Efficiently Explaining CSPs with Unsatisfiable Subset Optimization

Emilio Gamba\(^1\), Bart Bogaerts\(^1\) and Tias Guns\(^1,2\)

\(^1\)Vrije Universiteit Brussel, Belgium
\(^2\)KU Leuven, Belgium

emilio.gamba@vub.be, bart.bogaerts@vub.be, tias.guns@kuleuven.be

Abstract

We build on a recently proposed method for explaining solutions of constraint satisfaction problems. An explanation here is a sequence of simple inference steps, where the simplicity of an inference step is measured by the number and types of constraints and facts used, and where the sequence explains all logical consequences of the problem. We build on these formal foundations and tackle two emerging questions, namely how to generate explanations that are provably optimal (with respect to the given cost metric) and how to generate them efficiently. To answer these questions, we develop 1) an implicit hitting set algorithm for finding optimal unsatisfiable subsets; 2) a method to reduce multiple calls for (optimal) unsatisfiable subsets to a single call that takes constraints on the subset into account, and 3) a method for re-using relevant information over multiple calls to these algorithms. The method is also applicable to other problems that require finding cost-optimal unsatisfiable subsets. We specifically show that this approach can be used to effectively find sequences of optimal explanation steps for constraint satisfaction problems like logic grid puzzles.

1 Introduction

Building on old ideas to explain domain-specific propagations performed by constraint solvers [Sqalli and Freuder, 1996; Freuder et al., 2001], we recently introduced a method that takes as input a satisfiable constraint program and explains the solution-finding process in a human-understandable way [Bogaerts et al., 2020]. Explanations in that work are sequences of simple inference steps, involving as few constraints and facts as possible. The explanation-generation algorithms presented in that work rely heavily on calls for Minimal Unsatisfiable Subsets (MUS) [Marques-Silva, 2010] of a derived program, exploiting a one-to-one correspondence between so-called non-redundant explanations and MUSs. The explanation steps in the seminal work are heuristically optimized with respect to a given cost function that should approximate human-understandability, e.g., taking the number of constraints and facts into account, as well as a valuation of their complexity (or priority). The algorithm developed in that work has two main weaknesses: first, it provides no guarantees on the quality of the produced explanations due to internally relying on the computation of \(\subseteq\)-minimal unsatisfiable subsets, which are often suboptimal with respect to the given cost function. Secondly, it suffers from performance problems: the lack of optimality is partly overcome by calling a MUS algorithm on increasingly larger subsets of constraints for each candidate implied fact. However, using multiple MUS calls per literal in each iteration quickly causes efficiency problems, causing the explanation generation process to take several hours.

Motivated by these observations, we develop algorithms that aid explaining CSPs and improve the state-of-the-art in the following ways:

- We develop algorithms that compute (cost-)Optimal Unsatisfiable Subsets (from now on called OUSs) based on the well-known hitting-set duality that is also used for computing cardinality-minimal MUSs [Ignatiev et al., 2015; Saikko et al., 2016].
- We observe that many of the individual calls for MUSs (or OUSs) can actually be replaced by a single call that searches for an optimal unsatisfiable subset among subsets satisfying certain structural constraints. In other words, we introduce the Optimal Constrained Unsatisfiable Subsets (OCUS) problem and we show how \(O(n^2)\) calls to MUS/OUS can be replaced by \(O(n)\) calls to an OCUS oracle, where \(n\) denotes the number of facts to explain.
- Finally, we develop techniques for optimizing the O(C)US algorithms further, exploiting domain-specific information coming from the fact that we are in the explanation-generation context. One such optimization is the development of methods for information re-use between consecutive OCUS calls.

