Simplifying Step-Wise Explanation Sequences

Ignace Bleukx  DTAI, KU Leuven, Belgium
Jo Devriendt  DTAI, KU Leuven, Belgium
Emilio Gamba  Data Analytics Lab, VUB, Brussels, Belgium
Bart Bogaerts  Artificial Intelligence Lab, VUB, Brussels, Belgium
Tias Guns  DTAI, KU Leuven, Belgium
Data Analytics Lab, VUB, Brussels, Belgium

Abstract
Explaining constraint programs is useful for debugging an unsatisfiable program, to understand why a given solution is optimal, or to understand how to find a unique solution. A recently proposed framework for explaining constraint programs works well to explain the unique solution to a problem step by step. It can also be used to step-wise explain why a model is unsatisfiable, but this may create redundant steps and introduce superfluous information into the explanation sequence. This paper proposes methods to simplify a (step-wise) explanation sequence, to generate simple steps that together form a short, interpretable sequence. We propose an algorithm to greedily construct an initial sequence and two filtering algorithms that eliminate redundant steps and unnecessarily complex parts of explanation sequences. Experiments on diverse benchmark instances show that our techniques can significantly simplify step-wise explanation sequences.

2012 ACM Subject Classification  Theory of computation → Constraint and logic programming

Keywords and phrases  explanation, deduction, constraint programming, propagation

Digital Object Identifier  10.4230/LIPIcs.CP.2023.11

Supplementary Material  Software (Source code): https://github.com/ML-KULeuven/SimplifySeq
archived at swh:1:dir:a0de0a65a6c2c3a69d84a1de4047a360d8854195

Funding  This research was partly funded by the Flemish Government (AI Research Program), the Research Foundation - Flanders (FWO) project G070521N; the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt) and the European Union’s Horizon 2020 research and innovation programme under grant agreement No 101070149, project Tuples.

1 Introduction

As AI agents become more expressive and powerful, a growing need arises for explainable AI: methods that can explain the decisions of an automated agent. Such explanations can be needed for legal reasons, but are also essential to provide trust to users. Considerable research has gone into developing explanation techniques for black-box machine learning methods [17, 27], including techniques based on formal models [33].

1 E.g., the GDPR – https://gdpr.eu
11.2 Simplifying Step-Wise Explanation Sequences

But also explaining formal models themselves, such as constraint programs and satisfiability problems [37], is of high importance. Although individual constraints typically have a clear meaning, their interaction can be highly non-trivial, which led to the development of explanation methods for constraint programs [21]. A prominent branch of explainable constraint solving is occupied with explaining why a set of constraints is unsatisfiable. Most of these methods [30, 32, 26, 28, 31, 25, 29] extract a minimal unsatisfiable subset (MUS): a (minimal) subset of the constraints that renders the problem unsatisfiable. Such a MUS provides a user with a (potentially large) subset of constraints that yield inconsistency. Recently, there has been work on guiding users with respect to what can be done to restore feasibility [22, 39], but there is a lack of tools to better explain why a problem is inconsistent.

In a sense, traces or proof-logs of a solver (as common in SAT [24] and recently also finding its way into richer formalisms [2, 18, 3, 8, 4]) provide an explanation of why a model is unsatisfiable. Still, they would quickly overwhelm a user, as it involves constraint reformulations, auxiliary variables, and branching decisions. Other solver-generated explanations can be extracted from highly effective Lazy Clause Generation solvers [34] combining constraint propagation and SAT solving. In this setting, every propagation is explained by adding a clause to the SAT solver. However, these kinds of explanations are by nature restricted to explaining the propagation of a single constraint. Instead, we start from step-wise explanations [7, 15] where each explanation step in a sequence refines a partial assignment, using a minimal set of constraints as well as facts derived in previous steps.

These step-wise approaches were developed in the context of explaining the unique solution of satisfiable logic puzzles. Each step explains why a certain variable in the unique solution took that specific value. An example is explaining how to solve a Sudoku-puzzle [13, 20] where each step derives the value of a cell in the solution. In this context, the number of explanation steps is at most the number of variables in the problem.

But this explanation framework can be applied more broadly to constraint satisfaction problems (CSPs), in particular also to those that are unsatisfiable. A step-wise explanation of an over-constraint model allows to debug it, by listing steps similar to a debugger for programming languages. Each step shows a (preferably small) subset of constraints causing the removal of allowed values in variables domains, up to where a conflict is derived.

In contrast to explanation sequences for satisfiable problems, in the unsatisfiable case, only a subset of the variables and derived values contribute to the conflict and should therefore be explained. Therefore, directly applying the step-wise explanations framework to this new setting results in overly complex explanations.

Furthermore, finding the shortest sequence of explanations is a hard problem. A recent paper [6] touches upon the complexity of finding the shortest sequence of arc-consistency propagations steps. In that setting, the goal is to explain the full result of the arc-consistency algorithm. In the general case, where other/stronger propagation algorithms are used to find propagation steps, the problem may become even harder.

In this paper, we do not consider finding the best explanation sequence. Instead, we investigate how to find good – interpreted here as short and with small steps – step-wise explanations in the context of explaining why a CSP is unsatisfiable. The proposed techniques also apply to explaining the objective value of an optimization problem, explaining the solution(s) of constraint problems, and can be of use in interactive configuration problems [23].

For this, we contribute the following:

1. For the first time, we consider the quality of explanation sequences as a whole;
2. We formalize the properties that good explanation sequences, and explanation steps from which they are built, should adhere to;
We propose a new normal form for explanation sequences; in sections 5 and 6 we propose new algorithms to simplify sequences in this normal form; we show our methods significantly simplify explanation steps and sequences compared to current approaches.

2 Preliminaries

We now formalize constraint satisfaction problems and step-wise explanation sequences as used throughout this paper.

2.1 Constraint Satisfaction Problems (CSPs)

A CSP is a triple $(X, D, C)$ with $X$ a set of variables; $D$ a set of domains $D_x$ of allowed values for each variable $x$ of $X$, i.e., $D = \{D_x \mid x \in X\}$; $C$ a set of constraints, each over a subset of the variables.

A full assignment to a CSP $(X, D, C)$ is a mapping such that each variable takes a value from its domain. A constraint is a function mapping full assignments to true or false, typically described by a formula (e.g., $x + y \geq 1$). A constraint is satisfied by a full assignment if it maps the constraint to true. A solution to a CSP is a full assignment satisfying all constraints in $C$. A CSP is unsatisfiable if it has no solution. A set of constraints $S$ is a logical consequence of another set $S'$ if all solutions to $S'$ are solutions to $S$ – written as $S' \models S$. Constraints can be arranged in a constraint graph where nodes represent variables that are connected by an edge if they co-occur in a constraint.

