What Are the Rules? Discovering Constraints from Data

Boris Wiegand[1,2], **Dietrich Klakow**[2], **Jilles Vreeken**[3]

[1] SHS – Stahl-Holding-Saar, Dillingen, Germany
[2] Saarland University, Saarbrücken, Germany
[3] CISPA Helmholtz Center for Information Security, Germany
boris.wiegand@stahl-holding-saar.de, dietrich.klakow@lsv.uni-saarland.de, jv@cispa.de

Abstract

Constraint programming and AI planning are powerful tools for solving assignment, optimization, and scheduling problems. They require, however, the rarely available combination of domain knowledge and mathematical modeling expertise. Learning constraints from exemplary solutions can close this gap and alleviate the effort of modeling. Existing approaches either require extensive user interaction, need exemplary invalid solutions that must be generated by experts at great expense, or show high noise-sensitivity.

We aim to find constraints from potentially noisy solutions, without the need of user interaction. To this end, we formalize the problem in terms of the Minimum Description Length (MDL) principle, by which we select the model with the best lossless compression of the data. Solving the problem involves model counting, which is #P-hard to approximate. We therefore propose the greedy URPILS algorithm to find high-quality constraints in practice. Extensive experiments on constraint programming and AI planning benchmark data show URPILS not only finds more accurate and succinct constraints, but also is more robust to noise, and has lower sample complexity than the state of the art.

Introduction

Constraint programming, the holy grail of programming (Barták 1999), separates the concerns of modeling a problem and finding a solution. As modeling the problem requires the rarely available combination of both domain knowledge and mathematical modeling expertise, learning constraints from data enables broader application of constraint programming (O'Sullivan 2010). Handcrafted solutions are often recorded for real-world assignment problems like scheduling and staff rostering, and thus provide a promising knowledge base to mine constraints. Existing approaches do not satisfactorily solve this task. Active learning (Bessiere et al. 2013; Tsouros and Stergiou 2020; Belaid et al. 2022) needs thousands of queries even for simple problems, which is intractable if a human expert must label these queries. Passive learning approaches (Pawlak and Krawiec 2017; Kumar et al. 2020; Prestwich et al. 2021) need invalid examples, i.e., non-solutions, in their training set. Those are usually not collected and experts must create them at great expense.

State-of-the-art methods to learn constraints purely from valid solutions either suffer from a limited constraint language, resulting in a long list of hard-to-read constraints, and need a lot of data (Prestwich 2021), or cannot learn from real-world data because they are not robust to noise (Kumar et al. 2019; Kumar, Kolb, and Guns 2022). Furthermore, although learning conditions for actions in AI planning is closely related to learning constraints for constraint programming, none of the existing approaches is directly applicable to AI planning problems. Most of AI planning specific work (Arora et al. 2018; Aineto, Celorrio, and Onaindia 2019; Segura-Muros, Pérez, and Fernández-Olivares 2021) considers constraint learning as only one of many problems to solve. Not focusing on constraint learning prevents these methods from being effective on this task.

To overcome all these limitations, we formalize the problem of learning constraints from exemplary solutions in terms of the Minimum Description Length (MDL) principle, by which we select the model with the best lossless compression of the data. Since solving the problem exactly involves #P-hard model counting, we propose the greedy URPILS algorithm for Unveiling Rules from PosItive LabelS. Through extensive experiments on both constraint programming and AI planning benchmark data, we empirically show that URPILS discovers more accurate and succinct constraints with less constraint terms, is more robust to noise, and has lower sample complexity than the state of the art. In summary, our main contributions are as follows. We

(a) formalize the problem of learning constraints from exemplary solutions in terms of the MDL principle,

(b) propose an efficient heuristic to discover constraints for both constraint programming and AI planning,

(c) provide an extensive empirical evaluation,

(d) make code, data and additional details publicly available in the supplementary materials.

In the next section, we introduce necessary notation and basic concepts we use in the paper. Then, we formalize the problem in terms of MDL. Next, we propose our greedy URPILS algorithm and describe how to adapt URPILS to AI planning problems. After giving an overview of related work, we provide an extensive empirical evaluation on benchmark datasets. Finally, we discuss limitations, outline potential future work and draw a conclusion.

Preliminaries

Before we formalize the problem, we introduce notation and basic concepts we use in the paper.

Boolean Constraint Programming

Assume we are given a list of *object sets* $O_1, \dots, O_k$ and their Cartesian product $X = \prod_{i=1}^{k} O_i$. As an example, consider the 8-Queens problem, where we want to place eight queens on a 8×8 chess board, such that no two queens attack each other. We define an object set $O_1 = \{Q_1, \dots, Q_8\}$ for queens, and an object set $O_2 = \{S_1, \dots, S_{64}\}$ for squares on the board. An *assignment* is a boolean function $f_a : X \to \{0, 1\}$, e.g., $f_a(Q_1, S_{42}) = 1$ means queen Q_1 is on square S_{42}. We call f_a a *valid* assignment, if it satisfies a set of constraints, i.e., a *model* $M = \{C_1, \dots, C_m\}$, with $C_i : X \to \{0, 1\}$ and f_a is valid iff $\forall x \in X \; \forall C_i \in M : C_i(x) = 1$. For a given model M, we denote the set of valid assignments by $\mathcal{F}_M$. We define constraints by a boolean algebra over the assignment f_a, a set of boolean relations between objects $\mathcal{F}_{\mathbb{B}} = \{f_1, \dots, f_{|\mathcal{F}_{\mathbb{B}}|}\}$ with $f_i : \prod_{j \in \{1,\dots,k\}^+} O_j \to \{0, 1\}$, and arithmetic expressions over a set of numeric relations $\mathcal{F}_{\mathbb{R}} = \{f_1, \dots, f_{|\mathcal{F}_{\mathbb{R}}|}\}$ with $f_i : \prod_{j \in \{1,\dots,k\}^+} O_j \to \mathbb{R}$.

In the 8-Queens example, we assign rows and columns to squares. Formally, we define $\mathcal{F}_{\mathbb{R}} = \{f_x, f_y\}$ with $f_x : O_2 \to \{1, \dots, 8\}$ and $f_y : O_2 \to \{1, \dots, 8\}$. The constraint that no more than one queen may be placed in a row can then be written as $\forall (q_1, q_2, s_1, s_2) \in O_1 \times O_1 \times O_2 \times O_2 : (s_1 \neq s_2 \wedge f_x(s_1) = f_x(s_2)) \to (f_a(q_1, s_1) \to \neg f_a(q_2, s_2))$. Our goal is to find constraints like these from a dataset of exemplary valid assignments $D = \{f_a^1, \dots, f_a^n\}$.

Minimum Description Length Principle

We use the Minimum Description Length (MDL) principle (Rissanen 1978; Grünwald 2007) for model selection. MDL identifies the best model as the one with the shortest lossless description of the given data. In MDL, we only compute code lengths, but are not concerned with actual code words. Formally, given a set of models $\mathcal{M}$, the best model is defined by $\arg\min_{M \in \mathcal{M}} L(M) + L(D \mid M)$, in which $L(M)$ is the length in bits of the description of M, and $L(D \mid M)$ is the length of the data encoded with the model. This form of MDL is known as two-part or crude MDL. Although one-part or refined MDL provides stronger theoretical guarantees, it is only computable in specific cases (Grünwald 2007). Therefore, we use two-part MDL. Next, we formalize our problem in terms of MDL.

MDL for Constraint Learning

From a set of exemplary assignments, we aim to discover a succinct set of constraints fitting and explaining the observed data and generalizing well to unseen data. To account for potential noise in real-world data, we need a noise-robust discovery approach. Thus, we formalize the problem of constraint discovery from exemplary solutions in terms of the MDL principle. To this end, we define length of the data encoding $L(D \mid M)$, length of the model encoding $L(M)$, and finally give a formal problem definition.

Data Encoding for Constraint Programming

To encode a dataset D, we encode all its assignments, i.e.,

$$L(D \mid M) = \sum_{f_a \in D} L(f_a \mid M) \; .$$

An empty model without constraints has $|\mathcal{F}_M| = 2^{|X|}$ valid assignments, and we need $|X|$ bits to choose one. The more constraints the model contains, the smaller the set of valid assignments, and the cheaper it is to identify the actual one. As real-world data is often noisy, there may not exist a valid assignment matching the exemplary data exactly. To ensure a lossless encoding, we have to encode the errors of the best fitting assignment. We denote the number of errors by

$$\mathit{error}(M \mid f_a) = \min_{f'_a \in \mathcal{F}_M} \sum_{x \in X} \mathbb{1}_{f'_a(x) \neq f_a(x)}(x) \; .$$

To encode the errors, we first specify their number by the MDL-optimal encoding for integers $z \geq 1$ (Rissanen 1983) defined as $L_{\mathbb{N}}(z) = \log c_0 + \log z + \log \log z + \dots$, and we sum only the positive terms, and c_0 is set to 2.865064 to satisfy the Kraft inequality for a lossless encoding. Then, we encode the incorrect assignment values by a data-to-model code (Li and Vitányi 1993), i.e., an index to choose $\mathit{error}(M \mid f_a)$ out of $|X|$ values. In summary, we have

$$\begin{aligned} L(f_a \mid M) = {} & \log |\mathcal{F}_M| + L_{\mathbb{N}}\left(1 + \mathit{error}(M \mid f_a)\right) \\ & + \log \binom{|X|}{\mathit{error}(M \mid f_a)} \; . \end{aligned}$$

This gives us a lossless encoding of the data.