In this paper, we apply our OCUS algorithms to generate step-wise explanations of satisfaction problems. However, MUSs have been used in a variety of contexts, and in particular lie at the foundations of several explanation techniques [Junker, 2001; Ignatiev et al., 2019; Espasa et al., 2021]. We conjecture that OCUS can also prove useful in those settings, to take more fine-grained control over which MUSs, and eventually, which explanations are produced.
The rest of this paper is structured as follows. We discuss background on the hitting-set duality in Section 2. Section 3 motivates our work, while Section 4 introduces the OCUS problem and a generic hitting set-based algorithm for computing OCUSs. In Section 5 we show how to optimize this computation in the context of explanations and in Section 6 we experimentally validate the approach. We discuss related work in Section 7 and conclude in Section 8.

2 Background

We present all methods using propositional logic but our results easily generalize to richer languages, such as constraint languages, as long as the semantics is given in terms of a satisfaction relation between expressions in the language and possible states of affairs (assignments of values to variables).

Let \( \Sigma \) be a set of propositional symbols, also called atoms; this set is implicit in the rest of the paper. A literal is an atom \( p \) or its negation \( \neg p \). A clause is a disjunction of literals. A formula \( \mathcal{F} \) is a conjunction of clauses. Slightly abusing notation, a clause is also viewed as a set of literals and a formula as a set of clauses. We use the term clause and constraint interchangeably. A (partial) interpretation is a consistent (not necessarily truth valued) valuation of a propositional formula \( F \) of \( \Sigma \). Let \( \mathcal{F} \) be a set of propositional symbols, also called atoms; this set is implicit in the rest of the paper. A literal \( l \) is a consequence of a formula \( \mathcal{F} \) if \( l \) holds in all \( \mathcal{F} \)'s models. If \( \mathcal{I} \) is a set of literals, we write \( \overline{\mathcal{I}} \) for the set of literals \( \{ \neg l \mid l \in \mathcal{I} \} \).

Definition 1. A Minimal Unsatisfiable Subset (MUS) of \( \mathcal{F} \) is an unsatisfiable subset \( S \) of \( \mathcal{F} \) for which every strict subset of \( S \) is satisfiable. MUSs(\( \mathcal{F} \)) denotes the set of MUSs of \( \mathcal{F} \).

Definition 2. A set \( S \subseteq \mathcal{F} \) is a Maximal Satisfiable Subset (MSS) of \( \mathcal{F} \) if \( S \) is satisfiable and for all \( S' \subseteq S \subseteq \mathcal{F}, S' \) is unsatisfiable.

Definition 3. A set \( S \subseteq \mathcal{F} \) is a correction subset of \( \mathcal{F} \) if \( \mathcal{F} \setminus S \) is satisfiable. Such a \( S \) is a minimal correction subset (MCS) of \( \mathcal{F} \) if no strict subset of \( S \) is also a correction subset. MCSs(\( \mathcal{F} \)) denotes the set of MCSs of \( \mathcal{F} \).

Each MCS of \( \mathcal{F} \) is the complement of an MSS of \( \mathcal{F} \) and vice versa.

Definition 4. Given a collection of sets \( \mathcal{H} \), a hitting set of \( \mathcal{H} \) is a set \( h \) such that \( h \cap C \neq \emptyset \) for every \( C \in \mathcal{H} \). A hitting set is minimal if no proper subset of it is also a hitting set.

The next proposition is the well-known hitting set duality [Liffton and Sakallah, 2008; Reiter, 1987] between MCSs and MUSs that forms the basis of our algorithms, as well as algorithms to compute MSSs [Davies and Bacchus, 2013] and cardinality-minimal MUSs [Ignatiev et al., 2015].

Proposition 5. A set \( S \subseteq \mathcal{F} \) is an MCS of \( \mathcal{F} \) iff it is a minimal hitting set of MUSs(\( \mathcal{F} \)). A set \( S \subseteq \mathcal{F} \) is a MUS of \( \mathcal{F} \) iff it is a minimal hitting set of MCSs(\( \mathcal{F} \)).