For the remainder of this paper, we assume the set of variables and their domains are known. A positive literal is an equality $x = v$ and a negative literal is an inequality $x \neq v$ for variable $x$ and value $v$. Negative literals represent the constraint that a variable cannot be assigned to some value from its domain. We will often employ sets of negative literals, where we use $x \in R$ as a shorthand for $\{x \neq v \mid v \in D_x \setminus R\}$. For example, with $D_x = \{0, 1, 2, 3\}$, $x \in \{1, 2\}$ means the negative literals $\{x \neq 0, x \neq 3\}$. In other words, $x \in R$ denotes a set of negative literals enforcing that $x$ can only take values remaining in $R$. Note that a positive literal $x = v$ is equivalent to the set of negative literals $x \in \{v\}$. For the remainder of this paper, literals are assumed to be negative unless explicitly mentioned otherwise. With $\bot$ we denote the trivial inconsistency, i.e., the singleton set containing the literal false.

For simplicity, this paper only uses integer domains for variables. An integer domain is represented as either a finite range with a lower and upper bound, or an enumerated set. E.g., the range $[0..3]$ and the set $\{0, 1, 2, 3\}$ are identical. All ideas and algorithms in this paper apply to non-integer finite domains as well.

Given a set of constraints partitioned in soft constraints and a set of hard constraints, a minimal unsatisfiable subset (MUS) is a subset of the soft constraints that is unsatisfiable in conjunction with the hard constraints, and for which all strict subsets are satisfiable in conjunction with these hard constraints. MUS-calculation techniques are a well-studied research field and several algorithms for this exist [25, 32].

2.2 Step-wise explanations

The step-wise explanation framework was introduced in the context of first-order logic and Boolean satisfiability [10, 9, 7, 15]. We here reinterpret it from a finite-domain CSP perspective.
Given a CSP \((\mathcal{X}, \mathcal{D}, \mathcal{C})\), an explanation step is a triple \((\mathcal{E}, \mathcal{S}, \mathcal{N})\), where the input \(\mathcal{E}\) and the output \(\mathcal{N}\) are disjoint sets of literals, and \(\mathcal{S} \subseteq \mathcal{C}\) is the constraint subset, such that \(\mathcal{E} \cup \mathcal{S} = \mathcal{N}\). Informally, an explanation step consists of a subset of constraints which, together with some input literals, imply “new” output literals.

Given a CSP \((\mathcal{X}, \mathcal{D}, \mathcal{C})\), a target set of literals \(\mathcal{T}\) and a given set of literals \(\mathcal{G}\), an explanation sequence of length \(n\) from \(\mathcal{G}\) to \(\mathcal{T}\) is a sequence \(((\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i))_{1 \leq i \leq n}\) of explanation steps where \(\mathcal{E}_i \subseteq \mathcal{G} \cup \bigcup_{1 \leq j < i} \mathcal{N}_j\) and \(\mathcal{T} \subseteq \mathcal{G} \cup \bigcup_{1 \leq j \leq n} \mathcal{N}_j\). Informally, an explanation sequence is a sequence of explanation steps where each step derives some new literals and can do so using \(\mathcal{G}\) as well as previously derived outputs. Eventually, the sequence should derive all literals in \(\mathcal{T}\). As a whole, an explanation sequence explains how a target set is entailed by the union of a given set and the constraints of a CSP.

For satisfiable CSPs with a unique solution, we can let \(\mathcal{T}\) be a unique variables assignment. In general, we can set \(\mathcal{T}\) to the intersection of all solutions. An explanation sequence where \(\mathcal{G} = \emptyset\) and \(\mathcal{T}\) is inconsistent explains a CSP’s unsatisfiability.

We say an explanation step derives or explains a literal \(l\) if \(l \in \mathcal{N}\), and a sequence derives or explains a literal if one of its steps does.

The maximal output for an input \(\mathcal{E}\) and constraint subset \(\mathcal{S}\) is the set of all literals implied by \(\mathcal{E} \cup \mathcal{S}\), minus \(\mathcal{E}\). The maximal output can be calculated using a FullPropagate algorithm which finds the set of literals that are true in all solutions to the constraints. Such a FullPropagate function is implemented in multiple systems such as the Answer Set Programming system Clasp [16], the IDP system [11], and more recently in the pseudo-Boolean solver Exact [12, 14].

**Definition 2 (Maximal sequence).** An explanation step \((\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)\) in a sequence is maximal if (1) \(\mathcal{E}_i\) is the union of all previously derived literals and the given set, i.e., \(\mathcal{E}_i = \mathcal{G} \cup \bigcup_{j < i} \mathcal{N}_j\), and (2) \(\mathcal{N}_i\) is the maximal output of \(\mathcal{E}_i\) and \(\mathcal{S}_i\). An explanation sequence where all steps are maximal is a maximal sequence.

Given a sequence of constraint sets \((\mathcal{S}_i)_{1 \leq i \leq n}\) and a given set \(\mathcal{G}\), the maximal stepwise explanation sequence is the unique sequence \(((\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i))_{1 \leq i \leq n}\) where all explanation steps are maximal. Clearly, maximal sequences contain much more input/output literals than a user would care about. Moreover, calculating the maximal output of a step using FullPropagate is an expensive operation. In the general case, any sound propagation algorithm can be used to calculate the output of a step, but using a maximal one will provide us with a useful normal form.

