Learning Constraint Programming Models from Data using Generate-and-Aggregate

Mohit Kumar ✉
KU Leuven, Belgium

Samuel Kolb ✉
KU Leuven, Belgium

Tias Guns ✉
KU Leuven, Belgium

Abstract

Constraint programming (CP) is used widely for solving real-world problems. However, designing these models require substantial expertise. In this paper, we tackle this problem by synthesizing models automatically from past solutions. We introduce COUNT-CP, which uses simple grammars and a generate-and-aggregate approach to learn expressive first-order constraints typically used in CP as well as their parameters from data. The learned constraints generalize across instances over different sizes and can be used to solve unseen instances – e.g., learning constraints from a 4×4 Sudoku to solve a 9×9 Sudoku or learning nurse staffing requirements across hospitals. COUNT-CP is implemented using the CPMpy constraint programming and modelling environment to produce constraints with nested mathematical expressions. The method is empirically evaluated on a set of suitable benchmark problems and shows to learn accurate and compact models quickly.

2012 ACM Subject Classification Applied computing → Operations research

Keywords and phrases Constraint Learning, Constraint Programming, Model Synthesis

Digital Object Identifier 10.4230/LIPIcs.CP.2022.32

Funding This project was partially funded by the Research Foundation - Flanders (FWO) project G0G3220N and FWO-S007318N, as well as the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant No. 101002802: CHAT-Opt, and 694980: SYNTH). Samuel Kolb is supported by the "Agentschap Innoveren & Ondernemen" (VLAIO) as part of the innovation mandate HBC.2021.0246.

1 Introduction

Constraints play an important role in modelling many real-world decision problems. They are used widely in fields like cryptography [13, 15], complexity theory [1] and automatic theorem proving [14]. However, identifying the constraints of a problem and encoding them into a mathematical model requires both domain knowledge and modelling expertise. This non-trivial task is often the major bottleneck for the widespread application of constraint-based methods and solvers.

Consider, for instance, the case of scheduling nurses in a hospital, where the aim is to create a schedule for nurses every week. Modeling this problem requires domain knowledge to identify relevant constraints, such as, *every shift requires at least three nurses* or *nurses may work at most five days a week*. Next, these constraints have to be encoded as a mathematical model, e.g., a Constraint Satisfaction Problem (CSP). The skills required to achieve both these steps makes powerful techniques for efficiently solving such problems inaccessible to people without a mathematical or computer science background.

Constraint learning approaches aim to overcome this issue by instead learning constraint models from past solutions [19]. In the example of nurse scheduling, this means learning the constraint model from manually created past schedules. By automating the modeling step, constraint learning makes constraint solving techniques more accessible and makes the modeling process faster and cheaper. In the CP community there are a number of existing approaches to learn constraints from solutions [10, 2, 4] and – in some cases – non-solutions [17, 18]. A popular approach to learning constraint is the so called generate-and-test approach [21]. The idea behind generate-and-test is to generate candidate constraints and apply them to various subsets of the decision variables and test whether the constraint holds in the training data.

Most of these approaches learn constraints at the level that a constraint solver accepts: individual constraints, such as predicates with a fixed number of arguments. By listing all possible predicates and their signature, different predicate/variable combinations can be generated and tested. Learning more expressive constraints, however, often requires generating prohibitively large combinations of predicates and makes constraint learning very time-consuming. As a result several approaches design their constraint space instead as a flat catalog of more expressive candidate constraints, learning global constraints [2] or relational spreadsheet formulas [10]. By modeling constraints using an expressive, richer language, rather than acquiring individual lower-level constraints, these approaches are able to synthesize high quality models quickly. The key limitations are that only constraints from these catalogs can be learned and that parameters can only be inferred using constraint-specific parameter inference methods.

A recent approach (COUNT-OR [11]) to learning constraints for OR models, such as nurse scheduling problems, offered an alternative: Using a simple grammar of aggregation operators, different aggregation expressions are generated and applied to various slices of matrices and in general tensors of decision variables. By simply computing the lower and upper bounds of these expressions across all training examples, the method automatically identifies relevant parameters from data and learns constraints quickly.