Model Encoding

Next, we compute the length of the model encoding by

$$L(M) = L_{\mathbb{N}}(|M| + 1) + \sum_{C \in M} L(C) \; ,$$

i.e., we encode the number of constraints, which can be zero, and encode each constraint. We first define a grammar of our constraint language for complex real-world problems by

$$\begin{aligned}
& C \to \langle C_V \rangle \text{“}|\text{”} \langle C_F \rangle \; : \; \langle C_T \rangle \\
& C_V \to \epsilon \mid \forall x \in X \mid \forall x, y \in X \\
& C_F \to \epsilon \mid \langle v \rangle \; = \; \langle v \rangle \mid \langle v \rangle \; \neq \; \langle v \rangle \mid \langle f_{\mathbb{B}} \rangle(\langle v \rangle) \mid \langle \text{NF} \rangle \mid \\
& \qquad \neg \langle C_F \rangle \mid \langle C_F \rangle \wedge \langle C_F \rangle \mid \langle C_F \rangle \vee \langle C_F \rangle \\
& v \to x_{\langle i \rangle} \mid y_{\langle i \rangle} \qquad i \to 1 \mid \dots \mid k \qquad f_{\mathbb{B}} \to \text{one of } \mathcal{F}_{\mathbb{B}} \\
& \text{NF} \to \langle \text{NE} \rangle < \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \leq \langle \text{NE} \rangle \mid \langle \text{NE} \rangle = \langle \text{NE} \rangle \\
& \text{NE} \to \langle z \in \mathbb{R} \rangle \mid \langle f_{\mathbb{R}} \rangle(\langle v \rangle) \mid \langle \text{NE} \rangle \langle \odot \rangle \langle \text{NE} \rangle \mid \\
& \qquad \text{“}|\text{”} \langle \text{NE} \rangle \text{“}|\text{”} \mid \lfloor \langle \text{NE} \rangle \rfloor \mid \lceil \langle \text{NE} \rangle \rceil \\
& f_{\mathbb{R}} \to \text{one of } \mathcal{F}_{\mathbb{R}} \qquad \qquad \odot \to + \mid - \mid \cdot \mid / \\
& C_T \to f_a(x) \mid f_a(y) \mid f_a(X_{\langle j \rangle}) \mid \neg \langle C_T \rangle \mid \langle C_T \rangle \wedge \langle C_T \rangle \mid \\
& \qquad \langle C_T \rangle \vee \langle C_T \rangle \mid \langle \text{COUNT} \rangle \\
& j \to 1 \mid \dots \mid |X| \\
& \text{COUNT} \to \langle \text{NE} \rangle \leq \sum_{\langle v \rangle} f_a(x) \leq \langle \text{NE} \rangle \; .
\end{aligned}$$

We conceptually split a constraint C into three parts, i.e., $C = (C_V, C_F, C_T)$. In C_V, we can define *variables* of object tuples in X. In C_F, we *filter* the possible values of these variables: we can test for equality and inequality of variables, we can query values of boolean and numerical relations, and we can compose complex filters with boolean operators. A numeric filter NF compares the values of two numeric expressions NE, which are any real number, any numeric relation, or a composite of arithmetic operations.

The *target* of any constraint is to define the set of valid assignments. In C_T, we restrict the valid values of an assignment f_a by a boolean expression over f_a. In its simplest form, C_T requires f_a to be true for a variable defined by C_V and C_F. We can also require f_a to be true for one specific parameter combination X_j with $j \in \{1, \ldots, |X|\}$. We can compose more complex constraints using boolean operators. In many real-world problems, we can distribute some kind of budget. For instance, if we assign shifts to employees during rostering, employees require a minimal and maximal workload. We model such COUNT constraints by a lower and upper bound on a sum over the assignment values of f_a.

When computing the encoded length of a constraint, we want to avoid any undue bias and therefore assume that whenever we have multiple modeling choices, all options are equally likely. Formally, we use our defined constraint grammar to recursively compute $L(C)$ by

$$L(A) = \log |A| + \sum_{\langle \alpha \rangle \in A} L(\alpha) \ ,$$

where A is a nonterminal in the grammar, and we first encode which of the $|A|$ branches we produce, before we encode all remaining nonterminals. In the special case of $\langle z \in \mathbb{R} \rangle$, we compute the encoded length by $L_{\mathbb{R}}(z)$ (Marx and Vreeken 2019), where we represent z up to a user-specified precision p by the smallest integer shift s such that $z \cdot 10^s \geq 10^p$. We then encode shift, shifted digit and sign, i.e., $L_{\mathbb{N}}(s) + L_{\mathbb{N}}(\lceil z \cdot 10^s \rceil) + 1$. Altogether, this gives us a lossless encoding of the model.

Formal Problem Definition

Using our MDL score, we now formally state our problem.

Minimal Constraint Model Problem *Given a set D of assignments $f_a^1, \ldots, f_a^n$, find the constraint model M minimizing the total encoded cost $L(D, M) = L(D \mid M) + L(M)$.*

Solving this problem optimally is intractable in practice. Potentially, we have up to $2^{|X|}$ valid assignments, i.e., we face an exponentially growing search space for constraints. Moreover, our MDL score does not exhibit properties such as monotonicity or submodularity that we can exploit to efficiently find an optimal solution. We give a counterexample for both properties in the supplementary materials. Additionally, even computing $L(D \mid M)$ is hard by itself. Finding a valid assignment f'_a for M that is nearest to a given assignment f_a corresponds to finding a valid assignment having maximal Manhattan distance to f_a with negated values, which in general is NP-hard (Crescenzi and Rossi 2002).

Computing the number of valid assignments $|\mathcal{F}_M|$ is equivalent to counting the solutions of a boolean formula, which is #P-complete, i.e., at least as hard as NP-complete (Valiant 1979). Researchers have proposed algorithms like GANAK (Sharma et al. 2019), SHARPSAT-TD (Korhonen and Järvisalo 2021) or APPROXMC (Soos and Meel 2019) to tackle the problem. Dependent on the complexity of the formula, these approaches take seconds, minutes or even hours (Fichte, Hecher, and Hamiti 2021), which is too slow for evaluating many constraint candidates during search.

The URPILS Algorithm

Since solving the minimal constraint model problem optimally is intractable, we resort to greedy solutions.

Estimating the Number of Valid Assignments

To compute $L(f_a \mid M)$, we must count the number of valid assignments $|\mathcal{F}_M|$ for a given model M. We use an approximation, which is fast to compute and still enables useful comparison of constraint candidates. We estimate the number of valid assignments based on a standard algorithm for exact counting (Zhou, Yin, and Zhou 2010). First, we transform our constraint model M into a boolean function of conjunctive normal form (CNF), where each possible parameter combination of f_a corresponds to a boolean variable. Next, we compute the *constraint graph* G of the formula, which is an undirected graph with variables as nodes, and two variables are connected if they occur together in a clause. We count the number of valid assignments separately for disconnected, i.e., independent, components and get the total count by multiplying the result of each component.

If the graph is small enough and contains less than five variables, it is feasible to count the number of valid assignments exactly by enumeration. Otherwise, we use a polynomial-time approximation. If clauses in the CNF contain at most two variables, the number of valid assignments corresponds to the number of independent sets in G (Dahllöf, Jonsson, and Wahlström 2005), where an independent set is any set of non-adjacent nodes. We compute a lower bound by (Sah et al. 2019)

$$|\mathcal{F}_M| \geq \prod_{v \in V} (\deg v + 2)^{1/(\deg v + 1)},$$

with V being the set of nodes in G and $\deg v$ denotes the degree of node v. The number of variables per clause only depends on the target part C_T of a constraint. If C_T has the form $f_a(\cdot)$ or $\neg f_a(\cdot)$, we have one variable per clause, whereas implications like $f_a(x) \rightarrow f_a(y)$ result in two variables per clause. In these cases, our lower bound leads to valid results. As we will show later in the experiments, these unary and binary relationships between variables are sufficient to describe most of the constraints in many problems.

Count constraints, however, in general lead to clauses with more than two variables. For instance, let $x_1, \ldots, x_6$ be boolean variables and consider the constraint $\sum_{i=1}^{6} x_i = 3$. Then, the CNF of this constraint is $\prod_{i=1}^{4} \prod_{j=i}^{5} \prod_{k=j}^{6} (x_i + x_j + x_k)$, i.e., we have three variables per clause. Thus, we need to compute $|\mathcal{F}_M|$ differently for count constraints. For a single equality constraint $\sum_{i=1}^{n} x_i = a$, we have $\binom{n}{a}$ satisfying assignments. We generalize this for inequality constraints $a \leq \sum_{i=1}^{n} x_i \leq b$ by $\sum_{i=a}^{b} \binom{n}{i}$.

Since we do not know what the intersection of the valid assignments for multiple count constraints looks like, because enumerating them is intractable, we can only make assumptions. We assume all count constraints equally contribute to the final count, and thus divide the mean of the individual counts by the number of constraints, i.e.,

$$|\mathcal{F}_M| = \left\lceil \frac{\sum_{C \in M} |\mathcal{F}_{\{C\}}|}{|M|^2} \right\rceil .$$

By this, we can estimate the number of valid assignments.

Estimating the Best Fitting Valid Assignment

To compute $L(f_a \mid M)$, we also need to compute $error(M \mid f_a)$, i.e., the minimal number of values we need to change in a valid assignment of M to get f_a. Since we must repeat this computation many times during our search for constraints, we want this to be as fast as possible. As in counting the number of valid assignments, enumerating all assignments to find $error(M \mid f_a)$ is intractable.