**Example 3.** Consider the following unsatisfiable CSP and explanation sequences which we will use as a running example:

\[
\begin{align*}
\mathcal{X} & = \{x, y, z, v, w\} \\
\mathcal{D} & = \{D_x = D_y = [1..3], D_z = D_v = D_w = [0..3]\} \\
\mathcal{C} & = \{x + y + z \geq 7, x + y + w \leq 4, x \leq v, v + z \leq 3\}
\end{align*}
\]

## 3 Greedy initial sequence construction

Naively, generating explanation sequences involves constructing sequences of explanation steps that neatly match each other’s input and output. In existing work [7, 15], explanation sequences are constructed using an iterative loop that greedily searches for the best next explanation step to add. Each individual step in the explanation sequence is optimal with respect to an assumed cost function or heuristic. This cost function for example takes into account the number of constraints and the number of input literals used for each step.
This is precisely what is done in the algorithms presented in [7], by enumerating subsets

\[ 3.1 \] Properties

Example 4. The table below shows a maximal explanation sequence with given set \( G = \emptyset \) and target set \( T = \bot \) for the following unsatisfiable CSP:

\[ X = \{p, q, r, s\} \]
\[ D = \{D_p = D_q = D_r = D_s = [0..3]\} \]
\[ C = \{p + q \leq 1, q + 2r \leq 4, 3s + p + r \leq 1, alldiff(p, q, r, s)\} \]

To find a set of constraints and literals that explains a new literal \( n \), given a sequence of

\[ i = 1 \]
\[ E = \emptyset \]
\[ S \geq 7 \]
\[ N \neq 0 \]
\[ 2 \]
\[ E = \emptyset \]
\[ S \geq 7 \]
\[ z \in [3..3] \]
\[ 3 \]
\[ z \in [3..3] \]
\[ x \leq v \]
\[ v + z \leq 3 \]
\[ \bot \]

\[ (a) \]
\[ \begin{array}{|c|c|c|c|}
<table>
<thead>
<tr>
<th>i</th>
<th>E</th>
<th>S</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>x + y + z \geq 7</td>
<td>\bot</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>x + y + z \geq 7</td>
<td>z \in [3..3]</td>
</tr>
<tr>
<td>3</td>
<td>z \in [3..3]</td>
<td>x \leq v</td>
<td>v + z \leq 3</td>
</tr>
</tbody>
</table>
\end{array} \]

\[ (b) \]
\[ \begin{array}{|c|c|c|c|}
<table>
<thead>
<tr>
<th>i</th>
<th>E</th>
<th>S</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>x + y + z \geq 7</td>
<td>z \in [3..3]</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>x + y + w \leq 4</td>
<td>w \neq 3</td>
</tr>
<tr>
<td>3</td>
<td>v \in [1..3]</td>
<td>z \in [3..3]</td>
<td>v + z \leq 3</td>
</tr>
</tbody>
</table>
\end{array} \]

\[ \text{Table 1} \] Two explanation sequences for the same unsatisfiability for \( G = \emptyset \) and \( T = \bot \).

Such a construction method works well for explaining unique solutions, where the target

\[ \text{Properties} \]

\[ \text{Lemma 5.} \] For each step \( (E, S, N) \) in a sequence generated by \textsc{construct-greedy}, and for each \( S' \subseteq S \), the maximal output for \( S' \) and \( E \) is empty.

Otherwise \textsc{construct-greedy} would have picked \( S' \) to form a step with \( E \). A direct consequence of this is that the sequences constructed by \textsc{construct-greedy} are atomic:

\[ \text{Property 1 (Atomic).} \] An explanation step \( (E, S, N) \) is atomic if no explanation sequence from \( E \) to \( N \) exists in which each step uses a strict subset of \( S \). An explanation sequence is atomic if all its steps are.
Algorithm 1 Construct-greedy.

**Input:** CSP \((X, D, C)\), given \(G\), target \(T\)

1. \(Seq \leftarrow \) empty sequence
2. \(E \leftarrow G, i \leftarrow 0\)
3. **while** true **do**
4.   **forall** \(S \subseteq C\) where \(|S| = i\) and \(\text{CONNECTED}(S)\) **do**
5.     \(N \leftarrow \text{FULLPROPAGATE}(E \cup S) \setminus E\)
6.     **if** \(N \neq \emptyset\) **then**
7.         extend \(Seq\) with \((E, S, N)\)
8.         \(E \leftarrow E \cup N, i \leftarrow 0\)
9.     **if** \(T \subseteq E\) **then**
10.    **return** \(Seq\)
11.    **break**
12. \(i \leftarrow i + 1\)

Table 2 A maximal sequence that could have been constructed by Construct-greedy.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(E)</th>
<th>(S)</th>
<th>(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>(p + q \leq 1)</td>
<td>(p \in [0..1]) (q \in [0..1])</td>
</tr>
<tr>
<td>2</td>
<td>(p \in [0..1]) (q \in [0..1])</td>
<td>(q + 2r \leq 4)</td>
<td>(r \neq 3)</td>
</tr>
<tr>
<td>3</td>
<td>(p \in [0..1]) (q \in [0..1]) (r \neq 3)</td>
<td>(3s + p + r \leq 1)</td>
<td>(r \neq 2) (s \in [0..0])</td>
</tr>
<tr>
<td>4</td>
<td>(p \in [0..1]) (q \in [0..1]) (r \in [0..1]) (s \in [0..0])</td>
<td>(\text{alldiff}(p, q, r, s))</td>
<td>(\perp)</td>
</tr>
</tbody>
</table>

The third step of the first sequence of Example 3 is not atomic: it can be split into two steps over \(x \leq v\) and \(v + z \leq 3\) each, as is done in the second sequence of the same example.

We believe this property is desirable for easy-to-understand explanation sequences as it requires each explanation step to be as small as possible. Note that explanation sequences generated by the literature [7, 15] also exhibit atomicity.

3.2 Efficiency optimizations

The two main bottlenecks in this algorithm are enumerating large subsets of constraints and calculating the maximal output \(N\) for a given \(E\) and \(S\). To improve runtime, we employ two efficiency optimizations. First, we cache calls to the FullPropagate routine. When a maximal output call for some \(E\) and \(S\) is repeated, the output is just taken from the cache. This is useful as the domains of many variables are unchanged between calls and the propagation only considers the literals from \(E\) with regard to the variables occurring in \(S\).

A second optimization inspects the constraint graph of each enumerated subset \(S\) using the \(\text{CONNECTED}\) function call on line 4 in the algorithm. This function checks if the constraint graph is connected. If not, the graph can be split up into two or more components \(S_c\) and the maximal output \(N\) for \(S\) and \(E\) is exactly the union of the maximal outputs \(N_i\) for each component \(S_c\) and input \(E\). By Lemma 5, all smaller \(S_c\) did not imply any new literal, so \(S\) cannot imply any either. Naturally, this means Construct-greedy can skip subset \(S\).
3.3 Comparison to literature

The idea of \textsc{Construct-greedy} is similar to the first literature approach \cite{7}: iterate over increasingly larger subsets $S \subseteq C$ and check whether an explanation step $(E, S, N)$ exists.

