Minimal Infeasible Subset

Seminar: Satisfiability Checking

Idil Esen Zulfikar
Supervision: Prof. Dr. Erika Ábrahám

SS 2017

Abstract

In the domain of constraint satisfaction, extraction of unsatisfiable cores of unsatisfiable constraint satisfaction problems (CSPs) [4] has been studied in research. Most of the works, unsatisfiable subsets of constraint systems has been extracted, whereas there is not much research on the extraction of minimal unsatisfiable subsets (MUSs) has been done. In this paper, we describe a two-phase algorithm to provide all MUSs of unsatisfiable constraint systems through establishing a relation between correction cores and unsatisfiable cores of constraint systems. We illustrate the method on an example for Boolean satisfaction problem.

1 Introduction

Much research has been done on solving constraint satisfaction problems (CSPs). These research focused on deciding whether a constraint system is satisfiable or unsatisfiable. In order to solve CSPs, there are different methods to decide satisfiability or unsatisfiability of problems, but, in case of unsatisfiability, most of these methods cannot provide any explanation for why the problem is unsatisfiable. However, explanation of unsatisfiability of a problem is high practical relevance.

Reasons of unsatisfiability can be given via unsatisfiable subsets of problems. In the last years, most research focused on the extraction of a single unsatisfiable subset (US) that is not necessarily minimal. For a given unsatisfiable constraint system C, if we look at the system as a set of constraints, minimal unsatisfiable subsets (MUSs) of C are unsatisfiable and minimal subsets of C. MUSs are minimal because removing any element of a minimal unsatisfiable set (MUS) makes the remaining constraint set satisfiable. MUSs are important because they give explanations for infeasibility of unsatisfiable constraint systems. Moreover, MUSs are used with many SAT based applications for system analysis and verification and with other applications which utilize unsatisfiable cores [2].

This paper is based on the publication [5] in which the authors developed a sound and complete algorithm to extract all minimal unsatisfiable subsets (MUSs) of unsatisfiable constraint systems. This approach has two significant properties. First of all, it extracts *all* MUSs. Determining one MUS is not always sufficient because there might exist many MUSs in an unsatisfiable constraint systems, and after removing one of the reasons for unsatisfiability, the system can still be infeasible due to remaining MUSs. Secondly, this approach provides unsatisfiable subsets that are *minimal*. Previous methods provided only unsatisfiable subsets (USs) by applying resolution based approaches. Optimally, USs should be minimal cores so that they can be sufficient core explanations for unsatisfiability.

In this approach, extraction of all MUSs is achieved based on relationship between *correction subsets* and *unsatisfiable subsets* of infeasible constraint systems. Initially, we determine minimal correction subsets (MCSs) of infeasible constraint systems that means if one of the MCSs is removed from an infeasible constraint system, the remaining constraint system becomes satisfiable. Then, our approach extracts MUSs of the system through MCSs. Hence, our complete and sound algorithm has two phases. In the first phase, we provide MCSs, then we extract MUSs in the second phase.

2 Preliminaries

2.1 Boolean Satisfiability and CNF

Our approach can be implemented for any type of constraint satisfaction problems, but we create an example for Boolean satisfiability in order to present and explain our approach and algorithms. Boolean satisfiability problems are Boolean formulas in *Conjunctive Normal Form* (*CNF*). A formula in CNF is expressed in the following way:

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

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

Thus formula in CNF is a conjunction of disjunctions, where the disjunctions are called *clauses* C_i, where n is the number of clauses in the formula. Each clause is a disjunctions of *literals* a_{ij}, where k_i is the number of literals in the clause C_i. Each literal a_{ij} is either a positive or a negative(i.e., negated) Boolean variable which can have two values either 1 (TRUE) or 0 (FALSE). A CNF formula is *satisfiable* (TRUE) if some assignments of Boolean values to its variables exists, such that substituting those values for the variables evaluate the formula to true, otherwise a CNF formula is unsatisfiable that means there is no assignment that satisfies it.

In this paper, we use the following unsatisfiable CNF formula φ that has four clauses (C_1, C_2, C_3, C_4) to describe and explain our algorithm:

$$\begin{array}{ccccccccc} & & C_1 & & C_2 & & C_3 & & C_4 \\ \varphi & = & (x_1) & \wedge & (\neg x_1) & \wedge & (\neg x_1 \vee x_2) & \wedge & (\neg x_2) \end{array}$$

2.2 Clause-Selector Variables

Clause-selector variables are used in first phase of our algorithm for the extraction of MCSs. They are placed inside of each clause of a CNF formula φ, generating a new CNF formula φ'. Those variables are used to allow some of the clauses to be violated, i.e., to virtually remove some of the clauses from a formula.

