Step-Wise Explanations for Sudoku Puzzles Using ASP(Q)

Maximilian Moeller[1][0009−0008−5426−0445]✉, Alexander Ek[1,2][0000−0002−8744−4805], and Bart Bogaerts[1,3][0000−0003−3460−4251]

[1] KU Leuven, Dept. Computer Science, Leuven, Belgium
[2] ARC Training Centre OPTIMA, Melbourne, Australia
[3] Vrije Universiteit Brussel, Brussels, Belgium
{maximilian.moeller,alexander.ek,bart.bogaerts}@kuleuven.be

Abstract One avenue to increase trustworthiness and transparency of *constraint satisfaction problem* (CSP) solvers is that of *step-wise explanations*. A step-wise explanation bridges the gap between a CSP and its solution through a series of reasoning steps that are "simple" enough for a human to understand and verify. In prior work, the computational task of finding an explanation has been reduced to finding certain kinds of cost-optimal unsatisfiable subsets, called OCUSs, of a set of Boolean formulae. In this paper, we show how the recently proposed framework of *answer set programming with quantifiers* (ASP(Q)) can be used to compute such OCUSs and hence step-wise explanations. We demonstrate this for Sudoku puzzles and show how our declarative approach facilitates experimentation with various OCUS problem formulations. Finally, we empirically compare the ASP(Q)-based approach with the state of the art, and observe that ASP(Q) solves more puzzles to completion, and does so faster for many explanation steps, while the state-of-the-art approach is faster on high-cost explanation steps.

Keywords: Answer set programming · Constraint satisfaction · Optimal constrained unsatisfiable subsets

1 Introduction

As constraint solvers are becoming increasingly powerful and widely used in areas such as logistics, healthcare, and finance, the need for *explainability* is growing, both for legal reasons and to increase trustworthiness for the users of these solvers. State-of-the-art constraint solvers are highly optimized and employ specialized reasoning techniques from solving technologies such as *(integer) linear programming* ((I)LP) [24], *Boolean satisfiability* (SAT) [3], or *constraint programming* (CP) [23], involving search, backtracking, and constraint learning. While these techniques are sound and highly efficient when properly implemented, they lack explanatory power suitable for humans, regarding *why* a particular decision was made, *why* a particular decision was ruled out, or *how* the solver progressed in the solving process. Further, the problems addressed often involve hundreds to

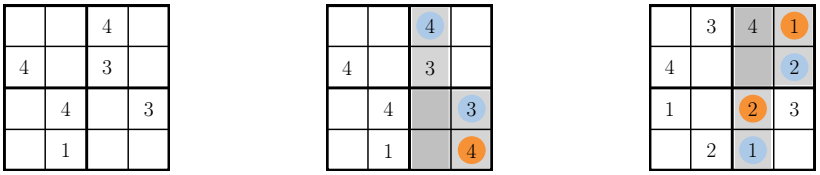

Figure 1: Left: a typical 4×4 Sudoku puzzle with six given digits. Middle: an optimal first explanation step for this puzzle. Using the constraints associated with column three and box four (highlighted in gray), and two given digits (highlighted in blue), one can derive that the bottom right cell must contain the digit four (highlighted in orange). Right: an explanation step from another Sudoku instance in which two facts can be derived simultaneously.

millions of variables. As a result, it is difficult for humans to verify the solutions provided by these solvers, and to understand the reasoning behind them.

These decision problems can each be formalized as a *constraint satisfaction problem* (CSP), with finite-domain variables and constraints over them. Explanations in the context of CSPs take various forms and shapes (see, e.g., the recent survey of Gupta et al. [17]). We focus here on explanations that are sequences of logically valid reasoning steps, that, starting from the CSP, arrive at the conclusion to be explained and are "easy" to follow. In particular, this kind of explanations aims *not* to interpret what the solver was doing, but rather to illustrate how a human could have found the solution themselves. Figure 1 exemplifies what explanation steps could look like for the logic puzzle Sudoku.

Prior Work. Much attention in SAT and CP research has been devoted to explaining why a problem instance is *unsatisfiable*, e.g., through the use of *minimal unsatisfiable subsets* (MUSs) [18,19,20]. In computer-assisted scheduling tasks, for example, human planners might be interested in *why* a partial assignment of workers to tasks makes the remaining problem unsatisfiable or necessarily results in a suboptimal schedule [4]. More recently, Bogaerts et al. [6] proposed a framework for explaining satisfiable CSPs, called *step-wise explanations* and applied it to logic grid puzzles. Within this context, an *explanation step* consists of some puzzle constraints and some previously derived facts that together derive a new fact in the puzzle. Several explanation steps can then be chained into an *explanation sequence* that starts from the initially given puzzle clues and constraints and derives the (unique) solution to the puzzle. Pen-and-paper puzzles such as logic grid puzzles or Sudoku lend themselves to this particular explanation technique, because they are constructed to be simple enough to solve without search. Each fact in the solution should be derivable through the use of previously derived facts and puzzle constraints alone, i.e., through (a powerful form of) *constraint propagation* that possibly combines multiple constraints. While these explanations were originally used to explain such logic puzzles, they have later also been applied to optimization problems [5] and can be of value for explaining a MUS in more detail: instead of just showing the user a set of

constraints that together are unsatisfiable, an explanation sequence illustrates how these constraints interact with each other to derive the contradiction.

In following research, Gamba et al. [14] gained insight into the computational problem that lies at the heart of finding a "simple" explanation step and called it the *optimal constrained unsatisfiable subset* (OCUS) problem. They showed that the OCUS problem lies at the second level of the polynomial hierarchy and devised an *implicit hitting set* (IHS) based algorithm for it.

