FMUS2: An Efficient Algorithm to Compute Minimal Unsatisfiable Subsets

Shaofan Liu and Jie Luo(✉)

State Key Laboratory of Software Development Environment,
School of Computer Science and Engineering, Beihang University,
Beijing 100191, China
{shaofanliu,luojie}@nlsde.buaa.edu.cn

Abstract. In the past few years, much attention has been given to the problem of finding Minimal Unsatisfiable Subsets (MUSes), not only for its theoretical importance but also for its wide range of practical applications, including software testing, hardware verification and knowledge-based validation. In this paper, we propose an algorithm for extracting all MUSes for formulas in the field of propositional logic and the function-free and equality-free fragment of first-order logic. This algorithm extends earlier work, but some changes have been made and a number of optimization strategies have been proposed to improve its efficiency. Experimental results show that our algorithm performs well on many industrial and generated instances, and the strategies adopted can indeed improve the efficiency of our algorithm.

Keywords: Minimal unsatisfiable subsets · Heuristic algorithm
Optimization strategy · SAT

1 Introduction

Given an unsatisfiable formula in Conjunctive Normal Form (CNF), a minimal unsatisfiable subset (MUS) is a subset of clauses which is (1) unsatisfiable, and (2) minimal, which means removing any one of its elements will make the remaining set satisfiable. Different classes of algorithms have been proposed to efficiently enumerate all or partial MUSes [1,16,19]. Early algorithms are based on subset enumeration [3,8]. In these algorithms, the power set of the input is enumerated in a tree structure and every subset is checked for satisfiability. A MUS can be easily identified by definition. Another class of algorithms [2,12,17] relies on the *hitting set duality*. First, all Minimal Correction Subsets (MCSes) are computed. Then, all MUSes are obtained by computing minimal hitting sets of these MCSes. CAMUS [12] is one of the state-of-the-art algorithms for computing all MUSes in this class. Recently, algorithms (e.g. eMUS/MARCO [11,14]) for partial MUS enumeration were proposed. These algorithms are able to produce the first MUS quickly and early, and the following MUSes are generally produced incrementally.

J. Fleuriot et al. (Eds.): AISC 2018, LNAI 11110, pp. 104–118, 2018.
https://doi.org/10.1007/978-3-319-99957-9_7

Most of the current algorithms rely on a SAT solver for checking the satisfiability of clause sets. The advantage is that they can utilize the power of highly optimized SAT solvers. But they also unavoidably introduce many duplicated computations. For example, if clause set $\{1, 2, 3\}$ is checked unsatisfiable, they should check the satisfiability of $\{1, 2\}, \{1, 3\}, \{2, 3\}$ for determining whether $\{1, 2, 3\}$ is indeed a MUS or not. Although many optimizations (e.g. using the hitting set duality) for these algorithms are proposed to reduce the number of SAT solver calls, there are still many duplicated computations. And when there are a larger number of MUSes in the input, the number of SAT solver calls will be enormous and the time used for duplicated computations will also be obviously large, which will cause a decrease in efficiency.

For the consideration of the shortcoming described above for those algorithms which are based on SAT solvers, we have adopted another approach for enumerating MUSes. This paper extends our earlier work [20] on computing MUSes for a decidable fragment of First-Order Formulas (FOL), and its main contributions can be summarized as follows. First, in contrast to most approaches which make use of variable assignments or an external SAT-solver to check satisfiability, this paper proposes a "decompose-merge" algorithm inspired by the process of logical deduction in belief revision [10,13]. It first decomposes clauses of the given formula into literals to easily identify all inconsistent relations between them, and then assembles all literals back to the original clauses to reveal the minimal inconsistent relations among them. Second, the proposed algorithm uses unification to accomplish "general instantiation". In other words, instead of instantiating all variables by all feasible values, a most general inconsistent subset is used to represent a class of instances which are equivalent under the *more general* relation, which can avoid generating of excessive instances and reduce the searching space. Another contribution of the paper is the optimization strategies used to improve the efficiency of our algorithm. Experimental results show that our algorithm is competitive and has the potential to be even better.

2 Preliminaries

This paper focuses on the function-free and equality-free fragment of first-order logic (FEF for short). Satisfiability of formulae from the FEF fragment is decidable, because it is a special case of effectively propositional logic (EPR), also known as the Bernays-Schönfinkel class [15] which is proved to be decidable. Hence it is feasible to design an algorithm to compute all MUSes in the FEF fragment.

Formulas in FEF are represented in CNF. That is, a CNF formula is a conjunction (AND, $\wedge$) of one or more clauses, and each clause is a disjunction (OR, $\vee$) of one or more literals. A literal is an atomic formula or its negation (NOT, $\neg$). The syntax is shown below.

$$\begin{aligned} F &::= C^1 \wedge \cdots \wedge C^n \\ C^i &::= L^1 \vee \cdots \vee L^{m_i} \\ L^j &::= A \mid \neg A \end{aligned}$$

Following the convention of many other papers (e.g. [4,7]), a CNF formula is treated as a (finite) set of clauses.