A clause-selector variable y_i is placed inside every clause C_i of a CNF formula φ as a negated variable to produce a new CNF formula φ'. If y_i is assigned TRUE, i.e., $\neg y_i$ is FALSE in the clause C_i, then C_i must be satisfied. Otherwise, if y_i is assigned FALSE which makes $\neg y_i$ TRUE in the clause C_i, so y_i makes C_i TRUE because one TRUE literal makes a clause satisfied. This way we can enable and disable any clause C_i in φ, without modifying the SAT solver's internal algorithm. In our example, the new CNF formula φ' that is generated from φ by placing clause-selector variables y_i is shown in the following:

$$\varphi' = (\neg y_1 \vee x_1) \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2)$$

2.3 AtMost Constraints

AtMost constraint is applied in the first phase of our algorithm to generate a counting-based constraint. For a given formula φ, AtMost constraint will be used to limit the number of false clause-selector variables in φ'. It is expressed as:

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

where $\{l_1, l_2, \ldots, l_n\}$ are n the literals and k is a positive integer. An AtMost constraint puts an upper bound on number of literals assigned TRUE, in other words, at most k number of literals in the set $\{l_1, l_2, \ldots, l_n\}$ can be assigned TRUE.

2.4 Hitting Sets

Let in the following $\Omega = \{S_1, \ldots, S_k\}$ be a collection of non-empty sets $S_i \subseteq D$ from a finite domain D. A hitting set H of Ω consists of elements from D such that at least one element is contained from each set $S_i \in \Omega$. It is defined as followings:

Definition 2.1 *A hitting set H of Ω is a set $H \subseteq D$ such that $\forall S \in \Omega$, $H \cap S \neq \emptyset$.*

In our algorithm, we need *minimal* hitting sets in the sense that those are irreducible. If any element of a hitting set H is removed, the hitting set property does not hold anymore. In other words, there exists no subset hitting set H' of H such that H' has at least one element common for all sets S. Formally:

Definition 2.2 [*1*] *A hitting minimal set H of Ω is a set $H \subseteq D$ such that $\forall S \in \Omega$, $H \cap S \neq \emptyset \wedge \nexists H' \subset H : H' \cap S \neq \emptyset$.*

Let us assume that S = {{1,2,3}, {2,5}}, then some hitting sets of S are {2}, {1,5}, {3,5}, {2,5}, but only {2}, {1,5}, {3,5} are minimal hitting sets of S (i.e., irreducible). {2,5} is not a minimal hitting set as it is reducible because its element 5 can be removed from set and the remaining set {2} is still hitting all sets of S. Notice that we are considering determining minimal hitting sets, not the *minimum* (smallest) hitting sets that is {2} in this example.

2.5 Useful Subsets of Constraint Systems

Our two-phased algorithm is based on a relationship between *Minimal Unsatisfiable Subsets* (MUSs) and *Minimal Correction Subsets* (MCSs).

MCSs are minimal subsets of an unsatisfiable constraint system such that if any MCS is removed from infeasible constraint system, the remaining set of constraints becomes satisfiable. This satisfiable set of constraints is called a *Maximal Satisfiable Subset* (MSS), i.e. any MCS is the complement of some MSS. MCSs and MSSs of a given infeasible constraint system C are defined as follows:

Definition 2.3 *A subset $M \subseteq C$ of an unsatisfiable constraint system C is an MCS if $C \setminus M$ is satisfiable and $\forall C_i \in M$, $C \setminus (M \setminus \{C_i\})$ is unsatisfiable.*

Definition 2.4 *A subset $S \subseteq C$ of an unsatisfiable constraint system C is an MSS if S is satisfiable and $\forall C_i \in (C \setminus S)$, $S \cup \{C_i\}$ is unsatisfiable.*

MUSs are irreducible in the sense that removing any element results in a satisfiable set. Notice that we aim at obtaining *minimal unsatisfiable subsets* (MUSs), not *minimum (smallest) unsatisfiable subsets* of infeasible constraint systems. Formally, MUSs and minimum unsatisfiable subsets C are defined as follows:

Definition 2.5 *A subset $U \subseteq C$ of an unsatisfiable constraint system C is an MUS if U is unsatisfiable and $\forall C_i \in U$, $U \setminus \{C_i\}$ is satisfiable.*

Definition 2.6 *[6] Let $\{U_1, U_2, \ldots, U_j\}$ be the set of all unsatisfiable subsets of an unsatisfiable constraint system C.Then, $U_k \in \{U_1, U_2, \ldots, U_j\}$ is a minimum unsatisfiable subset for C iff $\forall U_i \in \{U_1, U_2, \ldots, U_j\}$, $0 \leq i \leq j$: $|U_i| \geq |U_k|$.*

In Table 1 and Table 2, you can see MUSs, MCSs and MSSs of our example CNF formula φ. Note that the minimum unsatisfiable core is $\{C_1, C_2\}$ whose cardinality is smallest.