Logic programming [25] is a paradigm in which knowledge is expressed declaratively rather than imperatively through the use of *rules*. A set of such rules is commonly called a (logic) program. There are various logic programming frameworks which differ in syntax, semantics or underlying philosophies. One successful approach that has been primarily developed for NP-hard (search) problems is called *answer set programming* (ASP) [21]. ASP is based on the *stable model semantics* that, very roughly speaking, requires each fact in a model to the program to be "justified" by derivation through some rule in the program. There have been proposals to extend ASP's modelling capabilities to the second level of the polynomial hierarchy, most famously through so-called *disjunctive* rules and a technique called *saturation* [11]. However, these are widely regarded as unintuitive in the scientific community. More recently, proposals have been made to extend classical ASP to capture arbitrary levels in the polynomial hierarchy in a more natural way by combining multiple programs. This idea was first introduced as *stable-unstable semantics* [7] and later as *ASP with quantifiers* (ASP(Q)) [1]. The idea is to allow nesting answer set programs with alternating uses of quantification. E.g., in ASP(Q) notation, programs P_1, P_2 and P_3 could be nested into program $C = \exists P_1 \forall P_2 : P_3$. Intuitively, the answer sets to C are the stable models of P_1 such that for all stable models of P_2 there is at least one stable model to P_3, where programs "deeper" inside the nesting can make use of predicates defined in programs "higher up" than them.

Motivation and Contributions. One strength of the framework proposed by Bogaerts et al. [6] is how agnostic it is of the concrete problem being explained. We are motivated, however, by the presumption that further augmenting the problem formulation with actual domain-specific knowledge might help to (1) guide the solver(s) towards more favorable explanation steps and sequences and (2) improve run times through a reduction of the search space. Since it might not be immediately clear which additions to the problem formulation might lead to the desired effects some prototyping might be required. However, encoding this knowledge in the IHS approach requires considerable insight into the inner workings of the solvers and/or the overall algorithm, and might therefore be also prone to errors. In this article we propose a different paradigm for computing OCUSs for explanation sequences that is based on recent developments in logic programming, and argue that our technique allows us to declaratively express and modify the problem formulation without encoding this knowledge deep inside an imperative procedure.

The **main contributions** of this paper are threefold. First, we show how the task of computing an explanation step (and by iterative application, an

explanation sequence) can be mapped to ASP(Q). Second, we highlight the need for more effective optimization techniques for ASP(Q). In particular, we show that the standard solution-improving optimization implemented for ASP(Q) is insufficient for this problem, and easily outperformed by a naïve bound-driven method. Third, using recent developments in ASP(Q) solvers [10], we evaluate our approach against the previously proposed IHS algorithm on the well-known Sudoku puzzle. We observe that ASP(Q) performs better in many cases, while the IHS approach is still faster on some high-cost explanation steps.

Although our ASP(Q) encoding contains problem-specific components, it can be easily modified to accommodate other problems. Further, many high-level CSP modeling frameworks, such as MiniZinc [22], CPMpy [16], or Essence [13] have the capability to translate their models to other formalisms. Currently, ASP is not widely supported as a target formalism, but we hope that our work can contribute to the motivation for its adoption. We also hope that our use-case promotes further research and development of ASP(Q) and associated solvers, and maybe even contribute to benchmark datasets for ASP(Q).

The rest of the paper is structured as follows. First, we discuss relevant preliminaries in Section 2. We show how the OCUS problem can be mapped to ASP(Q) in Section 3 and highlight advantages of the declarative approach. We experimentally evaluate our approach against the previously proposed IHS algorithm in Section 4 and conclude in Section 5.

2 Preliminaries

As a running example, we use the well-known puzzle *Sudoku*. In a conventional Sudoku puzzle, we are given a 9×9 grid of cells to be filled with digits from 1 to 9, with some cells already filled with digits. The task is to fill the empty cells such that each digit from 1 to 9 appears exactly once in every row, column, and given 3×3 box. It is customary to number the rows, columns, and boxes from 1 to 9 in left-to-right, top-to-bottom order. Enough digits are given such that there is a unique solution to the puzzle. The (perceived) difficulty of a puzzle varies with the number of given digits and their placement. Sudoku can be generalized to an arbitrary $n \times n$ grid with "digits" from 1 to n and $\sqrt{n} \times \sqrt{n}$ boxes, such that $\sqrt{n}$ is an integer. Here, we will consider the 4×4 and 9×9 variants.

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) is a triple (X, D, C) where X is a set of *variables*, D is a set of initial *domains* mapping each variable $x \in X$ to its finite set of possible values $x \mapsto D(x)$, and C is a set of *constraints* over the variables X. An *assignment* $\mathcal{A}$ is a (partial) mapping from the variables $x \in X$ to values in their domains, i.e., $\mathcal{A}(x) \in D(x)$. If $\mathcal{A}(x)$ is undefined for any $x \in X$, we say it is a *partial* assignment, otherwise it is a *total* assignment. We denote the support of an assignment $\mathcal{A}$, i.e., the set of variables for which it is defined, by $\mathrm{dom}(\mathcal{A})$. We call assignment $\mathcal{A}'$ *more precise* than assignment $\mathcal{A}$ if $\mathrm{dom}(\mathcal{A}) \subseteq \mathrm{dom}(\mathcal{A}')$ and

both interpretations agree on $\mathrm{dom}(\mathcal{A})$. If, additionally, $\mathrm{dom}(\mathcal{A}') \neq \mathrm{dom}(\mathcal{A})$, we call $\mathcal{A}'$ *strictly more precise* than $\mathcal{A}$. If for any $x \in X$, we have $|D(x)| = 1$, we say that every $\mathcal{A}$ contains the variable assignment $x \mapsto d$ for the unique $d \in D(x)$. Each constraint $c \in C$ is a set of assignments over a subset of variables called the *scope* of c, denoted $\mathrm{vars}(c) \subseteq X$. An assignment $\mathcal{A}$ *satisfies* constraint c if $\{x \mapsto \mathcal{A}(x) \mid x \in \mathrm{vars}(c)\} \in c$. An assignment $\mathcal{A}$ *violates* constraint c if there is no total assignment $\mathcal{A} \cup \{x \mapsto d \mid d \in D(x),\, x \notin \mathrm{dom}(\mathcal{A})\}$ that satisfies c. A total assignment that satisfies all constraints C is a *solution*.