Here is an example formula in the FEF fragment.

Example 1. The uppercase letter X denotes a variable, while the lowercase letter a and b denote constants.

$$F = \overset{C^1}{(A(a))} \wedge \overset{C^2}{(\neg A(X) \vee B(X))} \wedge \overset{C^3}{(\neg B(b))} \wedge \overset{C^4}{(\neg B(a))}$$

3 Algorithm for Computing All MUSes

In this section, we will give an overview of the proposed FMUS2 algorithm for computing all MUSes for formulas in the FEF fragment, which is an improved version of our previous FMUS algorithm [20].

Both FMUS2 and FMUS adopt a constructive "decompose-merge" approach to compute MUSes. First, the clauses of the given formula are decomposed into literals and inconsistent pairs of decomposed literals are all computed, this is the "decompose" procedure. Thus, the initial intermediate results are created, which are sets of literals and indicate the contradictory relations among literals of all clauses. Then, by iteratively merging these intermediate results into larger sets, which are still unsatisfiable during the whole process, the original clauses are restored one by one, this is the "merge" procedure. The merging operation processes literal by literal and clause by clause. After all the literals are merged into original clauses, the final results will contain all MUSes of the clauses in the input formula.

FMUS2 and FMUS both use unification for instantiating clauses with the *most general unifier*, but through different approaches.

Definition 1 (Most general unifier). *A substitution σ is a most general unifier (MGU) of two literals L_1 and L_2 if σ unifies them, i.e. $(L_1, \sigma) = (L_2, \sigma)$, and for any unifier σ' of these two formulas, there exists a substitution ω such that $\sigma' = \omega \circ \sigma$.*

For FMUS, MGUs are kept along with the whole procedure. There will be a MGU for each intermediate result, which indicates how this intermediate result is unsatisfiable. For example, $I = \{A(a), \neg A(X)[a/X]\}$ is unsatisfiable if we substitute the constant a for the variable X. In other words, FMUS uses MGUs instead of explicit instantiation. The implicit way can cause difficulty for identifying whether two substitutions, which look different, are in fact equivalent sometimes. For example, if there are $I_1 = \{A(Y), \neg A(X)[Y/X]\}$ and $I_2 = \{A(Z), \neg A(X)[Z/X]\}$ among all intermediate results, they are equivalent when the variables Y and Z are substituted by the same constant a, but they are different when Y is substituted by a and Z is substituted by b. When the input formula is complex, the identification will be difficult. Some redundant branches will arise also because of its implicity.

So for FMUS2, we have tried to adopt a new way to solve this problem. We choose to explicitly instantiate the original clauses with ground term (i.e. terms without variables). Before the instantiation, MGUs of decomposed literals are computed, which will be used to confine the scope of instantiation and reduce the number of instantiations. For example, if the variable X from $\neg A(X)$ can be substituted by constants a, b, c but only substitute the constant a for the variable X can lead to contradiction, there is no need for replacing X with b or c. Thus the scope of instantiation is confined.

Based on the discussion above, the basic steps of FMUS2 are listed below.

1. **Preprocess.** For the given CNF formula, FMUS2 first parses and decomposes clauses of the CNF formula into literals with labels to indicate their origin, meanwhile overlapping bound variables are renamed to eliminate name ambiguity.
2. **Find initial contradictions.** For each decomposed literal it is checked whether there is another literal which is contradictory to it. This process is accomplished by unification to obtain a MGU.
3. **Instantiation.** If there are variables in the given CNF formula, literals will be instantiated and the MGUs already found will be used to confine the scope of instantiation. After instantiation, the previous step of finding initial contradictions will be processed again for these ground literals in newly instantiated clauses. If there is no variable in the given CNF formula, which means the formula is a ground formula, there is no need for instantiation.
4. **Merge.** After all steps above, the core process of FMUS2—the merge process begins. Literal instances of the same clause are merged to reconstruct instances of their original clause according to certain order, which will be further discussed in Sect. 4.1. The principle for deciding whether two intermediate results can be merged will be discussed in Sect. 4.1 too.
5. **Map back.** If the original CNF formula is a ground formula, then the result is all MUSes of the input. But if the original CNF formula contains variables, one original clause may have many corresponding clause instances. Thus after all steps above, the instance sets need to be mapped back into unsatisfiable subsets of the original clause set. Then all MUSes of the input can be obtained by extracting the minimal ones from the set of all those unsatisfiable subsets.

The pseudo-code for FMUS2 is shown in Algorithm 1.

FMUS2 takes a set of clauses (a CNF formula) F as input, and outputs all MUSes of F. If F is satisfiable, the output will be $\emptyset$. Lines 1–4 demonstrate the process of decomposing clauses into literals. Every clause C^i in F is $L_1^i \vee \cdots \vee L_{m_i}^i$ where m_i stands for the number of literals in C^i. Lines 5 enumerates all inconsistent pairs among decomposed literals to construct M_0. Lines 6–9 show, if F is in the field of first-order logic, all literals will be instantiated and the initial contradictions set M_0 will be computed again for L'.