We built on this approach to design a constraint learner (COUNT-CP) that applies the ideas of bounded expressions to learn CP constraints. First, we observe that constraint models over finite domain integers usually consist of Boolean expressions and numeric expressions with a comparison (e.g., `x = y` and `x <= y`). Because Boolean expressions in this context can be seen as a special case of numeric expressions that are equal to 1, we can use suitable bounded numeric expressions `lb <= expr <= ub` to express common types of constraints (e.g., `abs(x-y) <= 0` and `x-y <= 0`). Second, we observe that first-order constraints such as *each nurse works at most 5 days a week* or global constraints such as `alldifferent` can be decomposed into multiple bounded-expression constraints.

Based on these observations, we suggest to learn bounded-expressions using a COUNT-OR style approach which offers an obvious mechanism to infer the constants. COUNT-CP uses a simple grammar to generate suitable nested mathematical expressions and computes their lower and upper bounds. However, to produce expressive, first-order constraints, the learned bounded-expression constraints are then grouped together over structured sets of variables using simple grammars of `foreach` statements. This step also serves as an inductive bias in selecting which constraints should make up the final learned model. As a result, COUNT-CP is able to assemble first-order constraints, such as *each nurse works at most 5 days a week* and global constraints such as `alldifferent`. COUNT-CP allows users to provide background knowledge in the form of sets of variables that the user considers related – e.g., connected edges in a graph or shared skill levels of nurses – and adds these sets to the grammar used for grouping constraints.

Our developments have been inspired by the PTHG-21 Holy Grail Challenge, which contains variable-sized problem instances over the integer domain, and where a preliminary version of our approach was the winning (and only) entry. In principle, first-order constraints are independent of the instance size and can be used to solve different instances of unseen sizes. However, the numeric constants fitted by approaches such as COUNT-OR may be instance size dependent and would not apply to unseen instances. To resolve this issue, we propose to fit a symbolic bound expression across training instances, using generic problem features as well as semantic constants provided by a domain expert – e.g., minimal staffing requirements of a hospital.

To summarize, the key contributions of this paper are:

- Learning first-order bounded-expression constraints, which are expressive enough to capture many complicated constraints *and* can be learned using a simple and fast generate-and-aggregate procedure.
- Defining the language bias for constraints using simple and simple-to-extend grammars that are combined to learn intricate constraints.
- Replacing constants in the learned constraints by symbolic expressions, which allows learned model to generalise to different and unseen problem sizes.
- Allowing users to provide background knowledge using a simple interaction protocol: sets of related variables and semantic instance-level constants.
- Providing an effective strategy for removing redundant constraints to improve the interpretability and speed of learned models.

This paper is structured as follows: First, we review related work on constraint learning (Section 2). Second, we present our constraint learning approach (COUNT-CP, Section 3) by discussing its links to COUNT-OR, how it learns propositional constraints, first-order constraints and how we filter out constraints to produce compact models. Third, we empirically evaluate our approach on a set of suitable benchmark problems (Section 4). Fourth, and finally we summarize our conclusions (Section 5).

2 Related Work

Learning constraints from a given set of feasible examples has a long history. The first algorithm in this regard was given by Valiant [21], back in 1984. Given a set of feasible examples, this algorithm learns Boolean formulas consistent with the given examples. To do so, it enumerates all possible formulas upto a pre-defined complexity and keeps only those which are satisfied by all feasible examples. This is essentially a generate-and-test approach, where the algorithm generates all possible constraints and then tests whether they hold on the given dataset. This approach was later extended to first order logic under the banner of inductive logic programming [8]. Although important, these early works are limited to Boolean variables and logical formulas.

More recent works, like the series of work by Bessiere et al [4, 5, 6] extend these approaches to integer variables. For instance, Conacq [4] learns constraints, typically using the basic comparison relations $(=, <, \leq, \geq, >, \neq)$. The relations considered is called the *bias*. It basically searches for such constraints over every compatible subset of variables (called *scope*, e.g. all pairs) and defines a lattice structure of the comparison relations, based on the generalisation/specialisation relation between them. Feasible examples are used to remove relations from the lattice. Infeasible examples only say that there has to exist one constraint over all possible variable combinations that is violated; which is expressed as a meta constraint. The authors use the concept of *convergence* to denote if a lattice for a pair of variables only contains a single relation; if not, the default action is to take the most specific relation as constraints, e.g. in case of $\geq, \leq$ and $=$, $=$ is taken. The concept of the lattice comes from the version space algorithm where a lattice is defined over the whole program space. When considering a separate lattice for every variable pair, arguably its main purpose may be to identify which relations are redundant, e.g. subsumed by others. None of these works can learn the bounds in the data, let alone symbolic bounds. This work was later extended to learn generalized constraints [3], however, the assumption here was that the user knows which variables are supposed to be grouped together. It was further extended to detect the groups automatically in [7].