Table 1 MUSs of φ

MUSs(φ)	Clauses
$\{C_1, C_2\}$	$\{(x_1), (\neg x_1)\}$
$\{C_1, C_3, C_4\}$	$\{(x_1), (\neg x_1 \vee x_2), (\neg x_2)\}$

Table 2 MCSs and MSSs of φ

MCSs(φ)	MSSs(φ)
$\{C_1\}$	$\{C_2, C_3, C_4\}$
$\{C_2, C_3\}$	$\{C_1, C_4\}$
$\{C_2, C_4\}$	$\{C_1, C_3\}$

3 Computation of All Minimal Infeasible Subsets

In this section, we describe the two-phase algorithm to compute all minimal unsatisfiable subsets of an infeasible constraint system. Afterwards, we show the implementation of the algorithm and illustrate its application on our Boolean satisfaction problem φ.

3.1 MUS/MCS Duality

As already mentioned, the two-phase algorithm is based on a relationship between MCSs and MUSs of an infeasible constraint system. MCSs and MUSs are related to each other in the aspect of hitting set duality: MCSs are hitting sets of MUSs, and vice versa. Formally, hitting set duality between MUSs and MCSs is described in the following theorem:

Theorem 3.1 *Assume an unsatisfiable constraint system C.*

1. *A subset M of C is an MCS of C iff M is an irreducible hitting set of the MUSs(C) of all minimal unsatisfiable subsets of C;*

2. *A subset U of C is an MUS of C iff U is an irreducible hitting set of the MCSs(C) of all minimal correction sets of C.*

Proof 3.1 *The proof can be found in [3].*

The hitting set duality between MCSs and MUSs is consequence of minimality. Let assume we have an MUS of a given infeasible constraint system C. The nature of minimality of MUSs implies that removing any element from an MUS results in a satisfiable set. Therefore, if we remove at least one element from each MUSs of C, the remaining part of each MUS becomes satisfiable, i.e., removing at least one element from each MUS and aggregating those elements form a hitting set for the MUSs. Therefore, removing a minimal hitting set of MUSs from C results in a satisfiable constraint system. This satisfiable constraint system is an MSS. As a result of that, all MCSs of an infeasible constraint system C are all irreducible hitting sets of MUSs.

In our example φ, you can see MUSs and MCSs of φ in Table 1 and 2. By using MUS/MCS duality, minimal hitting sets of MCSs are $\{C_1, C_2\}$ and $\{C_1, C_3, C_4\}$ that are also MUSs of φ displayed in Table 1. Likewise, hitting sets of MUSs in Table 1 can be seen as MCSs in Table 2.

3.2 Our Approach

In the algorithm, we benefit from this hitting set duality argument explained in the previous Section 3.1. to extract all MUSs of an infeasible constraint system by obtaining all irreducible hitting sets of MCSs. In order to accomplish that, we first need to determine MCSs of infeasible constraint systems that is easier than determining MUSs. After that, the second phase of the algorithm provides all irreducible hitting sets of MCSs, i.e., all MUSs, by applying a recursive branching algorithm to MCSs obtained in the first phase.

3.3 Computing MCSs

In this section, we describe and explain the first phase of the algorithm that finds all MCSs of an unsatisfiable constraint system C. The examples used to explain the steps of the algorithm are based on our Boolean satisfiability problem φ.

Finding MCS of C is based on finding minimal subsets of C whose exclusion makes the remaining constraint system satisfiable, i.e. the remaining satisfiable constraint system is an MSS. In an iterative procedure, maximal satisfiable subsets (MSSs) of decreasing size are found by a solver until all of them are found. These can then be used to determine all MCSs.

In the algorithm for computing MCSs, there are two important points that should be ensured. The first important point is that in each iteration the algorithm should not find any MSS that has been already found in a previous iteration. The second important point is that the algorithm should find only correction sets that are minimal.

By noticing these important points, the iterative algorithm finds MCSs in sense of minimal and not duplicate correction sets, i.e. finds MSSs as complements of MCSs. The pseudocode algorithm used to find MCSs of φ is shown in Algorithm 1.

