What Are the Rules? Discovering Constraints from Data

Boris Wiegand1,2, Dietrich Klakow2, Jilles Vreeken3
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; Tsouras 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, Celorio, and Onaíndia 2019; Segura-Muros, Pérez, and Fernández-Oliva 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. 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 $O_1, \ldots, O_k$ 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 $8 \times 8$ chess board, such that no two queens attack each other. We define an object set $O_1 = \{Q_1, \ldots, Q_8\}$ for queens, and an object set $O_2 = \{S_1, \ldots, S_{64}\}$ for squares on the board. An assignment is a boolean function $f_a : X \rightarrow \{0, 1\}$, e.g., $f_a(Q_1, S_{32}) = 1$ means queen $Q_1$ is on square $S_{32}$. We call $f_a$ a valid assignment, if it satisfies a set of constraints, i.e., a model $M = \{C_1, \ldots, C_m\}$, with $C_i : X \rightarrow \{0, 1\}$ 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 $F_M$. We define constraints by a boolean algebra over the assignment $f_a$, a set of boolean relations between objects $F_B = \{f_1, \ldots, f_j\}$ with $f_j : \prod_{i=1}^k O_j \rightarrow \{0, 1\}$, and arithmetic expressions over a set of numeric relations $F_X = \{f_1, \ldots, f_j\}$ with $f_j : \prod_{i=1}^k O_j \rightarrow \mathbb{R}$.

In the 8-Queens example, we assign rows and columns to squares. Formally, we define $F_B = \{f_1, f_2\}$ with $f_1 : O_2 \rightarrow \{0, 1, \ldots, 8\}$ and $f_2 : O_2 \rightarrow \{1, \ldots, 8\}$. The constraint that no more than one queen may be placed in a row can then be written as $\forall(q_1, q_2, s_1, s_2) \in O_1 \times O_1 \times O_2 \times O_2 : s_1 \neq s_2 \land f_1(s_1) = f_2(s_2) \Rightarrow \neg f_1(q_1, s_1) \lor \neg f_2(q_2, s_2)$. Our goal is to find constraints like these from a dataset of exemplar valid assignments $D = \{f_a, \ldots, f_a\}$.

**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 $M$, the best model is defined by $\arg \min_{M \in M} L(M) + L(D \mid M)$, 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 $|F_M| = 2^{|X|}$ 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

$$\text{error}(M \mid f_a) = \min_{f_a \in F_M} \sum_{x \in X} I(f_a(x) \neq f_a(x)) \cdot x.$$

To encode the errors, we first specify their number by the MDL-optimal encoding for integers $z \geq 1$ (Rissanen 1983) defined as $L_B(z) = \log c_0 + \log z + \log \log z + \ldots$ 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 $\text{error}(M \mid f_a)$ out of $|X|$ values. In summary, we have

$$L(D \mid M) = \log |F_M| + L_B(1 + \text{error}(M \mid f_a)) + \log |X| \cdot \text{error}(M \mid f_a).$$

This gives us a lossless encoding of the data.

**Model Encoding**

Next, we compute the length of the model encoding by

$$L(M) = L_B(|M| + 1) + \sum_{C \in M} L(C),$$

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

$$C \rightarrow \langle C_V \rangle \mid \langle \text{NE} \rangle \mid \langle C_T \rangle \mid \langle C_F \rangle \mid \langle C_R \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle$$

$$C_V \rightarrow \epsilon \mid \forall x \in X \mid \forall x, y \in X$$

$$C_F \rightarrow \epsilon \mid \langle \nu \rangle \mid \langle \nu \rangle \mid \langle \nu \rangle \neq \langle \nu \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle$$

$$C_T \rightarrow \epsilon \mid \langle \nu \rangle \mid \langle \nu \rangle \mid \langle \nu \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle \mid \langle \text{NE} \rangle$$

$$f_R \rightarrow \text{one of } F_R \mid \circ \rightarrow + \mid - \mid \cdot$$

$$C_T \rightarrow f_a(x) \mid f_a(y) \mid f_a(X_{(j)}) \mid \neg \langle C_T \rangle \mid \langle C_T \rangle \wedge \langle C_T \rangle \mid \langle C_T \rangle \vee \langle C_T \rangle \mid \langle \text{COUNT} \rangle$$

$$j \rightarrow 1 \mid \ldots \mid |X|$$

$$\text{COUNT} \rightarrow \langle \text{NE} \rangle \leq \sum_{(\nu)} f_a(x) \leq \langle \text{NE} \rangle.$$
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 numerical 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 operators. In many real-world problems, we can distribute some

