MCS Extraction with Sublinear Oracle Queries

Carlos Mencía[2], Alexey Ignatiev[1,3], Alessandro Previti[1], and Joao Marques-Silva[1]

[1] LaSIGE, Faculty of Science, University of Lisbon, Portugal
aignatiev@ciencias.ulisboa.pt,apreviti.research@gmail.com
jpms@ciencias.ulisboa.pt
[2] University of Oviedo, Gijón, Spain
cmencia@gmail.com
[3] ISDCT SB RAS, Irkutsk, Russia

Abstract. Given an inconsistent set of constraints, an often studied problem is to compute an irreducible subset of the constraints which, if relaxed, enable the remaining constraints to be consistent. In the case of unsatisfiable propositional formulas in conjunctive normal form, such irreducible sets of constraints are referred to as Minimal Correction Subsets (MCSes). MCSes find a growing number of applications, including the approximation of maximum satisfiability and as an intermediate step in the enumeration of minimal unsatisfiability. A number of efficient algorithms have been proposed in recent years, which exploit a wide range of insights into the MCS extraction problem. One open question is to find the best worst-case number of calls to a SAT oracle, when the calls to the oracle are kept simple, and given reasonable definitions of simple SAT oracle calls. This paper develops novel algorithms for computing MCSes which, in specific settings, are guaranteed to require asymptotically fewer than linear calls to a SAT oracle, where the oracle calls can be viewed as simple. The experimental results, obtained on existing problem instances, demonstrate that the new algorithms contribute to improving the state of the art.

1 Introduction

The analysis of over-constrained systems finds a wide range of practical applications [8, 17, 20]. Given an inconsistent set of constraints, one is often interested in finding minimal explanations of inconsistency, or in finding maximally consistent sets of constraints (and so minimal sets of constraints to discard). For propositional formulas, a minimal set of clauses to discard to achieve consistency of the remaining clauses is referred to as a minimal correction subset (MCS). In general, selecting minimal sets of constraints is of prime importance, since this enables irrelevant constraints not to be considered. Many application areas of inconsistency analysis, including minimal explanations of inconsistency and maximally consistent subsets of constraints, can be viewed as instantiations of model-based diagnosis [36]. These include, among others, product configuration, axiom pinpointing in description logics, fault localization in software, and design debugging. Other applications of MCSes include the computation of minimal and maximal models [6, 23], enumeration of minimal unsatisfiability, approximation of maximum satisfiability and, as a result, also in relational inference [26]. In the concrete case of propositional formulas, the enumeration of MCSes has been used

for approximating maximum satisfiability (MaxSAT), enumeration of minimal unsatisfiability and also solving function problems in the second level of the function polynomial hierarchy, including minimum unsatisfiability [19]. Although the subject of over-constrained system analysis has been studied in different settings, in some cases for more than a century[1], recent years have seen a multitude of algorithms and algorithmic optimizations being proposed, both for finding minimal explanations of inconsistency [5, 17, 18, 22] and for finding minimal corrections of inconsistency (which enable the consistency of the remaining constraints) [1, 3, 14, 16, 25, 27, 32, 33, 39].

This paper studies the problem of computing MCSes of unsatisfiable propositional formulas in conjunctive normal form (CNF). Recent work proposed algorithms that analyze variables instead of clauses [16, 27, 32]. The main goal of most novel algorithms is to reduce the number of calls to a SAT solver (i.e. the oracle queries) in such a way that the SAT solver calls remain *simple*, i.e. changes to the original formula are mostly negligible. Accordingly, most recent algorithms require a worst-case number of SAT oracle queries that is linear in the number of variables. Given that the number of clauses can be exponentially larger than the number of variables, these algorithms are in the worst-case asymptotically as efficient as using maximum satisfiability [35], while ensuring *simpler* oracle queries, in the sense that no new variables are used and encodings of cardinality constraints are not required [24].

The main contribution of this paper is to develop three novel algorithms for MCS extraction, aiming at reducing the number of oracle queries, both in theory and in practice. Two of the proposed algorithms are shown to require asymptotically fewer than linear SAT oracle queries in the number of (soft) clauses. Nevertheless, for problem domains where the number of (soft) clauses is linear in the number of (interesting) variables, the algorithms are shown to require asymptotically fewer oracle queries than recent algorithms [16, 27, 32]. Concrete examples of problem domains where the number of (soft) clauses is linear on the number of relevant variables include computing minimal and maximal models [6, 23]. Besides providing theoretical guarantees in terms of the worst-case number of SAT oracle queries, the new algorithms are shown to be effective in practice. On the one hand, the new algorithms are shown to be more robust, being less dependent on practical optimizations commonly used in practice. In addition, the new algorithms contribute to improving the performance of portfolios of MCS extraction algorithms.

It should also be emphasized that the proposed new algorithms find application in settings other than MCS extraction. As shown elsewhere [21, 28, 29], MCS extraction is an instantiation of the problem of computing a minimal set subject to a monotone predicate (MSMP), with a specific predicate form. The algorithms proposed in this paper can be applied to any computational problem that can be reduced to the same MSMP predicate form. A list of problems with the same MSMP predicate form as MCS extraction can be found in [28], and include finding a minimal distinguishing subset (MDS), computing a minimal (and a maximal) model, and computing a maximum autarky.

The paper is organized as follows. Section 2 introduces the notation and definitions used throughout. Section 3 provides a brief overview of recent work on MCS extraction. Section 4 builds on recent algorithms and develops an alternative MCS extraction

[1] This is the case for example with the analysis of infeasible systems of linear inequalities [10].

algorithm that requires a worst-case linear number of queries to the SAT oracle on the number of clauses. The insights provided by this new algorithm enable the development of two novel algorithms in Section 5 which require worst-case sublinear number of queries to a SAT oracle, on the number of clauses. This section also shows that a number of important classes of problems are shown to require sublinear number of queries to a SAT oracle, on the number of variables. Section 6 evaluates the proposed algorithms on standard problem instances [1, 2, 16, 27, 32]. The results demonstrate that the new algorithms proposed in this paper enable further improvements to the state of the art in MCS extraction. Section 7 concludes the paper.

