Computing Minimal Unsatisfiable Subsets of Constraints

Seminar

Author: Shahriar Robbani
Supervision: Erika Ábrahám
RWTH Aachen University, LuFG Informatik 2

WS 16/17

Abstract

We present a procedure as published in [1] which computes all Minimal Unsatisfiable Subsets (MUSs) of an unsatisfiable constraint system. The authors have developed a complete algorithm to compute all MUSes in two phases, one is generating all Minimal Correction Subsets (MCSs) and another one is generating all MUSs. Removing an MCS makes the unsatisfiable constraint system satisfiable. Also removing one clause from each MUS implies that the remaining set of clauses is not unsatisfiable anymore.

1 Introduction

Many real-world problems arising for example in formal verification of hardware and software can be formulated as propositional logic formulas, which are Boolean combinations of atomic propositions. When transformed to conjunctive normal form (CNF), i.e., when formulas are conjunctions of disjunctions of propositions (clauses) and negated propositions (literals). Boolean satisfiability (SAT) solvers can be employed to determine whether a formula is satisfiable or unsatisfiable i.e., whether there exists an assignment of Boolean values for the propositions such that the formula evaluates to true. When a formula is unsatisfiable, it is required to find minimal unsatisfiable subsets of the problem's clauses. Minimal Unsatisfiable subsets (MUSes), also called Minimal Unsatisfiable Cores (MUCs), are essential to determine the reasons for the failure of the system. In the past few years, the interest and research on the extraction of MUSes of an unsatisfiable constraint system has been increasing, but most of the research has concentrated on the algorithms to extract a single MUS [1]. This single MUS may not provide complete information about the failure of constraint system, because the system might have multiple MUSes and the appearance of any of them makes the system unsatisfiable.
In this paper, we explaine some algorithms which are necessary to compute all MUSs of a system as in [1]. The paper is organized as follows. Section 2 presents some preliminaries with formal definitions. We introduce the algorithms in details in Section 3 and conclude in Section 4.

2 Preliminaries

2.1 Propositional Logic Formula

Propositional logic formulas are Boolean combinations of atomic propositions i.e., each atomic proposition is a statement which is either *true* or **false**. A well-formed propositional logic has following grammar:

$$\varphi \quad := \quad a \quad | \quad (\neg\varphi) \quad | \quad (\varphi \wedge \varphi)$$

Here a is a atomic proposition.
Syntactic sugar (a syntax which make propositional logic formulas easier to read or to express) for propositional logic formulas are given below:

$$\bot \quad := \quad (a \wedge \neg a)$$

$$\top \quad := \quad (a \wedge \neg a)$$

$$(\varphi_1 \vee \varphi_2) := \neg((\neg\varphi_1) \wedge (\neg\varphi_2))$$

$$(\varphi_1 \rightarrow \varphi_2) := ((\neg\varphi_1) \vee \varphi_2)$$

$$(\varphi_1 \leftrightarrow \varphi_2) := ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1))$$

$$(\varphi_1 \oplus \varphi_2) := (\varphi_1 \leftrightarrow (\neg\varphi_2))$$

We omit parentheses according to the operator following operator precedence:

$$\xleftarrow{\hspace{8em}}$$

$$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

Example 1 *Some propositional logic formulas are given below:*

- $(\neg a)$
- $((a \wedge b) \vee c)$
- $((a \rightarrow b) \rightarrow c)$

2.2 Conjunctive Normal Form

A propositional logic formula is said to be in Conjunctive Normal Form (CNF) if and only if it is a conjunction of clauses, where clause is a disjunction of literals. A literal could be a positive or negative instance of Boolean variable. A CNF formula φ is defined as follows:

$$\varphi = \bigwedge_{i=1\ldots n} C_i$$

$$C_i = \bigvee_{j=1\ldots m} a_{ij}$$

Here, C_i is the i^{th} clause and n is total the number of clauses in φ, a_{ij} is a literal and m is the total number of literals in C_i.