The loop in lines 10–16 of FMUS2 is the most interesting but bewildering part. In this loop, we iteratively merge clauses that contain multiple literals. In each iteration, literals from a certain clause are merged to the original form and the unsatisfiable subsets that contain these literals are merged to larger

Algorithm 1. FMUS2(F)

Input: F as a set of clauses $\{C^1, \ldots, C^n\}$
Output: The set of all MUSes of F
1: **for** $i = 1$ **to** n **do**
2: Decompose C^i to $\{L^i_1, \ldots, L^i_{m_i}\}$
3: **end for**
4: $L := \bigcup_{i=1}^{n} \{L^i_1, \ldots, L^i_{m_i}\}$
5: Find the initial inconsistent set M_0 of L
6: **if** there are variables in L **then**
7: Instantiate $L' := L$
8: Find the inconsistent set M_0 of L'
9: **end if**
10: **for** $i = 1$ **to** n **do**
11: **if** $m_i > 1$ **then**
12: $M_i := \text{Merge}(i, M_{i-1})$
13: **else**
14: $M_i := M_{i-1}$
15: **end if**
16: **end for**
17: Map instances in M_n back to their corresponding clauses and obtain M'_n
18: **return** M'_n

unsatisfiable sets. Each round of iteration is based on the result of the previous iteration. To give a clearer explanation, let us suppose that the ith clause (i.e. C^i) is going to be merged and M_{i-1} is the result of the last iteration. So clauses C^1 to C^{i-1} have already been merged, and clauses C^i to C^n still appear in the form of literals. The process of merging the ith clause is shown as Algorithm 2. Note that when merging, all literals are in propositional logic, which means all substitutions σ are empty now. So in Algorithm 2, we do not use the symbol σ.

In the Merge process, N_i is generated by extracting elements from M_{i-1} which have no intersection with literals in C^i (Line 2). Conversely, S_i is a set of m_i-tuples that represent all merging options with respect to C^i (Line 3). The jth item in each tuple $(\Phi^i_1, \ldots, \Phi^i_{m_i})$ is supposed to be an element of M_{i-1} that contains literal L^i_j. Then M'_i is constructed through merging all alternative $\Phi^i_1, \ldots, \Phi^i_{m_i}$ (Line 6). As a result, N_i consists of unsatisfiable subsets without C^i, while M'_i is formed of unsatisfiable subsets which contain C^i. The operation of `MS`() is to obtain those minimal elements under set inclusion. That is, if $\Theta = \{\Theta_1, \ldots, \Theta_n\}$, where $\Theta_1, \ldots, \Theta_n$ are different sets, then $\texttt{MS}(\Theta) = \{\Theta' \mid \Theta' \in \Theta$ and there is no $\Theta'' \in \Theta$ such that $\Theta'' \subset \Theta'\}$. After all formulas are merged, we get M_n, the set that contains all MUSes of instances of original clauses. Finally, by processing the **Map Back** step that is Algorithm 1 Line 17, all MUSes are extracted.

Since the input set consists of finite clauses, and the number of intermediate results generated during the procedure of FMUS2 is also finite, FMUS2 must

Algorithm 2. Merge(i, M_{i-1})

Input: the set of all MUSes after merging $i-1$ clauses of F
Output: the set of all MUSes after merging i clauses of F
1: $M_i^{'} := \emptyset$
2: $N_i := \{\phi \mid \phi \in M_{i-1}$ and $\phi \cap \{L_1^i, \ldots, L_{m_i}^i\} = \emptyset\}$
3: $S_i := \{(\Phi_1^i, \ldots, \Phi_{m_i}^i) \mid \Phi_j^i \in M_{i-1}, L_j^i \in \Phi_j^i, j \in [1, m_i]\}$
4: **for all** $(\Phi_1^i, \ldots, \Phi_{m_i}^i) \in S_i$ **do**
5: **if** $(\Phi_1^i, \ldots, \Phi_{m_i}^i)$ can merge **then**
6: $M_i^{'} := M_i^{'} \cup \left\{ \{C^i\} \cup \bigcup_{j=1}^{m_i} (\Phi_j^i - \{L_j^i\}) \right\}$
7: **end if**
8: **end for**
9: $M_i := \mathtt{MS}(N_i \cup M_i^{'})$
10: **return** M_i

terminate in finite steps. The output of FMUS2 will be the set which consists of all MUSes of the input. Besides, FMUS2 can be altered to a partial MUS enumerating algorithm by simply outputting all MUSes newly found after merging every clause. This is based on the fact that if there is a MUS $\{1, 3\}$ after merging clauses 1 to 3, $\{1, 3\}$ is also a MUS of the whole set of clauses 1 to n, where $n \geq 3$.

4 Optimization Strategies

In this section, we will discuss some optimization strategies used to improve the performance of FMUS2. The strategies can be divided into two categories. One is concerned with the order used in the merging procedure, and the other is concerned with pruning, i.e. reducing the number of intermediate results.