In contrast to $error(M \mid f_a)$, the number of unsatisfied clauses in the CNF formula of the model is cheap to compute. The more clauses are unsatisfied, the more variables we expect must be flipped to satisfy the formula, and hence the higher is $error(M \mid f_a)$. We estimate the number of variables we must flip to satisfy the formula by using the coupon collector's problem (Pólya 1930)(Feller 1968, p. 225): If we assume that for each of the m unsatisfied clauses, we draw one of $|V|$ variables with replacement to flip, the expected number of flipped variables is

$$error(M \mid f_a) = \left\lceil |V| - |V| \cdot (1 - \frac{1}{|V|})^m \right\rceil .$$

The value of $error(M \mid f_a)$ is 0 if no clause is unsatisfied, increases with m and does not exceed the number of variables $|V|$. By this, we can compute $L(D, M)$.

Discovering a Good Constraint Model

We now want to minimize $L(D, M)$ for a given dataset D, i.e., we want to discover a good constraint model in feasible time. Since computing $L(D \mid M)$ is harder for models with COUNT constraints, we search for these at the end. Many satisfiability and optimization problems contain a set of relatively simple constraints, even if they also contain a set of very complex constraints. Simple constraints typically involve none or only one feature relation in the filtering part C_F. Therefore, we propose our method URPILS, in which we split the search for constraints into three stages.

We give the pseudocode of URPILS in Algorithm 1. Starting with an empty model, we first search for the low-hanging fruit and generate simple constraint candidates. In constraint programming, we are often interested in modeling the pairwise relationship between variables. For example, we require in Sudoku that two cells in the same row do not have the same value. Hence, we generate a set of simple candidates with all constraints of the form $\forall x, y \in X \mid C_F : f_a$. In C_F, we compare the values of at most one boolean and numerical relation, e.g. $f(x_1) < f(y_1)$ with $f \in \mathcal{F}_{\mathbb{R}}$. To restrict the pairwise assignment values of x and y, we generate implications of the type $f_a(x) \rightarrow f_a(y)$ and $f_a(x) \rightarrow \neg f_a(y)$ for C_T. For further reference, we provide pseudocode for SIMPLECANDS in the supplementary.

Algorithm 1: URPILS

```
input  : dataset $D$
output: set of constraints $M$
$M \leftarrow \emptyset$;
$M \leftarrow$ FILTER$(M, D,$ SIMPLECANDS$(D))$;
$M \leftarrow$ FILTER$(M, D,$ COMPLEXCANDS$(M, D))$;
$M \leftarrow$ FILTER$(M, D,$ COUNTCANDS$(D))$;
return $M$;
```

Algorithm 2: FILTER

```
input  : current model $M$, dataset $D$, set of
         candidates $Q$
output: extended model $M$
sort $Q$ by $L(D, \cdot)$;
foreach $C' \in Q$ do
    foreach $C \in M$ do
        if $C'_V = C_V \wedge C'_T = C_T$ then
            $C'_F \leftarrow C'_F \vee C_F$;
            $M' \leftarrow M \backslash \{C\}$;
            break;
    $M' \leftarrow M' \cup \{C'\}$;
    if $L(D, M') < L(D, M)$ then
        $M \leftarrow M'$;
return $M$;
```

We filter the generated candidates in the FILTER subroutine for which we give the pseudocode in Algorithm 2. We test the most promising candidates first through prioritizing candidates by their individual gain. To minimize model complexity, we try to merge each candidate C' with an existing constraint $C \in M$. We can merge constraints if they share the same variable and target part. For example, we merge $\forall x, y \in X \mid g(x_1) < g(y_1) : f_a(x) \rightarrow \neg f_a(y)$ and $\forall x, y \in X \mid h(x_2) = h(y_2) : f_a(x) \rightarrow \neg f_a(y)$ into $\forall x, y \in X \mid g(x_1) < g(y_1) \vee h(x_2) = h(y_2) : f_a(x) \rightarrow \neg f_a(y)$. If a candidate improves our score, we add it to the model.

A model with simple constraints gives us a good baseline from which we search for constraints with a more complex filtering part. Instead of an intractable search over all possible filtering expressions, we map the problem to a simpler binary classification problem. We test for each pair $x, y \in X$ whether the single implication $f_a(x) \rightarrow f_a(y)$ improves the fit on the data, i.e., it leads to a lower $L(D \mid M)$. We later repeat the search for $f_a(x) \rightarrow \neg f_a(y)$. By this, we get a set of positive and a set of negative implications as targets of a binary classification. We generate features by a recursive enumeration of all possible C_F using our defined constraint grammar. To avoid infinite recursion and combinatorial explosion, we do not generate C_F with conjunctions or disjunctions, and we limit the number of numerical operators. Finally, we look for a set of features best explaining the division into positive and negative implications, which gives us a good candidate for C_F. For reference, we provide details and pseudocode for COMPLEXCANDS in the supplementary.

In the last stage of URPILS, we search for count constraints. To this end, we create candidates for different input partitions of f_a similar to the existing COUNTOR algorithm (Kumar et al. 2019). Formally, we create constraints of the form $\forall x \in X \mid C_F : a \leq \sum f_a(x) \leq b$, where we generate candidates for $a, b \in \mathbb{N}$ by observations in D. We generate an empty C_F, and we generate all possible $C_F = f(x_i)$ with $f \in \mathcal{F}_{\mathbb{B}}$ and $i \in \{1, \ldots, k\}$. Again, we use FILTER to select which candidates we add to our final model. This gives us a set of constraints from exemplary assignments.

URPILS for AI Planning

Next we show how to adapt URPILS for AI planning problems, in which actions change the state of an environment until a predefined goal state is reached. We reuse notation and define a state by boolean and numerical relations between objects from different object sets. We write f_i^j to refer to relation f_i at state j. W.l.o.g we consider a single action a. We denote the assignment at state j by f_a^j, and $f_a^j(x) = 1$ if a is executed with objects x at state j and $f_a^j(x) = 0$ else. As before, we aim to find constraints M for valid assignments and thus preconditions to execute a.

As valid assignments satisfy $\sum_{x \in X} f_a(x) = 1$, the empty model has $|X|$ instead of $2^{|X|}$ valid assignments. To encode errors efficiently, we specify for each assignment in the data if it is valid for M. If we knew the number of valid and invalid assignments beforehand, we could compute the lengths of optimal prefix-codes. To avoid any arbitrary choices, we use prequential codes (Grünwald 2007), which are asymptotically optimal without requiring initial knowledge of the code distribution. If an assignment is valid, we encode it via an index over all valid assignments, otherwise we use an index over all other assignments. Formally, we have

$$L_{\text{AI}}(D \mid M) = \sum_{i=1}^{|D|} - \log \begin{pmatrix} \text{usg}_i \, f_a^i \in \mathcal{F}_M + \epsilon \\ \text{usg}_i \, 0 + \text{usg}_i \, 1 + 2\epsilon \end{pmatrix} + \begin{cases} \log |\mathcal{F}_M| \,, & \text{if } f_a^i \in \mathcal{F}_M \\ \log(|X| - |\mathcal{F}_M|), & \text{otherwise,} \end{cases}$$

where $\text{usg}_i \, x$ is how often code x has been used up to the i-th assignment, and ϵ with standard choice 0.5 is for smoothing. This gives us an efficient encoding for AI planning data.

We also incorporate that $f_a(x) = 1$ for exactly one x into our search candidate generation. A single one in f_a means we neither need constraints on pairwise relationships of f_a nor count constraints. Instead, we search for constraints telling us when we are not allowed to execute an action. This means we create candidates $\forall x \in X \mid C_F : \neg f_a(x)$, where C_F compares boolean and numerical relations. We give pseudocode in the supplementary materials.

Related Work

Learning constraints for constraint programming is a widely studied problem. Active learning approaches (Bessiere et al. 2013; Tsouros and Stergiou 2020; Belaid et al. 2022) derive constraints by asking queries in the form of partial or complete solutions and non-solutions. Even for simple problems, these approaches may require thousands of queries, which limits their applicability if a human must label these queries. Therefore, researchers proposed to learn constraints from a static set of both solutions and non-solutions (Pawlak and Krawiec 2017; Kumar et al. 2020; Prestwich et al. 2021). While handcrafted solutions are usually recorded in real-world applications like scheduling and rostering, non-solutions representing forbidden behavior often are not collected. Thus, data and label acquisition as a bottleneck still can prevent application of such methods in practice.

Recent work finds constraints from solutions only. This often results in methods working in narrow contexts, such as integer linear programming (Meng and Chang 2021), scheduling sequences (Picard-Cantin et al. 2016) or tabular spreadsheets (Kolb et al. 2017). COUNTOR (Kumar et al. 2019) infers count constraints. It, however, cannot handle noise, can only create simple expressions, and does not consider redundancy between constraints. COUNTCP (Kumar, Kolb, and Guns 2022) extends COUNTOR by a richer modeling language and reduced redundancy, but still does not handle noise. MINEACQ (Prestwich 2021) selects constraints by permutation testing. In contrast to COUNTOR and COUNTCP, MINEACQ does not find quantified constraints, which can lead to a large result set. CABSC (Coulombe and Quimper 2022) also selects constraints by counting valid assignments, but needs user-provided knowledge of constraints and does not handle noise.

In AI planning, many approaches try to infer domain models from exemplary execution plans (Arora et al. 2018). Strictly assuming no noise, FAMA (Aineto, Celorrio, and Onaindia 2019) formalizes the problem as a planning problem itself. PLANMINER (Segura-Muros, Pérez, and Fernández-Olivares 2021) translates the problem to a rule-based classification task, and PLANMINER-N (Segura-Muros, Fernández-Olivares, and Pérez 2021) improves PLANMINER's noise-handling. AI planning domain acquisition methods tackle multiple tasks, treating the learning of action constraints as an unfocused subproblem. In contrast to all other methods above, URPILS discovers a succinct set of constraints with low sample complexity, is robust to noise, and can be applied to a broad domain of optimization, satisfiability and planning problems.