Table 2.1 Truth Table

x	y	$x \vee y$	$x \rightarrow y$
0	0	0	1
0	1	1	1
1	0	1	0
1	1	1	1

Example 2 *Assume the following CNF formula,*

$$\varphi = \underbrace{(x)}_{C_1} \wedge \underbrace{(\neg x)}_{C_2} \wedge \underbrace{(\neg x \vee y)}_{C_3} \wedge \underbrace{(\neg x \vee \neg y)}_{C_4}$$

where, $x, \neg x, y$ and $\neg y$ are the literals.

A propositional logic formula is said to be satisfiable if it is possible to find an assignment that makes the formula true, otherwise unsatisfiable.

Example 3 *Assume two propositional formulas be defined as $\varphi_1 = x \vee y$ and $\varphi_2 = x \rightarrow y$, $\alpha : \{x, y\} \rightarrow \{0, 1\}$ be an assignment with $\alpha(x) = 1$ and $\alpha(y) = 0$. Considering Table 2.1* (0 = **false** *and* 1 = true)*: So, φ_1 is satisfiable for the assignment, whether φ_2 is unsatisfiable.*

The aforementioned CNF formula which is unsatisfiable will be used as an example throughout this paper.

2.3 Satisfiability Checking

Satisfiability checking for propositional logic determines whether a given propositional logic formula is satisfiable. One of the most successful technologies for this test is SAT solving.
A SAT solver solves the above satisfiability checking problem by implementing a decision procedure. The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is the basis for most modern SAT solvers. The input formula of it is expected to be in CNF. It's not possible to explain SAT solver shortly.

2.4 Clause-Selector Variables

The authors of the paper [1] have used clause-selector variable to augment a CNF formula. A clause-selector variable is a variable, w_i which is negated to augment each clause C_i of a CNF φ such that $C'_i = (\neg w_i \vee C_i)$ are the clauses of the new formula φ'. Notice that each C'_i is an implication, $C'_i = (w_i \rightarrow C_i)$. It means if w_i is set to *true*, the original clause C_i must be satisfied. Conversely, assigning w_i to **false** means that C_i does not need to satisfied for satisfying C'_i, which corresponds to removing C_i from φ. So, adding clause-selector variables provides the SAT solver the ability to enable and disable clauses as a part of its normal search.

Example 4 *For our example formula $\varphi = (x) \wedge (\neg x) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$, after adding clause-selector variables we get the following augmented formula φ',*

$$\varphi' = (\neg w_1 \vee x) \wedge (\neg w_2 \vee \neg x) \wedge (\neg w_3 \vee \neg x \vee y) \wedge (\neg w_4 \vee \neg x \vee \neg y)$$

Table 2.2 All MUSes and MCSes

$MUSs(\varphi)$	$\{C_1, C_2\}, \{C_1, C_3, C_4\}$
$MCSs(\varphi)$	$\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}$

2.5 Minimal Unsatisfiable Subset and Minimal Correction Subset

An Unsatisfiable Subset (US) of φ for an unsatisfiable CNF formula φ is a subset of the clauses of whose conjunction is unsatisfiable. An US is said to be a Minimal Unsatisfiable Subset (MUS) if and only if removing any clause from the US makes it satisfiable. A Minimal Correction Subset (MCS) is a subset of the clauses of φ whose removal from φ makes it satisfiable and which is minimal.

Definition 2.1 (MUS) *For a propositional logic formula φ in CNF, a subset U of $\varphi's$ clauses is an US* for φ *if* $\bigwedge_{C \in U} C$ *is unsatisfiable. An US* for φ *is* minimal *(MUS) if for all* $U' \subset U$, $\bigwedge_{C \in U'} C$ *is satisfiable.*

Definition 2.2 (MCS) *For a propositional logic formula φ in CNF, a subset M of $\varphi's$ clauses is an MCS* for φ *if* $\bigwedge_{C \in M} C$ *is satisfiable and for all* $M' \subset M, (\varphi slashDiteHobe \bigwedge_{C \in M'} C)$ *is unsatisfiable.*

A formula can have multiple MUSs and MCSs. For any formula φ, the set of all MUSs and MCSs of φ are denoted by MUSs(φ) and MCSs(φ), respectively. For our example formula from [Example 1 linkKorteHobe], the MUSes and MCSes are shown in the Table 2.1.

