set of constraints, i.e. on the hypercube graphs. We state the basic definitions and explain the terminology that is used in the area of constraints processing. We give a list of existing approaches for solving the problem of MUS/MSS enumeration and pinpoint their strengths and weaknesses. Subsequently we gradually present our approach for solving the MUS/MSSes enumeration problem. Finally, an experimental evaluation of our approach and comparison with other state-of-the art algorithm is given.

2 General Approach

2.1 Introduction

In this part of the diploma thesis we describe our approach for finding maximal valid and minimal invalid configurations of configuration spaces that are described by arbitrary directed acyclic graphs. Let us, before we present the basic definitions and preliminaries and state our problem formally, motivate the need for solving this problem on two examples from areas of safety and requirements analysis.

Safety Analysis. The safety analysis techniques are widely used during the design phase of safety-critical systems. Their aim is to assure that the systems provide prescribed levels of safety via exploring the dependencies between a system-level failure and the failures of individual components. Traditionally, the various safety analyses are done manually and are based on an informal model of the systems. This leads to the process being very time-consuming and the results being highly subjective. The desire to alleviate such issues somewhat and to make the process more automated led to the development of Model-Based Safety Analysis (MBSA) approach [21]. This approach assumes the existence of a system model that is extended by an error model describing the way faults may happen and propagate throughout the system. One of the problems solved in MBSA is the computation of the so-called minimal cut-sets for a given failure, i.e. the minimal sets of low-level faults that cause the high-level failure to manifest in the system.

One can map the minimal cut-sets problem to our setting easily. The configurations are the possible sets of faults that may be enabled in the extended system model, their ordering is given by set inclusion. Note that there might be dependencies between some of the faults, which means that not all sets of faults are considered to be possible. The property of interest is the non-existence of failure and the valid configurations are exactly those sets of faults that do not cause the failure to happen. Clearly, in this case, the minimal cut-sets correspond exactly to the minimal invalid configurations. This means that the problem can be solved using our approach.

To illustrate the application on a simple example, we consider an avionics triplex sensor voter, described in [12]. The voter gains measurement data from three sensors as well as information whether the sensors are operational. It computes the differences between the sensor data and detects persistent miscompare, i.e. situations where two sensors differ above a certain threshold for a certain amount of time. If all three sensors are operational and two pairs of sensors have persistent miscompare, the common sensor is marked as invalid and data is no longer received from that sensor. If just two sensors are operational, a persistent miscompare between the two means that the output data is considered invalid.

For simplicity, let us assume that there are two kinds of faults per sensor and let us call these fault A and fault B. Fault A causes the sensor to transmit wrong data while fault B causes the sensor to stop working completely. Note that we may assume that both faults cannot occur on the same sensor, as once fault B happens,

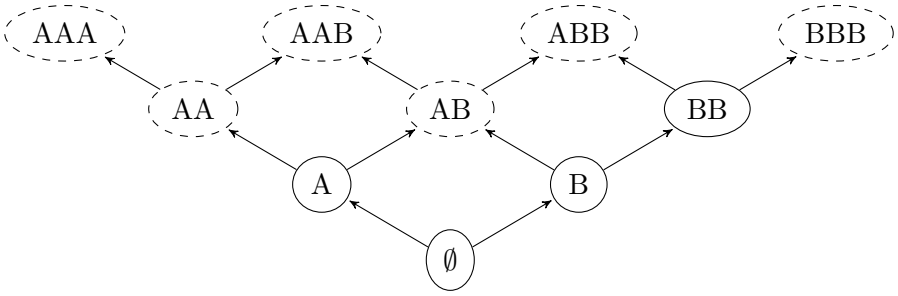


Figure 1: Illustration of the safety analysis example

the occurrence of fault A is irrelevant. In general, we thus have six possible faults and 27 sets of faults to be checked, including the empty set of faults. However, as the situation of sensors is symmetrical, we may get rid of this symmetry and simply count the number of fault-A sensors and fault-B sensors instead. This situation is illustrated in Fig. 1. The nodes in the graph represent the various fault configurations: $\emptyset$ represents that no faults occur, AB represents that fault A occurred on one sensor and fault B occurred on another sensor, etc. The graph is created from the inclusion ordering on the fault situations.

Let us now consider the failure to deliver data to the output. As explained above, the voter fails to deliver output if either all sensors stopped working or have been eliminated, or if there are just two sensors working with persistent miscompare. We assume that the persistent miscompare situation is detected once at least one of a pair of sensors starts transmitting wrong data, i.e. fault A occurred on that sensor. For this reason, the minimal invalid configurations (i.e. the minimal cut-set) are AA, AB, and BBB, while the maximal valid configurations are A and BB.

Requirements Analysis. Establishing the requirements is an important stage in all development. Although traditionally, software requirements were given informally, recently there has been a growing interest in formalising these requirements [19]. Formal description in a kind of mathematical logic enables various model-based techniques, such as formal verification. Moreover, we also get the opportunity to check the requirements earlier, even before any system model is built. This so-called requirements sanity checking [5] aims to assure that a given set of requirements is consistent and that there are no redundancies. If inconsistencies or redundancies are found, it is usually desirable to present them to the user in a minimal fashion, exposing the core problems in the requirements. As redundancy checking can be usually reduced to inconsistency checking [4], the goal is thus to find all minimal inconsistent subsets of requirements. Such a problem may be clearly seen as an instance of our setting, where the configurations are sets of requirements and the ordering is given by the subset relation.

We illustrate the inconsistency checking on an example. Assume that we are given a set of four requirements. These requirements consider one particular component in

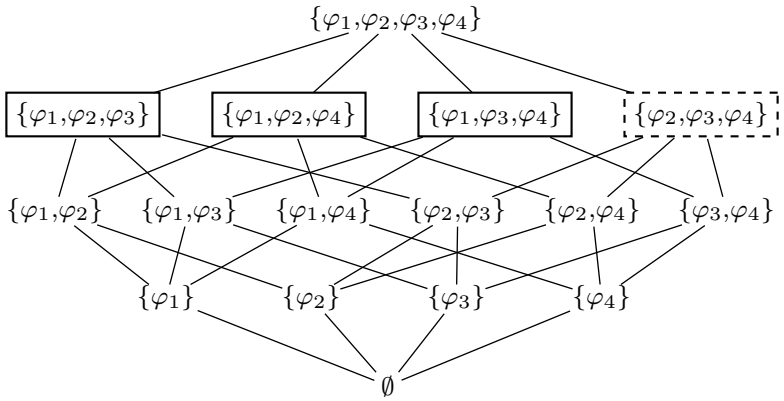


Figure 2: Illustration of the requirements analysis example. The subset with dashed outline is the maximal inconsistent one, the subsets with solid outline are the maximal consistent ones.

a system and constrain the way the component is used. We formalise the requirements using the branching temporal logic CTL [11]. In the formulae we use the atomic propositions q denoting that a *query* has arrived, r denoting that the component is *running*, and m denoting that the system is taken down for *maintenance*. Our first requirement states that whenever a query arrives, the component has to become active eventually, formally $\varphi_1 := \mathbf{AG}(q \rightarrow \mathbf{AF}\, r)$. The second requirement states that once the component is started, it may never be stopped. This may be a reasonable requirement e.g. if the component's initialisation is expensive, formally $\varphi_2 := \mathbf{AG}(r \rightarrow \mathbf{AG}\, r)$. The third requirement states that the system has to be taken down for maintenance once in a while. This also means that the component has to become inactive at that time. This is formalised as $\varphi_3 := \mathbf{AG}\,\mathbf{AF}\,(m \wedge \neg r)$. Our last requirement states that after the maintenance, the system (including the component we are interested in) has to be restarted, formally $\varphi_4 := \mathbf{AG}(m \rightarrow \mathbf{AF}\,(\neg m \wedge r))$. The situation is illustrated in Fig. 2. We discover that there is one minimal inconsistent subset of the four requirements, namely $\{\varphi_2, \varphi_3, \varphi_4\}$, and that there are three maximal consistent subsets of the requirements, namely $\{\varphi_1, \varphi_2, \varphi_3\}$, $\{\varphi_1, \varphi_2, \varphi_4\}$, $\{\varphi_1, \varphi_3, \varphi_4\}$. The consistency of the first set $\{\varphi_1, \varphi_2, \varphi_3\}$ might be surprising, as one would suspect the pair of requirements φ_2 and φ_3 to be the source of inconsistency. However, the first three requirements can hold at the same time – in systems where no queries arrive at all. In these situations we say that the requirements hold *vacuously*. There are ways of dealing with vacuity, such as employing the so-called vacuity witnesses [6].