2 Preliminaries

This section introduces the notation and definitions used throughout the paper. Standard propositional logic definitions apply (e.g. [7]). CNF formulas are defined over a set of propositional variables. A CNF formula $\mathcal{E}$ is a finite conjunction of clauses, also interpreted as a finite set of clauses. A clause is a disjunction of literals, also interpreted as a set of literals. A literal is a variable or its complement. $m \triangleq |\mathcal{E}|$ represents the number of clauses. The set of variables associated with a CNF formula $\mathcal{E}$ is denoted by $X \triangleq \mathrm{var}(\mathcal{E})$, with $n \triangleq |X|$. A CNF formula $\mathcal{E}$ is partitioned into a set of soft clauses $\mathcal{F}$, which can be relaxed (i.e. may not be satisfied), and a(n optional) set of hard clauses $\mathcal{H}$, which cannot be relaxed (i.e. must be satisfied). Thus, a CNF formula $\mathcal{E}$ is represented by a pair $\langle \mathcal{H}, \mathcal{F} \rangle$. Moreover, MUSes, MCSes and MSSes are defined over $\mathcal{F}$ taking into account the hard clauses in $\mathcal{H}$ as follows:

Definition 1 (Minimal Unsatisfiable Subset (MUS)). *Let $\mathcal{F}$ denote the set of soft clauses and $\mathcal{H}$ denote the set of hard clauses, such that $\mathcal{H} \cup \mathcal{F}$ is unsatisfiable. $\mathcal{M} \subseteq \mathcal{F}$ is a* Minimal Unsatisfiable Subset *(MUS) iff $\mathcal{H} \cup \mathcal{M}$ is unsatisfiable and $\forall_{\mathcal{M}' \subsetneq \mathcal{M}}$, $\mathcal{H} \cup \mathcal{M}'$ is satisfiable.*

Definition 2 (Minimal Correction Subset (MCS)). *Let $\mathcal{F}$ denote the set of soft clauses and $\mathcal{H}$ denote the set of hard clauses, such that $\mathcal{H} \cup \mathcal{F}$ is unsatisfiable. $\mathcal{C} \subseteq \mathcal{F}$ is a* Minimal Correction Subset *(MCS) iff $\mathcal{H} \cup \mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall_{\mathcal{C}' \subsetneq \mathcal{C}}$, $\mathcal{H} \cup \mathcal{F} \setminus \mathcal{C}'$ is unsatisfiable.*

Definition 3 (Maximal Satisfiable Subset (MSS)). *Let $\mathcal{F}$ denote the set of soft clauses and $\mathcal{H}$ denote the set of hard clauses, such that $\mathcal{H} \cup \mathcal{F}$ is unsatisfiable. $\mathcal{S} \subseteq \mathcal{F}$ is a* Maximal Satisfiable Subset *(MSS) iff $\mathcal{H} \cup \mathcal{S}$ is satisfiable and $\forall_{\mathcal{S}' \supsetneq \mathcal{S}}$, $\mathcal{H} \cup \mathcal{S}'$ is unsatisfiable.*

For query complexity analyses, the size of either the largest or the smallest MCS is denoted by k. Recent years have seen extensive work on extracting MUSes and MCSes. Recent work on MUS extraction is summarized for example in [2, 5, 29]. Recent work on MCS extraction is overviewed in the next section. Among the many applications of MCS extraction, a representative example is the computation of minimal and maximal models [6, 23]. Throughout the paper special emphasis will be given to problems for which the number of soft clauses is linear on the number of variables, with the computation of minimal and maximal models representing concrete examples.

Unless stated otherwise, in the remainder of the paper clauses are assumed to be half-reified [15], and the algorithms are described in terms of the reification variables. Concretely, the input to an MCS extraction algorithm is a pair of clauses $\langle \mathcal{H}, \mathcal{F} \rangle$, where $\mathcal{H}$ denotes the *hard* clauses, and $\mathcal{F}$ denotes the soft clauses, and such that $\mathcal{H} \not\models \bot$ and $\mathcal{H} \land \mathcal{F} \models \bot$. An initial assignment is selected from which two sets $\mathcal{S} \supseteq \mathcal{H}$ and $\mathcal{U} \subseteq \mathcal{F}$ are obtained, that denote respectively the satisfied and falsified clauses of $\langle \mathcal{H}, \mathcal{F} \rangle$. The clauses in $\mathcal{U}$ are half-reified using a set $\mathcal{R}$ of $\mathcal{O}(m)$ fresh variables [2, 31]. Each (soft) clause $c_i \in \mathcal{U}$ is replaced by a (hard) clause $(\neg r_i \lor c_i)$, which is added to $\mathcal{S}$, and the new variable r_i is added to $\mathcal{R}$. The set $\mathcal{U}$ of falsified soft clauses is replaced by a new set of unit (soft) clauses (r_i), one for each $r_i \in \mathcal{R}$. Given $\mathcal{U}$, each variable occurs with a single polarity (otherwise one of the clauses would be satisfied) [27]. The D clause is the disjunction of all literals in the clauses of $\mathcal{U}$; either the original literals or the reification literals can be considered. The D clause check step is to run a SAT oracle on $\mathcal{S} \land D$ (or on a subset of D), being used in recent MCS extraction algorithms [16,27,32].

Given a formula $\mathcal{E}$, a backbone literal l of $\mathcal{E}$ is such that $\mathcal{E} \models l$. Given an MCS $\mathcal{C}$ of a pair $\langle \mathcal{H}, \mathcal{F} \rangle$, any literal in $\mathcal{C}$ is the complement of a backbone literal of $\mathcal{H} \land (\mathcal{F} \setminus \mathcal{C})$ [27].

A SAT oracle is modeled as a function call that, in this paper, returns an outcome, which is either true (or SAT), for a satisfiable formula, or false (or UNSAT), for an unsatisfiable formula. For the cases the outcome is true, the SAT oracle also returns a witness of satisfiability μ (i.e. a satisfying truth assignment). Given a pair $\langle \mathcal{H}, \mathcal{F} \rangle$, a SAT solver call is thus represented as $(\text{st}, \mu) \leftarrow \text{SAT}(\mathcal{H} \land \mathcal{F})$, where st is either *true* (or SAT) or *false* (or UNSAT).

3 Related Work

The extraction of MCSes can be traced at least to the work of Reiter [36], in the form of minimal diagnoses[2]. A wealth of work has been developed since then. We emphasize the work more related with MCS extraction. Besides the work on algorithms for MCS and MUS extraction, there is a well-known hitting set duality relationship between MCSes and MUSes, which has been studied in different settings [3, 8, 25, 36].

A simple approach for MCS extraction is to use MaxSAT [25], with optimality guarantees in terms of the worst-case number of SAT oracle queries [9, 35]. Recent experimental results [27] indicate that the use of MaxSAT can perform poorly in practice, on representative classes of problem instances. As a result, most recent work focused on *simple* SAT oracle queries, where the modifications to the original formula are minimal. (For example, the use of cardinality constraints and the auxiliary variables required for CNF encoding cardinality constraints yields CNF formulas fairly different from the original CNF formula, which can be significantly harder.).

One of the most widely used MCS extraction algorithms is *linear search* [3, 27, 33]. Starting from a set $\mathcal{S}$ of satisfiable clauses and a set $\mathcal{U}$ of falsified clauses, the linear search algorithm iteratively tests the satisfiability of $\mathcal{S} \cup \{c\}$, for each clause $c \in \mathcal{U}$. If the outcome is true, the clause is added to $\mathcal{S}$; otherwise it is discarded, i.e. it is included in the MCS. The number of SAT oracle calls grows with the number of clauses,

[2] Nevertheless, irreducible inconsistent sets of linear inequalities have been studied for more than a century [10].

i.e. $\Theta(m)$. Recent work [21,28,29] has shown that linear search essentially corresponds to the well-known deletion-based MUS extraction algorithm [4, 11].