3 Motivation

Our work is motivated by the problem of explaining satisfaction problems through a sequence of simple explanation steps. This can be used to teach people problem-solving skills, to compare the difficulty of related satisfaction problems (through the number and complexity of steps needed), and in human-computer solving assistants.

Experiments have shown that many MUS calls are performed to increase the chance of finding a good MUS, and hence that algorithmic improvements are needed to make it more practical. We see three main points of improvement, all of which will be tackled by our generic OCUS algorithm presented in the next section:

- First of all, since the algorithm is based on MUS calls, there is no guarantee that the explanation found is indeed optimal (with respect to the given cost function). Performing multiple MUS calls is only a heuristic that is used to circumvent the restriction that there are no algorithms for cost-based unsatisfiable subset optimization.

- Second, this algorithm uses MUS calls for every literal to explain separately. The goal of all these calls is to find a single unsatisfiable subset of \( C \setminus I \setminus (I_{\text{end}} \setminus I) \) that contains exactly one literal from \( I_{\text{end}} \setminus I \). This begs the

---

**Algorithm 1:** EXPLAIN-ONE-STEP(\( C, f, I, I_{\text{end}} \))

1. \( X_{\text{best}} \leftarrow \text{nil} \)
2. for \( l \in (I_{\text{end}} \setminus I) \) do
3. \( X \leftarrow \text{MUS}(C \land I \land \neg l) \)
4. if \( f(X) < f(X_{\text{best}}) \) then
5. \( X_{\text{best}} \leftarrow X \)
6. end
7. end
8. return \( X_{\text{best}} \)
questions whether it is possible to compute a single (optimal) unsatisfiable subset subject to constraints. In our case, the constraint is to include exactly one literal from $\{ \text{f} \in \mathbb{I} \setminus \mathbb{T} \}$.

- Finally, the algorithm that computes an entire explanation sequence makes use of repeated calls to EXPLAIN-ONE-STEP and hence will solve many similar problems. This raises the issue of incrementality: can we re-use the computed data structures to achieve speed-ups in later calls?

4 Optimal Constrained Unsatisfiable Subsets

The first two considerations from the previous section lead to the following definition.

**Definition 6.** Let $\mathcal{F}$ be a formula, $f : 2^{\mathcal{F}} \rightarrow \mathbb{N}$ a cost function and $p$ a predicate $p : 2^{\mathcal{F}} \rightarrow \{ \text{true}, \text{false} \}$. We call $\mathcal{S} \subseteq \mathcal{F}$ an OCUS of $\mathcal{F}$ (with respect to $f$ and $p$) if

- $\mathcal{S}$ is unsatisfiable,
- $p(\mathcal{S})$ is true
- all other unsatisfiable $\mathcal{S}' \subseteq \mathcal{F}$ for which $p(\mathcal{S}')$ is true satisfy $f(\mathcal{S}') \geq f(\mathcal{S})$.

If we assume that the predicate $p$ is specified itself as a CNF over (meta-)variables indicating inclusion of clauses of $\mathcal{F}$, and $f$ is obtained by assigning a weight to each such meta-variable, then the complexity of the problem of finding an OCUS is the same as that of the SMUS (cardinality-minimal MUS) problem [Ignatiev et al., 2015]: the associated decision problem is $\Sigma_2^p$-complete. Hardness follows from the fact that SMUS is a special case of OCUS, containment follows - intuitively - from the fact that this can be encoded as an $\exists \forall$-QBF using a Boolean circuit encoding of the costs.

When considering the procedure EXPLAIN-ONE-STEP from the perspective of OCUS defined above, the task of the procedure is to compute an OCUS of the formula $\mathcal{F} := \mathcal{C} \land \mathcal{I} \land \mathbb{I} \setminus \mathbb{T}$ with $p$ the predicate that holds for subsets that contain exactly one literal of $\mathbb{I} \setminus \mathbb{T}$, see Algorithm 2.