2.6 Hitting Sets

Assume, $\Omega \subseteq 2^D$ is a collection (set) of sets of a finite set D. A hitting set of Ω is a subset of D such that it contains at least one element from each subset in Ω.

Definition 2.3 (Hitting Set) *Given a finite set D and a $\Omega \subseteq 2^D$, a hitting set (HS) H of Ω is $H \subseteq D$ such that $\forall S \in \Omega \; H \cap S \neq \emptyset$. A HS H of Ω is* minimal *if fro all $H' \subset H$ it holds that H' is not an HS.*

Example 5 *Let us consider $D = \{a, b, c, d\}$ and $\Omega = \{\{a, b\}, \{b, c, d\}\}$. Then $\{a, b\}, \{b, c, d\}, \{a, c, d\}, \{b\}, \cdots$ are hitting sets of ω where $\{b\}, \{a, c\}$ and $\{a, d\}$ are the minimal hitting sets.*

2.7 MUS \ MCS Duality

There is a relationship between MUSs and MCSs which is the foundation of paper [1]. The relationship states that the set of MUSs of formula φ is equal to the set of minimal hitting sets of the set of MCSs and vice-versa. This is the duality of MUS and MCS. Formally, it can said that:

1. A subset U of $\varphi's$ clause is an MUS if and only if U is a minimal hitting set of MCSs(φ).
2. A subset M of $\varphi's$ clause is an MCS if and only if M is an minimal hitting set of MUSs(φ).

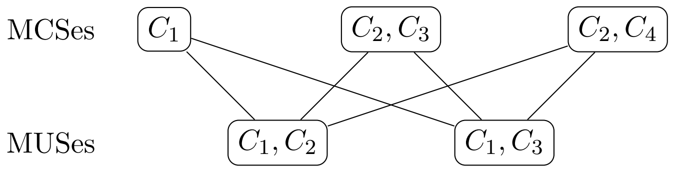

Figure 1: Duality of MUSes and MCSes.

Example 6 *To illustrate this duality we use our example formula:*

$$\varphi = \underbrace{(x)}_{C_1} \wedge \underbrace{(\neg x)}_{C_2} \wedge \underbrace{(\neg x \vee y)}_{C_3} \wedge \underbrace{(\neg x \vee \neg y)}_{C_4}$$

. First, the set of MCSs of φ $\{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$, whose minimal hitting sets are $\{\{C_1, C_2\}, \{C_1, C_{3,}C_4\}\}$ which is exactly the set of MUSs (given in Table 1). Similarly, we can show that the set of MCSes is the set of minimal hitting sets of the set of MUSSs. So, every MCS has at least one clause from every MUS and vice-versa. The duality is illustrated in Figure[1 linkKorte Hobe].

2.8 AtMost Constraints

An AtMost Constraint states that the number of the literals from a given set does not exceed a given upper bound. For $\{l_1, l_2, l_3, \cdots, l_\mathrm{n}\}$ being a set of n literals and k a non-negative integer, the corresponding AtMost constraint is defined as:

$$AtMost(\{l_1, l_2, \ldots, l_n\}, k) \equiv \sum_{i=1}^{n} val(l_i) \leqslant k$$

where $val(l_\mathrm{i})$ is 1 if l_i is assigned *true* and otherwise 0.The bound k tells that at most how many literals can be assigned]*true*.
the authors used an implementation of the AtMost constraints where the variables are watched and propagates the negation of each literal when k of them are assigned *true*. This is implemented with exactly n watched literals and a counter that is incremented or decremented whenever one of them is assigned or unassigned.

Example 7 *Consider formula $\varphi = (l_1 \vee l_2 \vee l_3)$ with the AtMost constraint $\varphi_{AM} = AtMost(\{l_1 \vee l_2 \vee l_3\}, 2)$. Solving the formula $\varphi \wedge \varphi_{AM}$ might assign true to l_1 and l_2, but φ_{AM} will assure that l_3 will be assigned* **false**.