Note that although in this example, the space of all sets of requirements had the particular shape of a hypercube, this might not always be the case. We might sometimes be interested in certain subsets of requirements instead of all of them. Such a situation may arise e.g. if there are some known implications between the requirements. Consider the example above with the added requirement that once

the component is started, it may only stop after 1 hour. This requirement is clearly implied by φ_2 and we would therefore omit all subsets that contain both φ_2 and this new requirement. Another way of obtaining a non-hypercube requirements graph is when considering requirements for several components at once in a component-based or software product line setting. In such cases, some of the components or product features may be incompatible and it thus only makes sense to consider subsets of requirements that reason about compatible components.

Outline of this part. The rest of this part of the thesis is organised as follows. In Section 2.2 we present the basic definitions and preliminaries and state our problem formally. In Section 2.3 we present our new algorithm to solve the problem and discuss several variants and heuristics of it, as well as we analyse its complexity. The algorithm is then evaluated on a set of experiments in Section 2.4.

2.2 Preliminaries and Problem Statement

In this section, we recall some basic notions that we use later in this part of the thesis. We also introduce the formalism of annotated directed acyclic graphs that forms the basic setting for our problem.

Definition 1. *(Directed acyclic graph) A* directed graph *G is a pair (V, E), where V is a finite set of vertices and $E \subseteq V \times V$ is a set of edges. An edge (u, v) is an* outgoing *edge of the vertex u and an* incoming *edge of the vertex v. The* indegree *(*outdegree*) of a vertex v is the number of incoming (outgoing) edges of v. A* path *from a vertex u to a vertex v in G is a sequence $\langle v_0, v_1, \cdots v_k \rangle$ of vertices such that $v_0 = u$, $v_k = v$, $k > 0$ and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \cdots, k-1$. We say that v is reachable from u if there is a path in G from u to v.*

A directed graph $G = (V, E)$ is called a directed acyclic graph (DAG) *if there is no path $\langle v_0, v_1, \cdots v_k \rangle$ in the graph such that $v_0 = v_k$. A DAG induces a strict partial order relation $\sqsubset_G$ on its vertices as follows: $u \sqsubset_G v$ if v is reachable from u. A vertex v is said to be a* minimal vertex *in G if there is no u such that $u \sqsubset_G v$. Dually, a vertex u is a* maximal vertex *in G if there is no v such that $u \sqsubset_G v$.*

Definition 2. *(Chain cover) A* chain *in a DAG G with its induced relation $\sqsubset_G$ is a sequence of one or more vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that $v_0 \sqsubset_G v_1 \sqsubset_G \cdots \sqsubset_G v_k$. A* chain cover *of a DAG is a set of chains $C = \{c_1, \cdots, c_l\}$ such that each vertex is included in exactly one chain from C. A* minimum chain cover *is a chain cover containing the fewest possible number of chains. Note that the minimum chain cover is not given uniquely.*

Definition 3. *(Annotated DAG) An* annotated directed acyclic graph (ADAG) *is a pair $(G, valid)$, where $G = (V, E)$ is a directed acyclic graph and $valid : V \to Bool$ is a validation function. The validation function is* monotone *on V, which means that for every pair $u, v \in V$ if $u \sqsubset_G v$ and $valid(u) = false$ then $valid(v) = false$.*

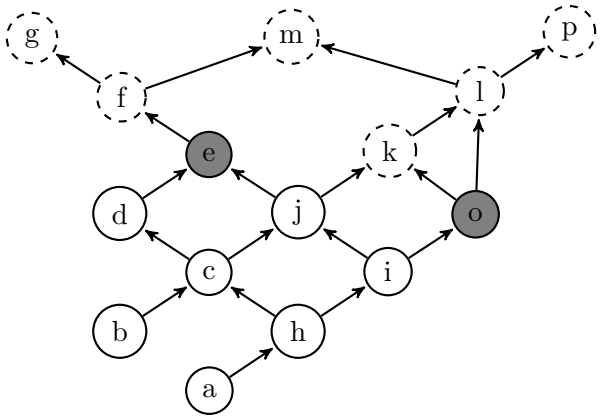


Figure 3: An example of an ADAG, the dashed vertices are the invalid ones, the grey vertices are the maximal valid ones.

The problem we are interested in can be stated as finding either a set of maximal valid vertices or a set of minimal invalid vertices. We present an algorithm to obtain the former. However, the algorithm can be also used to obtain the latter, as the two formulations are dual.

Definition 4. *(Maximal valid vertex and cut) Let $\mathcal{G} = ((V, E), valid)$ be an ADAG. A vertex $u \in V$ is a* maximal valid vertex *of G iff $valid(u) = true$ and $\forall v \in V$ such that $u \sqsubset_G v$ is $valid(v) = false$.*

The maximal valid cut *of $\mathcal{G}$ is a set of all its maximal valid vertices.*

Problem Formulation. Given an ADAG $\mathcal{G} = ((V, E), valid)$, find the *maximal valid cut* of $\mathcal{G}$.

As mentioned in the introduction, evaluating the function *valid* on a single configuration (a single vertex of the ADAG) might be an expensive operation. Therefore, our aim is to propose an algorithm minimising the number of evaluations of the *valid* function even for the price of the increased complexity of the algorithm with respect to the number of operations over the graph.

The problem formulation assumes that the graph is acyclic and that the validation function is monotone. We might, however, be also interested in cases where one of these preconditions is violated. We postpone the discussion of these possibilities to Section 2.3.5.

2.3 Algorithm

A naive solution of the maximal valid cut problem for a given ADAG G would be to evaluate the *valid* function on each vertex, compute the $\sqsubset_G$ relation for valid vertices, and choose the maximal ones. In this naive approach, the *valid* function is called once per each vertex.

2.3.1 Chain-Based Algorithm

Instead of dealing with each vertex of G separately we build our solution on a decomposition of G into a set of chains and we use the fact that the validation function is monotone. The algorithm takes as an input an ADAG G and one of its chain covers C. Then it iteratively handles chains and removes those vertices which cannot be the maximal valid ones.

From the definition, each vertex of the maximal valid cut of G belongs to exactly one chain from C. Moreover, every chain contains at most one maximal valid vertex of the graph and this vertex is at the same time the maximal valid vertex of the chain. Let us note that the opposite implication does not hold generally, the maximal valid vertex of a chain may not be a maximal valid vertex of the whole graph. Therefore, the set of maximal valid vertices of individual chains contains the maximal valid cut as its subset.

Let $c = \langle v_0, v_1, \cdots, v_l \rangle$ be an arbitrary chain of C. To find the maximal valid vertex v_h of this chain we use binary search. We take the *middle* vertex c_{mid} of c, $c_{mid} = v_{\lceil \frac{l}{2} \rceil}$ and evaluate the *valid* function on c_{mid}. If c_{mid} is valid, then we know for sure that none of the lower vertices from c can be the maximal valid vertex of this chain. In the other case, we claim that none of the higher vertices from c can be maximal valid vertex. This allows us to reduce c into half and recursively repeat the procedure. We finish with a chain consisting of only one vertex v_i. If v_i is a valid vertex then it is the maximal valid vertex of c, otherwise c does not have any valid vertex at all.