For example, in Sudoku, the variables are the cells in the grid, the domains are the possible values for each cell (initially $\{1, \ldots, 9\}$ for empty cells and a singleton set for given digits) and the constraints are the row, column, and box constraints denoting that their numbers are all distinct.

A *fact* is any constraint forcing some assignment to a variable, i.e., a constraint of the form $\{x \mapsto d\}$ for some variable x and value $d \in D(x)$. Further, we view an assignment $\mathcal{A}$ as a set of facts $\{x \mapsto \mathcal{A}(x)\}$ for $x \in \mathrm{dom}(\mathcal{A})$. We write $\textsc{DeriveFacts}(C, \mathcal{A})$ for the set of all facts that can be derived from the constraints C (possibly already including facts) and the assignment $\mathcal{A}$, i.e., the *maximal consequence* of C and $\mathcal{A}$. Note that this includes facts previously in C or $\mathcal{A}$. For the rest of the paper we use *constraint* to mean any constraint that is not a fact.

2.2 Explanations

Using the CSP viewpoint, we describe the task of computing a step-wise explanation [6] for a CSP (X, D, C) and a given solution $\mathcal{A}^*$ to this CSP. An *explanation step* from assignment $\mathcal{A}$ to a strictly more precise assignment $\mathcal{A}'$ is a triple (F, S, N) of non-empty sets, where $F \subseteq \mathcal{A}$ is a set of known facts, $S \subseteq C$ is a constraint subset, and $N = \mathcal{A}' \setminus \mathcal{A}$ is a set of newly derived facts such that the CSP $(X, D, F \cup S \cup \{\overline{n}\})$ is unsatisfiable for each $n \in N$ (where $\overline{n}$ denotes the negation of n). This guarantees that the facts F and constraints S entail each new fact in N. The step-wise explanation algorithm, shown in Algorithm 1, computes a sequence of such steps, explaining all facts in $\mathcal{A}^*$. We thereby obtain a sequence of assignments $\mathcal{A}_0 \subsetneq \mathcal{A}_1 \subsetneq \ldots \subsetneq \mathcal{A}_k$, where $\mathcal{A}_0$ is the initial state, i.e., $\mathcal{A}_0 = \{x \mapsto d \mid x \in X,\, D(x) = \{d\}\}$, and $\mathcal{A}_k = \mathcal{A}^*$. The algorithm also receives

Algorithm 1 Explanation Generation

```
function EXPLAIN(X, D, C, A*, g)
    i := 0
    A_0 := {x ↦ d | x ∈ X, D(x) = {d}}
    while A_i ≠ A* do
        (F_i, S_i, N') := EXPLAIN-ONE-FACT(A_i, A*, g)
        N_i := DERIVEFACTS(S_i, F_i) \ (A_i ∪ F_i)   ▷ all new facts derivable in this step
        i += 1
    return [(F_j, S_j, N_j)]_{j=0}^{i}
```

a function g assigning to each explanation step a cost representing the difficulty of that step. Notice that $(\mathcal{A}_0, C, \mathcal{A}^* \setminus \mathcal{A}_0)$ is a valid explanation step from $\mathcal{A}_0$ to $\mathcal{A}^*$, but perhaps no more insightful than the solution $\mathcal{A}^*$ was in the first place. Hence, on line 5, we make use of the function Explain-One-Fact$(\mathcal{A}, \mathcal{A}', g)$, which is described in more detail in Section 3. It returns an explanation step (F, S, N) from $\mathcal{A}$ to some strictly more precise assignment such that $N \subseteq \mathcal{A}'$, $|N| = 1$ and (F, S, N) is g-minimal among all such steps. Using the function DeriveFacts(S, F) on line 6, we compute all new facts that can be derived from the constraints and facts used in the explanation step, and add those to N_i. This allows us to derive multiple facts in one step without increasing the cost of the step. See Figure 1 (right) for an example step deriving multiple facts.

2.3 Optimal Constrained Unsatisfiable Subsets

Gamba et al. [14] presented a generalization of the problem of finding a MUS, called the *optimal constrained unsatisfiable subset* (OCUS) problem, and showed that it captures the problem of finding an optimal explanation step. In general, the OCUS problem, asks for a specific unsatisfiable *set* of constraints. In practice, an approach with *indicator variables* is commonly used as it lends itself well to implementation, using, e.g., an incremental solver. We present the indicator-perspective of the OCUS problem directly, since it also fits our needs better.

Definition 1. *Let $\mathcal{S}$ be a set of literals, called* soft literals*, and R be a set of* reified constraints*, i.e., of the form "$s \rightarrow c$", where $s \in \mathcal{S}$ and c is a constraint. Let $\mathcal{P}$ be a (CNF) formula over the variables in $\mathcal{S}$ and $g\colon 2^{\mathcal{S}} \rightarrow \mathbb{N}$ be a function associating each subset of $\mathcal{S}$ with a cost. The OCUS problem consists of finding a subset $\mathcal{O} \subseteq \mathcal{S}$ such that (1) $R \cup \mathcal{O}$ is unsatisfiable, (2) $\mathcal{O}$ satisfies $\mathcal{P}$, and (3) $\mathcal{O}$ is g-minimal among all subsets of $\mathcal{S}$ satisfying conditions (1) and (2).*

We will use the function Explain-One-Fact$(\mathcal{A}, \mathcal{A}', g)$ to refer to the special case of the OCUS problem as follows. The soft literals $\mathcal{S}$ contain the facts in $\mathcal{A}$, fresh indicator variables, and the negation of the facts in $\mathcal{A} \setminus \mathcal{A}'$. The set R contains the reified constraints (e.g., reified versions of the row, column, and box constraints in Sudoku), and $\mathcal{P}$ encodes the restriction that exactly one fact should be explained in this step. The function g is a weighted sum over the literals in $\mathcal{S}$, representing the cost of using facts and constraints in the explanation.