Another well known approach is ModelSeeker [2], which is also a generate-and-test approach; it does not just consider basic comparison relations, but a subset of all global constraints in the global constraint catalog. Furthermore, it does not search over all subsets of variables, but instead has candidate *generator* expressions that uses the structure of the decision variables (e.g. a list or matrix) to group the variables into meaningful subsets (e.g. per row, per column, all pairs). It then generates and tests which constraints are satisfied in each of the groups, over all positive instances. The use of 'generators' matches the programming style of 'foreach s in ...: constraint(s)' that CP modellers often use. If the constraint has additional parameters, then these need to be inferred separately using custom rules for every constraint. However, ModelSeeker requires the constraint catalog to be provided beforehand and does not learn symbolic parameters/bounds.

Finally the CPS [12] approach uses inductive logic programming (ILP) to find logic programming rules of the form $\mathit{condition} \implies \mathit{constraint}$. The condition can be seen as a *generator* too, for the learned logical rules have to be flattened into low level constraints for every possible substitution of the condition.

Our proposed approach does not go as far as ModelSeeker in considering a bias of a wide range of generic to very specialized global constraints. We do go beyond simple comparison relations between pairs of variables, by considering a comparison relation on a mathematical expression over variables. We use a grammar to generate the possible mathematical expressions. This captures the basic comparison relations, but also captures linear (unweighted) expressions and the use of constants such as in $x + y \geq 2$. This approach of mathematical expressions turned into constraints by identifying bounds on them, also generalizes well to the use of *generator* expressions.

A recent approach used for learning hard constraints given a set of feasible and infeasible examples is to encode the learning problem itself as a mathematical model [10, 17]. For instance, Pawlak et al. [17] learn constraints with linear, quadratic and trigonometric terms by encoding the learning as a MILP. The hard constraints of this MILP ensure that each example is correctly classified by the learnt model, while the objective tries to minimise the complexity of the learnt model. This ensures that the learnt model is concise and easily readable. This work was later extended to work with positive only training instances [16], where the idea is to fit a Gaussian mixture model on the given set of feasible points and use this model to sample infeasible points, the learning strategy then uses both feasible and the sampled infeasible instances to learn a model. The main drawback of these methods is that they do not learn symbolic expressions, and thus can not generalise to unseen problem sizes.

3 Learning constraints using COUNT-CP

3.1 Background on COUNT-OR.

Our approach for learning constraints is inspired by COUNT-OR [11], a constraint learning approach for acquiring personnel rostering constraints from example schedules. Instead of generating and testing a set of possible constraints, COUNT-OR instead generates a set of *expressions* that capture useful quantities in the data by applying aggregates to various slices of matrices and tensors. We use the term tensor for a multi-dimensional matrix, basically a matrix is a 2D tensor and a tensor can have more than 2 dimensions.

▶ **Example 1.** Consider a Boolean matrix of nurses (rows) and days (columns) that encodes whether a given nurse works on a given day (1) or not (0). Summing over values in a row of such a weekly schedule expresses the number of working days for a nurse.

By comparing the values of these expressions across different example schedules, COUNT-OR finds upper and lower bounds for every expression. Together, these bounds and expressions can be translated to constraints. For example, if the number of working days of nurse i in the example schedules is always between 0 and 5, COUNT-OR produces a constraint

```
0 <= sum(X[i, :]) <= 5
```

COUNT-OR also compares values across tensor slices, e.g., comparing working days for different nurses, to find generalized constraints, such as

```
foreach i: 0 <= sum(X[i, :]) <= 6
```

These generalized constraints can then be applied to schedules with different numbers of nurses.

We will adopt a similar strategy for learning CP models, however, we use specialized grammars to generating different types of expressions and to find slices that can be used for generalized constraints. Given the conceptual similarity to COUNT-OR, we call our approach COUNT-CP. First, we will explain how COUNT-CP generates *propositional* expressions for fixed problem sizes, that is, individual constraints that involve specific subsets of variables (their scope) and a relation between those variables. Second, we describe how to group these propositional expressions to find *generalized* constraints that can also be carried over to unknown problem sizes.

