Finding a Collection of MUSes Incrementally

Fahiem Bacchus[1] and George Katsirelos[2(✉)]

[1] Department of Computer Science, University of Toronto, Toronto, ON, Canada
fbacchus@cs.toronto.edu
[2] MIAT, INRA, Toulouse, France
george.katsirelos@toulouse.inra.fr

Abstract. Minimal Unsatisfiable Sets (*MUSes*) are useful in a number of applications. However, in general there are many different *MUSes*, and each application might have different preferences over these *MUSes*. Typical MUSER systems produce a single *MUS* without much control over which *MUS* is generated. In this paper we describe an algorithm that can efficiently compute a collection of *MUSes*, thus presenting an application with a range of choices. Our algorithm improves over previous methods for finding multiple *MUSes* by computing its *MUSes* incrementally. This allows it to generate multiple *MUSes* more efficiently; making it more feasible to supply applications with a collection of *MUSes* rather than just one.

1 Introduction

When given an unsatisfiable CNF $\mathcal{F}$, SAT solvers can return a core, i.e., a subset of $\mathcal{F}$ that remains unsatisfiable. Many applications, e.g., program type debugging, circuit diagnosis, and production configuration [6], need cores in their processing. In many cases these applications can be made much more effective if supplied with minimal unsatisfiable sets (*MUSes*), which are cores that are minimal under set inclusion. That is, no proper subset of a *MUS* is unsatisfiable.

This makes the problem of efficiently extracting a *MUS* an important and well studied problem, see [5,6,9,13,18,20,21] for a more extensive list. In fact, the problem of finding a minimal set of constraints sufficient to make a problem unsolvable is important in other areas as well. For example in operations research it is often useful to find irreducible inconsistent subsystems (IISes) of linear programs and integer linear programs [8,24], and in CP a minimal unsatisfiable set of constraints [12].

In various applications the preference for *MUSes* over arbitrary cores goes further, and some *MUSes* might be preferred to others. Most algorithms for computing *MUSes*, however, return an arbitrary *MUS*. There has been some work on the problem of computing specific types of *MUSes*. In [19] the problem of computing lexicographic preferred *MUSes* is addressed. Furthermore, the problem of computing the smallest *MUS* has been addressed in [10,11,15]. However, algorithms for extracting specific *MUSes*, especially those for extracting the smallest *MUS*, can be considerably less efficient than state-of-the-art *MUS* extraction algorithms returning an arbitrary *MUS*.

C.-G. Quimper (Ed.): CPAIOR 2016, LNCS 9676, pp. 35–44, 2016.
DOI: 10.1007/978-3-319-33954-2_3

In this paper we address this issue by trying to quickly return a collection of *MUSes*, rather than trying to compute a specific type of *MUS*. The application can then choose its best *MUS* from that collection. So, e.g., although our approach cannot guarantee returning the smallest *MUS*, the application can choose the smallest *MUS* from among the collection returned. This approach is advantageous when algorithms for computing the most preferred *MUS* are too costly (e.g., computing the smallest *MUS*), or when there is no known algorithm for computing the most preferred *MUS* (e.g., the application's preference criteria is not lexicographic).

We accomplish this task through an extension of a recent *MUS* algorithm [3]. The advantage of our algorithm is that it can exploit information computed when finding previous *MUSes* to speed up finding future *MUSes*. Hence, it can find multiple *MUSes* more efficiently. This algorithm has the drawback, however, that it cannot keep on finding more *MUSes* when given more time: it computes a set of *MUSes* of indeterminate size and then stops. Adopting the power set exploration idea of [14] we address this drawback, presenting a method that can eventually compute all *MUSes* while still enumerating them at a reasonable rate. We show that our algorithms improve on the state of the art.

2 Background

Let $\mathcal{F}$ be an unsatisfiable set of clauses.

Definition 1 (MUS). A **Minimal Unsatisfiable Set** (*MUS*) of $\mathcal{F}$ is a unsatisfiable subset $M \subseteq \mathcal{F}$ that is minimal w.r.t. set inclusion. That is, M is *unsat* but no proper subset is.

Definition 2 (MSS). A **Maximal Satisfiable Subset** (*MSS*) of $\mathcal{F}$ is a satisfiable subset $S \subseteq \mathcal{F}$ that is maximal w.r.t set inclusion.