In order to compute an OCUS of a given formula, we propose to build on the hitting set duality of Proposition 5. For this, we will assume to have access to a solver CONDOPTHITTINGSET that can compute hitting sets of a given collection of sets that are optimal (w.r.t. a given cost function $f$) among all hitting sets satisfying a condition $p$. The choice of the underlying hitting set solver will thus determine which types of cost functions and constraints are possible. In our implementation, we use a cost function $f$ as well as a condition $p$ that can easily be encoded as linear constraints, thus allowing the use of highly optimized mixed integer programming (MIP) solvers. The CONDOPTHITTINGSET formulation is as follows:

\[
\begin{align*}
\text{minimize}_S & \quad f(S) \\
\text{s.t.} & \quad p(S) \\
& \quad \sum_{H \in \mathcal{H}} H \geq 1, \quad \forall H \in \mathcal{H} \\
& \quad s \in \{0, 1\}, \quad \forall s \in S
\end{align*}
\]

where $S$ is a set of MIP decision variables, one for every clause in $\mathcal{F}$. In our case, $p$ is expressed as $\sum_{s \in \mathbb{I} \setminus \mathbb{T}} s = 1$.

Our generic algorithm for computing OCUSs is depicted in Algorithm 3. It combines the hitting set-based approach for MUSs of [Ignatiev et al., 2015] with the use of a MIP solver for (weighted) hitting sets as proposed for maximum satisfiability [Davies and Bacchus, 2013]. The key novelty is the ability to add structural constraints to the hitting set solver, without impacting the duality principles of Proposition 5, as we will show.

Ignoring Line 7 for a moment, the algorithm alternates calls to a hitting set solver with calls to a SAT oracle. If the SAT oracle returns true, i.e., the subset $S$ is satisfiable, the complement of $S$ is a correction subset of $\mathcal{F}$ and is added to $\mathcal{H}$.

As in the SMUS algorithm of Ignatiev et al. [2015], our algorithm contains an (optional) call to Grow. The purpose of the Grow is to expand a satisfiable subset of $\mathcal{F}$ further, to find a smaller correction subset and as such find stronger constraints on the hitting sets. In our case, the calls for hitting sets will also take into account the cost ($f$), as well as the meta-level constraints ($p$); as such, it is not clear a priori which properties a good Grow function should have here. We discuss the different possible implementations of Grow later and evaluate their performance in Section 6. For correctness of the algorithm, all we need to know is that it returns a satisfiable subset $S'$ of $\mathcal{F}$ with $S \subseteq S'$.

Soundness and completeness of the proposal follow from the fact that all sets added to $\mathcal{H}$ are correction subsets, and Theorem 7, which states that what is returned is indeed a solution and that a solution will be found if it exists.

**Theorem 7.** Let $\mathcal{H}$ be a set of correction subsets of $\mathcal{F}$. If $S$ is a hitting set of $\mathcal{H}$ that is $f$-optimal among the hitting sets of $\mathcal{H}$ satisfying a predicate $p$, and $S$ is unsatisfiable, then $S$ is an OCUS of $\mathcal{F}$.

If $\mathcal{H}$ has no hitting sets satisfying $p$, then $\mathcal{F}$ has no OCUSs.
A hitting set solver will search for a cost-minimal assignment of \( \mathcal{F} \) into \( \mathcal{H} \). Not all unsatisfiable subsets can be used. In practice however, the unsatisfiable subsets are specific to explanation sequence generation.

The main question is, suppose two OCUS calls are done, first with inputs \( F_1, f_1, \) and \( p_1 \), and later with \( F_2, f_2, \) and \( p_2 \): how can we make use as much as possible of the data computations of the first call to speed-up the second call? The answer is surprisingly elegant. The most important data OCUS keeps track of is the collection \( \mathcal{H} \) of correction subsets that need to be hit.