3.2 Learning propositional constraints

In this work, we consider the case of learning constraints of the following form:

```
lb <= expr <= ub
```

where `expr` is a mathematical expression over variables, such as `X[i] + X[j]`, or an aggregate expression over a group of variables, such as `sum(X[:])`. To learn these constraints, we defined a grammar that captures expressions frequently occurring in CP problems, and a mechanism to find suitable lower and upper bounds.

Our approach is different from current constraint learning approach in that our *bias*, the set of possible constraints that can be learned, is not determined by a fixed set of constraints (whose parameters might have to be infererd later). Rather, our bias consists of mathematical expressions on the one hand, and bound-constraints on these expressions on the other hand.

Expression grammar

To construct our grammar, we look at unary, binary and aggregate expressions that can be expressed in CP modelling languages such as MiniZinc and CPMpy. We consider the unary *identity* expression, the binary expressions *addition*, *subtraction* and *absolute difference*, and the aggregate *sum* expression.

Observe how this grammar does not include the 'traditional' constraint biases `x != y`, `x <= y`, `x < y`, etc. The reason is that we have constraints that subsume those, namely `abs(x - y) >= 1`, `x - y <= 0` and `x - y <= -1`, respectively. Hence, our constraint bias – inequalities over the expression grammar – can learn those traditional constraints. But it can also learn other constraints, like `abs(x - y) > 2`, as the bounds are automatically determined and not sequentially tested based on a predetermined list.

One of the few unary/binary constraints it can not learn is `x != c`, for some constant `c` which lies between (exclusive) the lower and upperbound of `x`. We believe it will be very rare that a constraint model intentionally excludes one individual value, without that value being specified as the 'input data' (more on this later). Hence, it is not part of our bias.

The bias also does not include n-ary global constraints such as `alldifferent()` or `increasing()`, however, these have decompositions into binary constraints, meaning that we can learn the decomposed versions. In Subsection 3.3 we will see how to group these decomposed constraints into generalized constraints that recreate such global constraints.

Also currently not included are tertiary constraints, such as `x + y = z` or, equivalently, bounds on `x + y - z` for arbitrary triples. We leave it open whether these constructs are commonly used, and how to best manage the large number of candidates.

This simple grammar already allowed us to learn a varied set of constraints. However, for more complicated problems, the grammar can trivially be extended with additional unary (e.g., `X[i]*X[i]`, `mod(X[i], 2)`), binary or n-ary operators. This increased expressiveness would, however, incur an additional computational cost during learning.

Learning algorithm

Our learning algorithm learns from a set of **positive** examples T, i.e., given true solutions. For the propositional learner, which learns individual constraints on specific subsets of variables, we expect all examples to have the same size and hence the same number of decision variables. Every positive example consists of a set of tensors which contain assignments to a given set of decision variables. For the sake of exposition we limit our discussion in this paper to at most two dimensional tensors (lists and matrices) and use lists for illustration whenever possible.

▶ **Example 2.** Consider the problem of graph coloring: Given a list of nodes, assign a color to every node such that nodes that share an edge are assigned different colors. This problem can be encoded using a list of n integer decision variables X – one per node – whose values corresponds to colors. Positive examples would be assignments to X that satisfy the graph coloring constraints. For an instance with 5 nodes and edges $1-2, 1-3, 2-4, 3-5$, examples could be assignments $t_1 =$ `[1, 2, 3, 1, 1]` and t_2 `[1, 2, 2, 1, 1]`.

To turn an expression into a constraint, COUNT-CP first generates all expressions in its grammar and computes their result for different decision variables. Unary expressions are simply applied to every single decision variable `X[i]`. Binary expressions are applied to every possible pair of decision variables `(X[i], X[j])`. In general, for n-ary expressions, all tuples of n variables are generated. We use lexicographical ordering to ensure pairs are not enumerated multiple times to avoid redundant constraints. For asymmetric expressions this optimization cannot be used as the position of variables are important. However, substraction is a special case because it holds that `lb <= a - b <= ub` can be rewritten as `-ub <= b - a <= -lb` which simply results in different bounds being learned.

Aggregates are applied to logical groups of variables, such as individual rows and columns of matrices or user provided groupings (see partitions in Subsection 3.3). For every expression e and set of variables V, COUNT-CP then computes the minimum and maximum result across all training examples. We represent these local lower- and upper-bounds as tuples $\langle e, V, \texttt{lb}, \texttt{ub}\rangle$, where – denoting the values of variables V in example t as V^t:

$$\texttt{lb} = \texttt{min}\{e(V^t) \mid t \in T\} \qquad \texttt{ub} = \texttt{max}\{e(V^t) \mid t \in T\}$$

It follows that, by design, the constraints learned by COUNT-CP are always satisfied by all training examples. In a sense, our approach learns the convex hull of all mathematical expressions that can be expressed by the grammar.