Definition 3 (MCS). A correction subset of $\mathcal{F}$ is a subset of $\mathcal{F}$ whose complement in $\mathcal{F}$ is *sat*. A **Minimal Correction Subset** (*MCS*) of $\mathcal{F}$ is a correction subset that is minimal w.r.t. set inclusion.

Note that if C is an *MCS* of $\mathcal{F}$ then its complement $\mathcal{F} \setminus C$ is an *MSS* of $\mathcal{F}$.

Definition 4. A clause $c \in \mathcal{F}$ is said to be **critical** for $\mathcal{F}$ (also known as a *transition* clause [7]) when $\mathcal{F}$ is *unsat* and $\mathcal{F} - \{c\}$ is *sat*.

Intuitively, a *MUS* is an unsatisfiable set that cannot be reduced without causing it to become satisfiable; a *MSS* is a satisfiable set that cannot be added to without causing it to become unsatisfiable; and an *MCS* is a minimal set of removals from $\mathcal{F}$ that causes $\mathcal{F}$ to become satisfiable.

A critical clause for $\mathcal{F}$ is one whose removal from $\mathcal{F}$ causes $\mathcal{F}$ to become satisfiable. If c is critical for $\mathcal{F}$ then (a) c must be contained in every *MUS* of $\mathcal{F}$ and (b) $\{c\}$ is an *MCS* of $\mathcal{F}$. Furthermore, M is a *MUS* if and only if every $c \in M$ is critical for M. Note that a clause c that is critical for a set S is not necessarily critical for a superset $S' \supset S$. In particular, S' might contain other *MUSes* that do not contain c.

Duality. A hitting set H of a collection of sets $\mathcal{C}$ is a set that has a non empty intersection with each set in $\mathcal{C}$: $\forall C \in \mathcal{C}.H \cap C \neq \emptyset$. A hitting set H of $\mathcal{C}$ is minimal (or irreducible) if no subset of H is a hitting set of $\mathcal{C}$.

Let $AllMuses(\mathcal{F})$ ($AllMcses(\mathcal{F})$) be the set containing all *MUSes* (*MCSes*) $\mathcal{F}$. There is a well known hitting set duality between AllMuses and AllMcses [22]. Specifically, $M \in AllMuses(\mathcal{F})$ iff M is a minimal hitting set of $AllMcses(\mathcal{F})$, and dually, $\mathcal{C} \in AllMcses(\mathcal{F})$ iff C is a minimal hitting set of $AllMuses(\mathcal{F})$. The duality also holds for non-minimal sets, e.g., any correction set hits all unsatisfiable subsets. It is useful to point out that if $\mathcal{F}' \subseteq \mathcal{F}$, then $AllMuses(\mathcal{F}') \subseteq AllMuses(\mathcal{F})$. Hence, if f is critical for $\mathcal{F}$ it is critical for all unsatisfiable subsets of $\mathcal{F}$. An *MCS* C' of $\mathcal{F}' \subset \mathcal{F}$, on the other hand, is not necessarily an *MCS* of $\mathcal{F}$, however C' can always be extended to an *MCS* C of $\mathcal{F}$ [3].

3 Enumerating *MUSes*

To the best of our knowledge the current state-of-the-art algorithm for the problem of quickly computing a collection of *MUSes* is the Marco system originally developed in [14] and later improved in [16]. Marco$^+$ (the new optimized version of Marco [16]) was compared with previous approaches [4,17] and shown to be superior at this task. Therefore we confine our attention in this paper to comparing with the Marco$^+$ approach.

Algorithm 1 shows the algorithm used by Marco$^+$. Marco$^+$ uses the technique of representing subsets of $\mathcal{F}$, the input set of clauses, with a CNF, *ClsSets*. *ClsSets* contains a variable s_i for each clause $c_i \in \mathcal{F}$. Every satisfying solution of *ClsSets* specifies a subset of $\mathcal{F}$: the set of clauses c_i corresponding to true s_i in the satisfying solution. Initially, *ClsSets* contains no clauses, and thus initially its set of satisfying solutions corresponds to $\mathcal{F}$'s powerset.

Marco$^+$ uses *ClsSets* to keep track of which subsets of $\mathcal{F}$ have already been tested so that each *MUS* it enumerates is distinct. When *ClsSets* becomes *unsat* all subsets of $\mathcal{F}$ have been tested and all *MUSes* have been enumerated. Otherwise, the truth assignment π (line 4) provides a subset S of unknown status.