Formal Problem Definition

The Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI-24)

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), $SHARP\text{-}TD$ (Korhonen and Järvisalo 2021) or APPROXMC (Soos and Meel 2019) to tackle the problem. Depending on the complexity of the formula, these approaches take seconds, minutes or even hours (Fichte, Hecher, and Hamiteli 2021), which is too slow for evaluating many constraint candidates during search.

The $URP1Ls$ 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.

The URP

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 $f_a^* \mid M$ 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).

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 $-f_a(\cdot)$, 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 constraints $\sum_{i=1}^{6} x_i = 3$.

Then, the CNF of this constraint is $\prod_{i=1}^{4} \prod_{j=1}^{5} \prod_{k=1}^{6} (x_i + x_j + x_k)$, 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 $\binom{b}{a}$.

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.,

$$|F_M| = \left[ \frac{\sum_{C \in M} |F_C|}{|M|^2} \right].$$

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

**Estimating the Best Fitting Valid Assignment**

To compute $L(f_a | M)$, we also need to compute $\text{error}(M | f_a)$, 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 computing the number of valid assignments, enumerating all assignments to find $\text{error}(M | f_a)$ is intractable.

In contrast to $\text{error}(M | f_a)$, 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 $\text{error}(M | f_a)$. We estimate the number of variables we must flip to satisfy the formula by using the coupon collector’s problem (Polya 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

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

The value of $\text{error}(M | f_a)$ 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 $L(D | M)$ 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 $\forall x, y \in X | C_F : f_a$. In $C_F$, we compare the values of at most one boolean and numerical relation, e.g. $f(x_1) < f(y_1)$ with $f \in F_R$. 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 $C_F$. For further reference, we provide pseudocode for SIMPLECANDS in the supplementary.

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 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 $\forall x, y \in X | g(x_1) < g(y_1) : f_a(x) \rightarrow \neg f_a(y)$ and $\forall x, y \in X | h(x_2) = h(y_2) : f_a(x) \rightarrow \neg f_a(y)$ into $\forall x, y \in X | g(x_1) < g(y_1) \lor h(x_2) = h(y_2) : f_a(x) \rightarrow \neg f_a(y)$. 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 $L(D | M)$. 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
grammars. 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 URPLs, 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 \( f \in F_B \) 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.

**URPLs for AI Planning**

Next we show how to adapt URPLs 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 \( f^a_j \) to refer to relation \( f_j \) 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 precedences 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 frequent 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

\[
L_{AI}(D \mid M) = \sum_{i=1}^{|D|} - \log \left( \frac{\text{usg}_a f^a_i \in F_M + \epsilon}{\text{usg}_a 0' + \text{usg}_a 1' + 2\epsilon} \right)
\]

\[
\begin{cases} 
\log |F_M|, & \text{if } f^a_i \in F_M \\
\log(|X| - |F_M|), & \text{otherwise,}
\end{cases}
\]

where usg\(_a\) \( x \) is how often code \( x \) has been used up to the \( i \)-th assignment, and \( \epsilon \) 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; Tsouras 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 (Ainet, Celorio, and Onaindia 2019) formalizes the problem as a planning problem itself. PLANMINEr (Segura-Muros, P´erez, and Fern´andez-Olives 2021) translates the problem to a rule-based classification task, and PLANMINEr-N (Segura-Muros, Fernández-Olives, and P´erez 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, URPLs 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 URPLs 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

1https://eda.rg.cispa.io/prj/urpils
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 $\tau$ and $\rho$ 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., $(x, y, z) = 1$ 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 Graph-Color, 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 $\frac{2}{5}$, 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 $5 \times 5$ 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 F1 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 F1 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 F1 score. Since MINEACQ does not look for quantified expressions, it produces sets with 1280 literals for 4x4-Sudoku and 10b 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, Celorio, and Oaindia 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 F1 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 F1 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 F1 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% F1 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