Inspired by the work on QuickXplain [22], FastDiag [14] represents an alternative for MCS extraction, which often requires a number of oracle calls smaller than linear (on the number of clauses), but which is linear in the worst-case. An alternative, similar to dichotomic search for MUS extraction [18], is the generation of corrective explanations [34], which is worse than linear in the worst-case.

The modification of the actual oracle, by introducing a preference on the literals was considered in different settings [1,37,38]. These approaches require a *single* SAT oracle call, with the main drawback being that the oracle is modified to solve SAT problems with preferences. Existing experimental evidence [32] (but also the results in this paper) indicate that the use of SAT with preferences does not scale for the more challenging MCS extraction problem instances.

Recent years have seen a number of alternative algorithms being proposed. The CLD algorithm [27] exploits the fact that falsified clauses do not have complemented literals, and so analyzes a single clause (i.e. the D clause) at each iteration. The D clause represents the disjunction of the literals in the falsified clauses of $\mathcal{U}$. An important observation is that the worst-case number of oracle calls for the CLD algorithm grows with the number of variables [32]. Moreover, it has been shown that the CLD can be applied to the more general setting of constraint programming [31], by using half reification [15]. The connection of backbone literals with MCSes was first investigated in [27] and further refined in [32]. These insights are used extensively throughout this paper. A variant of CLD was also recently proposed [2]. Similar to earlier work [31], half reification is also used.

The CMP algorithm [16] revisits the well-known insertion-based MUS extraction algorithm [12] in the context of MCS extraction[3]. Although apparently the CMP algorithm requires a close to quadratic number of oracle calls [16], it is also the case that the CMP algorithm checks the satisfiability of a (partial) D clause at every major iteration. The main consequence, as shown elsewhere [32], is that this technique reduces the worst-case number of oracle calls to linear, on the number of clauses.

A recent line of work has investigated algorithms that analyze falsified literals (which are bound by the number of variables) instead of clauses [32]. Although not claimed in earlier work, but an immediate consequence of this earlier work [32], is the observation that the number of oracle calls can be exponentially smaller than the size of the formula, because the number of clauses can be exponentially larger than the number of variables. The current state of the art algorithms for MCS extraction [16,32] both require a number of oracle calls that grows linearly with the number of variables. Concretely, LBX [32] requires $\Theta(n)$ SAT oracle calls and CLD requires $\mathcal{O}(\min\{n-r, m-k\})$, where k is the size of the smallest MCS and r is the smallest size of backbone literals in an MCS [32]. LBX, the currently best performing MCS extraction algorithm, is summarized in Algorithm 1. Inspired by CLD [27], and similarly to CMP [16], LBX performs a D clause check (i.e. the satisfiability of $\mathcal{S}$ conjoined with a global D clause) at line 5. If the function call returns true, then $\mathcal{S} \land D$ is unsatisfiable, and each literal in

[3] By exploiting the reduction of MCS extraction to MSMP [29], a wealth of different MUS extraction algorithms can be used for MCS extraction, including the insertion-based algorithm.

Algorithm 1: Literal-Based eXtractor (LBX) [32]

```
1  Function LBX(H, F)
2      (S, U) ← InitialAssignment(H, F)
3      (L, B) ← (Literals(U), ∅)
4      while L ≠ ∅ do
5          if CheckDClause(S, L) then break;
6          l ← RemoveLiteral(L)
7          (st, μ) = SAT(S ∪ B ∪ {l})
8          if st then
9              (S, U) ← UpdateSATClauses(μ, S, U)
10             L ← L ∩ Literals(U)
11         else
12             B ← B ∪ {¬l}
13     return F \ S                          // F \ S is MCS of F
```

Algorithm 2: Set up phase

```
1  Function Setup(H, F)
2      (S, U) ← InitialAssignment(H, F)
3      (S, U, R) ← ReifyClauses(U)
4      return (S, U, R)
```

D is the complement of a backbone literal of $\mathcal{S}$. It is important to note that the linear complexity of CMP is achieved only when a similar D clause check is executed at each iteration of the algorithm. (The experimental results Section 6 demonstrate the practical importance of the clause D check.) The extraction of preferred MCSes (and MUSes as well) has been investigated in [30, 39], extending earlier work on computing preferred explanations [22]. Finally, the algorithms proposed in this paper can be related with recent work on computing maximum autarkies [24].

4 MCS Extraction with Linear Oracle Calls

The algorithms described in this and the next section start with a set up phase shown in Algorithm 2. The set up phase picks an initial assignment that satisfies the clauses in $\mathcal{H}$ and some clauses in $\mathcal{F}$. The falsified clauses in $\mathcal{F}$, represented as set $\mathcal{U}$, are half-reified and added to $\mathcal{S}$. The set of reification variables is $\mathcal{R}$. Set $\mathcal{U}$ is rewritten to include the reification variables of the clauses originally in $\mathcal{U}$.

This section develops a new algorithm for MCS extraction. The algorithm is referred to as *Unit Clauses D* (UCD). Similarly to CLD [27], UCD uses D clauses. However, in contrast with CLD, all of the D clauses are *unit*. UCD is also tightly related with FastDiag [14] in that UCD iteratively splits a working set of clauses. Similar to earlier algorithms, an initial SAT call (see Algorithm 2) will split $\langle \mathcal{H}, \mathcal{F} \rangle$ into $\mathcal{S} \supseteq \mathcal{H}$ and $\mathcal{U} \subseteq \mathcal{F}$, respectively the satisfied and falsified clauses of $\langle \mathcal{H}, \mathcal{F} \rangle$. Let X_U be the set of falsified literals in the clauses of $\mathcal{U}$. Let $\mathcal{D}$ be the set of unit D clauses. A stack $\mathbb{T}$ of sets $\mathcal{D}$ is maintained, initially with a single set $\mathcal{D}$ containing one unit D clause for each falsified literal in $\mathcal{U}$. Moreover, $\mathcal{S}$ denotes the set of clauses that must be satisfied, starting from an initial assignment that includes the hard clauses $\mathcal{H}$. Algorithm 3 sum-

Algorithm 3: Unit Clauses D (UCD)