Marco$^+$ forces the sat solver to assign variables to true in each decision. Hence, if S is *sat* it is guaranteed to be an *MSS* (see [2] or [23] for a simple proof). S and all of its subsets are thus now known to be *sat* so they can be blocked in *ClsSets*. This means that all future solutions of *ClsSets* must have a non-empty intersection with $\mathcal{F} \setminus S$, i.e., they must hit the complement of S, a (minimal) correction set. The update of *ClsSets* is accomplished with the subroutine call **hitCorrectionSet**$(\mathcal{F} \setminus S)$ (line 6) which returns a clause asserting that some s_i corresponding to a clause in $F \setminus S$ must be true.

Otherwise S is *unsat* and it contains at least one *MUS*. Marco$^+$ then invokes a *MUS* finding algorithm to find one of S's *MUSes*. In addition, Marco$^+$ informs the *MUS* algorithm of all singleton *MCSes* it has found. The computed *MUS* M has to include the union of these singleton *MCSes* as it must hit every *MCS*.

M and all of its supersets are known to be *unsat* and are blocked in *ClsSets* by a clause computed by **blockSuperSets**(M) asserting that some s_i corresponding

Algorithm 1. MARCO$^+$ *MUS* enumeration algorithm

Input: $\mathcal{F}$ an **unsatisfiable** set of clauses
Output: All *MUSes* of $\mathcal{F}$, output as they are computed

$ClsSets \leftarrow \emptyset$ ◁ Initially, *ClsSets* admits all subsets of $\mathcal{F}$ as solutions.
while *true* **do**
  // If C is *sat*, **SatSolve**(C, π) returns true and puts truth assignment in π
  if SatSolve$(ClsSets, \pi)$ **then**
    $S \leftarrow \{c_i \in \mathcal{F} \mid \pi[s_i] = true\}$ ◁ All decisions set to *true* so S is maximal
    if SatSolve(S, π) **then**
      $ClsSets \leftarrow ClsSets \cup$ **hitCorrectionSet**$(\mathcal{F} \setminus S)$ ◁ $\mathcal{F} \setminus S$ is a *MCS*
    else
      $M \leftarrow$ **findMUS**$(S, \{\text{all singleton } MCSes\})$
      output(MUS)
      $ClsSets \leftarrow ClsSets \cup$ **blockSuperSets**(M)
  else return

to $c_i \in M$ must be false [14]. After all subsets of $\mathcal{F}$ have been identified as being *sat* or *unsat* (detected by *ClsSets* becoming *unsat*), the algorithm returns.

One advantage of MARCO$^+$ is that it can utilize any *MUS* algorithm. Thus once it has identified a subset of $\mathcal{F}$ to be *unsat* it can enumerate a new *MUS* as efficiently as finding a single *MUS*. Another advantage is that it will continue to enumerate *MUSes* until it has enumerated them all. On the negative side, each new *MUS* is computed with an entirely separate computation. This *MUS* computation only knows about the prior singleton *MCSes* but does not otherwise share much information with prior *MUS* computations (beyond some learnt clauses).

4 A New Algorithm for Enumerating *MUSes*

Algorithm 2 shows our new algorithm for generating multiple *MUSes* from a formula. The grayed out lines will be used when multiple initial calls are made to the algorithm, they will be discussed in the next section. For now it can be noted that these lines have no effect if *ClsSets* is initially an empty set of clauses.

The algorithm is a modification of the recently proposed state-of-the-art *MUS* algorithm MCS-MUS [3]. It extends MCS-MUS by performing a backtracking search over a tree in which the branch points correspond to the different ways the *MUSes* to be output can hit a just computed *MCS*.