2.4 Answer Set Programming (with Quantifiers)

We recall some basic notions of *answer set programming* (ASP) and refer the reader to further literature [8,15] for a more detailed introduction. In ASP, *constants* can be integers or strings starting with lowercase letters, and *variables* are written as strings starting with uppercase letters. For *predicate symbols* we use camelCase. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol, and each t_i is a constant or variable. A *literal* is an atom a or its

```
1 {useConstraint(T,N)} :- explainFact(X,Y,Z), in(X,Y,T,N).
2 :- {useFact(X,Y,Z) : fact(X,Y,Z)} < 2.
3 :~ useConstraint(T,N). [10@1,T,N]
```

Listing 1.1: Example of ASP syntax in CLINGO

negation not a. A (normal) *answer set program* P is a finite set of (normal) *rules* of the form

$$a \leftarrow b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots, \text{not } b_m.$$

The *head* a of the rule is a positive literal, and the expression $b_1, \ldots, \text{not } b_m$ is called the *body* of the rule. A rule with an empty body is called an *ASP-fact* (not to be confused with the 'fact' defined above for CSPs; to disambiguate, we will always use the term *ASP-fact*). A literal in which no variables appear is called *ground*, and a rule consisting of only ground literals is also called ground. The *Herbrand Base* B_P of program P is the set of all ground atoms that can be formed from predicates and constants appearing in P. A *stable model* or *answer set* of a program P is a subset $A \subseteq B_P$ such that A is the least model of the reduct of P with respect to A (which is the program obtained from P by removing all rules with a false body and all negative literals). We also use the common syntactic sugar for *integrity constraints*, *choice rules*, and *cardinality rules* in our listings; as well as the language extension *weak constraints* allowing optimization [9]. We give a short example in Listing 1.1 to explain the syntax informally. Line 1 is a choice rule, which intuitively says that when the fact that the cell at row `X`, column `Y` has a digit `Z` in it is to be explained, and this cell lies in the `N`th constraint of type `T` (row, column, box), then this constraint can either be used or not. Line 2 is an integrity constraint, stating that it cannot be the case that fewer than two facts are used in an explanation (for 4×4 Sudokus, see Section 3.3). Finally, line 3 is a weak constraint, which states that each instantiation of `T` and `N` such that the predicate `useConstraint(T,N)` holds, incurs a cost of 10 at the first lexicographic optimization level. Note that the rule operator is written as "`:-`" and that weak constraints are written using "`:~`".

ASP with quantifiers (ASP(Q)) [1] offers the extension to quantify over answer sets of ASP programs. An ASP(Q) program Π is of the form

$$\Box_1 P_1 \Box_2 P_2 \ldots \Box_n P_n \colon C \colon C_w,$$

where, for $1 \leq i \leq n$, P_i is a normal ASP program, $\Box_i \in \{\exists, \forall\}$, C is a stratified normal ASP program, i.e., the head predicates of C do not appear in the bodies of C in a way that would form a loop involving negation; and C_w is a program containing only weak constraints. The predicates defined in a program P_i, can appear in the rule bodies of programs P_j for $j \geq i$, C and C_w, but not in the heads, i.e., the programs define disjoint sets of predicates. For a stable model A_i to program P_i and a program P_j with $i < j$, we write $P_j \uplus_{P_i} A_i$ to mean the program $P_j \cup \{a. \mid a \in A_i\} \cup \{\leftarrow a. \mid a \in B_{P_i} \setminus A_i\}$.

For this paper, we will only consider programs of the form $\exists P_1 \forall P_2 \colon C \colon C_w$. We will call P_1 the *existential program*, P_2 the *universal program*, C the *checking*

program (instead of the conventional *constraint program* used in ASP(Q) literature and software to avoid confusion with *constraint programming*) and C_w the *optimization program*. The semantics of such a program is defined as a stable model A_1 to P_1 such that, for each stable model A_2 of $P_2 \uplus_{P_1} A_1$, all rules in $C \uplus_{P_1} A_1 \uplus_{P_2} A_2$ hold; and A_1 is optimal w.r.t. C_w among all those answer sets.

3 Problem Formulation and Encoding

We use a series of ASP(Q) programs to compute step-wise explanations for a given Sudoku puzzle and its unique solution $\mathcal{A}^*$. Each program is of the form $\Pi = \exists P_1 \forall P_2 \colon C \colon C_w$ and encodes a single explanation step. As is customary, we split the description of Π into a *problem formulation* part, encoding the computational problem of finding an explanation step; and an *instance* part, encoding the current state in the explanation process using only ASP-facts.

3.1 Instance Encoding

We encode an intermediate assignment $\mathcal{A}$ into ASP-facts `fact(X, Y, Z)` for each $\{x \mapsto \texttt{Z}\} \in \mathcal{A}$, where x denotes the cell in row `X` and column `Y` and `Z` is the digit that is assigned to that cell. Analogously, we encode the missing digits $\mathcal{A}^* \setminus \mathcal{A}$ using the predicate `remains(X, Y, Z)`. We also state the size of the Sudoku instance here using the predicate `size`. The instance part is merged with the existential program P_1 before solving.