This collection in itself is not useful for transfer between two calls, since – unless we assume that \( F_2 \) is a subset of \( F_1 \), there is no reason to assume that a set in \( \mathcal{H}_1 \) should also be hit in the second call. However, each set \( \mathcal{H} \) in \( \mathcal{H} \) is the complement (with respect to the formula at hand) of a satisfiable subset of constraints, and this satisfiability remains true. Thus, instead of storing \( \mathcal{H} \), we can keep track of a set \( SS_2 \) of satisfiable subsets (the sets \( S \) in the OCUS algorithm). When a second call to OCUS is performed, we can then initialize \( \mathcal{H} \) as the complement of each of these satisfiable subsets with respect to \( F_2 \), i.e.,

\[
\mathcal{H} \leftarrow \{F_2 \setminus S \mid S \in SS_2\}.
\]

The effect of this is that we bootstrap the hitting set solver with an initial set \( \mathcal{H} \).

For hitting set solvers that natively implement incrementality, we can generalize this idea further: we know that all calls to OCUS(\( F, f, p \)) will be cast with \( F \subseteq C \cup I_{end} \cup \overline{T_{end}} \). The hitting set solver will search for a cost-minimal assignment that hits all sets in \( \mathcal{H} \) and that furthermore contains exactly one of \( c_6 \) and \( c_7 \) (due to \( p \)). Table 1 shows the computed steps in the different iterations of Algorithm 3 given the above input. In this example, the GROW we used is the one called Max-Actual-Unif in Section 6.
cost function. In this way, the MIP solver will automatically maintain and reuse previously found sets-to-hit in each of its computations.

Explanations with Bounded OUS. Instead of working OCUS-based, we can now also generate optimal explanations by replacing the MUS call by an OUS call in Algorithm 1 (where OUS is computed as in Algorithm 3, but with a trivially true \( p \)). When doing this, we know that every OCUS of cost greater than or equal to \( f(X_{\text{best}}) \) will be discarded by the check on Line 4 of Algorithm 1. As such, a next optimization is to, instead of searching for an OUS, perform a bounded OUS check, which only computes an OUS in case one of cost smaller than a given bound \( ub \) exists. In our specific implementation, bounded OUS is performed by interrupting this OUS-call (after Line 3 Algorithm 3) if \( f(S) > ub \).

Since the bounding on the OUS cost has the most effect if cheap OUSs are found early in the loop across the different literals, we keep track of an upper bound of the cost of an OUS for each literal to explain. This is initialized to a value greater than any OUS, e.g., as \( f(C \land I_0 \land I_{\text{end}} \land I_0) \), and is updated every time an OUS explaining that literal is found; when going through the loop in Line 2 of Algorithm 3, we then handle literals in order of increasing upper bounds.

Domain-Specific Implementations of GROW. The goal of the GROW procedure is to turn \( S \) into a larger subformula of \( F \). The effect of this is that the complement added to \( H \) will be smaller, and hence, a stronger restriction for the hitting set solver is found.

Choosing an effective GROW procedure requires finding a difficult balance: on the one hand, we want our subformula to be as large as possible (which ultimately would correspond to computing the maximum satisfiable subformula), but on the other hand we also want the procedure to be very efficient as it is called in every iteration.

For the case of explanations we are in, we make the following observations:

- Our formula at hand (using the notation from the EXPLAIN-ONE-STEP-OCUS algorithm) consists of three types of clauses: (i) translations of the problem constraints (this is \( C \)) (ii) literals representing the assignment found thus far (this is \( I \)), and (iii) the negations of literals not-yet-derived (this is \( I_{\text{end}} \setminus I \)).
- \( C \) and \( I \) together are satisfiable, with assignment \( I_{\text{end}} \), and mutually supportive, by this we mean that making more constraints in \( C \) true, more literals in \( I \) will automatically become true and vice versa.
- The constraint \( p \) enforces that each hitting set will contain exactly one literal of \( I_{\text{end}} \setminus I \).