Once we have applied the binary search on each chain from C, we have the set H of maximal vertices of these chains. To obtain the maximal valid cut of G from H we just compute the $\sqsubset_G$ relation for each pair from H and remove from H all those vertices that are not maximal w.r.t. $\sqsubset_G$.

For an illustration of the chain based algorithm, assume that we are given the graph from Fig. 3 and as a chain cover we take these chains: $\langle b, c, d, e, f, g \rangle$, $\langle a, h, j, k, m \rangle$, $\langle i, o, l, p \rangle$. The vertices e, j, o are found to be the maximal valid vertices of these chains and the $\sqsubset_G$ relation is computed for these three vertices. Vertex j is found to be lower than e and vertices o, e are mutually unreachable, hence $\{e, o\}$ is the resulting maximal valid cut.

The number of calls to *valid* in this algorithm depends on the number of chains in C and the number of calls used in the binary searches. The number of calls is logarithmic in the length of the chain in every binary search. Therefore, the total number of calls is $\mathcal{O}(|C| \log L)$ where $|C|$ is the number of chains in C and L is the length of the longest chain in C.

Note that there are algorithms such as [10, 16] that compute the minimum chain cover of a given graph. We may thus make use of these algorithms to reduce the number of chains that need to be processed by this algorithm.

2.3.2 Cutoff-Based Algorithm

We now improve the efficiency of our algorithm by decreasing the chain lengths and possibly eliminating some of the chains completely. The main idea makes use of the fact that a vertex v_i is recognised as the maximal valid vertex of a chain $c = \langle v_0, v_1, \cdots, v_i, \cdots, v_l \rangle$ (if c has any). From this we can deduce that not only vertices from c lower than v_i cannot belong to the maximal valid cut, but neither do any vertices from G lower than v_i. Symmetrically, none of the vertices from G higher than v_{i+1} can belong to the maximal valid cut. Therefore, we can remove all vertices lower than v_i and higher than v_{i+1}, including v_{i+1}, from all chains and thus reduce their size and possibly the number of *valid* calls in the future.

Definition 5. *(Cutoff Transformation) Let G be an ADAG and C its chain cover. Let $c = \langle v_0, v_1, \cdots, v_i, \cdots, v_l \rangle$ be a chain from C and let v_i be its maximal valid vertex. Then the* cutoff *of G is a pair $\overline{G}$ and $\overline{C}$ generated from G and C, respectively, by removing:*

- *vertices which are lower than v_i,*
- *vertices which are higher than v_{i+1}, and*
- *the vertex v_{i+1}.*

In case that c does not have a maximal valid vertex we define the cutoff *of G to be a tuple $\overline{G}$ and $\overline{C}$ created from G and C, respectively, by removing:*

- *vertices which are higher than v_0 and*
- *the vertex v_0.*

As this vertex removal may make some chains empty, we also remove the empty chains from $\overline{C}$.

Theorem 1. *(Cutoff Property) Let G be an ADAG, C its chain cover, and $\overline{G}, \overline{C}$ be their cutoff. Then graphs G and $\overline{G}$ have the same maximal valid cuts, $\overline{C}$ is a chain cover of $\overline{G}$, and $|C| \geq |\overline{C}|$.*

Proof. None of the maximal valid vertices of G is removed from G during the cutoff transformation. On the other hand the set of edges of $\overline{G}$ is the subset of those of G and thus $\overline{G}$ cannot have any new maximal valid vertex.

As C is a chain cover of G we have that each vertex of $\overline{G}$ is in $\overline{C}$. Each sequence $\overline{c}$ from $\overline{C}$ emerges from a chain c from C by removing vertices. Due to transitivity of $\sqsubset_G$ only a prefix or postfix of c is removed and thus $\overline{c}$ is a chain again. □

Theorem 2. *(Maximal Cut Property) Let G be an ADAG and C its chain cover. Let us apply step by step the cutoff transformation on all chains from C and let $\overline{G}$ and $\overline{C}$ be the resulting graph and its chain cover respectively. Then every chain in $\overline{C}$ is just a single vertex and $\overline{C}$ is exactly the maximal valid cut of G.*

Proof. After the cutoff transformation of a chain $c \in C$, the chain has either one element or is empty.

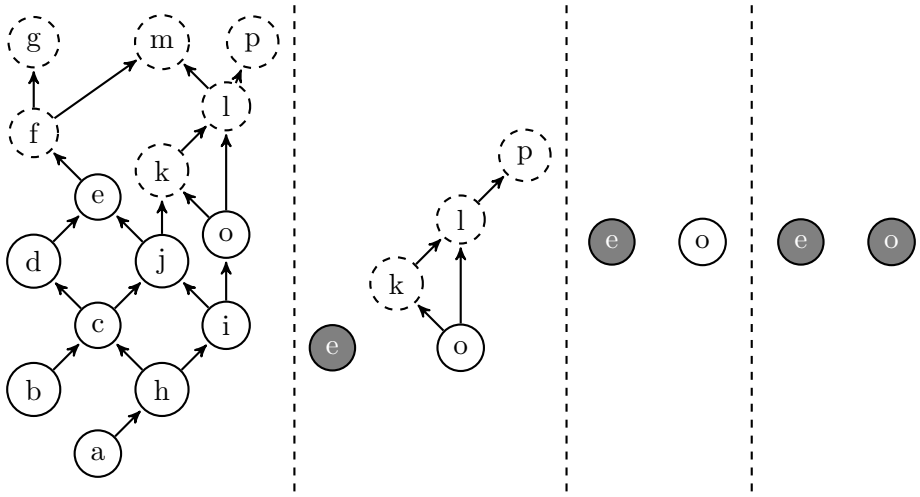


Figure 4: Illustration of the cutoff based algorithm. The graph is covered with three chains $\langle b, c, d, e, f, g\rangle$, $\langle a, h, j, k, m\rangle$, $\langle i, o, l, p\rangle$ and they are processed in this order. At first, the vertex e is found to be the maximal valid vertex of the first chain and the consequently made cutoff reduces the set of chains to $\langle e\rangle$, $\langle k\rangle$, $\langle o, l, p\rangle$. In the next step, the chain $\langle k\rangle$ is processed and no valid vertex on this chain is found, but the cutoff is made and the set of chains is reduced to $\langle e\rangle$, $\langle o\rangle$. In the last step, the chain $\langle o\rangle$ is processed and o is found to be valid. The result of the third cutoff is the maximal valid cut $\{e, o\}$. The grey nodes are nodes which have already been determined to be valid ones.

Theorem 1 implies that $\overline{G}$ contains all vertices from the maximal valid cut of G and so does its chain cover $\overline{C}$. On the other hand any two vertices from $\overline{C}$ are incomparable w.r.t. $\sqsubset_G$ due to the cutoff transformation. □

The algorithm based on the cutoff transformation is shown as Algorithm 1. The algorithm assumes that the reachability relation $\sqsubset_G$ is pre-computed. The relation is used both for computing the minimum chain cover and when detecting lower and higher vertices, however, bread-first-search can be also used for this detection. Instead of removing vertices from the graph we just mark them with a binary flag *cand* (for candidate) initially set to *true*. Once we have discovered that a vertex cannot be a maximal valid one, the flag is set to *false*.

Contrary to the previous algorithm based on chains, once the algorithm based on cutoffs processes the last chain from the chain cover of the original graph G, the set C contains exactly the maximal valid cut of G and no other computation is needed.