In the algorithm, initially φ' is generated by inserting clause-selector variables to φ as described Section 2.2. There is a positive integer k which is iteratively incremented to assure that the algorithm iteratively finds MSSs of φ in decreasing-size order, i.e., larger MSSs first. Thereby, k denotes the number of clauses that are removed from φ to achieve MSSs. Starting with $k = 1$, the algorithm iteratively finds all satisfiable subsets of size

Algorithm 1 Algorithm for finding all MCSs of a formula φ

```
MCSs($\varphi$)
 1  $\varphi' \leftarrow$ AddYVars($\varphi$)
 2  $MCSs \leftarrow \emptyset$
 3  $k \leftarrow 1$
 4  while SAT($\varphi'$)
 5  do $\varphi'_k \leftarrow \varphi' \wedge$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k$)
 6      while $newMSC \leftarrow$ IncrementalSAT($\varphi'_k$)
 7      do $MCSs \leftarrow MCSs \cup \{newMCS\}$
 8          $\varphi'_k \leftarrow \varphi'_k \wedge$ BlockingClause($newMCS$)
 9          $\varphi' \leftarrow \varphi'_k \wedge$ BlockingClause($newMCS$)
10      $k \leftarrow k + 1$
11  return $MCSs$
```

$|\varphi| - k$ of φ' in the while loop (lines 4-11) until all of them are found. In this while loop, the formula φ' is combined with an **AtMost** constraint that produces φ'_k to ensure that the algorithm finds only MSSs of size $|\varphi| - k$, which are dual to MCSs of size k. For example, in the first iteration, where $k = 1$, it finds all MCSs that consist of a single clause.

Afterwards, for each k value,the algorithm uses in **IncrementalSAT** a solver with a standard backtracking search algorithm to find satisfiable subsets of size of k in the inner while loop (lines 6-9). If the AtMost constraint is satisfied, it states that at most k of the y_i are assigned FALSE and thus at most k clauses are removed to achieve a MSS. The clauses that have y_i assigned FALSE become an MCS.

After finding an MCS, there are two important points to ensure: minimality and not producing same MCS. Firstly, the algorithm should not find the same MCS in the next iterations. In order to prevent that, **BlockingClause** adds the disjunction of the y_i variables that are FALSE as a new clause to φ'_k. For instance, one of MCSs found for $k = 2$ for our example is C_2 and C_3 whose y_i variables y_2 and y_3 are added to φ'_k as $(y_2 \vee y_3)$, so at least one of them must be TRUE in this as well as in the next iterations. Firstly, it assures that the same MCS $\{C_2, C_3\}$ will not be found again in this iteration $k = 2$. Secondly, the algorithm should not produce MCS that are reducible, in other words, in the next k value iterations supersets of any MCS which has been already found should not be extracted. Therefore, **BlockingClause** constraint with same y_i is also added to φ' to ensure MCS are minimal also for all later $k - values$. According to the last explained example, $(y_2 \vee y_3)$ is added to φ'_k, assuring that no supersets of $\{C_2, C_3\}$ will be found for any $k > 2$ as MCS.

Note that also beginning the algorithm with $k = 1$ and incrementing k consecutively are significant. To assure that minimal MCSs are extracted. Otherwise, the algorithm would not start searching subsets, so subsets would not be blocked for next iterations and minimality would not be maintained. Consider that we start execution of the first phase of the algorithm on our example with $k = 2$. In the first iteration, C_1 and C_2 can be obtained as an MCS since assigning false to y_1 and y_2 satisfies the **AtMost** constraint for $k = 2$ and the formula φ'_2 is satisfiable. However, $\{C_1, C_2\}$ is not a minimal correction set since C_2 is redundant which results from $\{C_1\}$ is not blocked as an MCS in previous iterations. So, incrementing k consecutively in the algorithm and starting searching subsets are significant in sense of minimality.

The algorithm terminates if any satisfiable subset of φ cannot be found in the sense

Algorithm 2 Algorithm for altering **MCSs** to make the choice of **thisClause** irredundant as the only element hitting **thisMCS**

```
PropagateChoice($MCSs, thisClause, thisMCS$)
for each $clause \in thisMCS$
do for each $testMCS \in MCSs$
      do if $clause \in testMCS$
            then $testMCS \leftarrow testMCS - \{clause\}$
for each $testMCS \in MCSs$
do if $thisClause \in testMCS$
      then $MCSs \leftarrow MCSs - \{testMCS\}$
MaintainNoSupersets($MCSs$)
```

that, the condition of the first while loop cannot be satisfied that indicates algorithm finds all MCSs of φ.