4.1 Merging Strategies

For FMUS2, the merging procedure is the most important and time-consuming part. Though different orders of merging do not affect the correctness of the algorithm, they do affect the number of intermediate results significantly. Thus the efficiency of the algorithm will be affected. A good order may solve an input rapidly while a bad order may timeout for the same input. We propose a simple heuristic merging strategy to determine the merging order.

The heuristic merging strategy is based on the theoretical maximum number $M(C)$ of intermediate results for each clause C when it is the first to merge. In detail, $M(C^i) = \prod_{j=1}^{m_i} n_j^i$. The m_i denotes the number of literals of C^i, and the numbers of contradictory literals of $C_1^i, \cdots, C_{m_i}^i$ are $n_1^i, \cdots, n_{m_i}^i$. In order to rein in the potentially exponential growth of intermediate results as much as possible, before merging, $M(C^i)$ will be calculated for every clause and then arranged from least to most which is the merging order. For the consideration of

comparison, a completely opposite order and a random order are implemented as contrast strategies.

Except for deciding the order of merging, the heuristic strategy will also renumber the clauses opposite to the merging order. The reasons are as follows.

While merging, we should decide whether two intermediate results can be merged. The principle is that, when merging clause i, if two intermediate results contains two different literals that come from the same clause $j(j \neq i)$ separately, they can not be merged. If they are merged, the unsatisfiability of the newly generated intermediate result can not be maintained.

Example 2. Considering

$$F = \{\overset{1.1}{x_1} \vee \overset{1.2}{x_2}, \overset{2.1}{\neg x_1} \vee \overset{2.2}{\neg x_2}\}.$$

The $x.y$ labels on the top of literals are identifiers. The x denotes the clause number which this literal belongs to, and the y denotes the literal number in clause. In particular, the $x.0$ label denotes the whole x-th clause.

It is obvious that F is satisfiable. Before merging, there are two intermediate results, $I_1 = \{\overset{1.1}{x_1}, \overset{2.1}{\neg x_1}\}$ and $I_2 = \{\overset{1.2}{x_2}, \overset{2.2}{\neg x_2}\}$. According to the principle above, I_1 and I_2 can not be merged. If they are merged, the result is $I_3 = \{x_1 \overset{1.0}{\vee} x_2, \neg x_1 \overset{2.0}{\vee} \neg x_2\}$, which is incorrect, in other words, satisfiable.

Because we should check whether two intermediate results can be merged, we should traverse and check all possible clauses, for which one or more literals are contained in these two intermediate results. The larger the $M(C)$ for a clause is, the more likely different literals of this clause will be contained in two different intermediate results. Thus when merging two intermediate results, the clause for which one or more literals are contained in these two intermediate results and $M(C)$ is larger, will be checked first.

We give an example to show how the merging strategy works and why we renumber the clauses opposite to the merging order.

Example 3. Considering

$$F = \{\overset{1.1}{x_1} \vee \overset{1.2}{\neg x_4}, \overset{2.1}{x_4} \vee \overset{2.2}{\neg x_3} \vee \overset{2.3}{x_2}, \overset{3.1}{\neg x_1} \vee \overset{3.2}{x_2} \vee \overset{3.3}{x_3}, \overset{4.1}{\neg x_2} \vee \overset{4.2}{x_4}, \overset{5.0}{\neg x_1}\}.$$

F is satisfiable. Before merging, there are 7 intermediate results I_1 to I_7.

$$\begin{aligned}
&I_1 = \{\overset{1.1}{x_1}, \overset{3.1}{\neg x_1}\}, I_2 = \{\overset{1.1}{x_1}, \overset{5.0}{\neg x_1}\},\\
&I_3 = \{\overset{1.2}{\neg x_4}, \overset{2.1}{x_4}\}, I_4 = \{\overset{1.2}{\neg x_4}, \overset{4.2}{x_4}\},\\
&I_5 = \{\overset{2.2}{\neg x_3}, \overset{3.3}{x_3}\}, I_6 = \{\overset{2.3}{x_2}, \overset{4.1}{\neg x_2}\}, I_7 = \{\overset{3.2}{x_2}, \overset{4.1}{\neg x_2}\}.
\end{aligned}$$

Because C^5 is a clause with only one literal, we just need to compute $M(C^1)$ to $M(C^4)$.

$$\begin{aligned}
&M(C^1) = n_1^1 \times n_2^1 = 2 \times 2 = 4,\\
&M(C^2) = n_1^2 \times n_2^2 \times n_3^2 = 1 \times 1 \times 1 = 1,\\
&M(C^3) = n_1^3 \times n_2^3 \times n_3^3 = 1 \times 1 \times 1 = 1,\\
&M(C^4) = n_1^4 \times n_2^4 = 2 \times 1 = 2.
\end{aligned}$$

Thus the merging order is $2, 3, 4, 1$.