Figure 4 illustrates the cutoff based algorithm on the graph from Fig. 3. The cutoffs significantly reduce the space of vertices that can be maximal valid ones. After processing of the first two chains only two vertices are left as the possible maximal valid ones.

```
1  Function MaxValid(c, IsValid())
2      if c is empty then
3          return nil
4      middle ← ⌈l/2⌉;
5      if IsValid(v_middle) then
6          x ← MaxValid(⟨v_{middle+1}, ..., v_l⟩);
7          if x = nil then
8              return middle
9          else
10             return x
11     else
12         return MaxValid(⟨v_0, ..., v_{middle−1}⟩)
```

```
1  Function Cutoff(G, c, i)
2      if i ≠ nil then
3          set v.cand = false for each v ∈ V lower than v_i;
4          set v.cand = false for each v ∈ V higher than v_{i+1};
5          set v_{i+1}.cand = false;
6      else
7          set v.cand = false for each v ∈ V higher than v_0;
8          set v_0.cand = false;
```

```
1  Function ProcessChain(G, IsValid(), c)
2      remove from c all vertices v with v.cand = false;
3      index ← MaxValid(c, IsValid());
4      Cutoff(G, c, index);
```

```
1  Function MaxValidCut(G = (V, E), IsValid())
       input  : graph G and a validation function IsValid()
       output : set of MSSes of G
2      set v.cand = true for each v ∈ V;
3      compute the relation ⊏_G;
4      ChainCover ← MinimumChainCover(G);
5      for each chain ∈ ChainCover do
6          ProcessChain(G, IsValid(), chain)
7      return {v ∈ V | v.cand = true};
```

Algorithm 1: Maximal Valid Cut Algorithm

2.3.3 Complexity

The time complexity analysis of the cutoff algorithm is given w.r.t. the size of the graph $G = (V, E)$ and we separately evaluate the number of *valid* calls and the number of all other operations.

The number of calls to the *valid* function depends on the number of chains in C and the number of calls used in the binary searches. The total number of calls is in the worst case the same as with the algorithm based on chains, i.e. $\mathcal{O}(|C| \log L)$ where $|C|$ is the number of chains in C and L is the length of the longest chain in C. Note that the size of the *minimum* chain cover can be bounded due to Dilworth's theorem.

Theorem 3. *(Dilworth's theorem [14]) The size of the minimum chain cover of graph G equals to the size of a maximal number of pairwise unrelated elements, where u is unrelated to v if neither $u \sqsubset_G v$ not $v \sqsubset_G u$.*

To evaluate the overall complexity of the algorithm we denote by T_{valid} the time needed for one evaluation of *valid*.

The reachability relation $\sqsubset_G$ is in fact equal to the transitive closure of the graph and can be computed in $\mathcal{O}(|V| \cdot |E|)$ with the help of, e.g., depth-first search starting from each node of the graph.

The procedure *ProcessChain* first removes from the chain all vertices that have been recognised as not maximal valid in some of the previous cutoff transformations. When starting the MaxValidCut algorithm, each vertex is included in exactly one chain of the chain cover. Each vertex is removed at most once, hence the overall number of removals is bounded by the size of V and the complexity of the removals only is $\mathcal{O}(|V|)$.

The procedure *MaxValid* is an analogy of the binary search. It calls the validation function on the middle vertex of the given chain c, splits the chain into two halves, and recursively continues on one of these halves. The complexity of *MaxValid* is $\mathcal{O}(T_{valid} \cdot \log |c|)$ where $|c|$ is the length of c. The procedure is called once for each chain of the chain cover C of G resulting in the overall complexity of $\mathcal{O}(T_{valid} \cdot |C| \cdot \log L)$ where L is the length of the longest chain from C.

The procedure *Cutoff* marks those vertices which cannot be maximal valid ones. Either bread-first-search or the $\sqsubset_G$ relation can be used to detect the vertices, which should be marked, and each vertex is marked as *false* at most once. Therefore all the markings (including the initialisation) take time $\mathcal{O}(|V|)$.

The most time consuming part of the algorithm (excluding the *valid* calls) is the computation of the minimum chain cover taking time $\mathcal{O}(|C| \cdot |V|^2)$. For details and complexity analysis please refer to [10, 16]. The total time complexity of the cutoff algorithm is thus $\mathcal{O}(|V|^3 + T_{valid} \cdot |C| \cdot \log L)$.

2.3.4 Heuristics

The cutoff algorithm works with the minimum chain cover, however, the algorithm does not prescribe the order in which individual chains are processed. Each cutoff transformation affects the chains that have not been processed yet. Therefore the order in which the chains are processed affects the total number of calls to the validation function.

Cutting Power Based Heuristics. The order which minimises the number of calls to the validation function cannot be determined without the information which vertices are valid and which are not. Instead, for each chain c we can identify the minimum and the maximum number of vertices that can be cut off as a result of its processing. Let us define for each vertex v_i from the chain $\langle v_0, v_1, \ldots, v_l \rangle$ its *cutting power* as the number of vertices of G lower than v_i plus the number of vertices higher than v_{i+1} plus 1 (for vertex v_{i+1}). Then the *maximum cutting power of chain* c is the maximum of cutting powers of its vertices. Average and median cutting power of a chain can be defined in a similar way. Cutting powers of vertices can be used to propose several heuristics decreasing the number of calls to the validation function.

The first heuristic sorts the chains in descending order according to their maximum cutting powers. This heuristic can lead to a large reduction of the graph while processing the first few chains. However, this happens only if the vertices with maximum cutting power are the maximal valid vertices of these chains.

As the second heuristic we propose to compute for each chain c its average cutting power which equals to the arithmetic mean of the cutting powers of its vertices. The heuristic sorts the chains in descending order according to their average cutting power. A similar heuristic is to order the chains according to the median of the cutting powers of its vertices. These two heuristics can speed up the average performance of the algorithm.

Note that to compute the cutting power of a vertex we need to know the reachability relation of the graph. The reachability relation is pre-computed when the minimum chain cover is constructed. The only additional computation required by the heuristics is thus the sorting which takes $\mathcal{O}(|C| \cdot \log |C|)$ time and does not increase the asymptotic complexity of the cutoff algorithm.

All heuristics can be improved if we recompute the cutting powers of vertices and sort the chains after each cutoff transformation. However, this requires recomputation of the reachability relation which is rather expensive and increases the complexity of the algorithm. As explained in the introduction, our goal is to minimise the number of calls to the validation function as it is assumed to be a very expensive operation. When choosing the appropriate heuristic we have to trade off between the number of validation function calls and the complexity of the heuristic.

Cutting Power Approximation. Yet another possibility is to approximate the cutting power of vertices by some easily computable characteristic. For instance, we can take the outdegree of a vertex as a high outdegree can indicate high cutting power.

The same holds for the indegree of a vertex. Again, we can sort chains according to out/indegrees, average degree or median. On the one hand, this approach could be less effective than the approaches based on cutting powers. On the other hand, it is relatively cheap and affords to recompute the ordering after each cutoff transformation.

Online Computed Chains. As the precomputation of the minimum chain cover is rather expensive, our last heuristic drops this precomputation. The chains are instead computed on the fly. To construct a chain we take an arbitrary unprocessed vertex (i.e. a vertex whose validity is not known yet) and by following its unprocessed predecessors and successors we extend it to a chain. This chain is then processed as described in the cutoff algorithm and we repeat this process as long as there are some unprocessed vertices. We call this heuristic the *online heuristic*. Obviously, the disadvantage of this approach is that the number of the on-the-fly constructed chains can be much higher than the size of the minimal chain cover. However, if we precompute the minimal chain cover, its minimality is guaranteed only before the first cutoff transformation is made as this transformation can shorten some chains of the cover and there can emerge some chains that can be joined together. The online heuristic always processes a chain that cannot be extended any more. It can thus possibly process even less chains than the original algorithm with the minimum chain cover precomputed. Moreover, the computation of the minimal chain cover is the most expensive operation of our algorithm besides the validation calls. The online heuristic does not need this precomputation and hence the $\sqsubset_G$ relation does not need to be computed. The time complexity of the algorithm is reduced to $\mathcal{O}(|V| + |E| + T_{valid} \cdot |C| \cdot \log L)$. We compare the online heuristic with the others in the next section.