3.4 Computing MUSs

As mentioned above, extraction of all MUSs is achieved by extracting all irreducible hitting sets of all MCSs. In order to fulfill that, the second phase of the algorithm is described in this section. The second phase of the algorithm operates one main algorithm to find all MUSs, and benefits from a subroutine to compute a single MUS. The main algorithm continues until all MUSs are computed. Therefore, we first explain how to compute a single MUS, then how to utilize a single MUS computation for finding all MUSs.

3.4.1 Computing a Single MUS

Computation of all MUSs is realized by computing all irreducible hitting sets of MCSs. It means that for a given collection of MCSs, we need to compute a set of clauses that hits each MCS and is irreducible. In order to hit each MCSs, the easiest approach can be iteratively selecting one MCSs that has not been hit yet, then hitting it by one of its clauses. This continues until all MCSs are hit. Although this approach hits all MCSs, it does not assure that the computed hitting set is irreducible. Let's utilize MCSs of our example $\{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$. None of MCSs are hit, so let's hit $\{C_1\}$ with C_1, $\{C_2, C_3\}$ with C_3 and $\{C_2, C_4\}$ with C_2. MUS of this example is $\{C_1, C_2, C_3\}$, but this MUS is reducible because C_2 hits both $\{C_2, C_3\}$ and $\{C_2, C_4\}$. As seen on this example, this approach should be improved to ensure being irreducible.

Algorithm 3 Algorithm for computing a single MUS from a set of MCSs

```
SingleMUS($MCSs$)
$MUS \leftarrow \emptyset$
while $MCSs \neq \emptyset$
do $selClause \leftarrow$ SelectRemainingClause($MCSs$)
   $selMCS \leftarrow$ SelectMCSContaining($MCSs, selClause$)
   $MUS \leftarrow MUS \cup \{selClause\}$
   PropogateChoice($MCSs, selClause, selMCS$)
return $MUS$
```

Irreducibility can be assured by modifying this approach. The main problem of this approach is redundant clauses, in other words a particular MCS is hit by more than one clause and one of them is redundant. To avoid having redundant clauses, when a clause hits a particular MCS that has not been hit yet, the other MCSs that have this clause are removed from all MCSs since they are also hit by this clause, and the remaining clauses of this particular MCS are removed from all other MCSs that include them, so the remaining clauses this particular MCS cannot be put in the MUS, i.e. they cannot hit another MCSs anymore, because they are removed. Let's imply this improved approach on our example $\{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$. Let's hit $\{C_1\}$ with C_1, then $\{C_1\}$ is removed from MCSs and there is no remaining clause in $\{C_1\}$ to be removed from other MCSs, so select again another MCS $\{C_2, C_3\}$, and hit with C_3, again there is no other MCS that contain C_3, but there is a remaining clause C_2. From all other MCSs that have C_2 we remove C_2, so from $\{C_2, C_4\}$ removing C_2 turns it into $\{C_4\}$ as altered MCSs which comprise MCSs that has not been hit. Hence, there is no chance for C_2 to be representative of another MCS. Finally, $\{C_4\}$ is hit with C_4, and there is no MCS that has not been hit, so a single MUS that hits all MCSs and irreducible is computed.

This approach computing a single MUS from all MCSs is described in Algorithm 3. Algorithm 3 generates a single MUS until all MCSs are hit by a clause and maintains to be irreducible. To maintain being irreducible, it utilizes the **PropagateChoice** subroutine as described in Algorithm 2. In Algorithm 2, lines 1-4 remove remaining clauses of `thisMCS` from MCSs to prevent them to hit any other MCSs. Lines 5-7 remove all MCSs that include `thisClause` because they are represented by this clause. Also, line 8 calls a subroutine **MaintainNoSupersets** to prevent having a superset in altered MCSs, because MCSs is defined as any set cannot be a superset of the other set without being reducible. Assume that we have a set of MCSs $\{\{C_1, C_2\}, \{C_2, C_3\}, \{C_3, C_4\}\}$ is `thisMCSs` and $\{C_1\}$ is `thisClause`. When **PropagateChoice** is executed, it alters MCSs to $\{\{C_3\}, \{C_3, C_4\}\}$ where $\{C_3, C_4\}$ is a superset of $\{C_3\}$, so $\{C_3, C_4\}$ is removed and the final altered MCSs is $\{\{C_3\}\}$. **PropagateChoice** returns the altered MCSs to Algorithm 3 which keeps being irreducible. Algorithm 3 checks altered MCSs that are not yet empty. If an altered MCS is not empty, i.e. it has not been hit yet, we continue execution of the steps above until there is MCS to be hit (i.e., all altered MCSs are empty). When all altered MCSs are empty, the algorithm terminates with the computed single MUS.