Experiments

Now, we evaluate URPILS on constraint programming and AI planning datasets. Since both domains have specialized state-of-the-art methods that are not applicable to both problems, we split the experiments into two. We conducted all our experiments on a PC with Windows 10, an Intel i7-6700 CPU and 32 GB of memory. To ensure reproducibility, we make code and data publicly available in the extra materials.[1]

[1] https://eda.rg.cispa.io/prj/urpils

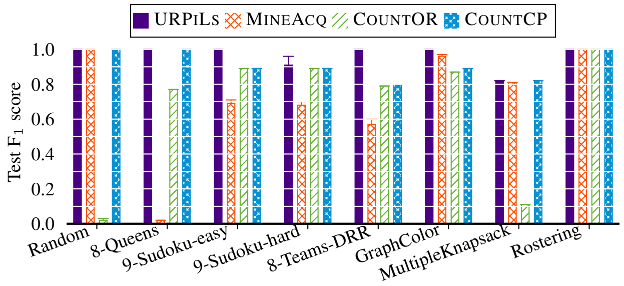

Figure 1: [URPILS discovers high-quality constraints] Average F_1 score on the test set for ten independent runs on training sets with 1000 randomly drawn examples, for constraints discovered by URPILS, MINEACQ, COUNTOR and COUNTCP. Error bars show standard deviation.

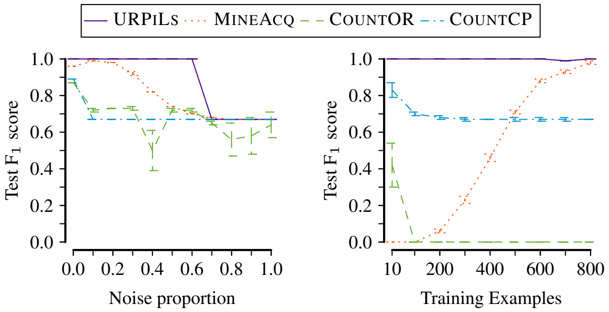

Figure 2: [URPILS is noise-robust with low sample complexity] Mean test F_1 score over ten independent runs on the GraphColor problem, for URPILS, MINEACQ, COUNTOR, and COUNTCP dependent on the proportion of noisy examples in the training set (left). Mean F_1 score on the 5-Queens test set with 10% noise on a varying number of training examples (right). Error bars show standard error.

Experiments on Constraint Programming Datasets

We start by comparing URPILS with the state of the art from related work. While COUNTOR and COUNTCP have no hyperparameters, we must generate candidate constraints for MINEACQ and set parameters τ and ρ to control the acceptance threshold of its permutation test for candidate selection. To ensure MINEACQ can find all necessary constraints to model the datasets without providing too much knowledge about the ground-truth constraints, we generate all pairwise implications $f_a(x) \rightarrow f_a(y)$ and $f_a(x) \rightarrow \neg f_a(y)$. By a manual hyperparameter search, we find $\tau = 10$ and $\rho = 0.001$ lead to the best results.

We experiment on datasets with different characteristics. To test if the constraint learners find spurious results, we create a synthetic dataset *Random*, where we uniformly and randomly sample values for f_a. We also uniformly and randomly sample values for boolean and numerical relations in the dataset, i.e., there is no dependency to f_a, and the ground truth is an empty model without any constraints.

Besides, we evaluate on datasets with non-empty ground-truth constraints. *8-Queens* contains examples for positioning eight queens on a chessboard such that no two queens attack each other. Since modelers may include knowledge about the problem into the modeled relations, we create two versions of a 9×9 Sudoku dataset. In *9-Sudoku-easy*, we specify for each cell its row, column and block number. In *9-Sudoku-hard*, we only specify row and column. For *8-Teams-DRR*, we generate data of eight teams in a double round-robin competition, i.e., $f_a(x, y, z) = 1$ if on match day x team y plays against team z, each team plays twice against each other on 14 match days, and we require symmetry between first and second half of the matches. In *GraphColor*, we generate a random undirected graph with ten nodes and twenty edges, and valid assignments are node colorings where two neighbors have different colors. For *MultipleKnapsack*, we assign twenty items of different weight and value to three knapsacks of limited size. Our last dataset, *Rostering*, contains an instance of a nurse rostering problem[2], where boolean relations differentiate shift types and numerical relations model the start times, end times and durations of shifts. We provide details about all datasets and their ground-truth constraints in the supplementary.

Quality of Discovered Constraints To see how well the discovered constraints match the ground-truth, we generate valid assignments for all datasets and split them into training and test set. For the test set, we additionally generate examples violating the ground-truth constraints. First, we run all methods on the training data. Then, we classify test examples *positive* if they satisfy all found constraints and *negative* otherwise. We report the F_1 score with 1000 training examples in Figure 1. We see that URPILS in contrast to its competitors achieves almost perfect F_1 score on all datasets. On 9-Sudoku-hard, URPILS does not find the block constraint in all runs, but on average still performs best.

Noise Robustness To evaluate noise-robustness, we inject noise into the training data by adding invalid assignments. We report test F_1 score on GraphColor dependent on noise proportion in Figure 2 (left). We see URPILS recovers the ground-truth for up to 60% noise and is on par for higher noise levels. The F_1 score of COUNTOR and COUNTCP drops for significantly less noise to $\frac{2}{3}$, i.e., a model that accepts all test examples with recall 1 and precision 0.5.

We also test noise-robustness on the queens problem. MINEACQ shows much better F_1 score with same training set size for lower dimensional problems. Furthermore, runtime of all methods is lower for smaller problems. To enable many runs, we evaluate on 5-Queens, reducing the problem to five queens on a 5×5 chessboard. We report F_1 score on the test set with 10% noise on the training set dependent on the number of training examples in Figure 2 (right). We see COUNTCP and especially COUNTOR pick up noise and discover bad generalizing constraints. MINEACQ performs better, but needs 800 examples to achieve 100% F_1 score. URPILS is not only robust to noise. It also achieves 100% F_1 score with ten training examples.

[2] http://www.schedulingbenchmarks.org/nrp/

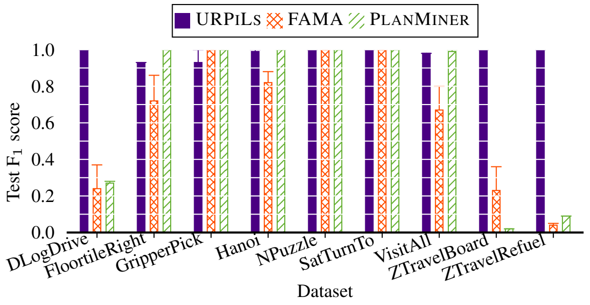

Figure 3: [AI Planning Results] Average F_1 score with standard error on the test sets of AI planning datasets for ten independent runs for URPILS, FAMA, and PLANMINER.

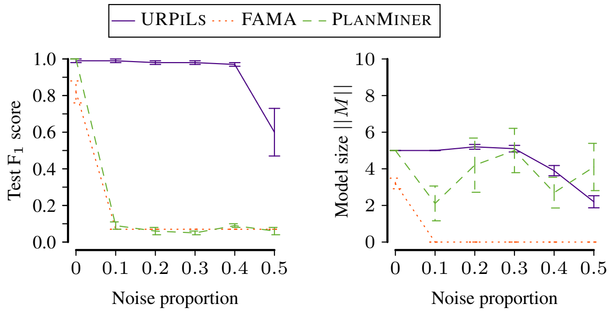

Figure 4: [URPILS is noise-robust on AI planning data] Average test F_1 score over ten independent runs for URPILS, FAMA and PLANMINER under varying noise proportion on the training data (left). Discovered model size under increasing noise (right). Error bars show standard error.

Model Size Across all datasets URPILS finds a compact set of constraints with a total of only 33 to 90 literals of our constraint grammar. COUNTOR and COUNTCP show similar results, but tend to need more constraint terms for equal F_1 score. Since MINEACQ does not look for quantified expressions, it produces sets with 1280 literals for 4x4-Sudoku and 10^6 literals for 8-Teams-DRR. For reference, we show exemplary discovered constraints for MINEACQ, COUNTOR and COUNTCP as well as complete results for model complexity in the supplementary materials.

Experiments on AI Planning Datasets

Finally, we evaluate URPILS on AI planning benchmark datasets (Aineto, Celorrio, and Onaindia 2019) and compare to the state-of-the-art methods FAMA and PLANMINER from related work. Unfortunately, the authors of PLANMINER-N have not published code for their method and did not respond to our emails. As before, we generate a test set for each dataset with valid and invalid executions of an action in the corresponding planning domain. We report the classification F_1 score for each method in Figure 3. We see that URPILS beats the state of the art by a wide margin.

In our last experiment, we evaluate noise-robustness on the Hanoi dataset. We report F_1 score and the number of relations in the discovered constraints for varying noise proportion in Figure 4. If the data contains noise, FAMA does not find any constraints. PLANMINER seems to pick up noise and finds constraints with a poor F_1 score on the test set. In contrast to that, URPILS is very robust to sensible amounts of noise. If the noise level increases, URPILS finds fewer constraints, i.e., it does not find spurious constraints.

Discussion

In our experiments, we empirically show URPILS not only finds more accurate constraints, but also finds more succinct constraints, is more robust to noise, and has lower sample complexity than the state of the art. Nonetheless, URPILS has its limitations, and we see interesting research directions to overcome them. First, despite using a rich modeling language, we cannot model everything. As we see in Figure 1, URPILS does not achieve a 100% F_1 score on MultipleKnapsack, because, with our current constraint language, we cannot model that the sum of the item weights in a knapsack must not exceed its capacity. We need a new type of constraint to model bounds on the sum of numerical relation values. However, we see computing the number of valid assignments for such models is even harder than for count constraints, and thus is a challenging problem. Ideally, we would extend our constraint language to the global constraint catalog (Beldiceanu, Carlsson, and Rampon 2012), which lists a large set of reusable constraints for constraint programming.