2.3.5 Relaxing the preconditions

The two main preconditions of our approach are that the graph is assumed to be acyclic and that the validation function is monotone on this graph. A natural question might arise whether we could relax one of these preconditions. Consider first an arbitrary annotated graph, i.e. a directed graph with a monotone validation function. The monotonicity implies that all vertices lying on one cycle are either all valid or all invalid. This means that we can preprocess the graph using any standard algorithm for decomposition into strongly connected components and work on the resulting (acyclic!) graph of strongly connected components.

Consider now a second possibility, where we retain the acyclic property of the graph yet relax the monotonicity precondition. If we run our algorithm on such a graph, we might not get the maximal valid cut of the graph. Nevertheless, the algorithm terminates and we obtain a set of vertices with the property that they are valid and their immediate successors in the graph are all invalid. We thus obtain at least partial evidence of the boundary between valid and invalid vertices. This can

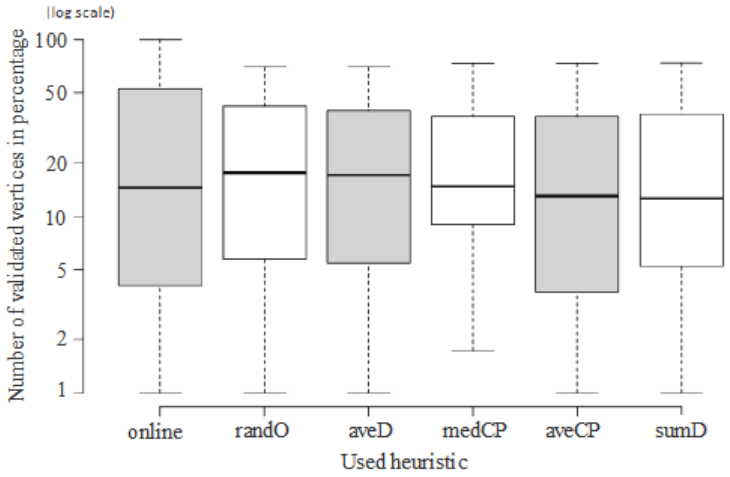


Figure 5: Efficiency of our algorithm with online computed chain cover (online), minimum chain cover (randO), and heuristics determining the order in which individual chains are processed. The graph is in log scale.

help us identify the source of errors in application areas such as software-product lines or in version control branches, which may not necessarily be monotone.

2.4 Experimental Evaluation

We implemented the cutoff algorithm and experimentally evaluated its behaviour on different types of graphs. While evaluating the algorithm we focused on the number of calls of the validation function as our aim is to minimise this number.

The first set of experiments was run on three different sets of randomly generated ADAGs of size up to 5000 vertices. The efficiency of the algorithm strongly depends on many factors like the relative number of pairwise unreachable vertices, the number and lengths of chains, density of the graph, etc. We tested the variant of our algorithm with online computed chain cover (online) and with precomputed minimum chain cover (randO). The results are shown in boxplot in Fig. 5, the boxplot shows the percentage of vertices which were validated. The online variant has higher third quartile but lower median.

Moreover, we tried the five heuristics described earlier. The heuristics sort chains from the minimum chain cover according to average cutting powers of individual chains (aveCP), medians of cutting powers of chains (medCP), average degrees of vertices of chains (aveD), and sum of the degrees of vertices of chains. The best performance was achieved using the sumD heuristic which has a median of 13 percent.

Note that there are ADAGs for which almost all vertices have to be validated, namely graphs where almost all vertices are pairwise unreachable. These types of graphs were not included in our data sets for the experimental evaluation.

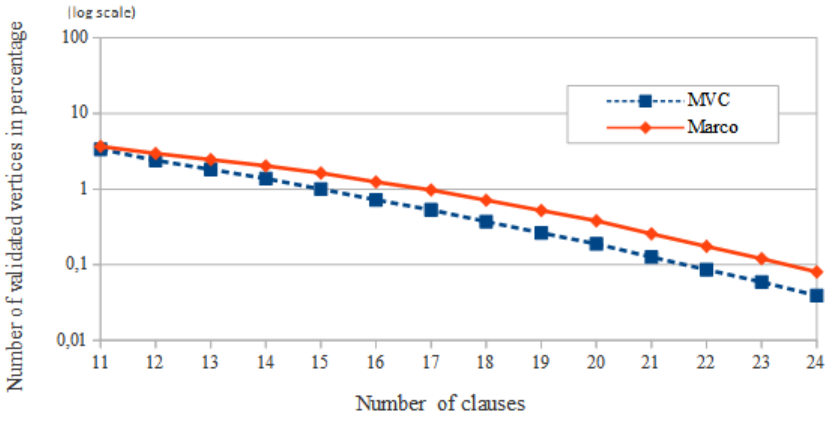


Figure 6: A log scale graph that compares our algorithm with the MARCO algorithm. The graph shows the percentage of subsets that were validated by the algorithms. Our algorithm is denoted by MVC (Maximal Valid Cut).

Requirements Checking. We now evaluate the performance of the algorithm on the graphs with the specific shape of a hypercube that represent all subsets of a set of requirements. For n requirements the hypercube consists of 2^n vertices and $n2^{n-1}$ edges. We used requirements specified in propositional logic and employed the SAT instances generator from [22] to generate experimental data. Experiments were run on requirements sets containing up to 24 requirements and hundred instances for each size. For these experiments we ran the online algorithm as it has shown to be the best one for hypercubes. The minimum-chain based approach performs worse on hypercubes as the minimal chain cover of a hypercube contains a large number of short chains. However, the binary search approach performs better on longer chains.

To provide a better insight into the qualitative parameters of our algorithm we compare its behaviour with two other tools solving the problem of finding the minimal unsatisfiable subsets of a set of requirements, namely [5] and [25]. Authors of [5] use the linear temporal logic (LTL) to specify requirements and report efficiency of around 10 percent (i.e. 10 percent of all vertices of the hypercube were validated). We were not able to repeat their experiments exactly as the authors do not provide their experimental data. Moreover, LTL is hard-coded in their tool. However, in our experiments with SAT instances the ratio of validated vertices decreases to 0.05 percent. The MARCO tool, presented in [25], is proposed to solve any constraint sets. We compared the efficiency of our algorithm against MARCO on the same sets of SAT instances. As can be seen in Fig. 6, our tool makes less queries to the SAT-solver.

3 Specialised Approach

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 infeasability. 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 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 [1].

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 part of the thesis to describe such an algorithm.

3.1 Related Work

The list of existing work that focuses on enumerating multiple MUSes is short as most of the related work focused just on an extraction of a single MUS or even a non-minimal unsatisfiable subset. For example all of [9, 27, 29] uses 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.

3.1.1 Explicit Checking

The first algorithm for enumerating all MUSes we are aware of was developed by Hou [20] 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 [18] and also by de la Banda et. al. [13].

3.1.2 CAMUS

A state-of-the-art algorithm for enumerating all MUSes called CAMUS by Liffiton and Sakallah [26] is based on the relationship between MUSes and the so-called minimal correction sets (MCSes), which was independently pointed out by [2, 8, 24]. This relationship states that $M \subseteq C$ is a MUS of C if and only if it is an irreducible hitting set of 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.