The algorithm maintains a current formula $F' \subseteq F$, such that F' is *unsat*, partitioned into a set of clauses known to be critical for F', *crits*, and a set of clauses of unknown status, *unkn*. It starts by identifying an *MCS*, *cs*, of $crits \cup unkn$, such that $cs \subseteq unkn$, using a slight modification of existing *MCS* algorithms [3]. If no such *MCS* exists, then *crits* is unsatisfiable and since all of its clauses are critical, it is a *MUS*. This *MUS* is reported and backtrack occurs. If *cs* does exist, it creates a choice point. By duality we know that every *MUS* must hit *cs*, and by minimality of *cs* we know that for every clause $c \in cs$ there is a *MUS* whose intersection with *cs* is only c. Hence, we select a clause $c \in cs$

Algorithm 2. MCS-MUS-BT $(unkn, crits, ClsSets)$: Output a collection of *MUSes* of $unkn \cup crits$ using *MCS* duality. To find some *MUSes* of $\mathcal{F}$ the initial call MCS-MUS-BT $(\mathcal{F}, \{\}, ClsSets = \emptyset)$ is used.

Input: *unkn* a set of clauses of unknown status such that $unkn \cup crits$ is *unsat*
Input: *crits* a set of clauses critical for $unkn \cup crits$
Input: *ClsSets* a CNF representing subsets of the input formula of unknown status.
Output: Some *MUSes* of $unkn \cup crits$, output as computed
Output: *ClsSets* is modified.

1 $crits \leftarrow crits \cup \{c_i \mid s_i \in UP(ClsSets \cup \{(\neg s_j) | c_j \notin crits \cup unkn\})\}$
2 $unkn \leftarrow unkn \setminus crits$
3 $(cs, \pi) \leftarrow$ **findMCS**$(crits, unkn)$ ◁ *Find cs, an MCS contained in unkn.*
4 **if** $cs =$ ***null*** **then**
5   output$(crits)$ ◁ *crits is a MUS of* $crits \cup unkn$
6     $ClsSets \leftarrow ClsSets \cup$ **blockSuperSets**$(crits)$
7   **return**
8 **else**
9     $ClsSets \leftarrow ClsSets \cup$ **hitCorrectionSet**$(\{c | \pi \not\models c\})$ ◁ *Correction set of* $\mathcal{F}$
10     $unkn \leftarrow unkn \setminus cs$
11   **for** $c \in cs$ **do**
12         $crits' \leftarrow crits \cup \{c\}$
13         $unkn' \leftarrow$ **refineClauseSet**$(crits', unkn)$
14         $C \leftarrow$ **recursiveModelRotation**$(c, crits, unkn, \pi)$
15   **MCS-MUS-BT** $(unkn' \setminus C, crits' \cup C)$

to mark as critical (line 12) removing the rest from *unkn* (line 10). This ensures that all *MUSes* enumerated in the recursive call contain *c* and hence hit *cs*.

Before the recursive call, we can use two standard techniques that are critical for performance, *clause set refinement* [21] and *recursive model rotation* [7].

Theorem 1. *All sets output by* MCS-MUS-BT *are MUSes of its input* $\mathcal{F} = unkn \cup crits$. *Furthermore, if* $\mathcal{F}$ *unsatisfiable a least one MUS will be output. Finally, if only one MUS is output, then* $\mathcal{F}$ *contains only one MUS.*

We omit the straightforward proof to save space. Although the theorem shows that MCS-MUS-BT will generate at least one *MUS* (as efficiently as the state-of-the-art MCS-MUS algorithm), the number of *MUSes* it will generate is indeterminate, as this depends on the *MCSes* it happens to generate. Furthermore, it cannot, in general, generate all *MUSes*. Intuitively, by removing *cs* from *unkn* at line 10, we block it from generating any *MUS* M with $|M \cap cs| > 1$.

The main advantage of this algorithm is that it shares computational effort among many *MUSes*. Namely, after the first *MUS* is generated, computation for the second *MUS* starts with at least one (potentially many) known *MCS*, and may also have several clauses in *crits* and a smaller set of clauses in *unkn*. Hence, it can more efficiently generate several *MUSes*.

4.1 Enumerating All *MUSes*

While MCS-MUS-BT may be able to generate a sufficiently large collection of *MUSes*, the unpredictability of the size of this collection might be unsuitable in some cases. In such cases we may of course fall back to MARCO$^+$, giving up the advantages of MCS-MUS-BT.

Another option is to embed MCS-MUS-BT in MARCO$^+$. It is straightforward to modify Algorithm 1 so that it uses MCS-MUS-BT instead of **findMUS** and blocks all *MUSes* discovered during one call. However, without modifying MARCO$^+$ this allows only limited information to flow between MARCO$^+$ and MCS-MUS-BT. In particular, sharing information beyond singleton correction sets is not supported.