In contrast to \cite{7}, \textsc{Construct-greedy} always greedily picks the first explanation step that can derive at least one unexplained literal to add to the explanation sequence, instead of generating multiple candidates and picking the optimal one under some quality function. The reasons for this difference are twofold: a simpler algorithm and less computational effort per step. Compared to logic puzzles, which are aimed to be solved by humans and contain at most hundreds of literals, for arbitrary CSPs the literals can easily grow into the millions (e.g., for large integer domains), so efficiency is more of a concern here. This is also the reason we do not employ the second literature approach \cite{15}, whose algorithm computes the optimal next explanation step according to a linear cost function directly, without explicit enumeration, by exploiting the hitting set duality between MUS’s and correction subsets.

Still, by iterating over small constraint subsets first, \textsc{Construct-greedy} adds steps that employ a small set of constraints, which is preferred to easily understand the explanation \cite{7}.

Note that \textsc{Construct-greedy} constructs maximal sequences. This contrasts with the literature approaches which minimize the input literals for each step during construction. The motivation for this difference is that we first want to minimize the number of steps and the number of constraints for each step. The more literals are derived by a step, the more literals future steps can use as input. Thereby potentially needing fewer constraints to derive new literals. Moreover, by deriving more literals in a single step, the quicker we might reach the target set.

One can also minimize the number of input literals for each step in the sequence during post-processing of the sequence, thereby simulating the sequences produced by literature approaches in terms of input literals \cite{7, 15}. In Section 7, this is precisely what is denoted as Filter simple.

4 Simple post-processing

In this section, we consider two straightforward ways to simplify explanation sequences requiring little to no computational effort.

4.1 Looseness

For an explanation sequence to be interpretable, we deem the length of the sequence to be an important metric. Each step in an explanation sequence requires effort by a user to process and understand. Naturally, this means no loose steps should be part of the sequence:

\begin{property}[Loose]
Given an explanation sequence $\langle (E_i, S_i, N_i) \rangle_{1 \leq i \leq n}$ of $T$ from $G$, the step $(E_j, S_j, N_j)$ is called loose if $\langle (E_i, S_i, N_i) \rangle_{i \neq j}$ still forms an explanation sequence of $T$ from $G$. An explanation sequence is loose if one of its steps is loose.
\end{property}

A step is loose if it can be left out of the sequence. This is the case for instance when none of a step’s output literals is used as input for later steps or is part of the target set.

The first step in the sequence 1a of Example 3 is loose, as it can just be removed without impacting the sequence. In sequence 1b of that same example, no loose steps are present as at least one output of each step is used later on.

Existing approaches \cite{7, 15} focused mostly on explaining a unique solution of a CSP, where every step derives at least one yet unexplained literal of that unique solution. This means every step is guaranteed to derive a literal in the target set. However, \cite{7} also describes
nested explanations. These clarify a single explanation step \((\mathcal{E}, \mathcal{S}, \mathcal{N})\) by creating a new explanation sequence from \(\mathcal{S}\) with \(\mathcal{E}\) and the negation of \(\mathcal{N}\) as the given set, and \(\bot\) as target set. The construction of nested explanations in [7] may lead to loose steps.

Loose steps are easy to detect in a sequence and can simply be removed from the explanation sequence until none remain.

4.2 Pertinence

After constructing a sequence and filtering out loose steps, the output of a step may still contain irrelevant literals as not all output literals have to be used later in the sequence. This requires a user to “waste” effort in understanding why useless literals are implied by the step. Clearly, this is undesired, motivating the following definition.

\begin{itemize}
  \item \textbf{Property 3 (Pertinent).} An explanation sequence is pertinent if (i) \(G\) and the \(N_i\) are pairwise disjoint, and (ii) all output literals are part of the target or the input of some following step.
\end{itemize}

The first condition states that each literal should only be derived once (and obviously, given literals should not be rederived). The second condition states that everything we derived should be used. To transform an explanation sequence into a pertinent one, we can simply loop over the sequence and remove any outputs not satisfying any of the conditions in the definition. Removing loose steps from a pertinent sequence corresponds to removing steps with an empty output set. In Example 3, the first step of sequence 1a is not pertinent, as the literal \(w \neq 3\) in its output is not part of any input or of the target. For satisfiable CSPs, the algorithms described in the literature [7] are guaranteed to produce a pertinent explanation sequence, but not so for unsatisfiable CSPs.

5 Relaxation-based filtering

Informally, a sequence is pertinent if it derives only those literals that are (indirectly) needed for the target. However, a step in a sequence may also contain input literals that are not needed to derive its output. Naturally, this is an undesired property of explanation steps as it requires the user to process more literals than needed.

\begin{itemize}
  \item \textbf{Property 4 (Sparse).} An explanation step \((\mathcal{E}, \mathcal{S}, \mathcal{N})\) is sparse if no step \((\mathcal{E}', \mathcal{S}, \mathcal{N})\) with \(\mathcal{E}' \subset \mathcal{E}\) exists. An explanation sequence is sparse if all its steps are.
\end{itemize}

The sequence of Example 4 contains non-sparse steps, e.g., the second step does not need the literals \(p \in [0..1]\) as input to derive \(z \neq 3\). The sequences in Example 7 are both sparse.

Previous approaches [7, 15] construct explanation steps by minimizing the input \(\mathcal{E}\), making the resulting steps and sequences sparse. In contrast, \textsc{Construct-Greedy} constructs a maximal sequence which typically has non-sparse steps due to non-useful output literals of previous steps. We address this by a relaxation-based filtering algorithm, which calculates a so-called relaxation for each step.

\begin{itemize}
  \item \textbf{Definition 6 (Relaxation).} A relaxation of a step \((\mathcal{E}, \mathcal{S}, \mathcal{N})\) is a step \((\mathcal{E}', \mathcal{S}, \mathcal{N})\) with \(\mathcal{E}' \subseteq \mathcal{E}\).
\end{itemize}

An explanation step is sparse if and only if it allows no strict relaxations. Given a set of implied literals \(\mathcal{N}'\), let \(\neg \mathcal{N}'\) be the constraint denoting that at least one of the literals in \(\mathcal{N}'\) is false, i.e., \(\neg \mathcal{N}' = \bigvee_{x \neq v \in \mathcal{N}'} x = v\). Then, \((\mathcal{E}, \mathcal{S}, \mathcal{N})\) is sparse if for every subset \(\mathcal{E}' \subseteq \mathcal{E}\) it holds that \(\mathcal{E}' \cup \mathcal{S} \cup \neg \mathcal{N}'\) is satisfiable, as in that case, \(\mathcal{E}' \cup \mathcal{S} \not\models \mathcal{N}'\) and \(\mathcal{E}'\) cannot form a relaxation for \((\mathcal{E}, \mathcal{S}, \mathcal{N})\).
To calculate sparse steps, we need to find a relaxation for each step. For this, we can calculate a MUS with $E$ as soft constraints and $S \cup \{¬N\}$ as hard constraints. This MUS is a subset of $E$ that yields a sparse step.