Second, the size of a satisfiability problem massively impacts the runtime and sample complexity of URPILS. While URPILS finds all constraints in the majority of runs from 40 examples of 4-Sudoku-hard, it only finds all constraints one out of ten times from 1000 examples of 9-Sudoku-hard. However, the rules of 4×4 and 9×9 Sudoku are basically the same, and many problems have constraints that are independent of the problem size. We, therefore, think it is promising to study how to reduce the size of a given problem as a preprocessing step. Other ways to improve performance on high dimensional problems may include expert knowledge to restrict the large search space of constraints, e.g. by symmetries in the assignments or active learning.

Conclusion

To close the gap between domain experts and mathematical modeling experts in constraint programming and AI planning, we studied the problem of discovering constraints from exemplary solutions. We formalized the problem in terms of the Minimum Description Length (MDL) principle, by which we select the model with the best lossless compression of the data. Since solving the problem involves #P-hard model counting, we proposed the greedy URPILS algorithm to find high-quality constraints in practice. Through extensive experiments on both constraint programming and AI planning benchmark datasets, we empirically showed URPILS not only discovers more accurate constraints, but also finds more succinct constraints, is more robust to noise, and has lower sample complexity than the state of the art. To apply URPILS on more complex problems, potential future work involves extending its modeling language and improving its efficiency on high dimensional problems.

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Journal of Artificial Intelligence (AIJ)*, 275: 104–137.

Arora, A.; Fiorino, H.; Pellier, D.; Métivier, M.; and Pesty, S. 2018. A review of learning planning action models. *The Knowledge Engineering Review*, 33: e20.

Barták, R. 1999. Constraint programming: In pursuit of the holy grail. In *Proceedings of the 8th Annual Conference of Doctoral Students (WDS), Prague, Czech Republic*, 555–564.

Belaid, M.-B.; Belmecheri, N.; Gotlieb, A.; Lazaar, N.; and Spieker, H. 2022. GEQCA: Generic Qualitative Constraint Acquisition. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI), Virtual Event*, 3690–3697.

Beldiceanu, N.; Carlsson, M.; and Rampon, J.-X. 2012. Global constraint catalog, 2nd edition (revision a). Technical report, Swedish Institute of Computer Science.

Bell, J.; and Stevens, B. 2009. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1): 1–31.

Bessiere, C.; Coletta, R.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.-G.; and Walsh, T. 2013. Constraint acquisition via partial queries. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), Beijing, China*, 475–481.

Coulombe, C.; and Quimper, C.-G. 2022. Constraint Acquisition Based on Solution Counting. In *28th International Conference on Principles and Practice of Constraint Programming (CP), Haifa, Israel*.

Crescenzi, P.; and Rossi, G. 2002. On the Hamming distance of constraint satisfaction problems. *Theoretical Computer Science*, 288(1): 85–100.

Dahllöf, V.; Jonsson, P.; and Wahlström, M. 2005. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1-3): 265–291.

Feller, W. 1968. *Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 3 edition.

Fichte, J. K.; Hecher, M.; and Hamiti, F. 2021. The Model Counting Competition 2020. *ACM Journal of Experimental Algorithmics (JEA)*, 26: 1–26.

Grünwald, P. 2007. *The Minimum Description Length Principle*. MIT Press.

Kolb, S.; Paramonov, S.; Guns, T.; and De Raedt, L. 2017. Learning constraints in spreadsheets and tabular data. *Machine Learning*, 106: 1441–1468.

Korhonen, T.; and Järvisalo, M. 2021. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters. In *27th International Conference on Principles and Practice of Constraint Programming (CP), Virtual Event*, 8:1–8:11.

Kumar, M.; Kolb, S.; and Guns, T. 2022. Learning Constraint Programming Models from Data Using Generate-And-Aggregate. In *28th International Conference on Principles and Practice of Constraint Programming (CP), Haifa, Israel*.

Kumar, M.; Kolb, S.; Teso, S.; and De Raedt, L. 2020. Learning MAX-SAT from Contextual Examples for Combinatorial Optimisation. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), New York, NY*, 4493–4500.

Kumar, M.; Teso, S.; De Causmaecker, P.; and De Raedt, L. 2019. Automating personnel rostering by learning constraints using tensors. In *Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI), Portland, OR*, 697–704.

Li, M.; and Vitányi, P. 1993. *An Introduction to Kolmogorov Complexity and its Applications*. Springer.

Marx, A.; and Vreeken, J. 2019. Telling cause from effect by local and global regression. *Knowledge and Information Systems*, 60(3): 1277–1305.

Meng, T.; and Chang, K.-W. 2021. An Integer Linear Programming Framework for Mining Constraints from Data. In *Proceedings of the 38th International Conference on Machine Learning (ICML), Virtual Event*, 7619–7631.

O'Sullivan, B. 2010. Automated modeling and solving in constraint programming. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI), Atlanta, GA*, 1493–1497.

Pawlak, T. P.; and Krawiec, K. 2017. Automatic synthesis of constraints from examples using mixed integer linear programming. *European Journal of Operational Research*, 261(3): 1141–1157.

Picard-Cantin, É.; Bouchard, M.; Quimper, C.-G.; and Sweeney, J. 2016. Learning parameters for the sequence constraint from solutions. In *22nd International Conference on Principles and Practice of Constraint Programming (CP), Toulouse, France*, 405–420.

Pólya, G. 1930. Eine Wahrscheinlichkeitsaufgabe in der Kundenwerbung. *Zeitschrift für Angewandte Mathematik und Mechanik*, 10(1): 96–97.

Prestwich, S. 2021. Unsupervised Constraint Acquisition. In *Proceedings of the 33rd International Conference on Tools with Artificial Intelligence (ICTAI), Virtual Event*, 256–262.

Prestwich, S. D.; Freuder, E. C.; O'Sullivan, B.; and Browne, D. 2021. Classifier-based constraint acquisition. *Annals of Mathematics and Artificial Intelligence*, 89: 655–674.

Rissanen, J. 1978. Modeling by shortest data description. *Automatica*, 14(1): 465–471.

Rissanen, J. 1983. A Universal Prior for Integers and Estimation by Minimum Description Length. *The Annals of Statistics*, 11(2): 416–431.

Sah, A.; Sawhney, M.; Stoner, D.; and Zhao, Y. 2019. The number of independent sets in an irregular graph. *Journal of Combinatorial Theory, Series B*, 138: 172–195.

Segura-Muros, J. Á.; Fernández-Olivares, J.; and Pérez, R. 2021. Learning Numerical Action Models from Noisy Input Data. arXiv:2111.04997.

Segura-Muros, J. Á.; Pérez, R.; and Fernández-Olivares, J. 2021. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, 51(11): 7973–7989.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), Macao, China*, 1169–1176.

Soos, M.; and Meel, K. S. 2019. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI), Honolulu, HI*, 1592–1599.

Tsouros, D. C.; and Stergiou, K. 2020. Efficient multiple constraint acquisition. *Constraints*, 25(3-4): 180–225.

Valiant, L. G. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3): 410–421.

Zhou, J.; Yin, M.; and Zhou, C. 2010. New worst-case upper bound for #2-SAT and #3-SAT with the number of clauses as the parameter. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI), Atlanta, GA*, 217–222.

Appendix

Here, we include supplementary material which could not be part of our main paper.

MDL for Constraint Learning

Here, we prove by counterexamples that our MDL score is neither monotone nor submodular. To this end, we define a dataset with $O_1 = \{o_1^1, \dots, o_{10}^1\}$, $O_2 = \{o_1^2, \dots, o_{10}^2\}$ and $D = \{f_a \mid \sum_{x \in X} f_a(x) = 1\}$. We further define constraints we use in the counterexamples by

$$C_1 = 1 \leq \sum f_a(\cdot) \leq 1$$
$$C_2 = 0 \leq \sum f_a(\cdot) \leq 1$$
$$C_3 = 1 \leq \sum f_a(\cdot) \leq 2.$$

The encoding of these constraints differs in the lower and upper bound only. We compute the encoded length of each constraint as defined by $L(A)$ in the *Model Encoding* Subsection. In our example, we have

$$L(C_1) = \log 3 + \log 8 + \log 7 + L_{\mathbb{N}}(2) + 2 + L_{\mathbb{N}}(2)$$
$$L(C_2) = \log 3 + \log 8 + \log 7 + L_{\mathbb{N}}(1) + 2 + L_{\mathbb{N}}(2)$$
$$L(C_3) = \log 3 + \log 8 + \log 7 + L_{\mathbb{N}}(2) + 2 + L_{\mathbb{N}}(3).$$

Next, we test if $L(D, M)$ is monotone.

Monotonicity A function f is monotone if $\forall T \subseteq S : f(T) \leq f(S)$. We compute

$$L(D, \emptyset) = L_{\mathbb{N}}(1) + 100 \cdot (100 + L_{\mathbb{N}}(1)) \approx 10153$$
$$L(D, \{C_1\}) = L_{\mathbb{N}}(2) + L(C_1) + 100 \cdot (\log 100 + L_{\mathbb{N}}(1)) \approx 833$$
$$L(D, \{C_1, C_2\}) = L_{\mathbb{N}}(3) + L(C_1) + L(C_2) + 100 \cdot (\log 100 + L_{\mathbb{N}}(1)) \approx 847,$$

where we first encode the number of constraints, then the content of the constraints, and for each of the 100 examples in the dataset, we compute the number of bits to select a valid assignment, and $L_{\mathbb{N}}(1)$ encodes the number of errors is 0. By this, we see that $L(D, M)$ is not monotone.