Function UCD ($\mathcal{H}$, $\mathcal{F}$)
  $(\mathcal{S}, \mathcal{U}, \mathcal{R}) \leftarrow \texttt{Setup}(\mathcal{H}, \mathcal{F})$
  $(\mathcal{B}, \mathcal{D}) \leftarrow (\emptyset, \texttt{CreateUnitClausesD}(\mathcal{R}))$
  $\texttt{Push}(\mathbb{T}, \mathcal{D})$
  while $\mathbb{T} \neq \emptyset$ **do**
    if $\texttt{CheckDClause}(\mathcal{S}, \mathcal{R})$ **then break**;
    $\mathcal{D} \leftarrow \texttt{Pop}(\mathbb{T})$
    $\mathcal{D} \leftarrow \texttt{UpdateDClauses}(\mathcal{D}, \mathcal{R}, \mathcal{B})$
    // **Invariant:** $\neg\text{SAT}(\mathcal{S} \wedge \mathcal{D})$
    if $|\mathcal{D}| \leq 1$ **then**
      $(\mathcal{B}, \mathcal{R}) \leftarrow \texttt{UpdateBackbones}(\mathcal{B}, \mathcal{R}, \mathcal{D})$
      continue
    $(\mathcal{D}_1, \mathcal{D}_2) \leftarrow \texttt{HalfSplit}(\mathcal{D})$
    $(\text{st}_1, \mu) \leftarrow \text{SAT}(\mathcal{S} \cup \cup_{D_i \in \mathcal{D}_1} \{D_i\})$
    if st_1 **then** $(\mathcal{S}, \mathcal{R}) \leftarrow \texttt{UpdateSATClauses}(\mu, \mathcal{S}, \mathcal{R})$
    $(\text{st}_2, \mu) \leftarrow \text{SAT}(\mathcal{S} \cup \cup_{D_i \in \mathcal{D}_2} \{D_i\})$
    if st_2 **then** $(\mathcal{S}, \mathcal{R}) \leftarrow \texttt{UpdateSATClauses}(\mu, \mathcal{S}, \mathcal{R})$
    // **Invariant:** $\neg\text{st}_1 \vee \neg\text{st}_2$
    if $\neg\text{st}_1$ **then** $\texttt{Push}(\mathbb{T}, \mathcal{D}_1)$
    if $\neg\text{st}_2$ **then** $\texttt{Push}(\mathbb{T}, \mathcal{D}_2)$
  return $\mathcal{F} \setminus \mathcal{S}$ // $\mathcal{F} \setminus \mathcal{S}$ is MCS of $\mathcal{F}$

marizes the organization of UCD. As can be observed, UCD shares a number of ideas with FastDiag [14] (and so with QuickXplain [22]), with the main differences summarized by the pseudo-code invariants. One invariant is that any set of D clauses in the stack is inconsistent with $\mathcal{S}$. Each set is split into two approximately equal sets of D clauses. Each such set is tested for consistency with $\mathcal{S}$. If the SAT formula is satisfiable, the set $\mathcal{S}$ is updated. The satisfied unit soft clauses (each with a single reified variable) become unit hard clauses. As a result, at least one call to the SAT solver is *guaranteed* to return false (i.e. the formula is unsatisfiable). Thus, at each iteration, if $|\mathcal{D}| > 1$, then at least one set of D clauses is added to the stack. The size of the sets of D clauses is strictly decreasing. Whenever a set with a single D clause is found, the complements of the associated literals are added to the set of backbone literals.

Proposition 1. *(Correctness of UCD) Algorithm 3 computes an MCS $\mathcal{C}$ of $\langle \mathcal{H}, \mathcal{F} \rangle$.*

Proof. (Sketch) By inspection the following invariants hold. First, the set of clauses $\mathcal{S}$ is satisfiable. Second, any set of unit D soft clauses consistent with $\mathcal{S}$ is added to $\mathcal{S}$, and consistency of $\mathcal{S}$ is preserved. If the set of unit clauses D is unsatisfiable given $\mathcal{S}$, then the set is split and the resulting sets of unit D clauses are pushed onto the stack. The size of the sets of D clauses in $\mathbb{T}$ is decreasing, and the stack is not updated when the size of the D clauses is less than or equal to 0. Thus the algorithm is terminating. Any clause $c \in \mathcal{F}$ not added to $\mathcal{S}$ is such that $\mathcal{S} \cup \{c\}$ is unsatisfiable. Finally, sets of D clauses of size not greater than 1 are either discarded (if there are no literals) or added as the (complement) of backbone literals, with $\mathcal{S} \wedge D \models \bot$. Thus, the UCD algorithm computes an MCS of $\langle \mathcal{H}, \mathcal{F} \rangle$. □

Proposition 2. *(Query Complexity of UCD) The worst case number of SAT oracle calls for the UCD algorithm is* $\mathcal{O}(k + k \log (\frac{m}{k}))$*, where* k *is the size of the largest MCS.*

Proof. (Sketch) The query complexity analysis ignores the additional SAT oracle query made by the D clause check set (see line 6). This additional call at most doubles the total number of SAT oracle queries. The analysis develops an upper bound on the number of oracle queries. The algorithm searches a tree with m possible leaves, with a total number of nodes bounded by $2m$. Since each node can make two oracle queries, except the leaves, then an upper bound on the number of oracle queries is $2m - 1$. Groups of literals that are not in the MCS, and so the outcome of the SAT query is true, are not pushed onto the stack. Groups of literals that contain one or more literals in the MCS, and so the outcome of the SAT query is false, are pushed onto the stack. Therefore, the number of oracle queries grows with the number of literals in the (largest) MCS, k with $k \geq 1$. To maximize the number of oracle queries the literals in the MCS must be as far apart as possible. Thus, we consider groups of $\frac{m-k}{k} + 1 = \frac{m}{k}$ literals, where the last one is a literal in the MCS. We now consider how each literal in the MCS is identified. The literal in the MCS needs to be selected after a number of queries that will select that literal out of $\frac{m}{k}$ literals. This corresponds to a tree with an upper bound of $2(\frac{m}{k}) - 1$ nodes. Of these, the SAT query will return false in $\log (\frac{m}{k}) + 1$ nodes and true in $\log (\frac{m}{k})$ nodes, because the subtree for the $\frac{m}{k}$ elements starts with an unsatisfiable call. Thus, the number of saved oracle queries becomes $2(\frac{m}{k}) - 1 - 2 \log (\frac{m}{k}) - 1$. Since there are k literals in the MCS, then the total number of oracle queries becomes $2m - 2k(\frac{m}{k}) + 2k \log (\frac{m}{k}) + 2k$, which can be simplified to $2k + 2k \log (\frac{m}{k})$. Thus, an upper bound on the number of oracle queries is $\mathcal{O}(k + k \log (\frac{m}{k}))$. □

For problem domains where the number of clauses are of the order of the number of variables (e.g. computing minimal and maximal models), the query complexity becomes asymptotically competitive with LBX [32] and CLD [27].

Corollary 1. *The query complexity of UCD is* $\mathcal{O}(k + k \log (\frac{n}{k}))$ *when* $m = \Theta(n)$.

5 MCS Extraction with Sublinear Oracle Calls

This section develops two novel algorithms for MCS extraction which, for specific classes of problems, will require asymptotically less than linear oracle calls. The two algorithms exploit the insights provided by the UCD algorithm (see Section 4). The algorithms also build on recent work on computing maximum autarkies [24].