Example 1. Consider the 4×4 Sudoku puzzle shown in Figure 1. The initial configuration of the puzzle is encoded as follows:

```
size(4).
remains(1,1,1). remains(1,2,3). fact(1,3,4).    remains(1,4,2).
fact(2,1,4).    remains(2,2,2). fact(2,3,3).    remains(2,4,1).
remains(3,1,2). fact(3,2,4).    remains(3,3,1). fact(3,4,3).
remains(4,1,3). fact(4,2,1).    remains(4,3,2). remains(4,4,4).
```

3.2 The Problem Formulation

As mentioned above, the problem formulation is split into the existential (P_1), the universal (P_2), the checking (C), and the optimization (C_w) program. Naïvely, one could encode our problem as follows. The existential program picks a single fact-to-explain and the constraints and facts to use in the explanation step, i.e., a subset $\mathcal{O} \subseteq \mathcal{S}$ that satisfies $\mathcal{P}$ (following Definition 1). The universal program goes over all possible assignments to the puzzle where the fact-to-explain is not part of the assignment but the other chosen facts are. Recall that we check for entailment of the fact-to-explain by the used facts and constraints via unsatisfiability when combining the latter with the negation of the fact-to-explain. The checking program verifies that the chosen assignment violates at least one Sudoku constraint. Since this has to hold for all answer sets of the universal

program (i.e., all assignments to the puzzle), this means that $R \cup \mathcal{O}$ (as in Definition 1) is unsatisfiable. Lastly, the optimization program calculates the cost of the choices in the existential program, i.e., the function g in Definition 1, forcing subsequent search to find cheaper explanations.

Notice, however, how a lot of work is "wasted" by the way the universal program picks an assignment, because no information on obvious contradictions is available at this level. Internally, an ASP(Q) solver would therefore need to make unnecessarily many calls to an underlying solver. We circumvent this by "pushing up" the assignment checking to the universal program. This improved solving time drastically in preliminary experiments. More specifically, we add rules enforcing that the chosen assignment *satisfies* all selected constraints and change the checking program to just contain a contradiction. In other words, instead of checking that all assignments violate at least one selected constraint, we say that all assignments that *do* satisfy all constraints also satisfy a contradiction. This means that there is no assignment that satisfies all constraints.

We now present the problem formulation shown in Listing 1.2 using Clingo syntax and following the Generate-Define-Test methodology. The comments in lines 1, 13, 33, and 35, which start with `%@`, demarcate the four subprograms for the ASP(Q) solver. In the existential program P_1, we define the Sudoku digits (line 2), predicates `row` denoting rows and `col` denoting columns (line 3), and constraint types (line 4; i.e., `r`(ow), `c`(olumn) and `b`(ox) constraints) appearing in the puzzle in a way that facilitates further machine processing of ASP(Q) solver output. To keep our problem formulation agnostic of the puzzle size, we define `boxSize` (line 5) and `boxIndex` (line 6) based on the `size` predicate from the instance part. The latter is used in the following (lines 7–9) to associate each box with an identifier. We use the predicate `box(B, X, Y)` to encode that the cell in row `X`, column `Y` is part of box `B` (when numbered in left-to-right top-to-bottom order). Lines 10–12 generate a candidate OCUS. They express that we want to explain exactly one fact (denoted by predicate `explainFact(X,Y,Z)`: the fact that row `X` column `Y` contains the digit `Z`), using some subset of the available facts (`fact(X,Y,Z)`: defined analogously) and constraints (`useConstraint(T,N)`: the `N`th constraint of type `T`). As such, the stable models of P_1 correspond to candidate explanation steps.

In the universal program, we use predicates `sol(X, Y, Z)` and `nsol(X, Y, Z)`, respectively denoting what is part of the assignment and what is not, to encode a total assignment to the puzzle. Lines 14–15 enforce that all the facts used in our explanation are part of the assignment and that the fact-to-explain is *not* part of the assignment. We define useful projection predicates of the assignment in lines 18–21. As described above, we also force the chosen assignment to satisfy all Sudoku constraints, namely that each Sudoku cell has exactly one digit in it (lines 22–23), and that all row (lines 24–25), column (lines 26–27), and box (lines 28–32) constraints are satisfied.

As described above, the checking program C (lines 33–34) just contains a contradiction. In the optimization program C_w, we choose a cost of 10 for each used constraint (line 36) and a cost of 1 for each used fact (line 37). This fol-

```
1  %@exists
2  num(1..X) :- size(X).                          ▷Define domain predicates
3  row(X) :- num(X). col(X) :- num(X).
4  conType(r). conType(c). conType(b).
5  boxSize(B) :- B = #max{X: num(X), num(X*X)}.
6  boxIndex(1..B) :- boxSize(B).
7  box(B*(N-1) + M, X, Y) :- boxSize(B), row(X), col(Y),
8          boxIndex(N), boxIndex(M),
9          B*(N-1) < X < B*N + 1, B*(M-1) < Y < B*M + 1.
10 {useFact(X,Y,Z) : fact(X,Y,Z)}.              ▷Generate candidate OCUS
11 {useConstraint(T,N) : conType(T), num(N)}.
12 1 <= { explainFact(X,Y,Z) : remains(X,Y,Z) } <= 1.
13 %@forall
14 sol(X,Y,Z)  :- useFact(X,Y,Z).              ▷Generate total assignment
15 nsol(X,Y,Z) :- explainFact(X,Y,Z).
16 sol(X,Y,Z)  :- row(X), col(Y), num(Z), not nsol(X,Y,Z).
17 nsol(X,Y,Z) :- row(X), col(Y), num(Z), not sol(X,Y,Z).
18 hasVal(X,Y) :- sol(X,Y,Z).                  ▷Define auxiliary predicates
19 inRow(X,N) :- sol(X,Y,N).
20 inCol(Y,N) :- sol(X,Y,N).
21 inBox(S, N) :- sol(X, Y, N), box(S, X, Y).
22 :- sol(X,Y,Z), sol(X,Y,Z'), Z>Z'.            ▷Test Sudoku constraints
23 :- row(X), col(Y), not hasVal(X,Y).
24 :- useConstraint(row,X), num(N), not inRow(X,N).
25 :- useConstraint(row,X), sol(X,Y,Z), sol(X,Y',Z), Y > Y'.
26 :- useConstraint(col,Y), num(N), not inCol(Y,N).
27 :- useConstraint(col,Y), sol(X,Y,Z), sol(X',Y,Z), X > X'.
28 :- useConstraint(box,S), num(N), not inBox(S, N).
29 :- useConstraint(box,S), box(S,X,Y), box(S,X',Y'),
30         X'>X, sol(X,Y,N), sol(X',Y',N).
31 :- useConstraint(box,S), box(S,X,Y), box(S,X',Y'),
32         Y'>Y, sol(X,Y,N), sol(X',Y',N).
33 %@constraint
34 contradiction.  :- contradiction.
35 %@global
36 :~ useConstraint(T,X).   [10@1, T,X]                ▷OCUS weights
37 :~ useFact(X,Y,Z).       [1@1, X,Y,Z]
```