▶ **Example 3.** Let us apply this approach to the graph coloring example. For the binary expression `abs(X[i] - X[j])`, COUNT-CP would compute the result of the expression for every pair in every example and then compute the bounds for every pair across the examples:

Pair	t_1	t_2	lb	ub	Pair	t_1	t_2	lb	ub
X[1], X[2]	1	1	1	1	X[2], X[4]	1	1	1	1
X[1], X[3]	2	1	1	2	X[2], X[5]	1	1	1	1
X[1], X[4]	0	0	0	0	X[3], X[4]	2	1	1	2
X[1], X[5]	0	0	0	0	X[3], X[5]	2	1	1	2
X[2], X[3]	1	0	0	1	X[4], X[5]	0	0	0	0

For all pairs of nodes with an edge between them, COUNT-CP will learn a constraint `abs(X[i] - X[j]) >= 1`, i.e., the nodes must have different colors.

3.3 Learning first-order constraints

Until now we have focused on learning propositional constraints for individual instances of a problem type. These local constraints can capture constraints over specific subsets of variables, however, these constraints cannot be used to find solutions for instances of different sizes (and hence different numbers of variables) and are prone to overfitting the training examples. Our goal is to address these shortcomings by learning *first-order* constraints that are independent of the instance size. That is, constraints of the form `foreach V in ...: lb <= expr(V) <= ub` This will allow us to learn constraints from, e.g., a 4×4 Sudoku and use these constraints to solve a 9×9 Sudoku. Additionally, we can find constraints, e.g., that the number of working days of nurses is at most 5, by generalizing across different nurses in a single example, even if some nurses always worked fewer days in the training examples.

Grouping constraints

The propositional constraint learning approach can learn constraints such as `alldifferent` by learning individual constraints `abs(X[i] - X[j]) >= 1` between each pair of decision variables. However, these local pairwise constraints are *hardcoded* for individual pairs of variables, and will not generalize to instances of different sizes that, for example, have more or less decision variables.

To overcome this limitation and learn constraints that are independent of the problem size, we find *index groups*, groups of decision variables or pairs of variables, that share a constraint. The concept is akin to the concept of generator expressions in ModelSeeker [2].

In this setting, for example, `alldifferent` can be encoded as:

```
foreach pairs (x[i], x[j]) in X:
    abs(x[i] - x[j]) >= 1
```

For a given expression e, e.g., absolute difference, and the set of learned local constraints C COUNT-CP uses a *sequence grammar* to generate *sequences*, i.e., sets $\mathbf{V}$ of decision variables to group over. For example, the sequence `all pairs` generates all pairs of decision variables. Next, COUNT-CP aggregates all the lower- and upper-bounds that had been found for local constraints to obtain a grouped or *first order* constraint $\langle e, \mathbf{V}, \texttt{lb}, \texttt{ub}\rangle$, where:

$$\texttt{lb} = \texttt{min}\{l \mid V \in \mathbf{V} \wedge \langle e, V, l, u\rangle \in C\}$$

$$\texttt{ub} = \texttt{max}\{u \mid V \in \mathbf{V} \wedge \langle e, V, l, u\rangle \in C\}$$

Our COUNT-CP implementation includes the following sequences: `all`) all individual unary variables; `all pairs`) all pairs of variables; and `full`) a singleton set with all variables. These sequences are used for unary, binary and aggregate expressions, respectively. The implementation can easily be extended with additional sequences, such as, variables with even indices, sequential pairs of variables, etc.

Partitioning groups

An `alldifferent` constraint will not usually be applied to *all* possible variables. A common pattern, instead, is that the variables are partitioned into groups with an `alldifferent` over each group. This pattern is used both by COUNT-OR, as well as the constraint learning system ModelSeeker [2]. For example, consider the example of Sudoku where the decision variables are arranged in a matrix. In this case the variables can be partitioned into rows, columns or blocks and within each partition the variables will be `alldifferent`.

COUNT-CP follows this pattern, too, and first considers different ways to partition the decision variables before searching for sequences and corresponding bounds within each partition. By default, COUNT-CP considers arbitrary slices of tensors as partitions. In the case of matrices this would be rows, columns as well as the entire matrix. Additionally, COUNT-CP allows users to provide custom partitions. Custom partitions are a powerful way for users to interact with the system and provide high-level background knowledge, such as blocks for Sudoku or edges of a graph coloring problem.