Submodularity A function f is submodular if $\forall X \subseteq \Omega : \forall x_1, x_2 \in \Omega \setminus X : f(x \cup \{x_1\}) + f(x \cup \{x_2\}) \geq f(X \cup \{x_1, x_2\}) + f(X)$. We compute

$$L(D, \{C_1, C_3\}) + L(D, \{C_2, C_3\}) \approx 1698$$
$$L(D, \{C_1, C_2, C_3\}) + L(D, \{C_3\}) \approx 2265,$$

and

$$L(D, \{C_1, C_2\}) + L(D, \{C_1, C_3\}) \approx 1697.1$$
$$L(D, \{C_1, C_2, C_3\}) + L(D, \{C_1\}) \approx 1696.6,$$

by which we see that $L(D, M)$ is not submodular.

The URPILS Algorithm

In this section, we show the pseudocodes for candidate generation in our URPILS algorithm.

Algorithm 3: SIMPLECANDS

```
   input  : dataset $D$
   output: set of candidates $Q$
1  $\mathcal{C}_F^1 \leftarrow \{x \neq y\} \cup \bigcup_{i=1}^{k} \{x_i = y_i \wedge \forall_{j \neq i} x_j \neq y_j\}$;
2  $\mathcal{C}_F^2 \leftarrow \{\epsilon\}$;
3  foreach $f \in \mathcal{F}_{\mathbb{B}}$ do
4      foreach $i \in \{1, \dots, k\}$ do
5          if dom $f = O_i$ then
6              $\mathcal{C}_F^2 \leftarrow \mathcal{C}_F^2 \cup \{f(x_i), \neg f(x_i)\}$;
7              foreach $j \in \{1, \dots, k\}$ do
8                  if dom $f = O_j$ then
9                      $\mathcal{C}_F^2 \leftarrow \mathcal{C}_F^2 \cup \{f(x_i) \wedge f(y_i)\}$;
10 foreach $f \in \mathcal{F}_{\mathbb{R}}$ do
11     foreach $i, j \in \{1, \dots, k\}$ do
12         if dom $f = O_i = O_j$ then
13             foreach $\odot \in \{<, \leq, =, >, \geq\}$ do
14                 $\mathcal{C}_F^2 \leftarrow \mathcal{C}_F^2 \cup \{f(x_i) \odot f(y_i)\}$;
15 $\mathcal{C}_T \leftarrow \{f_a(x) \rightarrow f_a(y), f_a(x) \rightarrow \neg f_a(y)\}$;
16 $Q \leftarrow \emptyset$;
17 foreach $C_F^1, C_F^2, C_T \in \mathcal{C}_F^1 \times \mathcal{C}_F^2 \times \mathcal{C}_T$ do
18     add $\left(\forall x, y \in X \mid C_F^1 \wedge C_F^2 : C_T\right)$ to $Q$;
19 return $Q$;
```

Generating Simple Constraint Candidates We separately create candidates for the filtering part and the target part of our constraints. To simplify generation of the filtering part, we split it into two parts. In the first part, we test equality and inequality between variables, and in the second part, we filter by boolean and numerical relations.

We give the pseudocode of SIMPLECANDS in Algorithm 3. We first initialize the candidates for the first part of C_F (line 1), requiring that x and y differ at all indices or at precisely one index. The candidates for the second part of C_F include an empty constraint (l. 2). We further compare values of boolean relations with domains compatible to the input variables (l. 3–9), and we do the same for numerical relations (l. 10–14). For the target part, we create implications of assignment values (l. 15). We generate simple constraint candidates from the cross product of the candidates for the individual constraint parts (l. 17–18).

Generating Complex Constraint Candidates To generate complex constraint candidates, we reuse concepts from how we create simple constraints. We give the pseudocode for COMPLEXCANDS as Algorithm 4. We first initialize the filtering part, C_F^1, and the target part, C_T, as in SIMPLECANDS (l. 1–2). For the second filtering part, we then create a set of basic filter elements Ω by producing all possible expressions from our grammar for C_F up to a user-defined depth d. In our experiments, we set $d = 3$. In the remainder of the search, we combine basic filter elements to more complex expressions, and hence require these elements to not contain $\wedge$ or $\vee$.

Algorithm 4: ComplexCands

```
input  : model with simple constraints $M$, dataset $D$,
         grammar production depth $d$
output: set of candidates $Q$
$\mathcal{C}_F^1 \leftarrow \{x \neq y\} \cup \bigcup_{i=1}^{k} \{x_i = y_i \wedge \forall_{j \neq i} x_j \neq y_j\}$;
$\mathcal{C}_T \leftarrow \{f_a(x) \rightarrow f_a(y), f_a(x) \rightarrow \neg f_a(y)\}$;
$Q \leftarrow \emptyset$;
foreach $C_F^1, C_T \in \mathcal{C}_F^1 \times \mathcal{C}_T$ do
    $\Omega \leftarrow \left\{ u \mid u \cap \{\wedge, \vee\} = \emptyset \wedge \langle C_F \rangle \overset{d}{\Rightarrow} u \right\}$;
    $C_F^2 \leftarrow \arg\min_{C_F^2 \subset \Omega} L_I(C_F^2 \mid C_F^1, C_T, D, M)$;
    add $\left(\forall x, y \in X \mid C_F^1 \wedge C_F^2 : C_T\right)$ to $Q$;
return $Q$;
```

Our goal is to find a conjunction of basic filter elements from Ω, that best explains which pairs $x, y \in X$ lead to an improvement of the encoded length of the data when a given C_T is added to the model. To find this conjunction C_F^2, we define an MDL score to evaluate candidates. Formally, we find the C_F^2 minimizing

$$\begin{aligned} L_I(C_F^2 \mid C_F^1, C_T, D, M) &= L_{\mathbb{N}}(\left|C_F^2\right|) + \log \binom{|\Omega|}{\left|C_F^2\right|} \\ &+ L_{\mathbb{N}}(1 + \mathit{error}_I(C_F^2 \mid C_F^1, C_T, D, M)) \\ &+ \log \binom{\left|\{x, y \mid C_F^1(x, y)\}\right|}{\mathit{error}_I(C_F^2 \mid C_F^1, C_T, D, M)}, \end{aligned}$$

where we compute the length of the model encoding by the number of elements in the conjunction and an index to select the elements from I. Then, we encode the number of pairs $x, y \in X$, for which $C_F^2(x, y)$ makes a classification error in predicting gain of C_T. We formally define

$$\begin{aligned} &\mathit{error}_I(C_F^2 \mid C_F^1, C_T, D, M) = \\ &\sum_{x, y \in X} C_F^1(x, y) \left| \mathbb{1}_{\Delta L(D \mid M \cup \{C_T\}) < 0}(x, y) - C_F^2(x, y) \right|, \end{aligned}$$

with $\Delta L\left(D \mid M \cup \{C_T\}\right)$ being the gain in the encoded length, after adding C_T to the model. This gives us candidates with complex filtering expressions.

Generating Constraint Candidates for AI Planning We search for constraints telling us when we are not allowed to execute an action. This means we create candidates $\forall x \in X \mid C_F : \neg f_a(x)$, where C_F compares boolean and numerical relations. We give the pseudocode of PlanningCands in Algorithm 5. We start by generating candidates with boolean relations. To generate syntactically valid constraints, we ensure that the parameters of the relation are a subset of the parameters of the assignment function f_a. We generate comparisons of numerical relation using different comparison operators and also ensure that the candidates are syntactically valid constraints.

Example on 4-Sudoku-hard To further simplify understanding what URPiLs is doing in all its steps, we show the generated candidates at each step and the corresponding

Algorithm 5: PlanningCands

```
input  : dataset $D$
output: set of candidates $Q$
$Q \leftarrow \emptyset$;
foreach $u \in \bigcup_{i=1}^{k} \{1, \ldots, k\}^i$ do
    foreach $f \in \mathcal{F}_{\mathbb{B}}$ do
        if dom $f = \prod_{i \in u} O_i$ then
            add $(\forall x \in X \mid f(x[u]) : \neg f_a(x))$ to $Q$;
            add $(\forall x \in X \mid \neg f(x[u]) : \neg f_a(x))$ to $Q$;
foreach $i, j \in \{1, \ldots, k\}$ do
    foreach $f \in \mathcal{F}_{\mathbb{R}}$ do
        if $i \neq j \wedge O_i = O_j =$ dom $f$ then
            foreach $\odot \in \{<, \leq, =, >, \geq\}$ do
                $C_F \leftarrow f(x_i) \odot f(x_j)$;
                add $(\forall x \in X \mid C_F : \neg f_a(x))$ to $Q$;
return $Q$;
```

MDL scores for an exemplary run on 40 exemplary solutions of a 4×4 Sudoku. The search starts with an empty model without any constraints, which has $L(D, M) \approx 2562$.