Listing 1.2: ASP(Q) problem formulation

lows the reasoning by Bogaerts et al. that facts should be easier to use than constraints [6]. Of course, the weights can be adjusted to the user's preference; for instance building on research on how to learn these weights from user feedback [12].

3.3 Adding Redundant Constraints

We can improve our encoding with redundant constraints. We incorporate the following observations into the existential program by replacing the choices for used facts and constraints (lines 10–11) with Listing 1.3. First, all used facts and the fact-to-explain must appear in a constraint that is used in the explanation.

```
1  chooseFactIn(T,N) :- useConstraint(T,N),                          ▷Define
2          useFact(X,Y,Z), in(X,Y,T,N).
3  overlap(T,N,T',N') :- in(X,Y,T,N), in(X,Y,T',N'), T>T'.
4  overlap(T,N,T',N') :- overlap(T',N',T,N).
5  {useConstraint(T,N)} :- explainFact(X,Y,Z), in(X,Y,T,N).    ▷Generate
6  {useConstraint(T,N) : overlap(T,N,T',N')} :- useConstraint(T',N').
7  {useFact(X,Y,Z)} :- fact(X,Y,Z), useConstraint(T,N), in(X,Y,T,N).
8  2 <= {useConstraint(T,N) : overlap(T,N,T',N')} :-
9          useConstraint(T',N'),  not chooseFactIn(T',N').
10 factsNeeded(X) :- boxSize(B), X=2(B-1).  ▷Sudoku specific constraint
11 :- {useFact(X,Y,Z) : fact(X,Y,Z) } < N, factsNeeded(N).
```

Listing 1.3: Improvements to the existential program P_1

Otherwise, such a fact could be dropped, contradicting optimality. Second, the set of used constraints must all be *connected* (either directly by sharing a joint variable or indirectly through other constraints). To encode these observations, we define a helper predicate `chooseFactIn` representing that some fact in a constraint is used (lines 1–2). Further, we define that constraints `overlap` when they share a cell (lines 3–4). We then recursively choose constraints to be used among those that either mention the fact-to-explain (line 5), or overlap with a used constraint (line 6) and we choose facts to be used among those appearing in chosen constraints (line 7). Further, if we use a constraint without using a fact in it, then it must "bridge" two other constraints (lines 8–9). Notice that these improvements apply to all CSPs, not only Sudoku puzzles.

To showcase, how problem specific information is easily represented in our approach, we also introduce a redundant, Sudoku-specific constraint in lines 10–11. For an $n \times n$ Sudoku, an explanation step must contain at least $2(\sqrt{n}-1)$ facts. To illustrate, notice that $\sqrt{n}$ is both the size of a box and the number of boxes per row or column. An explanation placing a digit somewhere in a specific box constraint must therefore rule it out from the $\sqrt{n}-1$ other rows and $\sqrt{n}-1$ other columns it could have gone into. Figure 1 (middle) shows such an explanation step. Analogously, the same holds for row and column constraints.

3.4 Bound-Driven Search

The default search strategy in ASP(Q) solvers is solution-improving search, which we observed spends a lot of time incrementally improving suboptimal solutions. Convergence to an optimal solution is slow, because there are many candidate explanation steps of high cost. To alleviate this, we use a simple, bound-driven search strategy. As the cost of an explanation step is dominated by the number of constraints it uses, we make calls to the ASP(Q) solver with an additional restriction that it should search for an optimal explanation step that uses a fixed number of constraints. This number of constraints is initially 1 and is increased until no optimal solution can exist with strictly more constraints (i.e., as soon as we have found a solution with cost below or equal to $10 \cdot n$, we know

that we should never consider explanation steps that use n or more constraints). To achieve this, we add the following line to the existential program:

```
:- {useConstraint(T,X)} != CONSTRAINT_COUNT.
```

where `CONSTRAINT_COUNT` denotes the number of constraints we are currently allowing. Notice that when a returned explanation step contains strictly more than 10 facts, the search will continue to the next level, as a better solution might still be possible (given that constraints have cost 10 and facts have cost 1).

4 Experiments

In this section, we first evaluate the proposed improvements regarding redundant constraints and bound-driven search. Then, we will compare our best encoding against the previous IHS-based approach.

Conveniently, the python library CPMpy [16] provides means for manipulation of literals, facts and constraints according to Algorithm 1, as well as a built-in IHS-based OCUS procedure. CPMpy also provides a translation of any CSP modelled in it to a CNF formula. Using this translation, we initially attempted to obtain an ASP(Q) encoding that was *fully agnostic* of the application CSP, i.e., the instance part would encode precisely the reified constraints R and soft literals $\mathcal{S}$ according to Definition 1, and the problem formulation part would encode conditions (1)-(3) specialized to the function Explain-One-Fact. The allocation of tasks to subprograms in our attempt was identical to the encoding presented above, and the addition of redundant constraints could be done in similar ways. However, even with sophisticated modeling techniques, we could not compensate for the increase in size in the instance encoding and complexity in the problem formulation. While the approach was feasible for 4×4 instances, it was completely unable to scale to 9×9 Sudokus, with some explanation steps requiring more than 20 hours of compute time. We therefore choose to exclude this method here from our further discussion. Rather unexpectedly, we found the built-in OCUS method of CPMpy to work remarkably well with this CNF translation of the high-level Sudoku modeling. Below, we compare our approach to the previously developed OCUS method in both a high-level formulation that we call "N-IHS" (for "natural"), and its CNF translation "CNF-IHS".