3 Algorithms for Computing Minimal Unsatisfiable Subsets

The approach of computing all MUSs of φ is to first find all MCSes(φ) and then to compute all minimal hitting sets of MCSs(φ), which are all MUSes(φ). So, there are two phases to generate all MUSes of an unsatisfiable CNF formula.

3.1 Computing MCSes

Algorithm 1 depicts that a method named **MCSs** which takes a CNF formula φ as input. This method finds all minimal correction sets whose removal makes φ satisfiable. In Line 1, φ is augmented with clause-selector variables by creating φ'. The augmentation increases

Algorithm 1 Algorithm for finding all MCSes of a formula φ

```
MCSes ( $\varphi$ )
 1  $\varphi' \leftarrow AddYVars(\varphi)$
 2  $MCSes \leftarrow \emptyset$
 3  $k \leftarrow 1$
 4  while SAT($\varphi$)
 5  do $\varphi'_k \leftarrow \varphi' \wedge AtMost(\{\neg w_1, \neg w_2, \dots, \neg w_n\}, k)$
 6      while $newMCS \leftarrow IncrementalSAT(\varphi'_k)$
 7      do $MCSes \leftarrow MCSes \cup newMCS$
 8          $\varphi'_k \leftarrow \varphi'_k \wedge BlockingClause(newMCS)$
 9          $\varphi' \leftarrow \varphi' \wedge BlockingClause(newMCS)$
10
11      $k \leftarrow k + 1$
12
13  return $MCSes$
```

the number of variables. Also the search space grows for finding a satisfying assignment. Initially, the variable ***MCSes*** is initialized to store the empty set and at the end we will get all MCSs in this variable. There is a counter variable, k which is initialized to 1.
Each iteration of the outer while loop (Lines 4-12) finds an MCS of size k, which is incremented by 1 after each iteration. A clause is added of the form $AtMost(\{\neg w_1, \neg w_2, \dots, \neg w_n\}, k)$ to φ' and a new formula φ'_k is created. k number of variables w_i are assigned to **false** so that k number of clauses could be disabled. The set of variables w_i are assigned to **false** by indicating the clauses as an MCS.
The inner while loop (Lines 6-10) searches all the satisfiable assignments of φ'_{k} and also finds all MCSs of size k. **IncrementalSAT** is an important part of this algorithm which is called to find satisfying assignment for φ'_{k}. This method finds satisfying assignment by invoking the solving function multiple times, but each time with a different set of assumption literals and then the solver checks the satisfiability of all the clauses provided with the current assumptions only [2]. Each satisfying assignment produces an MCS stored in a variable **newMCS** which contains the clauses having w_{i} assigned **false**. Each **newMCS** is recorded in **newMCSs**. After that a blocking clause of **newMCS** is added to φ'_k and φ', respectively to block that solution. Blocking clause is a clause containing the clause-selector variables which consist in the clauses of an MCS. It means that at least one of the clauses in the MCS will be enabled for future solution. For example, an MCS has the clauses C_2 and C_3 which means w_2 and w_3 are assigned **false** in satisfying assignment. Then the blocking clause will be $(w_2 \vee w_3)$. To make this clause satisfiable, *true* is needed to assign to at least one of the w_i variables which excludes the MCS and any of its supersets from any future solution.
All founded MCSs are *minimal* as the MCSs were found in increasing size and all supersets are excluded from the future solutions. After evaluating all the solutions with bound k enforces to find solutions with bound $k + 1$. Also, $k + 1$ disabled clause to be irreducible as potential subsets would have found and blocked.
The outer while loop checks if φ' is still satisfiable without any bound on w_{i}. Note that, φ' is augmented with all generated blocking clauses. Finding no satisfying assignments indicates that we have found all MCSes and there is no other way to remove clauses for satisfying formula. Finally, the algorithm terminates and returns all MCSes to caller.