A straightforward algorithm to make all steps sparse would be to iterate over the steps and calculate such a MUS. However, the disadvantage of this is that sequences may no longer be pertinent; if the input of some step is reduced, then the output of a previous step may no longer be needed. And conversely, naively making a sequence pertinent may make it no longer sparse, as a smaller output for a step may reduce the number of input literals needed.

Instead, we propose a method that makes a sequence sparse by relaxing its steps from back to front. Additionally, this will make some steps loose, allowing the method to remove these steps. Hence, we call it Relaxation-based filtering.

Algorithm 2 shows the pseudocode. It loops from back to front over the sequence and keeps a set of required literals $Req$ that still need to be explained by some step earlier in the sequence. $Req$ is initialized as the target set $T$. For each step in the sequence, the algorithm first keeps only those output literals that are also in $Req$, since keeping more would make the sequence non-pertinent (they are not required for the rest of the sequence). Next, it relaxes the step by calculating a MUS.

In the general case, many such MUSes may exist, and choosing one MUS over the other can have a profound impact on the sequence. This is clearly visible in Example 7 where the difference between both sequences arises from choosing a different MUS to relax the last step.

In this setting, we want a MUS allowing us to keep the set of required literals $Req$ as small as possible. The reason for this is twofold: we want to keep future relaxations as small as possible and increase the number of detected loose steps. To compute a MUS satisfying this preference, we can split the calculation into two parts: first, minimize the extra input literals needed, and next minimize the subset of $Req$. Other techniques for this exist and include the OCUS-algorithm from [15].

The above process yields a sparse step deriving all required literals the step originally derived. However, the sparse input and constraints may imply more output literals than the originally new literals. To detect this, we make $N$ maximal in line 8 of the algorithm. Any output this step can derive in this way can be removed from $Req$. But any input the step uses should be added to $Req$. This is what happens in the last line.

Relaxation-based filtering guarantees sparsity by design since the input of each step is generated by a MUS call. Transforming the sequence into a pertinent one can be done trivially by removing literals from each $N$ if they are not used later in the sequence or if they are derived as well at some point earlier in the sequence, as argued in Section 4.2.
Ensuring pertinence of this sequence in this way will not break sparsity in case the original sequence was maximal (which is guaranteed by our construction). To see this note that by definition in a maximal sequence, each literal is derived as soon as it can possibly be derived. Hence, any literal in $N$ after Line 3 of the algorithm will not be derived at any earlier point in the sequence. Since these literals are required, they are part of some later input, and hence will not be removed from the output. For this reason, the step will stay sparse after making the sequence pertinent.

If the input sequence was maximal, then after Relaxation-based filtering and making the sequence pertinent, there will also be no loose steps. To see this, note if Relaxation-based filtering keeps a step, it is because some of its newly derived literals were required. If the input sequence is maximal, all literals are derived as early in the sequence as possible, meaning that this literal will not be derived earlier (and hence the pertinence making of the sequence will never remove it).

Example 7. Below are two filtered versions of the maximal sequence in Example 4.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$E$</th>
<th>$S$</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>$p + q \leq 1$</td>
<td>$p \in [0..1]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q \in [0..1]$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>$3s + p + r \leq 1$</td>
<td>$r \neq 2, r \neq 3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$r \in [0..1]$</td>
</tr>
<tr>
<td>3</td>
<td>$p \in [0..1]$</td>
<td></td>
<td>$s \in [0..1]$</td>
</tr>
<tr>
<td></td>
<td>$q \in [0..1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$r \in [0..1]$</td>
<td>alldiff($p, q, r, s$)</td>
<td>$\perp$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>$E$</th>
<th>$S$</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>$3s + p + r \leq 1$</td>
<td>$p \in [0..1]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$r \in [0..1]$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$p \in [0..1]$</td>
<td></td>
<td>$r \in [0..1]$</td>
</tr>
<tr>
<td></td>
<td>$r \in [0..1]$</td>
<td>alldiff($p, q, r, s$)</td>
<td>$\perp$</td>
</tr>
</tbody>
</table>

### 6 Deletion-based filtering

The techniques proposed so far are essential to make explanation sequences more easily understandable by users. The experiments in Section 7 will show that Relaxation-based filtering significantly reduces the number of steps for many sequences. However, more steps could be removed if we allow to change the input and output of other, later steps. In the left sequence of Example 7, step 1 can be omitted from the sequence, as shown on the right in that same example. However, depending on the exact MUSes (and hence, relaxations) calculated by Relaxation-based filtering, it may miss this, and the step remains redundant.

Property 5 (Redundant). Given an explanation sequence $\langle (E_i, S_i, N_i) \rangle_{1 \leq i \leq n}$ of $T$ from $G$, the step $(E_j, S_j, N_j)$ is redundant if the maximal explanation sequence arising from $G$ and $\langle S_i \rangle_{i \neq j}$ is an explanation sequence of $T$. An explanation sequence is redundant if one of its steps is redundant.

Intuitively, a redundant step is a step for which all useful literals it derives could also be derived by later steps in the sequence. The first and second steps in Example 4 are redundant, as there exists a maximal sequence with the constraints of steps three and four: it is exactly the right sequence of Example 7.

The literature approaches [7, 15] may very well lead to redundant sequences, as once a step is added to the sequence under construction, no check is later made whether the step could be subsumed by one added later. Notice this also holds for the application of explaining the unique solution of a logic puzzle, whereas loose steps could not occur in that setting.
Algorithm 3 Deletion-based Filtering.