For an expression e, a partition P and a sequence s, COUNT-CP iterates over all partitions $p \in P$ and generates sets of indices by applying the sequence to obtain sets of indices $\mathbf{V}_p = s(p)$. Using the tuples $\langle e, \mathbf{V}, \texttt{lb}, \texttt{ub}\rangle$ found in the grouping step, it aggregates the bounds across partitions to obtain tuples $\langle e, P, s, \texttt{lb}, \texttt{ub}\rangle$, where:

$$\texttt{lb} = \texttt{min}\{l \mid p \in P \wedge \langle e, s(p), l, u\rangle \in C\}$$

$$\texttt{ub} = \texttt{max}\{u \mid p \in P \wedge \langle e, s(p), l, u\rangle \in C\}$$

The combination of partitions, sequences and bounded expressions allow us to learn complicated sets of first-order constraints, e.g., that the variables in columns are `alldifferent` or that each of the sums over rows never exceed an upper bound.

▶ **Example 4.** Building on the graph coloring problem introduced above, a user can provide a custom partition $P_{\textsf{edges}}$ for each instance that corresponds to the edge E of the graph used in an instance: $P_{\textsf{edges}} = \{\{X[i], X[j]\} \mid (X[i], X[j]) \in E\}$.

```
foreach group in P_edges:
    foreach (X[i], X[j]) in pairs(group):
        abs(X[i] - X[j]) >= 1
```

3.4 Symbolic expressions for bounds

So far, we talked about grouping constraints using partitions and sequence generators. In some cases, e.g., the column-wise all different, this grouping step is enough to learn first-order constraints that can be applied to instances of any size. However, in some cases the lower- and upper-bounds depend on the particular instance.

▶ **Example 5.** Reconsider the nurse scheduling example. When learning across schedules from different hospitals, the minimal staffing requirement, i.e., how many nurses have to work each day might differ and are an instance (hospital) dependent constant. Simply learning the smallest minimal staffing requirement across all hospitals will produce poor results.

To address this issue, COUNT-CP attempts to express bounds using symbolic expressions. These symbolic expressions can use computed features, e.g., the number of rows and columns, or custom features that the user provides for every instance, e.g., the minimal staffing requirements for hospitals. To keep the discussion clear, we focus on finding a symbolic expression for the upper-bound of a single first-order constraint $\langle e, P, s, \texttt{lb}_i, \texttt{ub}_i\rangle$ across instances i. The steps are repeated for each constraint and are analogous for lower-bounds.

COUNT-CP aims to find a simple, univariate symbolic expression of the form $f + b$, where f is a computed feature, a custom feature or 0, and b is a fixed offset. Given m candidate features f_j the goal is to find the feature and offset that minimize the error across all instances. By denoting the value of feature f_j in instance i as f_j^i and using a binary indicator variable α_j to select a feature, we can express the error E_i of a single instance as:

$$E_i = \sum_{j=1}^{m} \alpha_j f_j^i + b - \texttt{ub}_i$$

Finding the best symbolic expression now corresponds to finding the assignment to the indicator variables α_j and offset b that minimizes the overall error: $\texttt{sum}(|E_i|)$. COUNT-CP imposes an additional constraint that the symbolic bound must be an upper bound of the learned bounds. In other words, E_i cannot be negative. This ensures that learned constraints will be satisfied by every training example. We can now write the optimization problem as:

$$\min_{\alpha_j} \sum_i E_i \quad s.t. \sum_j \alpha_j = 1 \ \wedge \ \alpha_j \in \{0, 1\} \ \wedge \ \forall i : E_i \geq 0$$

In practice, this problem can be solved easily by computing the optimal offset b_j and resulting error E_{ij} for each feature f_j and picking the index j^* with the smallest error:

$$b_j = \texttt{max}\{\texttt{ub}_i - f_j^i \mid i\}$$
$$E_{ij} = f_j^i + b_j - \texttt{ub}_i$$
$$j^* = \texttt{argmin}_j E_{ij}$$

The resulting expression will then be: $f_{j^*} + b_{j^*}$. Since COUNT-CP also includes the constant 0 as a feature, it will still return a numeric bound for expressions that do not depend on a symbolic feature. In fact, in these cases the fitted expression will simply be $\texttt{max}_i \ \texttt{ub}_i$, the aggregation operation we have already applied for aggregating bounds across examples, sequences and partitions.