Each of the two phases can be accomplished by using any algorithm for computing MCS(C) and by using any algorithm for computing irreducible hitting sets, however the authors provide their own approach to solve these two phases. In the first phase, they repeatedly search for a satisfiable subsets of maximum size and blocking any solutions found before, thus they eventually find all MSSes whose complements are the MCSes. The second phase, computing MUSes from the MCSes, uses a recursive branching algorithm the authors developed to efficiently compute irreducible hitting sets, and it operates independently of the source of the 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. To combat this intractability of the first phase, variations of the core algorithm were proposed which can relax its completeness; however, the trade-off between time and completeness is crude [26].

3.1.3 DAA

Another algorithm that exploits the relationship between MUSes and MCSes is the Dualize and Advance (DAA) by Bailey and Stuckey [2]. DAA is an adaptation of the algorithm of the same name by Gunopulos, et al., for discovering collections of maximal frequent patterns in data mining [17]. Pseudocode for DAA is shown as Algorithm 2. It iteratively computes MCSes by "growing" MSSes from seeds (initially the empty set) and taking their complements. Subsequently it computes irreducible hitting sets of the MCSes found so far and test them for satisfiability. Unsatisfiable hitting sets are guaranteed to be MUSes and are immediately outputed, while any satisfiable hitting set can be used as a seed for the next iteration. The algorithm termines once there is no more satisfiable irreducible hitting set.

```
   input  : a set of constraints C
   output : all MSSes and MUSes of C
 1 MCSes, MUSes ← ∅;
 2 seed ← ∅;
 3 repeat
 4 |   MSS ← Grow(seed, C);
 5 |   yield MSS;
 6 |   MCSes ← MCSes ∪ {C \ MSS};
 7 |   seed ← ∅;
 8 |   for candidate ∈ (HittingSets(MCSes) \ MUSes) do
 9 |   |   if candidate is satisfiable then
10 |   |   |   seed ← candidate ;          // if sat., candidate is new seed
11 |   |   |   break;
12 |   |   else
13 |   |   |   yield candidate ;              // otherwise, candidate is MUS
14 |   |   |   MUSes ← MUSes ∪ {candidate};
15 until seed = ∅;
```

Algorithm 2: DAA algorithm

Contratry to CAMUS, DAA outputs both MUSes and MCSes (MSSes) throughout its execution, i.e., it can procude some MUSes "early". However, DAA suffers from another intractability; the number of irreducible hitting sets of a set of MCSes can be exponentially large which means that DAA can run out of memory in an early phase of its execution.

3.1.4 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 [23] and eMUS [28]. Both algorithms were later joined and presented in [25] 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 (i.e. an *unexplored subset*), 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 "grown" into a MSS. MARCO is thus suitable for both MUS and MSS online enumeration. Pseudocode for the basic variant of MARCO is shown as Algorithm 3. The algorithm can be supplied with any appropriate shrink and grow procedures; this makes MARCO applicable to any constraint satisfaction domain in general. There is also a variant of MARCO optimized for MUS enumeration; we describe this variant later in section 3.5.

```
   input  : a set of constraints C
   output : all MSSes and MUSes of C
 1 while there is an unexplored subset do
 2 |   N ←some unexplored subset;
 3 |   if N is satisfiable then
 4 |   |   MSS ← Grow(N);
 5 |   |   yield N;
 6 |   |   block all subsets of MSS
 7 |   else
 8 |   |   MUS ← Shrink(N);
 9 |   |   yield MUS;
10 |   |   block all supersets of MUS
```

Algorithm 3: The basic variant of the MARCO algorithm

CAMUS and MARCO were experimentally compared in [25] 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. DAA was also evaluated in these experiments; 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.

3.1.5 Our Contribution

In this part of the thesis, 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 binary-search-based approach. Similarly to MARCO, the algorithm is able to directly employ arbitrary shrinking and growing procedures. Moreover, our algorithm contains certain parameters that govern in which cases the shrinking and growing procedures are to be used. We evaluate our algorithm on a variety of benchmarks that show that the algorithm indeed outperforms MARCO.

Outline of this part. In Section 3.2 we state the problem we are solving in a formal way, defining all the necessary notions. In Section 3.3 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 3.5 provides an experimental evaluation on a variety of benchmarks, comparing our algorithm against MARCO.

3.2 Preliminaries

Our goal is to deal with arbitrary constraint satisfaction system. The input is given as a finite set of constraints $C = \{c_1, c_2, \ldots, c_n\}$ 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 6 (MSS, MCS, 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 correction set *(MCS) of C if $C \setminus N$ is a MSS of C. 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 that there can be multiple MSSes with different cardinality. We use MUS(C), MCS(C), and MSS(C) to denote the set of all MUSes, MCSes, 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). Note that due to the complementarity of MSS and MCS, this also enumerates all MCS(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 1. *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: $\{c_1, c_2\}$, $\{c_1, c_3, c_4\}$, three MSSes: $\{c_1, c_4\}$, $\{c_1, c_3\}$, $\{c_2, c_3, c_4\}$ and three MCSes: $\{c_2, c_3\}$, $\{c_2, c_4\}$, $\{c_1\}$.*

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 7. *Let C be a finite set of constraints. The sequence $K = \langle N_1, \ldots N_i \rangle$ is a* chain *in $\mathcal{P}(C)$ if $\forall j : N_j \in \mathcal{P}(C)$ and $N_1 \subset N_2 \subset \cdots \subset N_i$. 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.

3.3 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 satisifiability checks. Checking a given subset of C for satisfiability is usually an expensive task and the naive solution makes an exponentially many 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. It is therefore our goal to minimise the number of calls to the solver.

3.3.1 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 a 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 4. 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 to MSS_{can} or to MUS_{can}. *Explored* nodes are those, whose satisfiability the algorithm already knows and *unexplored* are the others. 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

```
   input  : a set of constraints C
   output : all MUSes and MSSes of C
 1 Unex ← P(C);
 2 MSS_can, MUS_can ← ∅;
 3 while Unex is not empty do
 4 |   Nodes ← some unexplored nodes;
 5 |   for each N ∈ Nodes do
 6 |   |   if N is satisfiable then
 7 |   |   |   MSS_can ← MSS_can ∪ {N};
 8 |   |   |   Unex ← Unex \ Sub(N) ;      // alternatively Unex ∩ ¬Sub(N)
 9 |   |   else
10 |   |   |   MUS_can ← MUS_can ∪ {N};
11 |   |   |   Unex ← Unex \ Sup(N) ;      // alternatively Unex ∩ ¬Sup(N)
12 extract MSSes from MSS_can;
13 extract MUSes from MUS_can;
```

Algorithm 4: The basic schema of our algorithm

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. The notation $\overline{Sup(N)}$, $\overline{Sub(N)}$ is used to denote the complements of $Sup(N)$ and $Sub(N)$.

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 a 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.

3.3.2 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 use the fact that the powerset lattice $\mathcal{P}(C)$ can be seen and manipulated as a Boolean algebra. We thus 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 $f(Nodes)$ 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 $Nodes = \{\{c_1\}, \{c_1, c_2\}, \{c_1, c_3\}\}$ be a set of three nodes. Using the Boolean variables representation of C, we can encode the set $Nodes$ using the Boolean formula $f(Nodes) = x_1 \wedge (\neg x_2 \vee \neg x_3)$.

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.

3.3.3 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 4. 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 a MSS, and it marks N_k as a 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

(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 a MSS candidate and N_{j+1} as a MUS candidate.

Algorithm 5 shows the extended schema of our algorithm which implements 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 $\langle N_1, N_2, \ldots, N_{k-1}, N_k \rangle$ by connecting these two nodes. The intermediate nodes $N_2, \ldots, N_{k-1}$ 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*.