5.1 Uniform Spliting with Binary Search

Instead of creating unit clauses D, the *Uniform Spliting with Binary Search* (UBS) algorithm, divides the reified variables into (approximately) $\sqrt{m}$ clauses D, each containing (approximately) $\sqrt{m}$ literals. The SAT oracle is run on $\mathcal{S}$ together with all the D clauses. If the formula is satisfiable, then at least $\sqrt{m}$ reification variables are satisfied, indicating that the corresponding clauses in $\mathcal{U}$ are consistent with $\mathcal{S}$. As a result, the sets of satisfied and falsified clauses, respectively $\mathcal{S}$ and $\mathcal{U}$, are updated. Moreover, the reification variables are dropped from $\mathcal{R}$. Otherwise, if the formula is unsatisfiable, a binary search step is used to find a culprit, i.e. a D clause which, together with $\mathcal{S}$ is

Algorithm 4: Uniform Splitting Binary Search (UBS)

Function UBS ($\mathcal{H}$, $\mathcal{F}$)
  $(\mathcal{S}, \mathcal{U}, \mathcal{R}) \leftarrow$ Setup$(\mathcal{H}, \mathcal{F})$
  $\mathcal{B} \leftarrow \emptyset$
  while $\mathcal{R} \neq \emptyset$ **do**
    if CheckDClause$(\mathcal{S}, \mathcal{R})$ **then break**;
    $\mathcal{D} \leftarrow$ CreateClausesD$(\mathcal{R})$ // Uniform split reif. variables
    $(\text{st}, \mu) \leftarrow \text{SAT}(\mathcal{S} \cup \cup_{D_i \in \mathcal{D}} \{D_i\})$
    if st **then**
      $(\mathcal{S}, \mathcal{R}) \leftarrow$ UpdateSATClauses$(\mu, \mathcal{S}, \mathcal{R})$
    else
      $(\mathcal{S}, \mathcal{B}, \mathcal{R}) \leftarrow$ FindCulprit$(\mathcal{S}, \mathcal{B}, \mathcal{R}, \mathcal{D})$
  return $\mathcal{F} \setminus \mathcal{S}$ // $\mathcal{F} \setminus \mathcal{S}$ is MCS of $\mathcal{F}$

Algorithm 5: Binary Search Step

Function FindCulprit $(\mathcal{S}, \mathcal{B}, \mathcal{R}, \mathcal{D})$
  while $|\mathcal{D}| > 1$ **do**
    // **Invariant:** $\neg\text{SAT}(\mathcal{S} \cup \cup_{D_i \in \mathcal{D}} D_i)$
    $(\mathcal{D}_1, \mathcal{D}_2) \leftarrow$ HalfSplit$(\mathcal{D})$
    $(\text{st}_1, \mu) \leftarrow \text{SAT}(\mathcal{S} \cup \cup_{D_i \in \mathcal{D}_1} \{D_i\})$
    if st_1 **then** $(\mathcal{S}, \mathcal{R}) \leftarrow$ UpdateSATClauses$(\mu, \mathcal{S}, \mathcal{R})$
    $(\text{st}_2, \mu) \leftarrow \text{SAT}(\mathcal{S} \cup \cup_{D_i \in \mathcal{D}_2} \{D_i\})$
    if st_2 **then** $(\mathcal{S}, \mathcal{R}) \leftarrow$ UpdateSATClauses$(\mu, \mathcal{S}, \mathcal{R})$
    // **Invariant:** $\neg\text{st}_1 \vee \neg\text{st}_2$
    $\mathcal{D} \leftarrow (\neg\text{st}_1)\ ?\ \mathcal{D}_1 : \mathcal{D}_2$
  $(\mathcal{B}, \mathcal{R}) \leftarrow$ UpdateBackbones$(\mathcal{B}, \mathcal{R}, \mathcal{D})$
  return $(\mathcal{S}, \mathcal{B}, \mathcal{R})$

unsatisfiable (or alternatively, $\mathcal{S}$ entails its complement). Algorithm 4 shows the main steps of the proposed approach. The binary search step is shown in Algorithm 5.

The operation of the binary search step used for finding the culprit is essential for correctness, and exploits the ideas used previously for the UCD algorithm. As shown in Algorithm 5, the loop invariant for the binary search procedure is such that the set of D clauses being analyzed is inconsistent with the current set $\mathcal{S}$.

Lemma 1. *The loop invariants of Algorithm 5 hold.*

Proof. (Sketch) Observe that any SAT oracle query with a satisfiable outcome causes the set $\mathcal{S}$ to be updated. Thus, one of the SAT oracle queries must return an unsatisfiable outcome in each loop iteration. □

Lemma 1 is used for arguing the correctness of the two algorithms investigated in this section.

Proposition 3. *(Correctness of UBS) Given $\langle \mathcal{H}, \mathcal{F} \rangle$, UBS (see Algorithm 4) computes an MCS $\mathcal{C}$ of $\langle \mathcal{H}, \mathcal{F} \rangle$.*

Proof. (Sketch) By inspection of Algorithm 4, any clause added to $\mathcal{S}$ is such that the resulting set of clauses is consistent. Given a SAT oracle query with an unsatisfiable

outcome, by Lemma 1 the binary search step in Algorithm 5 will find a single D clause which is inconsistent with (a possibly updated set) $\mathcal{S}$. Thus $\mathcal{S} \models \neg D$. This means that each literal in D is the complement of a backbone literal of $\mathcal{S}$, and so clauses composed of the complements of backbone literals are not consistent with $\mathcal{S}$. □

Proposition 4. *(Query Complexity of UBS) The number of SAT oracle calls for the UBS algorithm is* $\mathcal{O}(\sqrt{m}\log m)$.

Proof. (Sketch) The query complexity analysis ignores the additional SAT oracle query made by the D clause check (see line 5). This additional call at most doubles the total number of SAT oracle queries. By inspection, the algorithm removes $\mathcal{O}(\sqrt{m})$ literals in each iteration of the main loop. In the worst case, the algorithm runs the binary search step in each iteration of the main loop. The worst-case number of loop iterations for the binary search step is $\mathcal{O}(\log m)$. Thus, the overall number of calls to the SAT oracle is $\mathcal{O}(\sqrt{m}\log m)$. □

Corollary 2. *The query complexity of UBS is* $\mathcal{O}(\sqrt{n}\log n)$ *when* $m = \Theta(n)$.

Remark 1. For computing minimal or maximal models, the query complexity of UBS is sublinear on the number of variables.

5.2 Literal-Oriented Geometric Progression

For the UBS algorithm, there is no bias towards SAT or UNSAT outcomes. However, UNSAT outcomes have far greater cost, due to the need to find a culprit. As a result, it would be preferable to give preference to obtaining more SAT outcomes. Moreover, for the cases where the number of backbone literals in $\mathcal{U}$ is large, the uniform splitting of UBS may represent a drawback. A modification to the UBS algorithm is to start with larger D clauses, trying to mimic the behavior of the original CLD algorithm. The number of D clauses is increased by a geometric progression while instances are satisfied. Similarly, the number of literals in each D clause is reduced by a geometric progression. When an unsatisfiable instance is identified, the algorithm runs a binary search to identify a culprit. Thus, in contrast with UBS, the size and number of D clauses is not kept constant given the number of literals in $\mathcal{U}$. Algorithm 6 illustrates the proposed approach.