**Input:** Explanation sequence \( \text{Seq} \) with maximal inputs and outputs, given set \( \mathcal{G} \), target set \( \mathcal{T} \)

for \(( \mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i ) \in \text{reverse}( \text{Seq} )\) do

let \( \text{Sub} \) be a copy of \( \text{Seq} \) from \( i + 1 \) to end

if \( \text{TryDeletion}( \mathcal{E}_i, \text{Sub} ) \) then

shrink \( \text{Seq} \) to size \( i - 1 \)

append \( \text{Sub} \) to \( \text{Seq} \)

Function \( \text{TryDeletion}( \mathcal{E}_i, \text{Sub} ) \):

\( \mathcal{N}_i \leftarrow \emptyset \)

for \(( \mathcal{E}_j, \mathcal{S}_j, \mathcal{N}_j ) \in \text{Sub} \) do

\( \mathcal{E}_j \leftarrow \mathcal{G} \cup \mathcal{E}_j - 1 \cup \mathcal{N}_j - 1 \)

\( \mathcal{N}_j \leftarrow \text{FullPropagate}( \mathcal{E}_j \cup \mathcal{S}_j ) \)

if \( \mathcal{T} \subseteq \mathcal{N}_j \) then

return True

return False

Algorithm Deletion-based filtering filters redundant steps. For this, it assumes a maximal sequence as input – if not, the sequence can be made maximal by a linear iteration over all steps from front to back. Next, Deletion-based filtering iterates over the sequence from back to front. In every iteration, it leaves out the current step, calculates the resulting maximal explanation sequence, and checks if it still explains the target. Calculating the resulting maximal sequence requires a call to a full propagation routine for \( \mathcal{E} \cup \mathcal{S} \) for each later step to derive the new outputs.

In the worst case where no step can be removed from a sequence of length \( n \), this results in \( n(n - 1)/2 \) calls to a full propagation routine. Although the pseudo-Boolean solver Exact allows for stateful computation of these maximal outputs, this remains a computationally expensive task and several optimizations are needed to make the algorithm work in practice, as we now explain.

### 6.1 Necessary condition on the existence of a sequence

For a maximal sequence with given set \( \mathcal{G} \) and target set \( \mathcal{T} \), Deletion-based filtering has to check for a step \(( \mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i ) \) whether the maximal sequence determined by \( \langle \mathcal{S}_j \rangle_{i < j \leq n} \) and \( \mathcal{E}_i \) as given set, still derives \( \mathcal{T} \). A necessary condition is that \( \mathcal{E}_i \cup \bigcup_{i < j \leq n} \mathcal{S}_j \models \mathcal{T} \) – if not, \( \mathcal{T} \) cannot be explained with just the constraints from the remaining steps in the sequence. This can be checked via a simple solve call with the constraints \( \mathcal{E}_i \cup \bigcup_{i < j \leq n} \mathcal{S}_j \cup \neg \mathcal{T} \). In case this returns a solution, step \( i \) cannot be dropped from the sequence.

### 6.2 Partial propagation

Full propagation – calculating the maximal output of a step – is an expensive operation. However, weaker and more efficient propagation algorithms are researched \([5, 38]\) and are implemented in several state-of-the-art constraint solvers. When Deletion-based filtering has to check whether a target set \( \mathcal{T} \) can be derived by the maximal sequence arising from \( \langle \mathcal{S}_j \rangle_{i \leq j \leq n} \) and given set \( \mathcal{E}_i \), it can first compute a “partial maximal” sequence using a computationally cheaper propagation algorithm. If the target set is explained according to this weaker sequence, it also will be explained by the maximal sequence, so the step can be dropped and Deletion-based filtering can continue to the next iteration.
6.3 Caching

As explained in Section 3.2, we can cache the result of calls to a full propagation algorithm. In addition, we can cache the result of computing a maximal sequence (containing multiple maximal outputs), and we can cache the result of checking the necessary condition for a maximal sequence as described in Section 6.1. These caching strategies become particularly useful given the monotonicity of entailment, as formalized in the following lemma.

▶ Lemma 8. For any set of constraints \( S \) and any two sets of literals \( E \subseteq E' \), if \( E \cup S \models \mathcal{N} \) then \( E' \cup S \models \mathcal{N} \).

This entails that if a maximal sequence determined by \( (S_j)_{1 \leq j \leq n} \) with given set \( E_i \) entails a target set \( T \), so will the maximal sequence using the same constraints with a superset of \( E_i \). Conversely, if that maximal sequence does not entail \( T \), neither will the maximal sequence with any subset of \( E_i \).

We propose to generate explanation sequences using Construct-greedy and filter these sequences using Deletion-based filtering so no redundant steps remain. Lastly, these non-redundant sequences need to undergo Relaxation-based filtering so all steps are made sparse and can be made pertinent. In Appendix A, we construct a formal argument to prove why this pipeline ensures atomicity, sparsity, pertinence and non-redundancy for general explanation sequences.

Notice that filtering explanation sequences using Deletion-based filtering can potentially filter more steps compared to Relaxation-based filtering but may increase the complexity of individual steps as more literals are used/explained at a time. Nevertheless, the constraint sets of each step are not altered by any of the algorithms we proposed. These are entirely determined by the construction algorithm.

7 Experimental results

We evaluate and review the algorithms and ideas presented in previous sections using three benchmark sets consisting of unsatisfiable CSP instances.

Sudoku. 50 9x9 sudoku instances in which an empty cell is given a wrong value, such that this variable assignment is non-trivial, i.e., it does not directly falsify any constraint. The original puzzles were generated by the QQwing [35] tool using the “Intermediate” difficulty setting. The sudoku constraints are modelled with AllDifferent global constraints.

Jobshop. 50 jobshop instances generated by the approach in [40] with 5 machines and 5 tasks per job. The time horizon is equal to 50 units. Unsatisfiability arises from restricting the makespan to a better-than-optimal value. All instances are modelled using Cumulative constraints [1] as is usual in scheduling problems.

Debug. Using the diverse set of CSP models from Håkan Kjellerstrand,\(^2\) we construct a set of 202 unsatisfiable CSPs by introducing artificial errors in the models to mimic a modelling error by a user – an unsatisfiability bug. We follow a similar approach to [29]. The errors are introduced such that each constraint separately remains satisfiable.

\(^2\) http://www.hakank.org/cpmpy/
These three benchmark families correspond to three use cases where step-wise explanations can be of help. **Sudoku** simulates the counterfactual explanation provided during an interactive collaboration (e.g., [23]) by a human agent and an automated system. Here the user wants to know why the system derived that a certain value was no longer possible for a certain value. **Jobshop** corresponds to explaining to a user why an inferred objective value is optimal for some optimization problem. **Debug** simulates the situation where a user is modelling a CSP but discovers the model has no solutions and the user needs to “debug” it.