Lemma 1. *(Unexplored chain construction) Let (N_1, N_k) be an unexplored couple and $\langle N_1, N_2, \ldots, N_{k-1}, N_k \rangle$ be a chain that emerges from this couple using the construction described above. Then N_i is an unexplored node for each $1 \leq i \leq k$.*

Proof. Assume that there is $1 \leq i \leq k$ such that N_i is explored, i.e. there exists node N such that either $N \in MUS_{can} \wedge N_i \supseteq N$ or $N \in MSS_{can} \wedge N_i \subseteq N$. As $N_1 \subseteq N_i$ and $N_k \supseteq N_i$, then either $N_1 \subseteq N$ or $N_k \supseteq N$ which contradicts the assumption that (N_1, N_k) is 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 $f(Unex)$). Besides

input : a set of constraints C
output : all MUSes and MSSes of C

$Unex \leftarrow \mathcal{P}(C)$;
$MSS_{can}, MUS_{can} \leftarrow \emptyset$;
while *$Unex$ is not empty* **do**
  $K \leftarrow$ some unexplored chain;
  $Nodes \leftarrow$ `ProcessChain`(K);
  for *each* $N \in Nodes$ **do**
    if *N is satisfiable* **then**
      $MSS_{can} \leftarrow MSS_{can} \cup \{N\}$;
      $Unex \leftarrow Unex \setminus Sub(N)$; `// alternatively` $Unex \cap \overline{Sub(N)}$
    else
      $MUS_{can} \leftarrow MUS_{can} \cup \{N\}$;
      $Unex \leftarrow Unex \setminus Sup(N)$; `// alternatively` $Unex \cap \overline{Sup(N)}$
extract MSSes from MSS_{can};
extract MUSes from MUS_{can};

Algorithm 5: The extended schema of our algorithm

the capability of finding an arbitrary member of $Unex$, we require the following capability: For a given member $N_p \in Unex$, the SAT solver should be able to produce a *minimal* $N_q \in Unex$ such that $N_q \subseteq N_p$, where *minimal* means that there is no other $N_r \in Unex$ 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 [15] 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 $N_l \in Unex$ 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. In order to alleviate this disadvantage of the Basic approach we propose

```
1 Function Shrink(C, N_u)
      input  : set of constraints C
      input  : unsatisfiable node N_u
      output : an MUS of C
2     for c ∈ N_u do
3         if N_u \ {c} is unsatisfiable
          then
4             N_u ← N_u \ {c}
5     return N_u
```

Algorithm 6: A simple implementation of the shrink procedure

```
1 Function Grow(C, N_s)
      input  : set of constraints C
      input  : satisfiable node N_s
      output : an MSS of C
2     for c ∈ C \ N_s do
3         if N_s ∪ {c} is satisfiable
          then
4             N_s ← N_s ∪ {c}
5     return N_s
```

Algorithm 7: A simple implementation of the grow procedure

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 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 create a random partial valuation by randomly fixing values of some variables and ask the SAT solver for a node that complies with this partial valuation. If the solver returns a node, we use it as the pivot. Clearly, giving the SAT solver a partial valuation may make it fail to find a node despite the fact that there still are some. Therefore, if the solver return nothing, we try to get the unexplored couple using the Basic approach.

3.3.4 Online MUS/MSS Enumeration

The algorithm as presented until now is only able to provide MUSes and MSSes in the second phase, after it finished 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 into MSS N_s. A simple variant of these two procedures is shown in Algorithms 6 and 7. The 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 [7] which implements the shrink method for Boolean constraints systems.

```
1 Function ProcessChain(C, K = ⟨N_1, ..., N_k⟩)
      input  : unexplored chain K = ⟨N_1, ··· , N_k⟩
      output : set of (unexplored) nodes
2     find local MSS N_s and MUS N_u of K using binary search;
3     if u < S(|K|) then
4         N_u ← Shrink(N_u);
5         yieldMUS(N_u)
6     if s > |K| − G(|K|) then
7         N_s ← Grow(N_s);
8         yieldMSS(N_s)
9     return {N_u, N_s} ;            // Note that N_u or N_s may not exist
```

Algorithm 8: Extended version of the *processChain* method

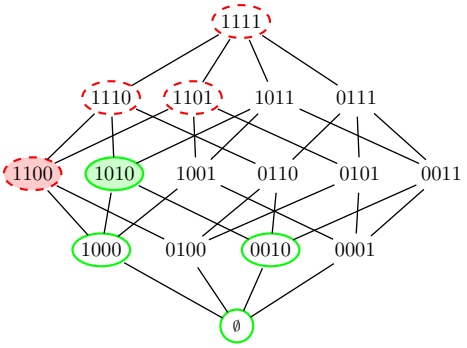
Recall that as a result of a processing a single chain K, our algorithm finds either a local MUS N_u, or a local MSS N_s, or both of them. To get a MUS (MSS) we propose to employ the shrink (grow) method on this local MUS (MSS). However, performing shrink (grow) on each local MUS (MSS) can be quite expensive 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 $S : \mathbb{N} \to \mathbb{N}$ 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 $G : \mathbb{N} \to \mathbb{N}$. 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 (and grows) 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 8 shows an extended version of the method *processChain* which is able to produces MUSes and MSSes during its execution based on the above mechanism.

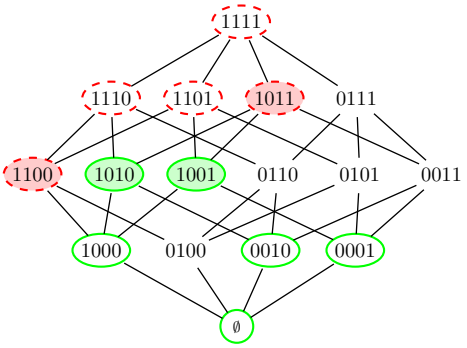
3.4 Example

The following example explains the execution of our algorithm on a simple set of constraints $C = \{c_1 = a,\ c_2 = \neg a,\ c_3 = b,\ c_4 = \neg a \vee \neg b\}$ and with the S, G functions set to $S(x) = x$, $G(x) = x$. The example is illustrated in Fig. 7.

I. iteration
– Unex. couple $\langle 0000, 1111\rangle$
– Unex. chain $\langle 0000, 1000, 1100, 1110, 1111\rangle$
– A local MSS 1000 and local MUS 1100 are found
– 1000 is grown to the MSS 1010
– 1100 is shrunk to the 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, local MUS 1011
– 1001 is grown to the MSS 1001
– 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, local MUS *undefined*
– 0111 is grown to the MSS 0111
– $MSS_{can} \leftarrow MSS_{can} \cup \{0111\}$
– $f(Unex) \equiv (x_2 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

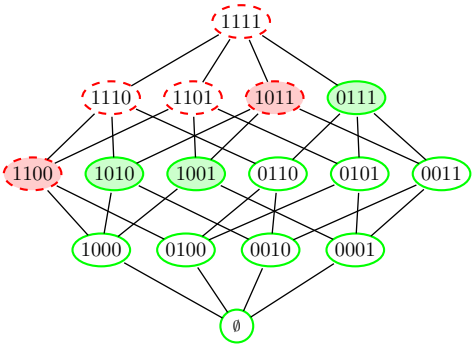


Figure 7: An example execution of our algorithm

Initially $MSS_{can} = \emptyset$, $MUS_{can} = \emptyset$ and all nodes are unexplored, i.e. $f(Unex) = True$. Figure 7 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.

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 (due to the definition of MSS) that all its supersets are unsatisfiable. One would 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.

3.5 Experimental Evaluation

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[1]. We also give a comparison with the MARCO algorithm [25].