Since the restriction on the third type of elements of \( F \) are already strong, it makes sense to use the GROW \((S,F)\) procedure to search for a maximal satisfiable subset of \( C \cup I \) with hard constraints that \( S \) should be satisfied, using a call to an efficient (partial) MaxSAT solver. Furthermore, we can initialize this call as well as any call to a SAT solver with the polarities for all variables set to the value they take in \( I_{\text{end}} \).

We evaluate different grow strategies in the experiments section, including the use of partial MaxSAT as well as weighted partial MaxSAT based on the weights in the cost function \( f \).

Example 1 (cont.) Consider line 0 in table 1. During the GROW procedure, the MaxSAT solver Max-Actual-Unif with polarities set to \( I_{\text{end}} \) branches when multiple assignment to a literal are possible. By hinting the polarities of the literals, we guide the solver and it assigns all values according to the end interpretation and neither \( c_6 \) nor \( c_7 \) is taken.

6 Experiments

We now experimentally validate the performance of the different versions of our algorithm. Our benchmarks were run on a compute cluster, where each explanation sequence generation was assigned a single core on a 10-core INTEL Xeon Gold 61482 (Skylake) processor, a timelimit of 120 minutes and a memory-limit of 4GB. Everything was implemented in Python on top of PySAT\(^1\) and is available at https://github.com/ML-KULeuven/ocus-explain. For MIP calls, we used Gurobi 9.0, for SAT calls MiniSat 2.2 and for MaxSAT calls RC2 as bundled with PySAT (version 0.1.6.dev11). In the MUS-based approach we used PySAT’s deletion-based MUS extractor MUSX [Marques-Silva, 2010].

All of our experiments were run on a direct translation to PySAT of the 10 puzzles of Boegaerts et al. [2020]\(^2\). We used a cost of 60 for puzzle-agnostic constraints; 100 for puzzle-specific constraints; and cost 1 for facts. When generating an explanation sequence for such puzzles, the unsatisfiable subset identifies which constraints and which previously derived facts should be combined to derive new information. Our experiments are designed to answer the following research questions:

Q1 What is the effect of requiring optimality of the generated MUSs on the quality of the generated explanations?

Q2 Which domain-specific GROW methods perform best?

Q3 What is the effect of the use of constrainedness on the time required to compute an explanation sequence?

Q4 Does re-use of information across the different iterations improve efficiency?

Explanation quality. To evaluate the effect of optimality on the quality of the generated explanations, we re-implemented a MUS-based explanation generator based on Algorithm 1. Before presenting the results, we want to stress that this is not a fair comparison with the implementation of Boegaerts et al. [2020], since there – in order to avoid the quality problems we will illustrate below – an extra inner loop was used that employs even more calls to MUS for a selected set of subsets of \( C \) of increasing size. While this yields better explanations, it comes at the expense of computation time, thereby leading to several hours to generate the explanation of a single puzzle.

To answer Q1, we ran the MUS-based algorithm as described in Algorithm 1 and compared at every step the cost

\(^1\)https://pysathq.github.io

\(^2\)In one of the puzzles, an error in the automatic translation of the natural language constraints was found and fixed.
of the produced explanation with the cost of the optimal explanation. These costs are plotted on a heatmap in Figure 1, where the darkness represents the number of occurrences of the combination at hand. We see that the difference in quality is striking in many cases, with the MUS-based solution often missing very cheap explanations (as seen by the two dark squares in the column around cost 60), thereby confirming the need for a cost-based OUS/OCUS approach.

**Domain-specific GROW.** In our OCUS algorithm, we do not just aim to find any satisfiable subsets, but we prefer high quality satisfiable subsets: subsets that impose strong constraints on the assignments the optimal hitting set solver can find. This induces a trade-off between efficiency of the GROW strategy and quality of the produced satisfiable subset.