Table 3.1 Running MCSs(φ) on our example formula φ

Step No.		k	φ'	MCSs
1	Initialization	1	$(x) \wedge (\neq x) \wedge (\neq x \vee y) \wedge (\neq x \vee \neq y)$	$\emptyset$
2	Run1		$\varphi'(step1) \wedge w_1$	$\{C_1\}$
3	Run2	2	$\varphi'(step2) \wedge (w_2 \vee w_3)$	$\{C_2, C_3\}$
4	Run3		$\varphi'(step3) \wedge (w_2 \vee w_4)$	$\{C_2, C_4\}$

Example 8 *Let us consider our example formula $\varphi = (x) \wedge (\neg x) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$. So, $\varphi' = (\neg w_1 \vee x) \wedge (\neg w_2 \vee \neg x) \wedge (\neg w_3 \vee \neg x \vee y) \wedge (\neg w_4 \vee \neg x \vee \neg y)$. Initially, MCSes $= \emptyset$ and $k = 1$.*
In the first iteration, it will add an AtMost bound with $k = 1$ by creating $\varphi'_1 = \varphi' \wedge AtMost(\{l_1, l_2, l_3, l_4\}, 1)$. At first $false$ is assigned to w_1 and the incremental solver finds a solution for φ'_k. Now, we get the followings:

$$MCSs = \{C_1\}$$

$$\varphi'_1 = \varphi'_1 \wedge w_1$$
$$\varphi' = \varphi' \wedge w_1$$

After adding the blocking clause, (w_1) the incremental solver will be unable to find any further solution with $k = 1$. There is no other way to remove one clause to satisfy φ. It means no other single clause covers all of its MUSes. The inner while loop is exited.
As φ' is still satisfiable, the bound is incremented to 2 and the search for soluitons of $\varphi'_2 = \varphi' \wedge AtMost(\{l_1, l_2, l_3, l_4\}, 2)$ is continued.For $k = 2$, we get the followings, respectively:

- *$w_2 = false$ and $w_3 = false$,*

$$MCSs = \{\{C_1\}, \{C_2, C_3\}\}$$

$$\varphi'_2 = \varphi'_2 \wedge (w_2 \vee w_3)$$
$$\varphi' = \varphi' \wedge (w_2 \vee w_3)$$

- *$w_2 = false$ and $w_4 = false$,*

$$MCSs = \{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$$

$$\varphi'_2 = \varphi'_2 \wedge (w_2 \vee w_4)$$
$$\varphi' = \varphi' \wedge (w_2 \vee w_4)$$

We will not get any other MCSes for $K = 2$. Also, the outer while loop will exit as φ' with all the added blocking clauses is no longer satisfiable. So, final MCSs are, $\{C_1\}, \{C_2, C_3\}$ and $\{C_2, C_4\}$.

Algorithm 2 Algorithm for computing the complete set of MUSes from a set of MCSes

```
AllMUSes ( MCSes, currentMUS )
 1  if MCSes = $\emptyset$
 2    then print(currentMUS)
 3         return for each selClause $\varepsilon$ RemainingClauses(MCSes)
 4  do
 5     $newMUS \leftarrow currentMUS \cup selClause$
 6     for each selMCS $\varepsilon$ MCSes such that selClause $\varepsilon$ selMCS
 7     do
 8        $newMCSes \leftarrow MCSes$
 9        $PropagateChoise(newMCSes, selClause, selMCS)$
10        $AllMUSes(newMCSes, newMUS)$
11        return
```

3.2 Computing MUSes

Once the entire collection of MCSs has been computed, the second phase produces all MUSs of the given instances and this phase is independent of first phase as it does not depend on the semantics of the input of Algorithm 1. The second phase can be applied to any minimal hitting set. The authors presented a recursive algorithm which takes two inputs, one is the set of all MCSs and another is the MUS currently being constructed in each branch of the recursion. The outputs are the set of all MUSs.
Algorithm 2 is the recursive algorithm which takes **MCSs** and **currentMUS** as inputs. The terminating condition of the recursion is if **MCSs** is empty. If it is, prints the MUS generated in the current recursion branch and returns to look for other MUSs in another branches. The outer for loop (Lines $5-12$) is iterated through all possible choices of clause selected from **MCSs** and recorded in a variable **selClause**. Then this selected clause is stored in a growing MUS **newMUS**. The outer for loop (Lines $7-11$) is iterated for all possible choices of MCSs recorded in **selMCS** such that it contains **selClause**. **MCSs** is copied in **newMCSs**.
For every **selMCS** a method, **PropagateChoice** (Algorithm 3) is called with **newMCSs**, *selClause* and *selMCS.*