A third option then is deeper integration of MCS-MUS-BT into a MARCO-like algorithm. We show this in Algorithm 3, which is based on the MCS-MUS-ALL algorithm of [3]. The outline of MCS-MUS-ALL-BT is broadly similar to that of MARCO$^+$. Like MARCO$^+$ it uses a CNF *ClsSets* to represent subsets of $\mathcal{F}$ with unknown status and uses the same **hitCorrectionSet** and **blockSuperSets** procedures to block *MSSes* and *MUSes*, respectively. When *ClsSets* becomes unsatisfiable all *MUSes* have been enumerated (line 3). Each solution π of *ClsSets* yields a set S of unknown status, which is then tested for satisfiability.

If it is satisfiable, S is guaranteed to be an *MSS* since we require the solver to assign variables to true in each decision as in MARCO$^+$. We can then block S and all of its subsets by forcing *ClsSets* to hit its complement with **hitCorrectionSet**.

If S is unsatisfiable, then it is given to MCS-MUS-BT to extract some of its *MUSes*. We generalize MARCO$^+$, however, by providing all previously discovered correction sets to MCS-MUS-BT, not just the singleton *MCSes*. These correction sets can be exploited to discover new critical clauses. In particular, all previously discovered correction sets result in clauses being added to *ClsSets* by **hitCorrectionSet**. We can use unit propagation (line 1 of Algorithm 2) to determine if the clauses currently excluded from the *MUSes* being enumerated ($\mathcal{F} \setminus (crits \cup unkn)$) make some prior correction set cs a singleton (of course all correction sets that are already singleton will also be found, so this method obtains at least as much information as MARCO$^+$). If so then all *MUSes* of the current subset $crits \cup unkn$ must include that single remaining clause $c \in cs$ since all *MUSes* must hit cs; i.e., c is critical for $crits \cup unkn$.

Thus our algorithm has two advantages over using MCS-MUS-BT in the MARCO$^+$ framework. First, individual calls to MCS-MUS-BT may produce *MUSes* more quickly because our generalization of MARCO$^+$'s technique of exploiting singleton *MCSes* (at line 1) can detect more critical clauses, either initially or as *unkn* shrinks. Second, the multiple correction sets that can be discovered within MCS-MUS-BT are all added to *ClsSets*. Hence, their complementary satisfiable sets will not appear as possible solutions to *ClsSets* in the main loop of Algorithm 3. This can reduce the time spent processing satisfiable sets.

Algorithm 3. MCS-MUS-ALL-BT($\mathcal{F}$): Enumerate all *MUSes* of $\mathcal{F}$.

Input: $\mathcal{F}$ an **unsatisfiable** set of clauses
Output: All *MUSes* of $\mathcal{F}$, output as there are computed
ClsSets $\leftarrow \emptyset$ ◁ *Initially, ClsSets admits all subsets of* $\mathcal{F}$ *as solutions.*
while *true* **do**
  if ***not*** *SatSolve(ClsSets,π)* **then return**; ◁ All *MUSes* enumerated
  $S \leftarrow \{c_i \mid c_i \in \mathcal{F} \wedge \pi \models s_i\}$ ◁ *All decisions set to true so S is maximal*
  if *SatSolve(S,π)* **then**
    ClsSets $\leftarrow$ *ClsSets* $\cup$ **hitCorrectionSet**$(\mathcal{F} \setminus S)$ ◁ *S is an MSS*
  else MCS-MUS-BT ($\mathcal{F}$, *crits*, *unkn*, *ClsSets*)

5 Empirical Results

In this section we evaluate our algorithms which we implemented in C++ on top of MINISAT. We used the benchmark set of [1] containing 300 problems. We used a cluster of 48-core 2.3 GHz Opteron 6176 nodes with 378 GB RAM available.

First we tested MCS-MUS-BT (Algorithm 2) against the MARCO$^+$ system [16]. MCS-MUS-BT can only generate some *MUSes*, while MARCO$^+$ can potentially generate all. So in the scatter plot (a) of Fig. 1 we plotted for each instance the time each approach took to produce the first k *MUSes*, where k is the minimum of the number of *MUSes* produced by the two approaches on that instance when run with a 3600 s timeout. In the plot, points above the 45° line are where MCS-MUS-BT is better than MARCO$^+$. The data shows that MCS-MUS-BT outperforms MARCO$^+$ on most instances.