▶ **Example 6.** For the nurse scheduling example, given a custom feature *minimal-staffing-requirement* (`msr`), COUNT-CP can now learn that the sum of every column (=nurses working on a day) is lower bounded by the `msr` leading to `foreach column: sum(column) >= msr`.

This approach can be applied on any of the `lb` or `ub` of the tuples $\langle e, \cdot, \texttt{lb}, \texttt{ub}\rangle$ found.

3.5 Filtering constraints

Filtering out useless constraints

By computing bounds over expressions, COUNT-CP ensures that learned constraints are always satisfied by training examples. However, by computing bounds over every expression, partition and sequence, COUNT-CP will always find valid lower- and upper-bounds. This can cause COUNT-CP to return many constraints which are true by default or trivially entailed by another constraint.

First, let us have a look at trivial constraints. As an example of a constraint that is true by default, consider `x[i] + x[j] <= c`, where `c` is the sum of the maximal values of the domains of variables `x[i]` and `x[j]`. COUNT-CP filters out trivial constraints by detecting them during the propositional learning step. Whenever COUNT-CP learns a local constraint $\langle e, V, \mathtt{lb}, \mathtt{ub}\rangle$, it also computes the minimal and maximal values of the expression e for the variables V and their domains: $l = \mathtt{min}\ e(V), u = \mathtt{max}\ e(V)$. If $\mathtt{lb} = l$ (or $\mathtt{ub} = b$) the bound is marked as trivial and the corresponding constraint, as well as any first-order constraint that includes that bound, is removed.

Second, let us consider trivial entailment. Consider two constraints: 1) for each column, the absolute difference of every pair of variables in the column in at least 1 ($\langle \mathtt{abs}, \mathtt{columns}, \mathtt{all\ pairs}, 1, \_\rangle$); and 2) for the entire matrix, the absolute difference of every pair of variables in the matrix in at least 1 ($\langle \mathtt{abs}, \mathtt{all}, \mathtt{all\ pairs}, 1, \_\rangle$). Because the pairs of variables in the first constraint is a subset of the pairs of variables in the second constraint, the first constraint is entailed by the second one, *unless* it has a stricter bound.

Consider two first-order constraints $c_1 = \langle e, P_1, s_1, l, u\rangle$ and $c_2 = \langle e, P_2, s_2, l, u\rangle$ with shared bounds (in practice entailment is computed for upper and lower bounds separately). If both constraints share the same partition $P = P_1 = P_2$ but one of the sequences is a subset of the other sequence: $\forall p \in P : s_1(p) \subset s_2(p)$, then c_1 is entailed by c_2. Since the sequence grammar is fixed, entailment between sequences can easily be computed in an offline step before learning. More generally, c_1 is entailed by c_2 if the union of sets of indices from c_1 is a subset of the union of sets of indices from c_2 (as in the example above): $\bigcup_{p\in P_1} s_1(p) \subset \bigcup_{p\in P_2} s_2(p)$. Because we allow users to provide custom partitions, the more general entailment cannot be fully pre-computed. COUNT-CP uses these entailment checks to filter out entailed constraints.

Because first-order constraints are made up of many local constraints, filtering out first-order constraints can drastically reduce the number of local constraints. This decreases the time it takes to solve a learned model and find new solutions, without affecting the quality of the model.

Filtering out overly restrictive constraints

COUNT-CP learns constraints that are satisfied by all training examples. However, there is a risk that the learned constraints exclude valid *unseen* solutions. Ideally, unconstrained expressions are detected by the trivial constraint detection step. However, given few training instances, COUNT-CP might find spurious constraints and produce bounds for unconstrained expressions. This may lead to the incorrect rejection of valid solutions.

Unfortunately, this problem is much more pertinent when learning constraints across instances and extrapolating to unseen instances. An incorrect, loose bound for an unconstrained expression might reject a few unseen solutions of the same size, however, it may reject large amounts of solutions for larger, unseen instances. COUNT-CP attempts to alleviate this issue by monitoring the errors E_i computed during the symbolic bound computation. If

Problem	User Input	
	Custom partitions	Semantic constants
Sudoku	Blocks of variables	-
Magic Square	-	-
N-Queens	Diagonals	-
Graph Coloring	Edges of the graph	-
Nurse Rostering	-	Staffing requirements

Table 1 List of problems used in the experiments along with the background knowledge provided as the user input to COUNT-CP

the errors exceed a given threshold, COUNT-CP opts to reject the bound and produce no constraint instead.