All our experiments[4] were run on identical machines equipped with 12th Gen Intel® Core™i7-12700 CPUs and 64GB of memory running Linux 5.15. We use existing puzzle instances for our experiments [14], picking all 9 4×4 puzzles and the 5 first 9×9 puzzles of each difficulty level. We added a tenth randomly generated 4×4 instance of our own. For computing a complete explanation sequence we set a time limit of 15 minutes (resp. 90 minutes) for 4×4 (resp. 9×9) puzzles. As ASP(Q) solver, we use Casper [10]. We let explanation sequences diverge, i.e., the compared techniques do not necessarily compute the same explanation sequences. This means that the nth explanation step is not necessarily comparable across methods for the same Sudoku puzzle if the previous steps differ. Thus,

[4] The source code is available at: https://doi.org/10.5281/zenodo.21070133

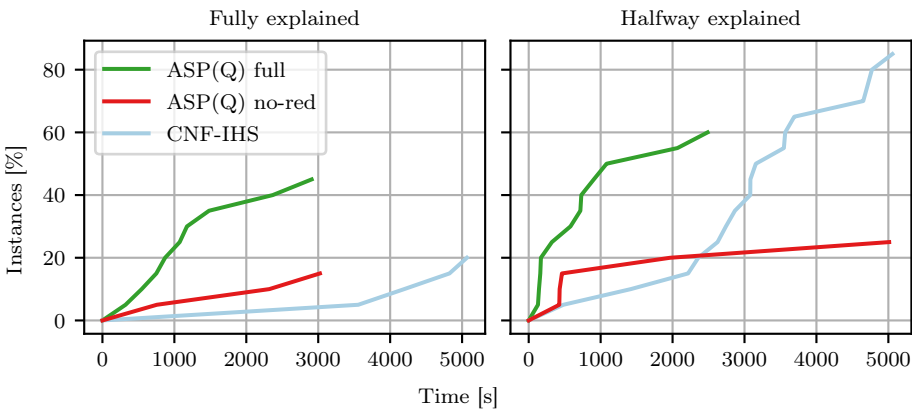

Figure 2: The share of 9×9 Sudokus that could be fully explained (left) and "halfway" (right), i.e., to 40 filled cells before timeout, using ASP(Q) and IHS methods, respectively. Both, the no-bds ASP(Q) variant and the N-IHS variant did not explain a single instance even halfway and are hence not shown.

we compare methods on whole explanation sequences, rather than on individual explanation steps.

4.1 Comparison of ASP(Q) Variations

We test our encoding presented in Section 3, which we call the "full" encoding, against the two variations in which we either *retract the redundant constraints addition* ("no-red") or *disable the bound-driven search strategy* ("no-bds"). For 4×4 instances we observed that the no-red variant finished only 4 out of 10 instances within the given time limit, whereas both other encodings explained all instances fully. For the 9×9 case, the no-bds encoding did not achieve more than 3 explanations steps for any instance, while the no-red and full encodings finished 3 and 9 complete explanation sequences, respectively. Figure 2 shows how many puzzles could be fully and "halfway" explained, respectively.

4.2 Comparison to the Previous Approach

Our observation that the larger instance size in the CNF encoding drastically slows down the ASP(Q) solver, does not seem to transfer to the IHS based methods. In fact, out of 20 9×9 instances, our "full" ASP(Q) approach explained 9 fully, CNF-IHS finished 4 sequences, and N-IHS none. ASP(Q) struggles on instances of high cost and hence either breezes through the whole sequence if it only contains easy steps or gets stuck on a hard step. On the other hand, CNF-IHS takes longer to find easy steps, but manages to also compute hard steps without timing out (c.f. Figure 3). Interestingly, this leads to CNF-IHS explaining fewer instances fully, but more instances to the midway point (c.f. Figure 2).

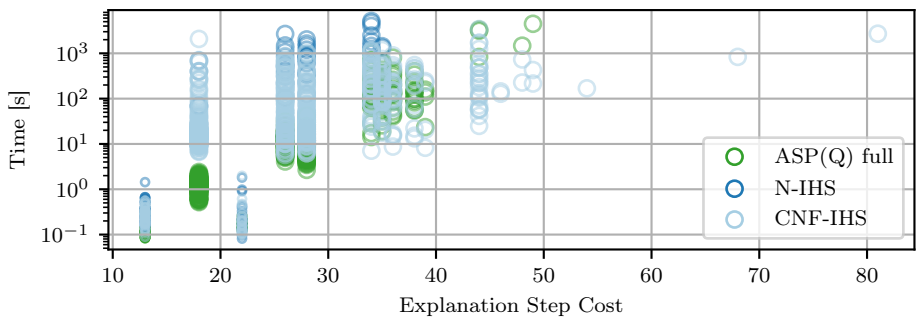

Figure 3: Times to compute calls to Explain-One-Fact depending on the optimal cost of that step and the method used. Steps across all 4×4 (resp. 9×9) instances are shown as small (resp. big) circles.

For each 9×9 instance, we recorded the explanation sequences that explained the most facts and later *re-played* it using both the full ASP(Q) and CNF-IHS methods. By this we mean that sequences no longer diverge, i.e., regardless of the explanation step found by the respective method, we update the current state in the explanation process using the previously recorded step. Thereby, we can (1) validate that both sequences indeed find the same optimal cost and (2) draw meaningful comparisons regarding their respective runtimes on the level of explanation steps (instead of whole sequences). For these experiment we chose a timeout of 10 minutes per explanation step, (over-)estimating a maximum user patience. Our results are visualized in Figure 4. A clear majority ($\approx 85\%$) of steps are solved faster using our proposed approach, especially relatively easy steps. For the few harder steps we encountered, ASP(Q) routinely times out, while CNF-IHS is more likely to yield an explanation in acceptable time.