Figure 1 illustrates the application of this algorithm to our example from Section 2.1.

3.4.2 Computing All MUSs

At this point, we already know how to compute a single MUS as an irreducible hitting set of MCSs, but our aim is computing all MUSs. Hence, the second phase of the algorithm computes all MUSs with the help of the algorithm for computing a single MUS as described above. There are two important decision points when computing a single MUS algorithm:

- Selection of the clause set MCS that would be hit is arbitrary.
- Selection of the clause in a particular MCS that would hit the MCS is arbitrary.

Hence, all MUSs result from all different possible decision how to select these clauses. The algorithm should include all different decisions, so it can be accomplished generating a recursive algorithm which branches all possible different selections of clauses.

The second phase of algorithm is described in Algorithm 4. It starts with the set of all MCSs and an empty set of MUSs. In each iteration, it arbitrarily selects a clause on line 4, then the algorithm branches for each MCS that owns this clause. This means the

Algorithm 4 Algorithm for computing the complete set of MUSs from a set of MCSs

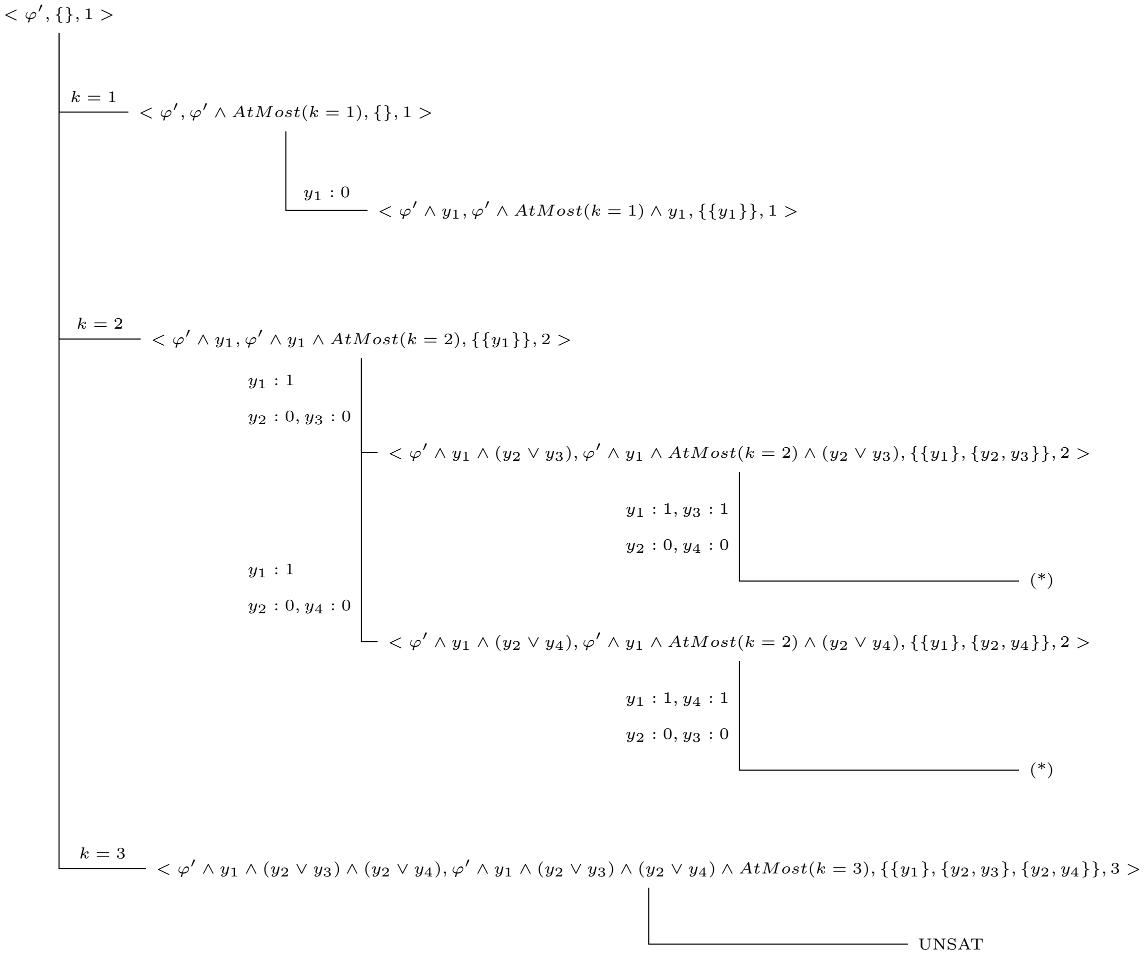
```
AllMUSs($MCSs, currentMUS$)
 1  if $MCSs = \emptyset$
 2    then print($currentMUS$)
 3           return
 4  for each $selClause \in$ RemainingClause($MCSs$)
 5  do $newMUS \leftarrow currentMUS \cup selClause$
 6     for each $selMCS \in MCSs$ such that $selClause \in selMCS$
 7     do $newMCSs \leftarrow MCSs$
 8        PropogateChoice($newMCSs, selClause, selMCS$)
 9        AllMUSs($newMCSs, newMUS$)
10  return
```