Proposition 5. *(Correctness of LOPZ) Given* $\langle \mathcal{H}, \mathcal{F} \rangle$*, LOPZ (see Algorithm 6) computes an MCS* $\mathcal{C}$ *of* $\langle \mathcal{H}, \mathcal{F} \rangle$.

Proof. (Sketch) By inspection of Algorithm 6, any clause added to $\mathcal{S}$ is such that the resulting set of clauses is consistent. Given a SAT oracle query with an unsatisfiable outcome, by Lemma 1 the binary search procedure in Algorithm 5 will find a single D clause which is inconsistent with (a possibly updated set) $\mathcal{S}$. Thus $\mathcal{S} \models \neg D$. This means that each literal in D is the complement of a backbone of $\mathcal{S}$, and so clauses composed of the complements of backbone literals are not consistent with $\mathcal{S}$. □

Proposition 6. *(Query Complexity of LOPZ) The number of SAT oracle calls for the LOPZ algorithm is* $\mathcal{O}(\sqrt{m}\log m)$.

Algorithm 6: Literal-Oriented Geometric Progression (LOPZ)

```
1  Function UBS (H, F)
2      (S, U, R) ← Setup(H, F)
3      (i, B) ← (0, ∅)
4      while R ≠ ∅ do
5          if CheckDClause(S, R) then break
6          D ← CreateClausesD(R, min(1, |R|/2^i))
7          (st, μ) ← SAT(S ∪ ∪_{D_i ∈ D} {D_i})
8          if st then
9              (S, R) ← UpdateSATClauses(μ, S, R)
10             i ← i + 1
11         else
12             (S, B, R) ← FindCulprit(S, B, R, D)
13             i ← 0
14     return F \ S                                  // F \ S is MCS of F
```

Proof. (Sketch) The query complexity analysis ignores the additional SAT oracle query made by the D clause check (see line 5). This additional call at most doubles the total number of SAT oracle queries. We first count the number of removed literals for a sequence of satisfiable oracle calls. If the current number of literals is m_i and the i^{th} iteration takes j_i steps, i.e. the number of satisfiable instances until the first unsatisfiable instance, then the number of removed literals is at least, $\Delta_i = 2^{j_i} - 1 + \frac{m_i}{2^{j_i+1}}$. Thus, we can define the variation in the number of elements with a recurrence:

$$m_i = \begin{cases} m & i = 0 \\ m_{i-1} - 2^{j_{i-1}} + 1 - \frac{m_{i-1}}{2^{j_{i-1}+1}} & i \geq 1 \end{cases}$$

where m is the initial number of distinct literals in $\mathcal{R}$. Observe that the value of each j_i cannot exceed $\log m$. The LOPZ algorithm can be viewed as a sequence of phases, where a phase ends when the index i is reset to 0. For each phase, an upper bound on the number of queries to the SAT oracle is $\mathcal{O}(\log m)$. This results from the number of SAT oracle calls returning true being $\mathcal{O}(\log m)$ and, to find a culprit, binary search also makes $\mathcal{O}(\log m)$ oracle calls. Now, the number of phases is maximized when the number of elements removed in each phase is minimized. Thus, we want to find the value of j_i that minimizes Δ_i given some initial number of elements m_i. To find the minima of the function above, we can differentiate with respect to j_i and equate to 0. Thus, the minimum number of removed elements is achieved for $j_i = \frac{1}{2}\log\frac{m_i}{2} = \log\sqrt{\frac{m_i}{2}}$. This means that the minimum number of literals removed in each phase is $\sqrt{\frac{m_i}{2}} - 1 + \sqrt{\frac{m_i}{2}} = 2\sqrt{\frac{m_i}{2}} - 1$, and so the number of phases is $\mathcal{O}(\sqrt{m})$. Thus, an upper bound on the number of queries to the SAT oracle is $\mathcal{O}(\sqrt{m}\log m)$. □

Although the query complexity of LOPZ and UBS are the same, we expect LOPZ to perform better in practice, since preference is given to SAT outcomes.

Corollary 3. *The query complexity of LOPZ is* $\mathcal{O}(\sqrt{n}\log n)$ *when* $m = \Theta(n)$.

Remark 2. For computing minimal or maximal models, the query complexity of LOPZ is sublinear on the number of variables.

Remark 3. As pointed out earlier (see Section 1), the algorithms described in this and the previous section, namely UCD, UBS and LOPZ can be used in settings other than MCS extraction. Concretely, the algorithms can be formulated in the general setting of solving the MSMP problem [29], and used with any computational problem with the same predicate form as MCS extraction [21, 28]. Concretely, the UCD, UBS and LOPZ algorithms can be formulated in terms of computing a minimal set subject to a monotone predicate for predicates of form $\mathscr{L}$.

6 Experimental Results

This section conducts an experimental evaluation of the algorithms proposed in this paper with state-of-the-art MCS extractors. Representative sets of problem instances were considered [32]. The experiments were performed in Ubuntu Linux running on an Intel Xeon E5-2630 2.60GHz processor with 64GByte of memory. The time limit was set to 1800s and the memory limit to 10GByte. All the described algorithms were implemented in a prototype tool named MCSXL[4] (***MCS eXtractor with subLinear number of oracle calls***). The MCSXL tool is written in C++ on top of the MINISAT SAT solver [13]. Besides, the experiments also used the state-of-the-art MCS extractors: CLD [27], CMP (with all its optimizations) [16], RS [1], as well as LBX [32]. In all cases, the original binaries provided by the authors were run according to their instructions. Other well-known MCS extraction algorithms including basic linear search [3, 33], MaxSAT [25], SAT with preferences [37, 38] (resp. nOPTSAT and SAT&PREF), and FastDiag [14] have been shown to be outperformed by recent approaches. A recently proposed MCS extraction algorithm [2] is not publicly available. In addition, the experimental results in [2] show gains with respect to CLD [27] and RS [1], but these gains are not competitive with other recent approaches [16, 32].

6.1 Performance Comparison

The benchmark suite considered in this paper is referred to as *IJCAI15* and comprises three different sets of unsatisfiable instances considered previously in [32]. These include *ijcai13-bench* containing 866 plain and partial MaxSAT instances taken from [27], *mus-bench* with 295 plain instances from the 2011 MUS competition proposed in [16], as well as 179 plain and partial instances from the *maxsat-bench* considered in [32] and taken from the 2014 MaxSAT evaluation. All the three benchmark sets were filtered by removing *easy* instances. An instance was declared to be easy if it was solved by each of the considered algorithms within 5 seconds. As a result, after filtering easy instances the total number of instances in the IJCAI15 benchmark suite is 556.