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 a 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 a MSS. We used the optimised variant in our experiments. The pseudocode of the optimised variant is shown as Algorithm 9

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 the MARCO algorithm in C++ to ensure that the two algorithms use the same implementations of the shrink and grow methods and the same solvers (the original MARCO is implemented in Python and has shown to be slightly slower than our reimplementation). As both the SAT solver and constraint solver we used the miniSAT tool [15] 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 these methods. That is why we used the simple implementations.

[1]The implementation is available at `http://www.fi.muni.cz/~xbendik/mvc/`

```
input  : a set of constraints C
output : all MSSes and MUSes of C
1 while Unex is not empty do
2 |   N ←some maximal unexplored node;
3 |   if N is satisfiable then
4 |   |   yield N;
5 |   |   Unex ← Unex \ Sub(N)
6 |   else
7 |   |   MUS ← Shrink(N);
8 |   |   yield MUS;
9 |   |   Unex ← Unex \ Sup(N)
```

Algorithm 9: An optimized MARCO using maximal unexplored nodes

As an experimental data we used a collection of 294 unsatisfiable Boolean CNF Benchmarks that were taken from the MUS track of the 2011 SAT competition[2]. The benchmarks 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

[2] `http://www.cril.univ-artois.fr/SAT11/`

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 approach	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**					

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. 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 $S(x) = 0.4x$. 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. In this comparison, there is already some notable difference between the Basic and the Pivot based approach. The Pivot based approach seems to be better for MUS enumeration whereas the Basic approach is more suitable for the MSS enumeration. As the Pivot based approach is randomized its performance may vary if it is run repeatedly on the same instances; 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.

Besides the number of outputted MUSes/MSSes within a given time limit, we also compared our algorithm with MARCO in the case of complete MUS/MSS

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 approach	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**					

enumeration. We used the generator of Boolean CNF formulae from [22] to generate tractable instances with a size of 30 to 40 constraints, 15 instances per each size. The graphs in Fig. 8 show the time comparison of MARCO and our algorithm using the Pivot based approach (PBA) with S and G set to $S(x) = 0.2x$ and $G(x) = 0.8x$. All of the instances were tractable which means that both phases of our algorithm were executed. Some of the MUSes and MSSes were outputed in the online manner, the rest of them were extracted from the candidate sets in the second phase.

Summarised, our algorithm outperformed MARCO both in the online MUS/MSS enumeration and in the complete MUS/MSS enumeration. Also, the results show that the choice of the functions S and G greatly affect the efficiency of our algorithm. The faster the S (G) grows, the more effort is made to output MUSes (MSSes). Also, it may be worth to always perform at least the "easy" grows (shrinks) even if we want to output only MUSes (MSSes), because each shrink (grow) also helps to reduce the space of unexplored nodes.

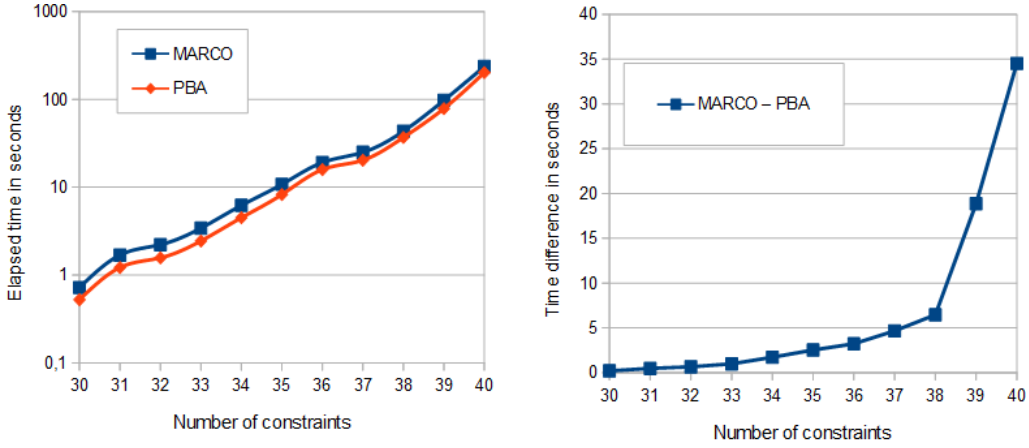


Figure 8: The time comparison of the MARCO and our algorithm. The chart on the left side shows the running times and it is logarithmically scaled. The other chart shows the difference in the running times between the two algorithms.

4 Conclusion

In this thesis, we have focused on finding boundary elements in partially ordered sets, seen as directed acyclic graphs. We have discussed the mapping of this problem to various activities in software engineering; we have shown applications in safety and requirements analysis. We have presented a new algorithm to solve this problem, including several variants and heuristics. The results of the experiments that we have made show that the efficiency of the heuristics depends on the structure of the input graph.

We have also focused on finding boundary elements in hypercube graphs which has an application in the area of Constraint Satisfaction Problems (CSPs). Namely, we have aimed on the problem of online enumeration of Minimal Unsatisfiable Subsets (MUSes) and Maximal Satisfiable Subsets (MSSes) of unsatisfiable set of constraints. We have given a detailed list of existing approaches for solving the problem of MUS/MSS enumeration and pinpointed their strengths and weaknesses. Subsequently we have gradually presented our novel algorithm for online enumeration of MUSes and MSSes. The core idea of the algorithm is based on a novel binary-search-based approach which 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 is better both in online enumeration and also in the case of complete enumeration. Our algorithm can be built on a top of any consistency solver and can employ any implementation of the *shrink* and *grow* methods, therefore any future advance in these areas can be reflected in the performance of our algorithm.

4.1 Future work

As a future work, we consider several improvements of both the algorithm for finding boundary elements in general graphs and the algorithm for online MUS/MSS enumeration. Especially a future developlment of the latter algorithm is well motivated as there is a growing interest in the area of CSPs nowadays. One possible direction of future research is to aim at parallel processing of the configuration space in order to improve the performance of our approach; there are usually several disjoint unexplored chains that can be processed concurrently.

We also want to consider more applications of our approach, such as software product line engineering and discovering incompatibilities in component-based designs. We also believe that our method can be applied to various other domains, such as the parameter synthesis for biological systems [3]. We intend to explore these applications in more detail.

References

[1] 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.

[2] 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.

[3] J. Barnat, L. Brim, A. Krejci, A. Streck, D. Safranek, M. Vejnar, and T. Vejpustek. On parameter synthesis by parallel model checking. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3):693–705, 2012.

[4] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing*, doi: 10.1007/s00165-015-0348-9, 2016.

[5] Jiri Barnat, Petr Bauch, and Lubos Brim. Checking sanity of software requirements. In *SEFM 2012 Proceedings*, volume 7504 of *LNCS*, pages 48–62. Springer, 2012.

[6] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Form. Methods Syst. Des.*, 18(2):141–163, 2001.

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

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

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

[10] Yangjun Chen and Yibin Chen. On the decomposition of posets. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 134–138, 2012.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[12] Samar Dajani-Brown, Darren D. Cofer, Gary Hartmann, and Steve Pratt. Formal modeling and analysis of an avionics triplex sensor voter. In *SPIN 2003*, volume 2648 of *LNCS*, pages 34–48. Springer, 2003.

[13] 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.

[14] Robert P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

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

[16] D. R. Fulkerson. Note on Dilworth's decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4):701–702, 1956.

[17] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharm. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003.

[18] 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.

[19] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. Software Engineering and Formal Methods. *Commun. ACM*, 51:54–59, 2008.

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

[21] Anjali Joshi, Steven P Miller, Michael Whalen, and Mats PE Heimdahl. A proposal for model-based safety analysis. In *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*, volume 2. IEEE, 2005.

[22] Massimo Lauria. CNFgen formula generator. `http://massimolauria.github.io/cnfgen/`. Accessed: 2016-01-11.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

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