algorithm computes MUS for all MCSs which can be hit by this clause (line 6). After that, the **PropogateChoice** subroutine is called on line 8 for altering set of MCSs to maintain irreducibility. The algorithm considers the altered MCSs that **PropogateChoice** returns as a new set of MCSs, and it calls itself again to hit this new set of MCSs on line 9. The recent recursive call starts applying the same steps and it continues recursively branching until an empty set of MCSs remains to be hit. When an empty set of MCSs is obtained, it prints the computed MUS and turns back to the point at the last recursive call on lines 1-3 to find another MUS. This procedure is repeated for all possible MCS and clause selections. As a result of these different selections, same MUSs can be printed out many times. There are many clause sets that have a same clause, so the same MUS can be printed out for all of them. For example, in our example $\{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}$, a MUS $\{C_1, C_2\}$ is printed out multiple times because both $\{C_2, C_3\}, \{C_2, C_4\}$ have C_2 and the algorithm operates both of them as selected MCS that leads to same MUS $\{C_1, C_2\}$. Since the computation of the same MUS more than once is inefficient, optimizations are needed. Some optimizations were proposed, but they are not explained in this paper.

Figure 2 shows the search tree for application of this algorithm to our example CNF formula introduced in Section 2.1.

4 Conclusion

In this paper, an algorithm to compute all minimal unsatisfiable subsets (MUSs) of an infeasible constraint system is explained. The algorithm is based on the hitting set duality between *minimal unsatisfiable subsets* (*MUSs*) and *minimal correction sets* (*MCSs*). It is a two-phase algorithm whose first phase extracts all MCSs of an infeasible constraint system and its second phase computes all MUSs from those MCSs. This two-phase algorithm is valuable because it finds all MUSs, but it has an inefficient second phase. Execution of the second phase obtains duplicated MUSs which cause inefficiency. To reduce inefficiency, some optimizations are developed but these optimizations are not described in this paper.


$< \varphi', \{\}, 1 >$
$k = 1$
$< \varphi', \varphi' \wedge AtMost(k = 1), \{\}, 1 >$
$y_1 : 0$
$< \varphi' \wedge y_1, \varphi' \wedge AtMost(k = 1) \wedge y_1, \{\{y_1\}\}, 1 >$
$k = 2$
$< \varphi' \wedge y_1, \varphi' \wedge y_1 \wedge AtMost(k = 2), \{\{y_1\}\}, 2 >$
$y_1 : 1$
$y_2 : 0, y_3 : 0$
$< \varphi' \wedge y_1 \wedge (y_2 \vee y_3), \varphi' \wedge y_1 \wedge AtMost(k = 2) \wedge (y_2 \vee y_3), \{\{y_1\}, \{y_2, y_3\}\}, 2 >$
$y_1 : 1, y_3 : 1$
$y_2 : 0, y_4 : 0$
(*)
$y_1 : 1$
$y_2 : 0, y_4 : 0$
$< \varphi' \wedge y_1 \wedge (y_2 \vee y_4), \varphi' \wedge y_1 \wedge AtMost(k = 2) \wedge (y_2 \vee y_4), \{\{y_1\}, \{y_2, y_4\}\}, 2 >$
$y_1 : 1, y_4 : 1$
$y_2 : 0, y_3 : 0$
(*)
$k = 3$
$< \varphi' \wedge y_1 \wedge (y_2 \vee y_3) \wedge (y_2 \vee y_4), \varphi' \wedge y_1 \wedge (y_2 \vee y_3) \wedge (y_2 \vee y_4) \wedge AtMost(k = 3), \{\{y_1\}, \{y_2, y_3\}, \{y_2, y_4\}\}, 3 >$
UNSAT