After merging C^2, we will get $I_8 = \{\overset{1.2}{\neg x_4}, x_4 \vee \overset{2.0}{\neg x_3} \vee x_2, \overset{3.3}{x_3}, \overset{4.1}{\neg x_2}\}$. And the set of all intermediate results is $\Gamma = \{I_1, I_2, I_4, I_7, I_8\}$.

Then we will merge C^3. I_1, I_7 and I_8 involve L_1^3, L_2^3 and L_3^3 separately. First we get $I_9 = \{\overset{1.1}{x_1}, \overset{3.1}{\neg x_1}, \overset{3.2}{x_2}, \overset{4.1}{\neg x_2}\}$ by merging I_1 and I_7. Then we try to merge I_8 and I_9. Because there are literals of C^1 and C^4 in I_8 and I_9, we should check whether I_8 and I_9 can be merged. Because $M(C^1)$ is larger than $M(C^4)$, we first check literals of C^1. I_8 contains L_2^1 and I_9 contains L_1^1, thus they can not be merged. I_8 and I_9 will be discarded. The remaining $\Gamma = \{I_2, I_4\}$.

If we do not check opposite to the merging order, we could first check literals of C^4, and we will see that I_8 and I_9 both contain L_1^4. Then we should also check literals of C^1. As a result, a useless check is processed.

The remaining I_2 and I_4 can be merged, but there is no another intermediate result contains L_1^4. Thus no MUS is found, and the original F is satisfiable.

4.2 Pruning Strategies

Two strategies are applied to prune the search space of MUSes, i.e., eliminate useless intermediate results. The core of these two strategies is keeping every intermediate result I *minimal*, in other words, I is not a superset of any other intermediate result I_2 or any already obtained MUS. If I is a superset of a MUS, it is obvious that I can never be merged (expanded) to a MUS. If I is a superset of another intermediate result I_2, for any larger intermediate result I' that contains I by merging it with some other intermediate result $I_3, I_4, \ldots$, there will be another I_2' that contains I_2 by merging it with the same $I_3, I_4, \ldots$. So I' is not minimal, and so it cannot be a MUS.

Strategy 1 focuses on eliminating useless intermediate results after merging every literal. The merging operation processes literal by literal and clause by clause. After merging every literal, each newly generated intermediate result will be checked whether it is a superset of any other intermediate result that is not used while merging this literal. And after merging every clause, each remaining newly generated intermediate result will be checked whether it is a superset of any already obtained MUS.

The ideal situation is that no useless intermediate result will be generated. But Strategy 1 cannot prevent the appearance of useless intermediate results. It can only discard them after their appearance. Though it can benefit the following merging steps, time and space are spent to generate the useless intermediate results and check whether they are useless. So we propose the next strategy to partly prevent the appearance of useless intermediate results.

Strategy 2 is recording an affirmative propositions set and a negative propositions set for every intermediate result, and then these sets will be used to decide whether two intermediate results can merge. For every intermediate result, its affirmative propositions set is a set that contains every *single* affirmative proposition which belongs to this intermediate result, and its negative propositions set is a set that contains every *single* negative proposition which belongs to this

intermediate result. The *single* proposition means proposition contained in this intermediate result, and not the whole clause which this proposition belongs to is contained in this intermediate result.

While merging, the intersection of two intermediate results' affirmative propositions sets P' and the intersection of two intermediate results' negative propositions sets N' will be computed first. Then, we will find literals contained in both intermediate results, and remove their corresponding propositions in P' or N'. Finally, if P' and N' are both $\emptyset$, these two intermediate results can be merged. If not, these two intermediate results can not be merged.

P' contains propositions in these two intermediate results' affirmative propositions sets both. If it is not empty, it means there are propositions with the same name but coming from different clauses, i.e., duplicate propositions. If we merge these two intermediate results and get a new intermediate result I, I cannot be *minimal*. Because there are duplicate propositions in I, we at least can remove one duplicate proposition without changing the unsatisfiability of I.

Example 4 shows how Strategy 2 works.

Example 4.

$$F = (\overset{1.1}{x_1} \vee \overset{1.2}{x_2}) \wedge (\overset{2.1}{\neg x_1} \vee \overset{2.2}{\neg x_2}) \wedge (\overset{3.1}{\neg x_3} \vee \overset{3.2}{\neg x_2}) \wedge (\overset{4.1}{\neg x_3} \vee \overset{4.2}{x_2}) \wedge (\overset{5.0}{x_3}).$$

After merging the first clause, we get an intermediate result I_1 with its affirmative propositions set $\emptyset$ and its negative propositions set $\{\neg x_1, \neg x_2\}$.

$$I_1 = \{x_1 \overset{1.0}{\vee} x_2, \overset{2.1}{\neg x_1}, \overset{3.2}{\neg x_2}\}.$$

And then, we merge the second clause, i.e., merge I_1 and $I_2 = \{\overset{2.2}{\neg x_2}, \overset{4.2}{x_2}\}$. The affirmative propositions set of I_2 is $\{x_2\}$, and the negative propositions set is $\{\neg x_2\}$. The proposition $\neg x_2$ appears in both I_1 and I_2, but it comes from different literals of different clauses (clause 2 and clause 3). So we choose not to merge I_1 and I_2.