5 Discussion & Conclusion

In this paper, we presented a novel application of ASP with quantifiers. We showed that it is feasible to use ASP(Q) to compute optimal explanation steps for constraint satisfaction problems, with a specific focus on Sudoku puzzles. We presented a direct encoding of the Sudoku problem in ASP that is competitive with the previously proposed IHS-based approach and even outperforms it on several instances in our empirical evaluation. We also discussed a more general encoding that can be applied to any problem that can be translated to a CNF formula, allowing it can be integrated directly into a library such as CPMpy which supports translations into CNF. Unfortunately, this generic approach did not scale well to realistically sized puzzles.

This work highlights opportunities for future work. First, designing translators of high-level constraint models into ASP could result in fully-general and efficient explanation computation methods, circumventing the need for translation

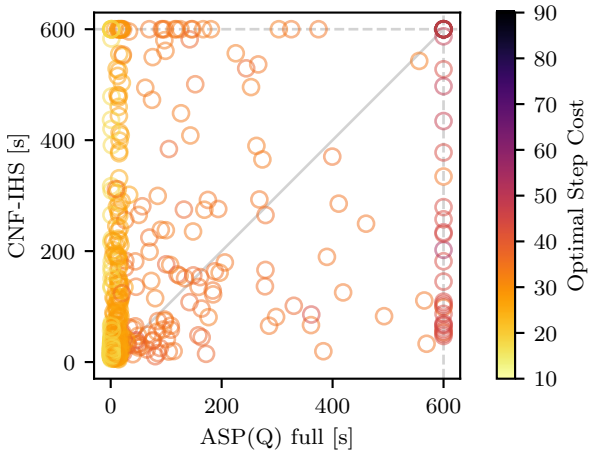

Figure 4: Comparison of runtimes to compute optimal explanations steps using ASP(Q) and CNF-IHS, respectively. Histograms show distributions with bucket sizes of 3 seconds. The optimal cost of an explanation step is indicated by color.

to CNF. Second, the performance of our hand-crafted optimization compared to the standard solution-improving search stresses the need for programmable search strategies directly in ASP(Q) solvers. Third, we conjecture that our approach would benefit greatly from an ASP(Q)-specific core-guided optimization algorithm [2]. Finally, our optimization loop makes use of repeated calls to the solver with only a single constraint changing, meaning a lot of work probably needs to be redone. As such, we conjecture that an incremental solving interface for ASP(Q) would be beneficial.

Acknowledgments. This work was partially supported by Fonds Wetenschappelijk Onderzoek — Vlaanderen (project G064925N) and the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Amendola, G., Ricca, F., Truszczynski, M.: Beyond NP: quantifying over answer sets. Theory Pract. Log. Program. **19**(5-6), 705–721 (2019)
2. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: ICLP (Technical Communications). LIPIcs, vol. 17, pp. 211–221. Schloss Dagstuhl (2012)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability - Second Edition, FAIA, vol. 336. IOS Press (2021)

4. Bleukx, I., Boumazouza, R., Guns, T., Laage, N., Povéda, G.: Modeling and explaining an industrial workforce allocation and scheduling problem. In: CP. LIPIcs, vol. 340, pp. 6:1–6:24. Schloss Dagstuhl (2025)
5. Bleukx, I., Devriendt, J., Gamba, E., Bogaerts, B., Guns, T.: Simplifying step-wise explanation sequences. In: CP. LIPIcs, vol. 280, pp. 11:1–11:20. Schloss Dagstuhl (2023)
6. Bogaerts, B., Gamba, E., Guns, T.: A framework for step-wise explaining how to solve constraint satisfaction problems. Artif. Intell. **300**, 103550 (2021)
7. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Stable-unstable semantics: Beyond NP with normal logic programs. Theory Pract. Log. Program. **16**(5-6), 570–586 (2016)
8. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
9. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., Schaub, T.: ASP-Core-2 input language format. Theory Pract. Log. Program. **20**(2), 294–309 (2020)
10. Cuteri, A., Mazzotta, G., Ricca, F.: 2-ASP(Q) solving based on CEGAR. In: AAAI. pp. 19030–19038. AAAI Press (2026)
11. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Ann. Math. Artif. Intell. **15**(3-4), 289–323 (1995)
12. Foschini, M., Defresne, M., Gamba, E., Bogaerts, B., Guns, T.: Preference elicitation for step-wise explanations in logic puzzles. In: AAAI. pp. 14225–14233. AAAI Press (2026)
13. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints An Int. J. **13**(3), 268–306 (2008)
14. Gamba, E., Bogaerts, B., Guns, T.: Efficiently explaining CSPs with unsatisfiable subset optimization. J. Artif. Intell. Res. **78**, 709–746 (2023)
15. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2012)
16. Guns, T.: Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In: ModRef. vol. 19 (2019)
17. Gupta, S.D., Genc, B., O'Sullivan, B.: Explanation in constraint satisfaction: A survey. In: IJCAI. pp. 4400–4407. ijcai.org (2021)
18. Junker, U.: QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In: AAAI. pp. 167–172. AAAI Press / The MIT Press (2004)
19. Leo, K., Tack, G.: Debugging unsatisfiable constraint models. In: CPAIOR. LNCS, vol. 10335, pp. 77–93. Springer (2017)
20. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
21. Lifschitz, V.: What is answer set programming? In: AAAI. pp. 1594–1597. AAAI Press (2008)
22. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: CP. LNCS, vol. 4741, pp. 529–543. Springer (2007)
23. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier (2006)
24. Sierksma, G.: Linear and integer programming - theory and practice, Pure and applied mathematics, vol. 198. Dekker (1996)
25. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4), 733–742 (1976)