(*) $< \varphi' \wedge y_1 \wedge (y_2 \vee y_3) \wedge (y_2 \vee y_4), \varphi' \wedge y_1 \wedge AtMost(k = 2) \wedge (y_2 \vee y_3) \wedge (y_2 \vee y_4), \{\{y_1\}, \{y_2, y_3\}, \{y_2, y_4\}\}, 2 >$

Figure 1: Computation of MCSs(φ) on our example,describing φ', φ'_k, the current MCS set and k at different execution points

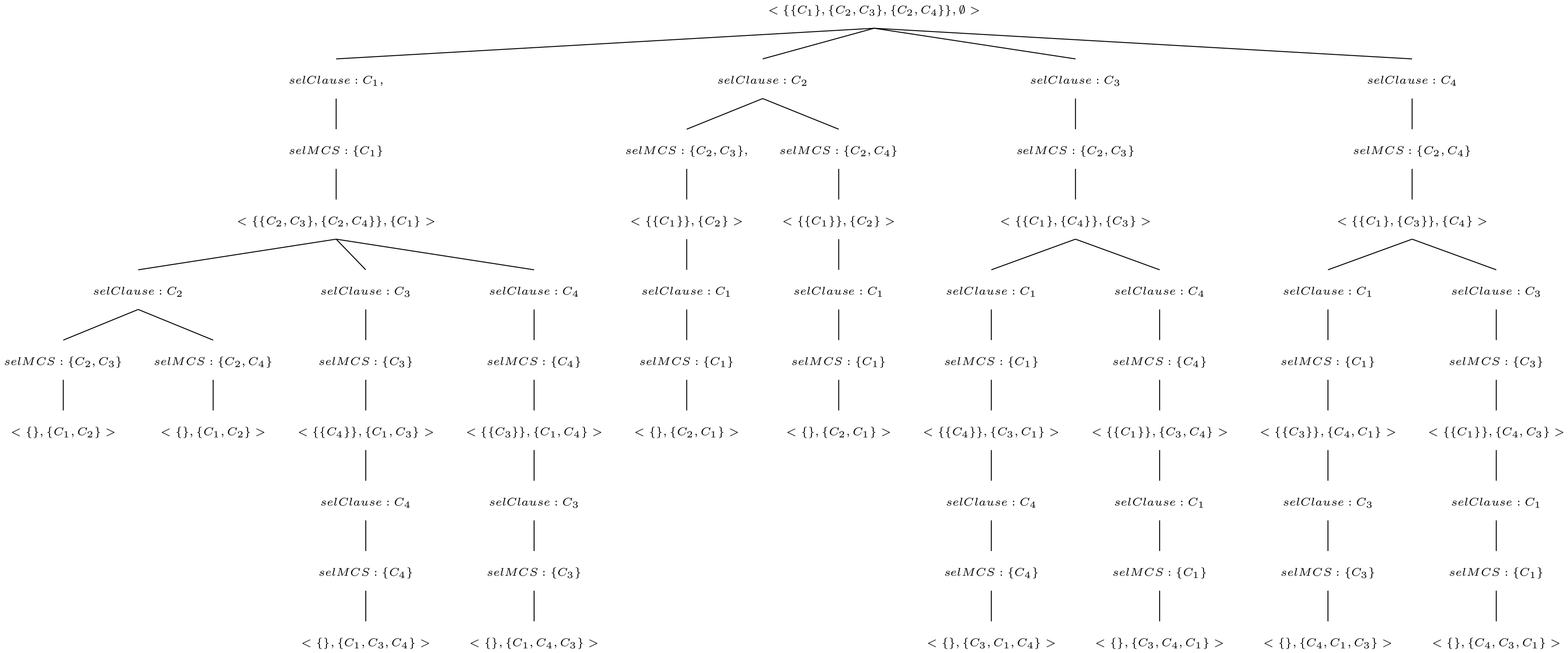


Figure 2: All MUSs of our example with all possible branches

References

[1] R. Abreu and A. J. Van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *SARA*, volume 9, pages 2–9, 2009.

[2] F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Proc. of International Conference on Computer Aided Verification*, pages 70–86. Springer, 2015.

[3] E. Birnbaum and E. L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental & Theoretical Artificial Intelligence*, 15(1):25–46, 2003.

[4] M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, pages 1–31, 2012.

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

[6] I. Lynce and J. P. Marques-Silva. On computing minimum unsatisfiable cores. 2004.