We also tested how many *MUSes* are typically produced by MCS-MUS-BT. When run on the 300 instances it yielded no *MUSes* on 20 instances (in 3600 s), 1 on 111 instances, 2–10 on 29 instances, and more than 10 on 140 instances. On 6 instances it yielded over 10,000 *MUSes*. So we see that MCS-MUS-BT often yielded a reasonable number of *MUSes*, but in some cases perhaps not enough.

To go beyond MCS-MUS-BT, potentially generating all *MUSes*, we used two variations of our complete algorithms. The first we call MARCO-MANY. This is MCS-MUS-BT integrated into an implementation of the MARCO$^+$ algorithm, with MCS-MUS-BT called when a *MUS* is to be computed and returning multiple *MUSes*. The second variation is MCS-MUS-ALL-BT, from the previous section. We also compare these against MARCO$^+$ [1] and our previous *MUS* enumeration algorithm MCS-MUS-ALL [3].

Figure 1(b) compares MCS-MUS-ALL-BT with MARCO$^+$. Here we plotted for each instance the number of *MUSes* produced by each approach within a 3600 s timeout. Points above the line represent instances where MCS-MUS-ALL-BT generated more *MUSes* than MARCO$^+$. The picture here is not completely clear. However, overall MCS-MUS-ALL-BT showed better performance: it generated more *MUSes* in 170 cases, an equal number in 48 cases, and fewer in 82 cases. Furthermore, notice that as we move up the x and y axis the instances

[1] Version 1.1, downloaded from https://sun.iwu.edu/~mliffito/marco/.

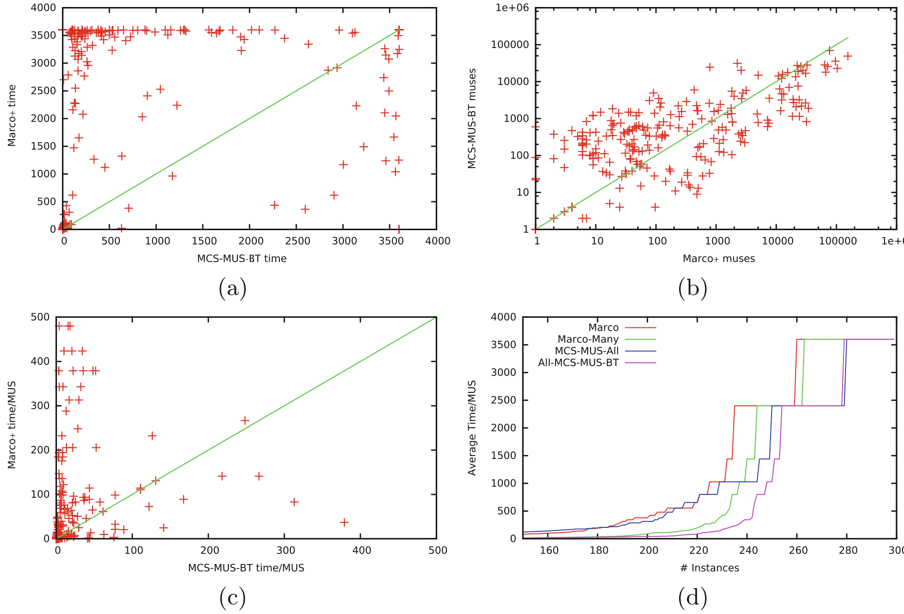

Fig. 1. (a) Time for Marco^{+} to generate as many *MUSes* as MCS-MUS-BT (b): number of *MUSes* MCS-MUS-All-BT against Marco^{+} (logscale). (c) Average time/*MUS* MCS-MUS-All-BT against Marco^{+}. (d) Cactus plot of Average time/*MUS* of all solvers.

become easier, i.e., many more MUSes can be generated per second in these instances. The instances in which Marco^{+} outperformed MCS-MUS-All-BT tend to be towards the upper right of the plot.

Besides the number of instances we are also interested in the rate at which *MUSes* are generated. For each instance we calculated the average time needed to generate a *MUS* by MCS-MUS-All-BT and Marco^{+}. Figure 1(c) shows a scatter plot of these points. The cactus plot of Fig. 1(d) elaborates on this data showing the other algorithms as well.