If we merge I_1 and I_2, we will get an intermediate result I_3.

$$I_3 = \{x_1 \overset{1.0}{\vee} x_2, \neg x_1 \overset{2.0}{\vee} \neg x_2, \overset{3.2}{\neg x_2}, \overset{4.2}{x_2}\}.$$

I_3 is a superset of another intermediate result $I_4 = \{\overset{3.2}{\neg x_2}, \overset{4.2}{x_2}\}$. According to the reason described above, I_3 will be discarded.

As a result, this strategy prevents I_1 and I_2 merging, instead of merging and discarding the newly generated intermediate result. Because it will not carry out the merging and discarding process, which needs to traverse the intermediate result set to decide whether one should be discarded or not, runtime will be saved. When the original formula becomes complex, the situation similar to Example 4 will occur many times. So a lot of runtime will be saved.

5 Experiments

In this section, a series of experiments are performed to evaluate the general performance of FMUS2 by comparing it with the state-of-the-art algorithms and verify the effectiveness of the heuristic merging and pruning strategies we adopted. All experiments were performed on a Ubuntu 16.04 LTS Linux server with an Intel Xeon E5-4607 v2 2.6 GHz CPU and 15 GB main memory. Timeout is set to 300 s for all test cases. For timeout instances, we use the Penalized Average Runtime (PAR-10) [9], where a timeout counts 10 times the time limit. That is, the runtime for every timeout instance is set to 3000 s.

5.1 Performance

As mentioned earlier, the FEF fragment is a special case of EPR. Since many implementations of MUS enumeration algorithms only deal with propositional logic, we shall first evaluate the performance of FMUS2 on FEF by comparing its performance on industrial benchmarks with one of the state-of-the-art partial MUSes enumerators—MARCO [11], which support enumerating MUSes in the EPR fragment by using Z3 [5] as a SAT-solver. In this experiment, MARCO and Z3 are both open source and the version of MARCO is 2.0.1. The evaluation is performed on 100 instances from the EPR division of the TPTP Problem Library [18]. The majority of instances considered are originally from realistic problems, including geometry, puzzles, and software verification.

Figure 1 shows the the runtime of FMUS2 and MARCO for each instance. The x-coordinate represents the number of solved instances, and the y-coordinate

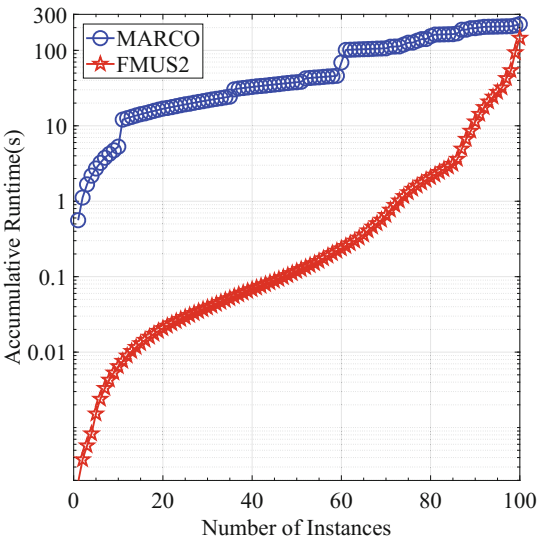

Fig. 1. Comparing FMUS2 against MARCO on industrial benchmarks.

represents the accumulative runtime spent by MARCO or FMUS2 when solving these instances. The line for FMUS2 is always below the line for MARCO, which implies that FMUS2 is faster than MARCO in general for these instances. Although FMUS2 does not spend less time than MARCO for every instance, the accumulative runtime, in other words, the average runtime for FMUS2 is less than MARCO (the average runtime for FMUS2 and MARCO are 2.230 s and 1.460 s respectively).

The experimental results also reveal that FMUS2 is still not optimized enough to compete with methods which utilize highly optimized SAT-solvers when dealing with large-scale formula sets which have complex inconsistency relations between clauses of formulas, i.e. hard instances for FMUS2.

Since FMUS2 is a complete MUSes enumeration algorithm, we shall do a further comparison with one of the state-of-the-art complete MUSes enumeration algorithm CAMUS [12]. Because CAMUS only supports propositional logic and the above industrial instances for comparing with MARCO are relatively scattered and smaller in their scales, randomly generated benchmarks in propositional logic are adopted to further comparison on large-scale instances. Note that in this experiment, MARCO uses its built-in SAT solver—MiniSAT [6].

The randomly generated benchmarks are divided into classes such that all instances in each class have the same number of formulas, which can be found in https://github.com/luojie-sklsde/MUS_Random_Benchmarks. Each class contains 200 unsatisfiable formulas, denoted as the form "mus$x$-y", where the first number x stands for the number of clauses of instances in this class and the second number y stands for the average number of literals in each instance. For example, class "mus400-798" is composed of instances (formulas) containing 400 clauses, where the average number of literals in these instances is 798. Although the number of clauses (i.e. x) is fixed in each class, the number of literals within clauses can vary (so y is an average number), which allows us to simulate as many cases as possible.