Thus, to answer Q2, we compared variations of OCUS that only differ in which GROW strategy they use. Figure 2 depicts the average (over all the puzzles) cumulative explanation time to derive a number of literals. Note that most puzzles only contain 150 literals, except for 2, which contain 96 and 250 literals respectively. When a method times out for a puzzle at one step, a runtime value of 7200 is used in computing the averages for all future steps. The configurations used are as follows:

- **Max** refers to growing with a MaxSAT solver and Greedy to growing using a heuristic method implemented by repeated sat calls, while no-grow refers to skipping the GROW step.

- **Full** refers to using the full unsatisfiable formula $F$ while Actual refers to using only the constraints that hold in the final interpretation (see Section 5). For instance, for the MaxSAT-based calls, Actual means that only the previously derived facts and the original constraints are taken into account when computing optimality.

- The MaxSAT solver (Max) is combined with different weighing schemes: uniform weights (unif), cost-function weights (pos) (equal to the weights in $f$), or the inverse of these costs (inv) defined as $\max_j(w_j)+1-w_i$.

We can observe that not using a grow strategy performs badly, as do weighted MaxSAT grows with costs $w_i$ (-Pos). Greedy grow strategies improve on not using a grow strategy, but not substantially. The two approaches that work best use the domain-specific knowledge of doing a MaxSAT grow on $C \cup I$, with the unweighted variant the only one that never times out.

**Constrainedness and incrementality.** To answer Q3 and Q4, we compare the effect of constrainedness in the search for explanations (C) and incrementality. Next to OCUS, we also include the bounded OUS approach (OUSb), where we call the OUS algorithm for every literal in every step, but we reuse information by giving it the current best bound $f(X_{best})$ and iterating over the literals that performed best in the previous call first. Based on the previous experiment, we always use (both for OUSb and OCUS) Max-Actual-Unif as grow strategy. For OCUS, incrementality (+Incr. HS) is achieved by reusing the same incremental MIP hitting set solver throughout the explanation calls, as explained in Section 5. To have a better view on how incrementality also affects OUSb, we add incrementality to it in the following ways:

- **SS. caching** keeps track of the satisfiable subsets, which are used to initialize $\mathcal{H}$ for a fresh hitting set solver instance each time.

- **Lit. Incr. HS** uses a separate incremental hitting set solver for every literal to explain, throughout the explanation calls. Once the literal is explained, the hitting set solver is discarded.

Figure 3 shows the results, where In this figure, the configurations are compared in a similar fashion to Figure 2.

When comparing the configurations, we see that the plain MUS-based implementation is faster than the $O(C)US$ imple-
mentations, as it solves a simpler problem (with worse quality results as shown in Q1). Replacing MUS by bounded OUS calls (orange line) leads to a much larger computation cost. The generic SS caching technique adds additional overhead.

The OCUS variants significantly improve runtime compared to those two bounded OUS approaches, by reducing the number of OUS calls. For OCUS, using an incremental hitting set solver across all steps seems to be slightly faster for deriving literals earlier in the sequence, while inducing a small overhead for the literals at the end of the sequence.

When looking at the runtime to explain the entire sequence, best results are obtained with OUSb + Lit. Incr. HS, that is, using an incremental hitting set solver for every individual literal to explain. However, for the first literals, we can see that it takes much more computation time and that reducing the number of OUS calls from \(n\) to 1 per explanation step improves runtime (OCUS). However, making each of the \(n\) calls incremental and bounded across the entire explanation sequence generation process leads to an even faster process overall (OUSb+Lit. Incr. HS).

7 Related Work

In the last few years, driven by the increasingly many successes of Artificial Intelligence (AI), there is a growing need for explainable artificial intelligence (XAI) [Miller, 2019]. In the research community, this need manifests itself through the emergence of (interdisciplinary) workshops and conferences on this topic [Miller et al., 2019; Hildebrandt et al., 2020] and American and European incentives to stimulate research in the area [Gunning, 2017; Hamon et al., 2020; FET, 2019].