For every instance, we compute three explanation sequences using different random seeds. As metrics, we consider the number of steps in sequences, as well as the complexity of individual steps. This first metric is different from what literature approaches consider [7, 15] as in their use-case of explaining a unique solution, they focus only on the complexity of individual steps.

**RQ1.** How many redundant steps do explanation sequences of unsatisfiable CSPs contain?

**RQ2.** How well can the proposed filtering techniques simplify explanation sequences?

**RQ3.** How much time do the proposed algorithms take to run?

**RQ4.** How does simplifying the set of input constraints affect the explanation sequence?

### 7.1 Experimental setup

All experiments were run on a single core of an Intel(R) Xeon(R) Silver 4214 CPU with 128GB of RAM. **FULLPROPAGATE** is handled by the stateful **pruneDomains** functionality of the Exact solver [12, 14] v1.0.0. Partial propagation is calculated with OR-Tools [36] v9.6 presolve routine. All algorithms are implemented using a custom branch of CPMPy [19] v0.9.12, embedded in Python v3.10.9. To calculate MUSes we employ the **mus** tool provided by CPMPy. Cardinal-minimal MUSes are computed using the hitting-set approach of [25]. All code and benchmarks are available on GitHub: https://github.com/ML-KULeuven/SimplifySeq

### 7.2 Approaches under investigation

Throughout this section, we investigate all algorithms and combinations of them presented in this paper. All sequences are generated using **CONSTRUCT-GREEDY** and are indicated using **Original** in each of the plots. By making each step sparse, we can mimic the explanation sequences generated by approaches described in the literature [7, 15]. This can simply be done by finding a minimal unsatisfiable subset with hard constraints \( S \cup \{ \neg N \} \) and soft constraints \( E \). Afterwards filtering loose steps and making the sequence pertinent using the approach described in Section 4 is denoted as **Filter simple** and is considered current state-of-the-art. We consider two versions of **RELAXATION-BASED FILTERING** differing in the type of MUS extracted: a MUS and a smallest MUS (SMUS), shown as **MUS Relax** resp. **SMUS Relax**. Lastly, we investigate sequences exposed to the entire pipeline using **DELETION-BASED FILTERING** and both versions of **RELAXATION-BASED FILTERING**. These are shown as **Del + MUS Relax** and **Del + SMUS Relax**.

### 7.3 Redundant steps

The first research question is straightforward: how many redundant steps are there in naively generated sequences, and how many can we filter using our proposed techniques? From Figure 1 it is clear greedy explanation sequences contain a lot of redundant steps. For all

---

3 [https://gitlab.com/JoD/exact](https://gitlab.com/JoD/exact) git commit 34c4fa46
benchmarks, the deletion-based method leads the way in terms of the number of steps filtered from the sequence. The price to be paid is a significant increase in computational effort for some benchmarks and more complex individual steps as outlined in the next sections. Simple filtering of loose steps as described in Section 4 is able to reduce the number of steps, but falls short compared to fully-fledged relaxation-based methods. This is most prominently visible in the Jobshop benchmark set. Interestingly, both variants of RELAXATION-BASED FILTERING (MUS Relax and SMUS Relax) show similar performance in terms of steps left in the sequence after filtering.

7.4 Complexity of individual steps

Next, we investigate how different filtering methods compare in terms of input and output literals for each step. The simple filtering method is able to keep the number of input literals relatively low. As expected, it again falls short compared to relaxation-based methods as Filter simple does not modify the set of input literals based on the output literals that are actually useful. While the deletion-based methods are able to remove more steps, it increases the number of input literals for the remaining steps compared to relaxation-based methods. This is similar to the number of output literals, shown in Figure 2b. Deletion-based methods require steps to derive more output while relaxation-based methods work better for this metric. We argue this larger number of output literals is less of a concern compared to the increase of input literals as steps can be split up so they derive fewer literals at a time.
The left-hand side of Table 4 shows that the number of constraints is low for each individual step, ranging from 1 to a few constraints propagated at a time. Notice how during filtering, some steps occupying little constraints are deleted, while the largest step for each sequence is always present after filtering still.

### 7.5 Explaining a MUS

Table 4 Effect of explaining a MUS of the unsatisfiable problem instead of all constraints in the problem. Metrics are shown before filtering $\rightarrow$ after filtering using DELETION-BASED FILTERING.

<table>
<thead>
<tr>
<th></th>
<th>Explaining all constraints</th>
<th>Explaining a MUS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#steps</td>
<td>Max</td>
</tr>
<tr>
<td>Sudoku</td>
<td>79.62</td>
<td>26.33</td>
</tr>
<tr>
<td>Jobshop</td>
<td>118.03</td>
<td>35.69</td>
</tr>
<tr>
<td>Debug</td>
<td>18.70</td>
<td>5.91</td>
</tr>
</tbody>
</table>

In the literature, the most common way to explain why a model is unsatisfiable is to first extract a MUS. What happens to step-wise explanations when we first extract a MUS and build the explanation sequence from that MUS?

The result of this experiment is summarized in Table 4. Here we observe that when extracting a MUS, the explanation sequence is much shorter before any filtering is applied. Interestingly, explaining the full CSP may actually lead to shorter sequences after filtering, as observed for Sudoku and Jobshop. Equally remarkable, limiting the explanation sequence to a MUS increases the number of constraints used in individual steps. This is most clear from Sudoku where the maximal number of constraints in a step increases from 1 constraint to 2.04 constraints on average. Intuitively, limiting the constraints with which to build an explanation sequence to a smaller unsatisfiable subset reduces the ease with which explanations can be built. E.g., the Sudoku problem contains many redundant constraints that are implied by some subset of other constraints. Nevertheless, these redundant constraints can be very useful to elegantly explain something to a user.

### 7.6 Runtime and optimizations

Figure 3 displays the runtime for each of the filtering algorithms for all benchmark sets. For Jobshop, the deletion-based method is computationally more expensive compared to any of the relaxation-based variants. This is not the case for Sudoku and Debug. As Cumulative constraints are harder to fully propagate compared to AllDifferent constraints, the quadratic number of full-propagation calls is much more prominent for the Jobshop benchmark. For all benchmarks, calculating a smallest-minimal MUS requires more computational effort while not impacting the sequence enough to justify this extra cost.