Table 1 shows experimental results of CAMUS, MARCO and FMUS2 on the randomly generated benchmarks.

Table 1. Comparing among CAMUS, MARCO and FMUS2

Benchmarks	CAMUS			MARCO			FMUS2		
	N_{TO}	N_{best}	T_{Ave}	N_{TO}	N_{best}	T_{Ave}	N_{TO}	N_{best}	T_{Ave}
mus100-200	25	10	379.344	12	0	183.483	**3**	**187**	**45.798**
mus200-401	121	0	1822.921	76	0	1150.496	**16**	**184**	**242.506**
mus400-798	194	0	2911.875	182	0	2736.391	**34**	**166**	**511.340**
mus600-1200	200	0	3000	200	0	3000	**56**	**144**	**841.770**
mus800-1601	200	0	3000	200	0	3000	**64**	**136**	**961.467**
mus1000-2002	200	0	3000	200	0	3000	**69**	**131**	**1035.995**

The first column of Table 1 are different classes of the benchmarks, followed by statistical runtime data for CAMUS, MARCO and FMUS2. N_{TO} is the number of instances which are timeout after 300s, N_{best} is the number of instances which get the best runtime among the 3 approaches, and T_{Ave} is the average runtime (in seconds) of all instances. The bold number in each row represents the best results among different approaches. It is clear that FMUS2 outperforms CAMUS and MARCO in all three numbers, i.e. N_{TO}, N_{best}, and T_{Ave}. FMUS2 has the smallest number of timeout instances in all six classes and gets the best runtime for most of instances in each class of benchmarks, which means the performance of FMUS2 is stable among different instances. Based on a detailed analysis of the experimental data, we find that FMUS2 is especially efficient when dealing with instances that contain multiple MUSes, which are exactly the ideal targeting input of the MUSes enumeration problem.

The performance experiment shows the competitive power of FMUS2, that is, FMUS2 can perform better than the state-of-the-art algorithms MARCO and CAMUS in some industrial and randomly generated cases.

5.2 Effectiveness of the Optimization Strategies

To evaluate whether the strategies are effective or not, we carried out a series of experiments. Table 2 shows experimental results of FMUS2 and FMUS on the same benchmarks with Table 1.

Table 2. Comparing FMUS2 with FMUS on randomly generated benchmarks

Benchmarks	FMUS			FMUS2		
	N_{TO}	N_{best}	T_{Ave}	N_{TO}	N_{best}	T_{Ave}
mus100-200	**3**	2	46.713	**3**	**195**	**45.798**
mus200-401	19	3	287.875	**16**	**182**	**242.506**
mus400-798	36	0	542.194	**34**	**166**	**511.340**
mus600-1200	57	1	860.813	**56**	**143**	**841.770**
mus800-1601	68	1	1026.261	**64**	**135**	**961.467**
mus1000-2002	71	0	1070.605	**69**	**131**	**1035.995**

The result shows that these optimizations adopted are effective in this randomly generated benchmark.

To evaluate the impact of different merging strategies on the performance of the proposed FMUS2 algorithm, a series of experiments are performed on the industrial benchmarks from TPTP Problem Library.

Table 3 shows statistical data of experimental results. Note that the contrast merging strategy adopts an opposite strategy to the heuristic merging strategy. In Table 3, N_{TO}, T_{Ave} are the same as Table 1, while T'_{Ave} is the average runtime of all instances which are solved in time. From the average runtime data, we can see that different merging strategies greatly affect the performance

Table 3. Statistical data for different merging strategies on industrial benchmarks

Benchmarks	Random			Heuristic			Contrast		
	N_{TO}	T_{Ave}	T'_{Ave}	N_{TO}	T_{Ave}	T'_{Ave}	N_{TO}	T_{Ave}	T'_{Ave}
TPTP instances (100)	6	185.685	6.047	**0**	**1.460**	**1.460**	9	271.479	1.626

of our algorithm. It is obvious that the heuristic strategy yields the best performance overall, especially obvious when timeout instances are also counted. Hence, adopting the proposed heuristic merging strategy greatly improves the performance of FMUS2 on practical problems in general, which we view as a reasonable metric of its effectiveness.