In the SimpleCands phase, URPiLs creates 46 candidates, i.e., all constraints with at most one relation. Adding 43 of these candidates would lead to an increased MDL score. Therefore, only 3 candidates remain. We select the candidate with the highest compression gain: Adding

$$\begin{aligned} &\forall c, v_1, v_2 \in \text{Cell} \times \text{Value} \times \text{Value} \mid v_1 \neq v_2 : \\ &\quad f_a(c, v_1) \rightarrow \neg f_a(c, v_2) \end{aligned}$$

to the model decreases the MDL score to $L(D, M) \approx 1570$. Next, URPiLs adds the remaining two simple candidates to the model in order of their individual MDL gain. Adding

$$\begin{aligned} &\forall c_1, c_2, v \in \text{Cell} \times \text{Cell} \times \text{Value} \mid \\ &\quad c_1 \neq c_2 \wedge f_y(c_1) = f_y(c_2) : \\ &\quad f_a(c_1, v) \rightarrow \neg f_a(c_2, v) \end{aligned}$$

to the model decreases the MDL score to $L(D, M) \approx 1213$. The last remaining candidate constraint is

$$\begin{aligned} &\forall c_1, c_2, v \in \text{Cell} \times \text{Cell} \times \text{Value} \mid \\ &\quad c_1 \neq c_2 \wedge f_x(c_1) = f_x(c_2) : \\ &\quad f_a(c_1, v) \rightarrow \neg f_a(c_2, v), \end{aligned}$$

which we can merge with the previous constraint to

$$\begin{aligned} &\forall c_1, c_2, v \in \text{Cell} \times \text{Cell} \times \text{Value} \mid \\ &\quad c_1 \neq c_2 \wedge (f_y(c_1) = f_y(c_2) \vee f_x(c_1) = f_x(c_2)) : \\ &\quad f_a(c_1, v) \rightarrow \neg f_a(c_2, v), \end{aligned}$$

which decreases the total encoded cost to $L(D, M) \approx 1015$.

In the ComplexCands phase, URPiLs creates 37 constraint candidates. Most of them increase the MDL score if we add them to the model. Therefore, we reject most of them. Replacing the last constraint in the model by

$$\begin{aligned}&\forall c_1, c_2, v \in \text{Cell} \times \text{Cell} \times \text{Value} \mid \\ &\quad c_1 \neq c_2 \wedge (f_y(c_1) = f_y(c_2) \vee f_x(c_1) = f_x(c_2) \\ &\quad \left\lfloor \frac{f_x(c_1)}{2} \right\rfloor = \left\lfloor \frac{f_x(c_2)}{2} \right\rfloor \wedge \left\lfloor \frac{f_y(c_1)}{2} \right\rfloor = \left\lfloor \frac{f_y(c_2)}{2} \right\rfloor) : \\ &\quad f_a(c_1, v) \rightarrow \neg f_a(c_2, v)\end{aligned}$$

decreases the MDL score to $L(D, M) \approx 1011$.

In the COUNTCANDS phase, URPILS creates two count constraint candidates. Adding

$$1 \leq \forall c \in \text{Cell} : \sum_{v \in \text{Value}} f_a(c, v) \leq 1$$

decreases the MDL score to $L(D, M) \approx 881$, and adding

$$4 \leq \forall v \in \text{Value} : \sum_{c \in \text{Cell}} f_a(c, v) \leq 4$$

decreases the MDL score to $L(D, M) \approx 846$. As no more candidate constraints are left, we terminate the search and return the model. For more details about the rejected generated constraints, we refer to our source code, which provides a more detailed software execution log.

Experiments on Constraint Programming Datasets

We now provide further information about our experiments on constraint programming datasets, which we could not include in the main paper.

Dataset Characteristics We experiment on datasets with different characteristics as we show in Table 1. Most of our datasets consist of two object sets except for 8-Teams-DRR, where we have three object sets. Since every combination of objects $x \in X$ is one parameter of an assignment, we have $2^{|X|}$ possible assignments per dataset. We see that the set of valid assignments for the ground-truth model is usually much smaller. For example, without any constraints, there are 2^{512} ways to position eight queens on a 8×8 chessboard, from which $8! \cdot 92 \approx 2^{22}$ are valid assignments of the ground-truth model (Bell and Stevens 2009). Learning the constraints to describe these 2^{22} from a total of possible 2^{512} assignments appears to be a hard problem.

Ground-Truth Constraints We now show for each dataset in Table 1, how me modeled the dataset and give the ground-truth constraints. Random is a uniformly randomly generated dataset without ground-truth constraints. For 8-Queens, we define one object set $O_1 = \{Q_1, \ldots, Q_8\}$ containing the queens, and one object set $O_2 = \{S_1, \ldots, S_{64}\}$ containing all squares. For each square, we specify a row $f_x : O_2 : \{1, \ldots, 8\}$ and a column $f_y : O_2 : \{1, \ldots, 8\}$. We assign queens to squares, i.e., $f_a(q, s) = 1$ means queen q is on square s. We define the ground-truth constraints by

Dataset	$\lvert X\rvert$	$\lvert\mathcal{F}_{\mathbb{B}}\rvert$	$\lvert\mathcal{F}_{\mathbb{R}}\rvert$	$2^{\lvert X\rvert}$	$\lvert\mathcal{F}_M\rvert$
Random	10×10	4	4	2^{100}	2^{100}
8-Queens	8×64	0	2	2^{512}	$\sim 2^{22}$
4-Sudoku-easy	16×4	0	3	2^{64}	288
4-Sudoku-hard	16×4	0	2	2^{64}	288
9-Sudoku-easy	81×9	0	3	2^{729}	$\sim 2^{73}$
9-Sudoku-hard	81×9	0	2	2^{729}	$\sim 2^{73}$
8-Teams-DRR	$14 \times 8 \times 8$	0	1	2^{896}	$\sim 2^{19}$
GraphColor	10×10	1	0	2^{100}	$\sim 2^{22}$
Rostering	8×168	3	3	2^{1344}	$\sim 2^{31}$

Table 1: [Constraint programming datasets] For all datasets, we show the number of objects per dimension $|X|$ of f_a, number of boolean $|\mathcal{F}_{\mathbb{B}}|$ and numerical $|\mathcal{F}_{\mathbb{R}}|$ relations, total number of possible assignments $2^{|X|}$, and the number of valid assignments $|\mathcal{F}_M|$ for the ground-truth model.

$$\forall q \in O_1 : \sum_{s \in O_2} f_a(q, s) = 1$$

$$\forall s \in O_2 : \sum_{q \in O_1} f_a(q, s) \leq 1$$

$$\begin{aligned}&\forall q_1, q_2, s_1, s_2 \in O_1^2 \times O_2^2 \mid q_1 \neq q_2 \wedge s_1 \neq s_2 \\ &\quad \wedge (f_x(s_1) = f_x(s_2) \vee f_y(s_1) = f_y(s_2) \\ &\quad \vee |f_x(s_1) - f_x(s_2)| = |f_y(s_1) - f_y(s_2)|) \\ &\quad : f_a(q_1, s_1) \rightarrow f_a(q_2, s_2),\end{aligned}$$

where the first constraint ensures that a queen is assigned to exactly one square, the second constraint ensures that at most one queen is assigned to the same square, and the third constraints ensures queens do not attack each other.

In 4-Sudoku-easy, we define one object set $\text{Cell} = \{S_1, \ldots, S_{16}\}$ containing all cells, and one object set $\text{Value} = \{V_1, V_2, V_3, V_4\}$ for the values. We assign values to cells, i.e., $f_a(c, v) = 1$ means that cell c has value v. For each cell, we specify row f_x, column f_y and block f_b. We define the ground-truth constraints by

$$\forall c \in \text{Cell} : \sum_{v \in \text{Value}} f_a(c, v) = 1$$

$$\forall v \in \text{Value} : \sum_{c_1, c_2 \in \text{Cell} \mid f_x(c_1) = f_x(c_2)} f_a(c, v) = 1$$

$$\forall v \in \text{Value} : \sum_{c_1, c_2 \in \text{Cell} \mid f_y(c_1) = f_y(c_2)} f_a(c, v) = 1$$

$$\forall v \in \text{Value} : \sum_{c_1, c_2 \in \text{Cell} \mid f_b(c_1) = f_b(c_2)} f_a(c, v) = 1,$$

i.e., we ensure each cell gets assigned one value, and we require uniqueness of values in each row, cell and block. In 4-Sudoku-hard, we do not have the block information and must replace the last ground-truth constraint. Instead of comparing f_b, we define

$$\left\lfloor \frac{f_x(c_1)}{2} \right\rfloor = \left\lfloor \frac{f_x(c_2)}{2} \right\rfloor \wedge \left\lfloor \frac{f_y(c_1)}{2} \right\rfloor = \left\lfloor \frac{f_y(c_2)}{2} \right\rfloor$$

which is equivalent to $f_b(c_1) = f_b(c_2)$. We accordingly define object sets and ground-truth constraints for 9-Sudoku-easy and 9-Sudoku-hard.

For 8-Teams-DRR, we define an object set $O_1 = \{M_1, \ldots, M_14\}$ with match days, and an object set $O_2 = \{T_1, \ldots, T_8\}$ for teams. Each match day is assigned a unique number $f_m : O_1 \rightarrow \{1, \ldots, 14\}$. We assign teams and match days, i.e., $f_a(m, t_1, t_2) = 1$ means on match day m, team t_1 plays against t_2. We define the ground truth by

$$\begin{aligned}
&\forall m, t \in O_1 \times O_2 : \neg f_a(m, t, t) \\
&\forall t_1, t_2 \in O_2 \mid t_1 \neq t_2 : \sum_{m \in O_1} f_a(m, t_1, t_2) = 1 \\
&\forall m_1, m_2, t_1, t_2 \in O_1^2 \times O_2^2 \mid \\
&\quad |f_m(m_1) - f_m(m_2)| = 7 : \sum_{m \in O_1} f_a(m, t_1, t_2) = 1,
\end{aligned}$$

where the first constraint ensures no team plays against itself, the second constraint ensures each team plays against each other twice, and the third constraint ensures symmetry of home and away team between first and second half of the match days.