While the main focus of XAI research has been on explaining black-box machine learning systems [Lundberg and Lee, 2017; Guidotti et al., 2018; Ignatiev et al., 2019], also model-based systems, which are typically considered more transparent, are in need of explanation mechanisms. Indeed, by advances in solving methods in research fields such as constraint programming [Rossi et al., 2006] and SAT [Biere et al., 2009], as well as by hardware improvement, such systems now easily consider millions of alternatives in short amounts of time. Because of this complexity, the question arises how to generate human-interpretable explanations of the conclusions they make. Explanation methods for model-based systems have been considered mostly for explain unsatisfiable problem instances [Junker, 2001], and have recently seen a rejuvenation in various subdomains of constraint reasoning [Fox et al., 2017; Čyraš et al., 2019; Chakraborti et al., 2017; Bogaerts et al., 2020].

In this context, we recently introduced step-wise explanations [Bogaerts et al., 2020] and applied them to Zebra puzzles; similar explanations, but for a wider range of puzzles, have been investigated by Espasa et al. [2021]. Our current work is motivated by a concrete algorithmic need: to generate these explanations efficiently, we need algorithms that can find optimal MUSs with respect to a given cost function, where the cost function approximates human-understandability of the corresponding explanation step. The closest related works can be found in the literature on generating or enumerating MUSs [Lynce and Silva, 2004; Lifiton et al., 2016]. Different techniques are employed to find MUSs, including manipulating resolution proofs produced by SAT solvers [Goldberg and Novikov, 2003; Gersmann et al., 2008; Dershowitz et al., 2006], incremental solving to enable/disable clauses and branch-and-bound search [Oh et al., 2004], or by BDD-manipulation methods [Huang, 2005]. Other methods work by means of translation into a so-called Quantified MaxSAT [Ignatiev et al., 2016], a field that combines the expressivity of Quantified Boolean Formulas (QBF) [Kleine Büning and Bubeck, 2009] with optimization as known from MaxSAT [Li and Manya, 2009], or by exploiting the so-called hitting set duality [Ignatiev et al., 2015] bootstrapped using MCS ENUMERATION [Marques-Silva et al., 2020]. An abstract framework for describing hitting set-based algorithms, including optimization was developed by Saikko et al. [2016]. While our approach can be seen to fit the framework, the terminology is focused on MaxSAT rather than MUS and would complicate our exposition. To the best of our knowledge, only few have considered optimizing MUSs: the only criterion considered yet is cardinality-minimality [Lynce and Silva, 2004; Ignatiev et al., 2015].

8 Conclusion, Challenges and Future work

We presented a hitting set-based algorithm for finding optimal constrained unsatisfiable subsets, with an application in generating explanation sequence for constraint satisfaction problems. We extended our methods with incrementality, as well as with a domain-specific method for extending satisfiable subsets (GROW). This domain-specific GROW method was key to generating explanation sequences in a reasonable amount of time. We noticed that, independently, incrementality and constrainedness have major benefits on explanation generation time. The best method on the tested puzzles was the incremental, bounded but non-constrained variant. It remains an open question how to make constrainedness and incrementality work together more effectively, as well as how to further optimize O(C)US-based explanations, for instance using disjoint MCS enumeration [Marques-Silva et al., 2020].

With the observed impact of different ‘GROW’ methods, an open question remains whether we can quantify precisely and in a generic way what a good or even the best set-to-hit is in a hitting set approach. The synergies of our approach with the more general problem of QMaxSAT [Ignatiev et al., 2016] is another open question.

The concept of bounded (incremental) OUS and OCUS are not limited to explanations of satisfaction problems and we are keen to explore other applications too. A general direction here are explanations of optimization problems and the role of the objective function in explanations.

Acknowledgments

This research received partial funding from the Flemish Government (AI Research Program); the FWO Flanders project G070521N; and funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-OPT).