The cactus plot reporting the performance of the considered algorithms measured for the IJCAI15 problem instances is shown in Figure 1. The algorithms proposed in this paper perform comparably to LBX, solving one more instance than LBX. The plot also includes a VBS (virtual best solver) based on picking the best result out of LBX, UCD, UBS and LOPZ. It is important to emphasize that the good performance of some algorithms, more concretely of LBX and CMP, but also of UCD, is in part explained by the regular clause D checks. The cactus plot shown in Figure 2 illustrates the effect of removing the clause D check step in LBX, UCD, UBS and LOPZ. As can be

[4] Available at `http://logos.ucd.ie/web/doku.php?id=mcsxl`

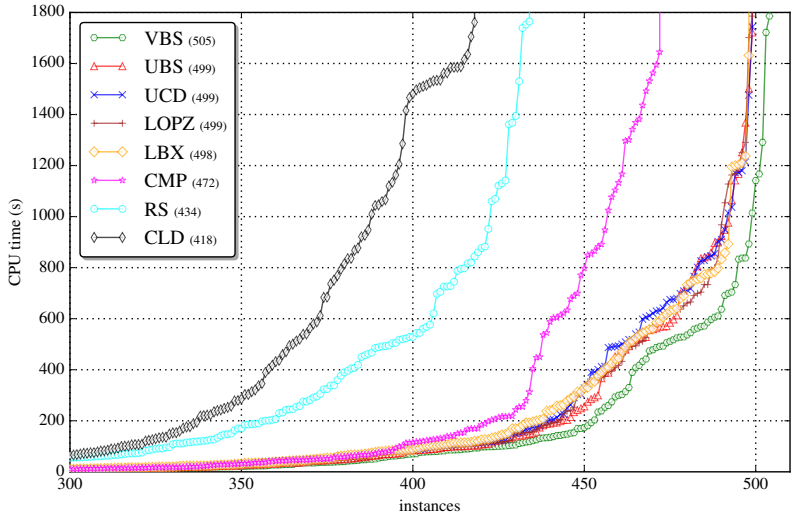

Fig. 1: Results for the IJCAI15 [32] problem instances.

Table 1: Average number of SAT oracle calls for the IJCAI15 [32] problem instances.

	CLD	CMP	LBX	UCD	UBS	LOPZ	$\text{LBX}_{\text{w/o D}}$	$\text{UCD}_{\text{w/o D}}$	$\text{UBS}_{\text{w/o D}}$	$\text{LOPZ}_{\text{w/o D}}$
S	91.9	5.9	16.5	11.5	9.3	8.4	131.9	30.5	9.1	7.4
U	3651.3	24.2	5.5	7.4	2.5	1.8	799.8	1641.9	71.5	2.8
A	3743.2	30.0	21.9	18.8	11.8	10.2	931.6	1672.4	80.6	10.2

observed, the performance of LBX and UCD is strongly dependent on the use of the clause D check step. However, this is not the case with LOPZ and UBS. Our interpretation of these results is that both LOPZ and UBS are more robust than either LBX and UCD, since both are less dependent upon the clause D check step. Another measure of robustness of the MCS extraction algorithms is the number of SAT oracle calls. This information is summarized in Table 1 (**S**, **U**, **A** represent the number of SAT, UNSAT and total calls). The reduction in the number of SAT oracle calls observed in practice confirms the theory, with LOPZ and UBS achieving the smallest number of SAT oracle calls. The large number of SAT oracle calls for CLD results for the identification of disjoint unsatisfiable cores as a preprocessing step [27], being used in the default configuration. Furthermore, it can be observed that the number of SAT oracle calls increases significantly (i.e. between one and two orders of magnitude) when the clause D check step is not used in LBX and UCD. For UBS there is a modest increase, whereas for LOPZ the average number of SAT oracle calls remains unchanged (with a reduction in the number of SAT outcomes and an increase in the number of UNSAT outcomes).

6.2 Analysis of the VBS

The VBS in both Figure 1 and Figure 2 shows gains when compared to each separate algorithm. This confirms that the new algorithms have an observable contribution

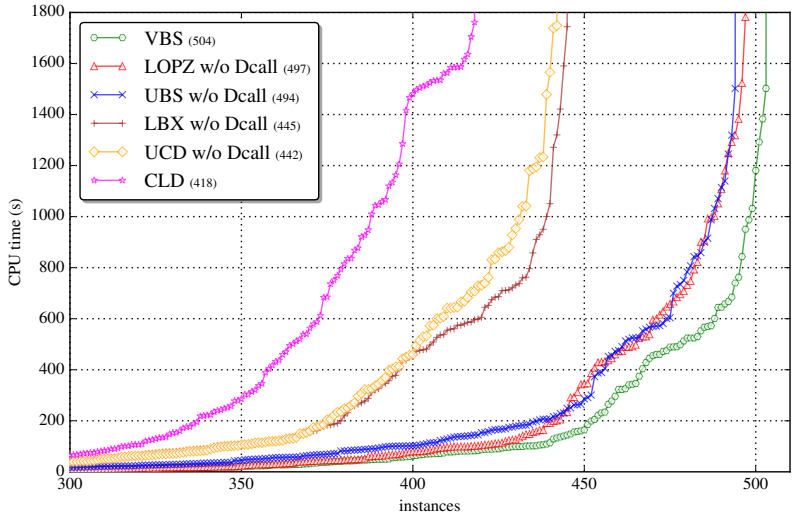

Fig. 2: Results for the IJCAI15 [32] problem instances, without the D clause check.

Table 2: Contribution to the VBS by LBX, UCD, UBS, and LOPZ. [* Observe that the algorithms make use of D calls for Figure 1 but do not use D calls for Figure 2.]

	LBX*	UCD*	UBS*	LOPZ*	Total
Figure 1	82	108	120	194	504
Figure 2	66	81	124	232	503

to improving the state of the art in MCS extraction. Table 2 summarizes the contribution to the VBS. Regarding the VBS in Figure 1, and as can be observed, the percentage of instances for which the selected solver is either LOPZ, UBS, UCD or LBX is, respectively, 38.5%, 33.8%, 21.4% and 16.3%. For Figure 2, without the use of the clause D check step, the numbers are even more conclusive. The data shown provides additional evidence of the contribution of LOPZ, UBS and UCD to the state of the art in MCS extraction.

Figure 3 shows a scatter plot comparing LOPZ with LBX. As can be observed, there is an observable performance gain of LOPZ when compared to LBX. More importantly, LOPZ adds to the robustness of MCS extraction, with fewer outliers than LBX. Figure 3 also includes a scatter plot highlighting the run times of the instances for which the selected solver is not LBX. As can be observed, the new algorithms LOPZ, UBS and UCD contribute for the VBS for the larger run times. Again, the conclusion is that the new algorithms contribute to increase the robustness of MCS extraction. Table 3 complements the scatter plots of Figure 3 and shows an analysis of the percentage of pairwise performance wins between the different algorithms. As can be concluded, LOPZ performs better than LBX for every around 2 out of 3 problem instances.