However, there are still some cases where the heuristic merging strategy is beaten by the random strategy, and the runtime for some instances can be shortened, which means there is still a lot of potential for further optimizing of the heuristic merging strategy. Figure 2 demonstrates the change of the numbers of intermediate results for different merge orders while running the "HWV003-3" instance from TPTP. The x-axis represents the number of merged clauses, and the y-axis represents the number of intermediate results during each after each merging. More specifically, there are 61 clauses that need to be merged in this test case, thus the y-value becomes zero when the x-value increases to 61, indicating the end of the merging. The line labeled with order3 represents the status of our current heuristic merging strategy. On the one hand, a slight change to order3 can result in order4, which maintains a large amount of intermediate results from merging 26 clauses to merging 54 clauses such that the runtime increases dramatically. Further change to order4 can lead to the merge order order5, which triggers a visible explosion of intermediate results and run out

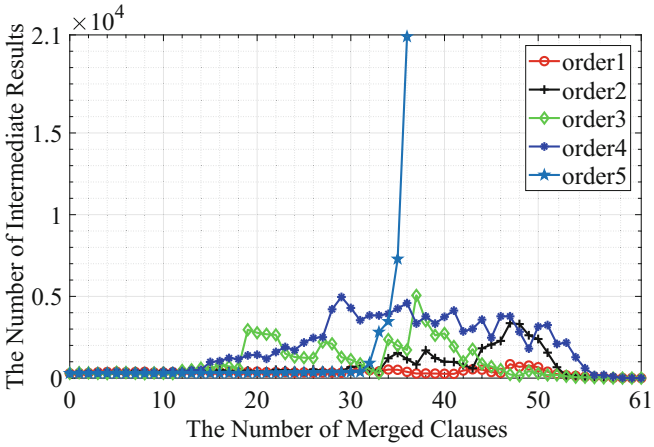

Fig. 2. Variation trends of intermediate results for different merge orders.

of memory at the end. This is one of main reasons for the timeout of some instances in these benchmarks. On the other hand, changes to order3 may also lead to merge orders such as order2 and order1 which is the best merge order we obtained for the HWV003-3 instance. Hence there is still much room left for the optimization of the merging strategy, especially when dealing with hard instances for FMUS2.

6 Conclusions

In this paper, we proposed a "decompose-merge" algorithm to enumerate all minimal unsatisfiable subsets for a CNF formula in the field of propositional logic and FEF fragment of first-order logic. A heuristic merging strategy and two pruning strategies are adopted to improve the performance of the algorithm. Experimental results show that our algorithm FMUS2 is competitive, and can perform better on some industrial and randomly generated cases when compared with two other state-of-the-art MUS enumerating algorithms. And the optimization strategies adopted has proved to be effective.

For future work, further improvements to FMUS2 will be one of our focuses. As mentioned above, there are still some weaknesses in FMUS2 when dealing with hard instances, i.e. large-scale formulas which have a complex inconsistency relations between clauses. The experimental results also show that the current heuristic merging strategy can be optimized. There is still a lot of room to improve. For instance, it would be interesting to explore better merging strategies and techniques to intelligently select a strategy according to the characteristics of the input set. Besides, we would like to investigate whether our algorithm can be applied to larger fragments of first-order logic in future work.

Acknowledgments. This work was supported by National Natural Science Foundation of China (Grand No. 61502022) and State Key Laboratory of Software Development Environment (Grand No. SKLSDE-2017ZX-17).

References

1. Bacchus, F., Katsirelos, G.: Finding a collection of MUSes incrementally. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 35–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_3
2. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30557-6_14
3. de la Banda, M.G., Stuckey, P.J., Wazny, J.: Finding all minimal unsatisfiable subsets. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming, pp. 32–43. ACM (2003)
4. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2), 97–116 (2012)

5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
7. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability solvers. Found. Artifi. Intell. **3**, 89–134 (2008)
8. Hou, A.: A theory of measurement in diagnosis from first principles. Artif. Intell. **65**(2), 281–328 (1994)
9. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Tradeoffs in the empirical evaluation of competing algorithm designs. Ann. Math. Artif. Intell. **60**(1–2), 65–89 (2010)
10. Li, W., Shen, N., Wang, J.: R-calculus: a logical approach for knowledge base maintenance. Int. J. Artif. Intell. Tools **4**(01n02), 177–200 (1995)
11. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016)
12. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
13. Luo, J., Li, W.: R-calculus without the cut rule. Sci. China Inf. Sci. **54**(12), 2530–2543 (2011)
14. Previti, A., Marques-Silva, J.: Partial MUS enumeration. In: AAAI (2013)
15. Ramsey, F.P.: On a problem of formal logic. In: Gessel, I., Rota, G.C. (eds.) Classic Papers in Combinatorics. MBC, pp. 1–24. Springer, Boston (2009). https://doi.org/10.1007/978-0-8176-4842-8_1
16. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_15
17. Stern, R.T., Kalech, M., Feldman, A., Provan, G.M.: Exploring the duality in conflict-directed model-based diagnosis. In: AAAI, vol. 12, pp. 828–834 (2012)
18. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reason. **59**(4), 483–502 (2017)
19. Xiao, G., Ma, Y.: Inconsistency measurement based on variables in minimal unsatisfiable subsets. In: European Conference on Artificial Intelligence 2012 (ECAI 2012) (2012)
20. Xie, H., Luo, J.: An algorithm to compute minimal unsatisfiable subsets for a decidable fragment of first-order formulas. In: 2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 444–451. IEEE (2016)