For the GraphColor dataset, we define an object set for nodes $O_1 = \{N_1, \ldots, N_10\}$ and for colors $O_1 = \{C_1, \ldots, C_10\}$. We further define a boolean relation $f_n : O_1^2 \rightarrow \{0, 1\}$ with $f_n(n_1, n_2) = 1$ if there is an edge between nodes n_1 and n_2. We assign colors to nodes, i.e., $f_a(n, c) = 1$ if node n has color c. Additionally, we define the ground-truth constraints by

$$\begin{aligned}
&\forall n \in O_1 : \sum_{c \in O_2} f_a(n, c) = 1 \\
&\forall n_1, n_2, c \in O_1^2 \times O_2 \mid f_n(n_1, n_2) : \\
&\quad f_a(n_1, c) \rightarrow \neg f_a(n_2, c),
\end{aligned}$$

i.e., every node is assigned one color and two neighbored nodes must not have the same color.

For the Rostering dataset, we define an object set for employees $O_1 = \{E_1, \ldots, E_8\}$ and an object set for shifts $O_2 = \{S_1, \ldots, S_168\}$. One boolean relation f_e marks whether a shift is of type early shift, and another boolean f_l marks late shifts. We mark optional shifts by the boolean relation f_o. For every shift, we give the relative start time in hours by the numerical relation f_s, the relative finish time in hours f_f, and the duration in hours f_d, where relative means that the first shift starts at time 0. We assign employees to shifts, i.e., $f_a(e, s) = 1$ if employee works on shift s. Our ground-truth model is

$$\begin{aligned}
&\forall e \in O_1 : 160 \leq \sum_{s \in O_2} f_a(e, s) f_d(s) \leq 168 \\
&\forall s \in O_1 \mid f_o(s) : 0 \leq \sum_{e \in O_1} f_a(e, s) \leq 1 \\
&\forall s \in O_1 \mid \neg f_o(s) : \sum_{e \in O_1} f_a(e, s) = 1 \\
&\forall s_1, s_2 \in O_1 \mid f_s(s_1) = f_s(s_2) : f_a(e, s_1) \rightarrow \neg f_a(e, s_2),
\end{aligned}$$

i.e., we define a minimal and maximal workload for employees, every optional shift can be assigned to one employee, every mandatory shift must be assigned to one employee, and an employee must not be assigned to parallel shifts.

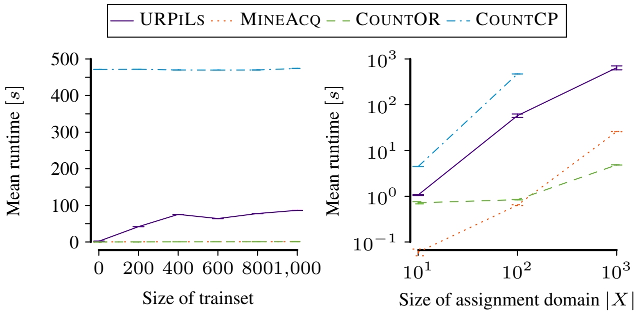

Figure 5: [Runtime on Random dataset] We show mean runtime in seconds for URPiLs, MineAcq, CountOR and CountCP on the Random dataset for $|X| = 100$ and varying number of training examples (left). We show mean runtime in seconds for all methods on the random dataset for 1000 examples and varying $|X|$ (right). CountCP did not finish within 12 hours for $|X| = 10^3$.

Runtime We report wall-clock running times for single-threaded execution of all methods on all datasets in Table 2. We see that URPiLs is not the fastest but still shows reasonable runtime on all datasets. In particular, it is significantly faster than CountCP. CountOR and MineAcq are magnitudes faster than URPiLs, but CountOR cannot deal with noise in the data and MineAcq produces models with significantly more constraint terms and needs more examples to discover constraints.

To evaluate runtime scaling behavior of all methods, we use the Random dataset and vary the number of training examples and the size of the assignment function domain $|X|$ as a proxy for problem size in Figure 5. We see that URPiLs shows a linear runtime behavior in the number of training examples, while the other methods work on a compressed representation of a dataset and thus remain relatively constant in their runtime. Furthermore, we see that the size of the assignment function domain has a high impact on the runtime of all methods. Since $|X| = \prod_{i=1}^{k} |O_i|$, adding a few objects to the problem can lead to a significant growth in runtime. Furthermore, the size of the space of possible assignments increases significantly, which makes it harder to discover the right constraints, and we need more examples for constraint discovery. We observe all these effects when comparing 4×4 and 9×9 Sudoku in Table 2. Therefore, future work should examine how to reduce the size of a given problem.

Discovered Constraints for 4-Sudoku-hard We show the constraints found by URPiLs, MineAcq, CountOR and CountCP on the 4-Sudoku-hard dataset in Figure 6. We see URPiLs produces a succinct constraint set, which matches the ground truth and URPiLs expresses the block constraint by the available row and column relation.

				URPILS		MINEACQ		COUNTOR		COUNTCP	
Dataset	$\|X\|$	$\|\mathcal{F}_\mathbb{B}\|$	$\|\mathcal{F}_\mathbb{R}\|$	$\|\|M\|\|$	t [s]	$\|\|M\|\|$	t [s]	$\|\|M\|\|$	t [s]	$\|\|M\|\|$	t [s]
Random	10×10	4	4	0	87	0	1	136	1	51	473
8-Queens	8×64	0	2	52	1543	10^5	8	46	2	90	12350
4-Sudoku-easy	16×4	0	3	40	8	1280	1	87	1	58	192
4-Sudoku-hard	16×4	0	2	45	9	1280	1	66	1	48	195
9-Sudoku-easy	81×9	0	3	40	8107	50674	13	87	3	58	24931
9-Sudoku-hard	81×9	0	2	40	3735	50774	12	66	2	48	25217
8-Teams-DRR	$14 \times 8 \times 8$	0	1	90	950	10^6	19	72	1	44	38425
GraphColor	10×10	1	0	33	14	30162	2	18	1	28	471
Rostering	8×168	3	3	78	5930	10^6	46	81	14	83	87459

Table 2: [Model size and runtime on constraint programming datasets] We show for each dataset the cardinality $|X|$ of the assignment function f_a and the number of boolean $|\mathcal{F}_\mathbb{B}|$ and numerical relations $|\mathcal{F}_\mathbb{R}|$. For each of the datasets, we report the number of constraint terms $||M||$ and the average discovery runtime over ten runs in seconds t for URPILS, MINEACQ, COUNTOR and COUNTCP.

URPILS

$$\forall c, v_1, v_2 \in \text{Cell} \times \text{Value} \times \text{Value} \mid v_1 \neq v_2 : f_a(c, v_1) \rightarrow \neg f_a(c, v_2)$$

$$\forall c_1, c_2, v \in \text{Cell} \times \text{Cell} \times \text{Value} \mid c_1 \neq c_2 \wedge (f_x(c_1) = f_x(c_2) \vee f_y(c_1) = f_y(c_2) \vee \left\lfloor \frac{f_x(c_1)}{2} \right\rfloor = \left\lfloor \frac{f_x(c_2)}{2} \right\rfloor \wedge \left\lfloor \frac{f_y(c_1)}{2} \right\rfloor = \left\lfloor \frac{f_y(c_2)}{2} \right\rfloor) : f_a(c_1, v) \rightarrow \neg f_a(c_2, v)$$

$$\forall c \in \text{Cell} : \textstyle\sum_{v \in \text{Value}} f_a(c, v) = 1$$

MINEACQ

$$f_a(c_{00}, 1) \rightarrow \neg f_a(c_{00}, 2)$$

$$f_a(c_{00}, 1) \rightarrow \neg f_a(c_{01}, 1)$$

$$\dots$$

$$f_a(c_{33}, 3) \rightarrow \neg f_a(c_{33}, 4)$$

COUNTOR

$$\forall c \in \text{Cell} : \textstyle\sum_{v \in \text{Value}} f_a(c, v) = 1$$

$$\forall v \in \text{Value} : \textstyle\sum_{c \in \text{Cell}} f_a(c, v) = 4$$

$$\forall c \in \text{Cell} : \textstyle\sum_{v \in \text{Value} \mid f_a(c,v)} f_x(c) \leq 3$$

$$\forall c \in \text{Cell} : \textstyle\sum_{v \in \text{Value} \mid f_a(c,v)} f_y(c) \leq 3$$

$$\forall v \in \text{Value} : \textstyle\sum_{c \in \text{Cell} \mid f_a(c,v)} f_x(c) = 6$$

$$\forall v \in \text{Value} : \textstyle\sum_{c \in \text{Cell} \mid f_a(c,v)} f_y(c) = 6$$

COUNTCP

$$\forall c, v_1, v_2 \in \text{Cell} \times \text{Value} \times \text{Value} \mid v_1 \neq v_2 : f_a(c, v_1) \rightarrow \neg f_a(c, v_2)$$

$$\textstyle\sum_{c, v \in \text{Cell} \times \text{Value}} f_a(c, v) = 16$$

$$\forall c \in \text{Cell} : \textstyle\sum_{v \in \text{Value}} f_a(c, v) = 1$$

$$\forall v \in \text{Value} : \textstyle\sum_{c \in \text{Cell}} f_a(c, v) = 4$$

$$\forall x \in \text{dom } f_x : \textstyle\sum_{c, v \in \text{Cell} \times \text{Value} \mid f_x(c) = x} f_a(c, v) = 4$$

$$\forall y \in \text{dom } f_y : \textstyle\sum_{c, v \in \text{Cell} \times \text{Value} \mid f_y(c) = y} f_a(c, v) = 4$$

Figure 6: [Discovered Sudoku models] We show the constraints found by URPILS, MINEACQ, COUNTOR and COUNTCP on the 4-Sudoku-hard dataset. We show variable definitions in orange, relation comparisons in blue and constraints on f_a in black.

MINEACQ finds pairwise implications between assignment values. This means, MINEACQ can rediscover the ground truth, but needs many pairwise implications to describe the data, which is hard to read for domain experts. COUNTOR and COUNTCP do not find the block constraint.