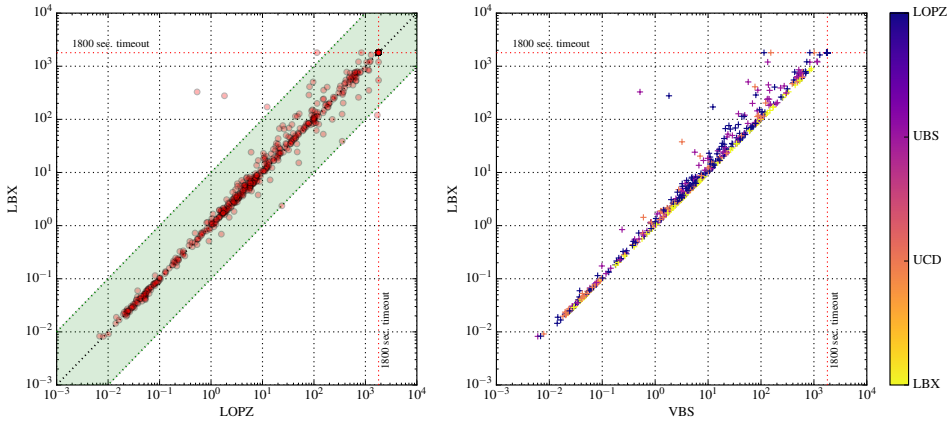

Fig. 3: Scatter plots comparing LBX with LOPZ and the VBS.

Table 3: Comparison of the percentage of pairwise wins between LBX, UCD, UBS, and LOPZ.

	LBX	UCD	UBS	LOPZ
LBX wins		46.2	44.4	35.1
UCD wins	53.6		48.2	36.5
UBS wins	55.0	51.4		34.5
LOPZ wins	64.3	63.1	65.1	

7 Conclusions

The paper extends recent work on MCS extraction [1,16,27,32]. Recently proposed highly efficient algorithms have been shown to be linear on the number of variables [32]. This paper develops a new algorithm with worst-case linear number of oracle calls, on the number of clauses. In addition, the paper develops two new algorithms which require asymptotically fewer than linear number of oracle calls, on the number of clauses. However, for representative classes of problems, including the computation of minimal and maximal models, the algorithms are shown to require asymptotically fewer than linear oracle calls on the number of variables. This improves earlier results for specific problem domains. The paper also argues that the proposed algorithms can be easily adapted to solving specific predicate forms of the MSMP problem [21,28,29].

The experimental results show that the new algorithms perform comparably to the state of the art in MCS extraction. More importantly, the new algorithms are shown to enable observable performance gains when portfolios of algorithms are considered. The experimental results also confirm that, in practice, far fewer oracle calls may not guarantee large performance gains, even when the calls are expected to be simple. Earlier work [27] had reached similar conclusions when using MaxSAT for MCS extraction. Finding the best possible balance between fewer oracle calls and best SAT solver performance remains an open research subject.

Acknowledgement. This work was partly funded by grant TIN2013-46511-C2-2-P.

References

1. F. Bacchus, J. Davies, M. Tsimpoukelli, and G. Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841, 2014.
2. F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV*, pages 70–86, 2015.
3. J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186, 2005.
4. R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI*, pages 276–281, 1993.
5. A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2):97–116, 2012.
6. R. Ben-Eliyahu and R. Dechter. On computing minimal models. *Ann. Math. Artif. Intell.*, 18(1):3–27, 1996.
7. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
8. E. Birnbaum and E. L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1):25–46, 2003.
9. C. Cayrol, M.-C. Lagasquie-Schiex, and T. Schiex. Nonmonotonic reasoning: From complexity to algorithms. *Ann. Math. Artif. Intell.*, 22(3-4):207–236, 1998.
10. J. W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*, volume 118. Springer Science & Business Media, 2008.
11. J. W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.
12. J. L. de Siqueira N. and J. Puget. Explanation-based generalisation of failures. In *ECAI*, pages 339–344, 1988.
13. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003. Available from: https://github.com/niklasso/minisat.
14. A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012.
15. T. Feydy, Z. Somogyi, and P. J. Stuckey. Half reification and flattening. In *CP*, pages 286–301, 2011.
16. É. Grégoire, J. Lagniez, and B. Mazure. An experimentally efficient method for (MSS, coMSS) partitioning. In *AAAI*, pages 2666–2673, 2014.
17. É. Grégoire, B. Mazure, and C. Piette. On approaches to explaining infeasibility of sets of boolean clauses. In *ICTAI*, pages 74–83, 2008.
18. F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *ECAI*, pages 113–117, 2006.
19. A. Ignatiev, A. Previti, M. H. Liffiton, and J. Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP*, pages 173–182, 2015.
20. M. Jampel. A brief overview of over-constrained systems. In M. Jampel, E. C. Freuder, and M. J. Maher, editors, *Over-Constrained Systems*, volume 1106 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 1995.
21. M. Janota and J. Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233:73–83, 2016.
22. U. Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.
23. D. J. Kavvadias, M. Sideri, and E. C. Stavropoulos. Generating all maximal models of a boolean expression. *Inf. Process. Lett.*, 74(3-4):157–162, 2000.
24. O. Kullmann and J. Marques-Silva. Computing maximal autarkies with few and simple oracle queries. In *SAT*, pages 138–155, 2015.

25. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
26. R. Mangal, X. Zhang, A. Kamath, A. V. Nori, and M. Naik. Scaling relational inference using proofs and refutations. In *AAAI*, 2016.
27. J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622, 2013.
28. J. Marques-Silva and M. Janota. Computing minimal sets on propositional formulae I: problems & reductions. *CoRR*, abs/1402.3011, 2014.
29. J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in Boolean formulae. In *CAV*, pages 592–607, 2013.
30. J. Marques-Silva and A. Previti. On computing preferred MUSes and MCSes. In *SAT*, pages 58–74, 2014.
31. C. Mencía and J. Marques-Silva. Efficient relaxations of over-constrained CSPs. In *ICTAI*, pages 725–732, 2014.
32. C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979, 2015.
33. A. Nöhrer, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In *VaMoS*, pages 83–91, 2012.
34. B. O'Callaghan, B. O'Sullivan, and E. C. Freuder. Generating corrective explanations for interactive constraint satisfaction. In *CP*, pages 445–459, 2005.
35. C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
36. R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
37. E. D. Rosa and E. Giunchiglia. Combining approaches for solving satisfiability problems with qualitative preferences. *AI Commun.*, 26(4):395–408, 2013.
38. E. D. Rosa, E. Giunchiglia, and M. Maratea. Solving satisfiability problems with preferences. *Constraints*, 15(4):485–515, 2010.
39. R. Walter, A. Felfernig, and W. Kuchlin. Inverse QUICKXPLAIN vs. MAXSAT – a comparison in theory and practice. In *Configuration Workshop*, volume CEUR 1453, pages 97–104, 2015.