Algorithm 3 Algorithm for altering MCSes to make the choice of thisClause irredundant as the only element hitting thisMCS

```
PropagateChoice ( MCSes, thisClause, thisMCS )
1  for each clause $\varepsilon$ thisMCS
2  do for each testMCS $\varepsilon$ MCSes
3     do if clause $\varepsilon$ testMCS
4           then $testMCS \leftarrow testMCS - \{clause\}$
5                for each testMCS $\varepsilon$ MCSes
6  do if thisClause $\varepsilon$ testMCS
7       then $testMCS \leftarrow testMCS - \{clause\}$
8            $MCSes \leftarrow MCSes - \{testMCS\}$
9            $MaintainNoSupersets(MCSes)$
```

The goal of this algorithm is to alter the copy of the current MCSs (**newMCSs**). It al-

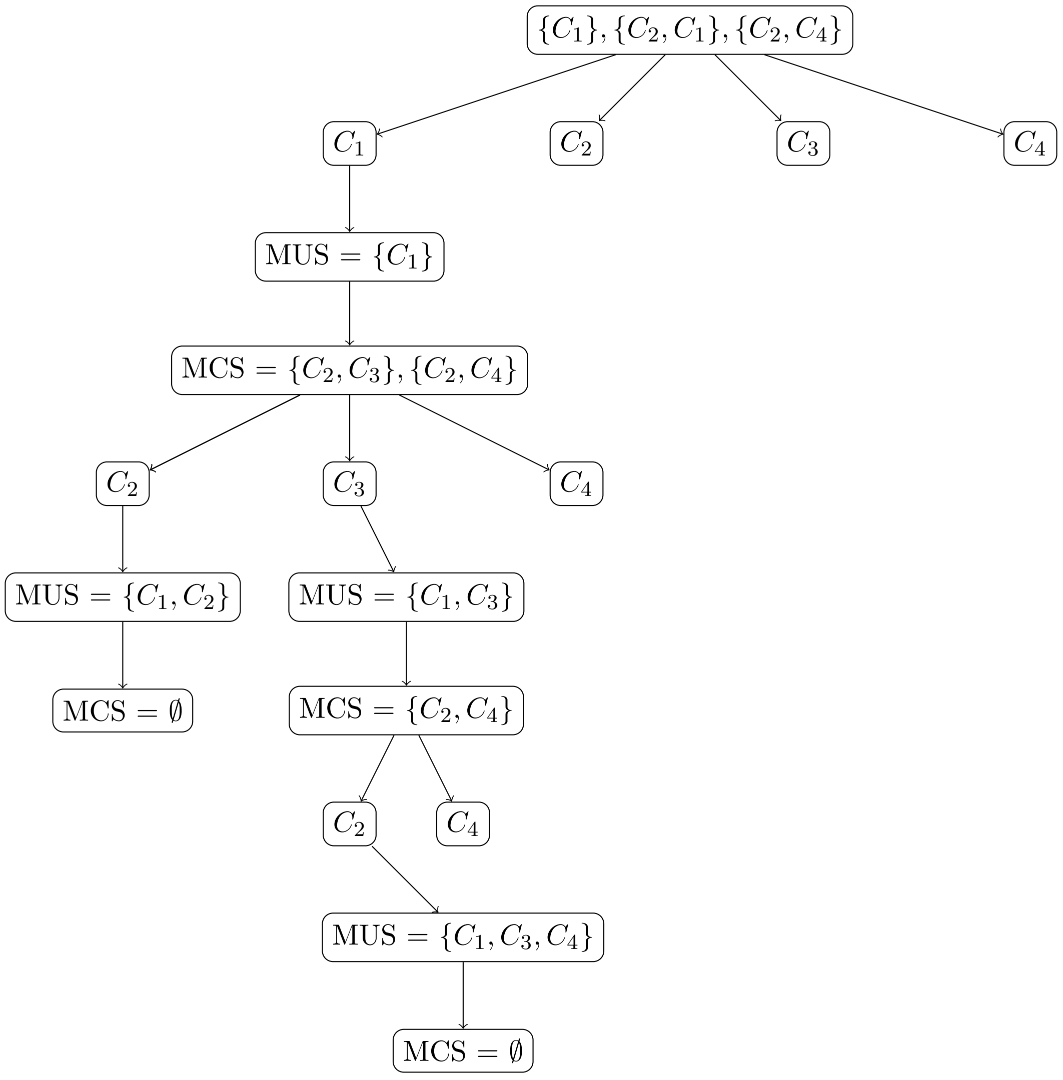

Figure 2: Illustration of all MUSes

ters by removing an MCS from the copy hitted by **thisClause**. At the end **MaintainNoSupersets** is invoked which removes an MCS from *mathbfMCSes* which is a superset of others. This is needed as we are dealing with the *minimal* set of MCSs and no MCS cannot be a superset of others. For example, let we have $MCSs = \{\{C_2\}\{C_1,C_3\}\{C_2,C_4\}\}$. Here, $\{C_2,C_4\}$ is a superset of $\{C_2\}$. If we pass this MCSs through **MaintainNoSupersets**, $\{C_2,C_4\}$ will be removed.
The recursion of Algorithm 2 continues by calling **AllMUSs** with **newMCSs** and **newMUS**.

Example 9 *Let us consider, our example formula linkkortehobe. We have already found $MCSs(\varphi) = \{\{C_1\},\{C_2,C_3\},\{C_2,C_4\}\}$. Let, C_2 is chosen as* **selClause** *in the first iteration. So, C_2 is added in the growing MUS. Now, for each set of MCSs containing C_2 another iteration will occur which calls* **PropagateChoice** *to create a modified copy of current MCSs and makes a recursive call to