In scatter plot (c) the axes have been inverted so that once again points above the line represent instances in which MCS-MUS-All-BT is better than Marco^{+}. We zoomed this plot into the range [0,500] s per *MUS* as most of the data was clustered into this region. These instances show a convincing win for MCS-MUS-All-BT. The plot excludes 100 instances. Of these, 43 instances could not be plotted as one or both algorithms produced zero *MUSes*: on 18 both produced zero *MUSes*; on 22 MCS-MUS-All-BT generated a *MUS* but Marco^{+} did not; on 3 the inverse happened. The other 57 instances were excluded because of the plot range. Among them 3 were below the line, 23 above the line and 31 on the line. Of these excluded instances the most extreme win for Marco^{+} was an instance where Marco^{+} generated 3 *MUSes* and MCS-MUS-All-BT only 1; and the most extreme win for MCS-MUS-All-BT was an instance where Marco$^+$ generated only 1 *MUS* and MCS-MUS-All-BT generated 800.

We see that with few exceptions, the average time to generate a *MUS* with MCS-MUS-All-BT is smaller. This is confirmed by the cactus plot (d), where we see that the average time to generate a *MUS* by MCS-MUS-All-BT remains well below that of other algorithms. The corresponding lines only meet for the hardest instances, where all methods generate one or no *MUSes*. The cactus plot also confirms that simply integrating MCS-MUS-BT into a Marco-like algorithm (i.e., Marco-Many) is not sufficient. Additionally, we see that the MCS-MUS-All-BT provides a good improvement over the previous MCS-MUS-All.

References

1. MUS track of the sat competition (2011). http://www.maxsat.udl.cat
2. Bacchus, F., Davies, J., Tsimpoukelli, M., Katsirelos, G.: Relaxation search: a simple way of managing optional clauses. In: Proceedings of the AAAI National Conference (AAAI), pp. 835–841 (2014)
3. Bacchus, F., Katsirelos, G.: Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 70–86. Springer, Heidelberg (2015)
4. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005)
5. Belov, A., Heule, M.J.H., Marques-Silva, J.: MUS extraction using clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 48–57. Springer, Heidelberg (2014)
6. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2), 97–116 (2012)
7. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 37–40 (2011)
8. Chinneck, J.W.: Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods. International Series in Operations Research and Management Sciences, vol. 118. Springer, USA (2008). ISBN 978-0387749310
9. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of boolean clauses. In: International Conference on Tools with Artificial Intelligence (ICTAI), pp. 74–83 (2008)
10. Ignatiev, A., Janota, M., Marques-Silva, J.: Quantified maximum satisfiability: a core-guided approach. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 250–266. Springer, Heidelberg (2013)
11. Ignatiev, A., Previti, A., Liffiton, M., Marques-Silva, J.: Smallest MUS extraction with minimal hitting set dualization. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 173–182. Springer, Heidelberg (2015)
12. Junker, U.: QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In: Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, San Jose, California, USA, 25–29 July 2004, pp. 167–172 (2004)

13. Lagniez, J.-M., Biere, A.: Factoring out assumptions to speed up MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 276–292. Springer, Heidelberg (2013)
14. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple muses quickly. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 160–175. Springer, Heidelberg (2013)
15. Liffiton, M.H., Mneimneh, M.N., Lynce, I., Andraus, Z.S., Marques-Silva, J., Sakallah, K.A.: A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. Constraints **14**(4), 415–442 (2009)
16. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2015)
17. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
18. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 592–607. Springer, Heidelberg (2013)
19. Marques-Silva, J., Previti, A.: On computing preferred MUSes and MCSes. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 58–74. Springer, Heidelberg (2014)
20. Nadel, A., Ryvchin, V., Strichman, O.: Efficient MUS extraction with resolution. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 197–200 (2013)
21. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated deletion-based extraction of minimal unsatisfiable cores. J. Satisfiability Boolean Model. Comput. (JSAT) **9**, 27–51 (2014)
22. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
23. Di Rosa, E., Giunchiglia, E.: Combining approaches for solving satisfiability problems with qualitative preferences. AI Commun. **26**(4), 395–408 (2013)
24. van Loon, J.: Irreducibly inconsistent systems of linear equations. Eur. J. Oper. Res. **8**(3), 283–288 (1981)