The optimizations implemented for DELETION-BASED FILTERING as described at the end of Section 6 are most effective with the Debug benchmark, where only 7% of the theoretic number of full-propagation calls are executed. For Sudoku and Jobshop, this is 19% resp. 34%. Overall, the proposed optimizations are necessary to reach the observed performance.

### 8 Conclusion and Future work

We investigated the problem of step-wise explaining unsatisfiable problems, with applications in debugging unsatisfiability, explaining optimality, and explanations in interactive configuration. To ensure a simple step-wise explanation, both the number of steps and the
number of constraints and literals in each explanation step must be kept low. We proposed the formal properties of atomicity, pertinence, sparsity, and (ir)redundancy, to which an explanation sequence should strive to adhere. For this, we proposed a workflow combining greedy construction, deletion-based filtering and relaxation-based filtering that guarantees all properties are satisfied.

We presented extensive experimental results comparing current approaches to our proposed techniques. This showed filtering of explanation sequences is especially essential in the challenging setting of explaining unsatisfiability and is key in creating easy-to-understand explanation sequences. Our methods are able to considerably shorten explanation sequences with deletion-based filtering leading to sequences without any redundant steps. Compared to relaxation-based techniques which may produce longer sequences, deletion-based methods may lead to more complex steps after filtering.

For future work, observe that none of the filtering algorithms considered in this paper alter the set of constraints for explanation steps, nor the order of explanation steps. These are entirely determined by the construction algorithm. Changing either of these aspects might further simplify an explanation sequence. We leave these areas for future work, where both new sequence construction methods and more elaborate post-processing are viable options to explore.

Finally, the small number of constraints in most explanation steps shows that computing explanation sequences seem a worthwhile approach to help a user understand unsatisfiable constraints. Still, the expressivity of explanation steps deserves further study with our results opening the door for further investigation into what domain experts perceive as “simple” explanation sequences. User studies or learning techniques can investigate how it helps them to understand the interplay of complex constraints.

References


Simplifying Step-Wise Explanation Sequences


To transform a maximal sequence generated by Construct-greedy to one that is sparse, pertinent, atomic and contains no redundant (and hence no loose) steps, we propose to generate sequences with Construct-greedy, then filter those with Deletion-based filtering, finally followed by Relaxation-based filtering.

The application of Relaxation-based filtering in the last step guarantees sparsity. Modifying the output of steps to guarantee pertinence will not break sparsity as the output of Deletion-based filtering is maximal (see the argument of Section 5).

After the application of Deletion-based filtering, all redundant steps have been removed. In other words, for any $i$, there exists no strict subsequence $\langle S_i \rangle_{1 \leq i \leq n, i \neq j}$ for which its maximal sequence with the same given set $G$ explains the same target set $T$. Hence, Relaxation-based filtering will not actually filter any more steps, as also shown in the experiments in Section 7. Moreover, Relaxation-based filtering does not change any of the constraint subsets $S_i$, so the maximal sequence after Relaxation-based filtering is equal to the one after Deletion-based filtering. Therefore, Relaxation-based filtering preserves the guarantee of Deletion-based filtering that no redundant steps exist.

To argue atomicity, we will consider three sequences
\[ \langle (E_i, S_i, N_i) \rangle_{1 \leq i \leq n} \text{ denotes the sequence obtained after construction,} \]
\[ \langle (E'_i, S_i, N'_i) \rangle_{i \text{ not deleted}} \text{ the sequence obtained after DELETION-BASED FILTERING, and} \]
\[ \langle (E''_i, S_i, N''_i) \rangle_{i \text{ not deleted}} \text{ the sequence obtained after RELAXATION-BASED FILTERING.} \]
Moreover, we will write \( I_i \) (resp. \( I'_i, I''_i \)) for \( G \cup \bigcup_{j<i} N_j \) (resp. \( G \cup \bigcup_{j<i} N'_j, G \cup \bigcup_{j<i} N''_j \)).

Intuitively, \( I_i \) represents the set of all facts available prior to step \( i \). By definition of an explanation sequence, we have that \( E_i \subseteq I_i \) (and similar for the other two sequences).

To argue that \( (E''_i, S_i, N''_i) \) is still atomic, we first prove four claims.

1. \( I_i \supseteq I'_i \supseteq I''_i \);
2. \( N'_i \subseteq I_{i+1} \)
3. For each literal \( l \) in \( N_i \setminus I_i \) (i.e., every newly derived literal in step \( i \)), and every subset \( S \subseteq S_i, I_i \cup S \neq l; \)
4. If step \( i \) is not deleted, then \( N'_i \cap (N_i \setminus I_i) \neq \emptyset \) and \( N''_i \cap (N_i \setminus I_i) \neq \emptyset \), i.e., at least one literal that was newly derived at step \( i \) is still derived at that step.

Claim 1 follows from the fact that, by construction, the first two sequences are maximal (in their inputs as well as outputs), and hence derive everything that is derivable at each point. Since the constraint sets in the three sequences are the same, nothing extra can then be derived in the trimmed sequences. Claim 2 again follows directly from the fact that the initial sequence is maximal: what is derived at step \( i \) can be at most \( I_i \cup N_i \), which equals \( I_{i+1} \). Claim 3 holds by construction: our greedy construction algorithm only generates steps for which there is no smaller set of constraints that can derive something. Claim 4 holds since, using Claim 2, if that intersection would be empty, \( N'_i \) would consist only of literals of \( I_i \), in which case DELETION-BASED FILTERING would clearly delete this step.

Now, take any \( i \). We continue to show that \( (E''_i, S_i, N''_i) \) is in fact atomic. Using Claim 4, \( N''_i \cap (N_i \setminus I_i) \neq \emptyset \). From Claim 2 it follows that \( N''_i \) is the union of two sets: \( O := N''_i \cap I_i \) and \( D := N''_i \cap (N_i \setminus I_i) \), where \( O \) is the set of literals that were originally “old” at step \( i \) (derived before \( i \)) and \( D \) are those that were actually derived at step \( i \). By Claim 4, \( D \) is non-empty. Now assume towards contradiction that our step \( (E''_i, S_i, N''_i) \) is not atomic. In that case, there is an explanation sequence that derives \( N''_i \) from \( E''_i \) in which each step uses a strict subset of \( S_i \). From that sequence, consider the first step that derives an element from \( D \). By definition, the input of that step can consist at most of \( E''_i \cup O \). Since \( E''_i \cup O \subseteq E_i \), we find an explanation step that violates Claim 3, and we can conclude our proof by contradiction.