itself. Modified MCSs contain $\{C_1\}$ which becomes current MCSs after the recursive call. The recursive call continues until empty MCSs is not found. For this branch empty MCSs is found by generating a MUS, $\{C_1,C_2\}$. Now, it will look for other MUSs in other branches.*

Again consider another iteration where C_3 *is chosen as* **selClause** *and* **newMUS** $= \{C_3\}$. *Inner for loop runs for only* **selMCS** $= \{C_2, C_3\}$ *as* **selClause** $\in$ **selMCS** *and creates a modified copy of current MCSs by calling* **PropagateChoice**. *Recursive call continues, because we have non-empty* $MCSs = \{\{C_1\}, \{C_2, C_4\}\}$. *For this branch an empty MCSes is found after generating another MUS* $\{C_1, C_3, C_4\}$ *If we search other branches for new MUSes, we would not find as* $\{C_1, C_2\}$ *and* $\{C_1, C_3, C_4\}$ *are only two MUSes. In other word these are two* minimal *sets of clauses that hits each set of MCSs. Finally, we get all MUSs,* $\{C_1, C_2\}$ *and* $\{C_1, C_3, C_4\}$.

4 Conclusion

We have discussed a set of algorithms which generate MUSs and also the authors made a framework consisting of the algorithms. The generation of MUSs is based on the duality of MCSs and MUSs. The algorithms can be used to work with different types of constraint solvers. The algorithms (may be with small modification) can be used on top of new constraint solvers to provide the functionality of finding MUSs for new types of constraints.

References

[1] Mark H. Liffiton, Karem A. Sakallah, *Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints*. Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor 48109-2121.

[2] Alexander Nadel, Vadim Ryvchin, Ofer Strichman *Ultimately Incremental SAT*. `https://ie.technion.ac.il/~ofers/publications/sat14.pdf`