In theory, this type of filtering can occur at every step where different bounds are aggregated – over training examples, over sequences, across partitions – however, since bounds naturally vary, a lot of training data is required to avoid rejecting *valid* bounds.

4 Experiments

In this section, we empirically answer the following research questions:

Q1 How well does COUNT-CP perform on instances used during training?
Q2 Do models learned by COUNT-CP generalize to unseen instances?
Q3 How does the performance change with the size of the training set?
Q4 How fast is COUNT-CP and how does the run-time scale with the number of training examples?
Q5 How effective is the filtering step in COUNT-CP?

To answer these questions, we use COUNT-CP to learn models for a set of benchmark problems and evaluate its performance according to different metrics. The code is available online[1] and uses the CPMpy modeling library [9]. The benchmark problems (see Table 1) consist of problems selected from CSPLib[2], which is a library of test problems for constraint solvers, and an adapted nurse scheduling problem used to evaluate COUNT-OR [11]. The language bias used in COUNT-CP is not expressive enough to model all the CSBLib problems, therefore, we selected problems that COUNT-CP should be able to learn successfully. We hope these experiments will showcase the capabilities of our approach and the viability of our architecture across different problem domains. The language bias of our approach can be extended to cover more complicated constraints by adding building blocks to the various grammars at the cost of increasing the run-time. We leave the exercise of crafting biases to cover larger benchmarks to future work.

Performance measures

The performance of the learned models are measured in terms of *Precision* and *Recall*. Precision tells us what percentage of the learned feasible region is actually feasible in the target model, while recall tells us what percentage of the target feasible region is captured

[1] https://github.com/ML-KULeuven/COUNT-CP
[2] `https://www.csplib.org/`

Training Size	1		5		10	
	Precision	Recall	Precision	Recall	Precision	Recall
Sudoku	100%	100%	100%	100%	100%	100%
Magic Square	100%	100%	100%	100%	100%	100%
N-Queens	100%	100%	100%	100%	100%	100%
Graph Coloring	100%	57.5%	100%	100%	100%	100%
Nurse Rostering	100%	15.5%	100%	100%	100%	100%

Table 2 Performance of COUNT-CP across different problems and different training sizes. The results are shown only for training instances.

by the learned model. Ideally, having high precision is more desirable, as it ensures that the solutions generated using learned model have higher chances of being feasible, while high recall means we can generate many feasible solutions.

Calculating these performance measures is not trivial. We sample 100 solutions from the learned model and compute how many satisfy the target model – this gives us precision. The recall is computed by instead sampling 100 solutions from the target model and computing how many satisfy the learned model. Sampling uniformly from the feasible region of a model is extremely hard [20], however, the CPMPy constraint modeling framework[3] allows us to instruct the constraint solver to find solutions close to a given starting point. By generating each of the 100 solutions using different random starting points, we try to obtain a representative sample, which provides better estimates of the true precision and recall.

Setup

For each problem, we include two different *training* instances to learn from and another unseen *test* instance to evaluate the performance on unseen instances. Problems instances are instantiations of problems to a specific size or setting. For example, for the Sudoku problem, the training instances are of size 4×4 and 9×9, while the test instance is of size 16×16. For nurse rostering, different instances correspond to different hospitals, which have different staffing requirements and numbers of nurses.

For every problem instance, e.g., a 9×9 Sudoku, we use a set of training examples – solutions of the problem instance – to learn from. Specifically, we learn from $1, 5$ or 10 examples per instance. The performance is then evaluated using the sampling procedure described above.

Q1. How well does COUNT-CP perform on instances used during training?

To answer Q1, we report the precision and recall on the training instances (see Table 2). Using just a single training example per instance, COUNT-CP already learns models that have 100% precision. For two of the problems, a single example is not enough to obtain 100% recall. However, when given 5 training examples, COUNT-CP achieves 100% recall for all benchmark problems.

Training Size	1		5		10	
	Precision	Recall	Precision	Recall	Precision	Recall
Sudoku	100%	100%	100%	100%	100%	100%
Magic Square	100%	100%	100%	100%	100%	100%
N-Queens	100%	100%	100%	100%	100%	100%
Graph Coloring	100%	61%	100%	100%	100%	100%
Nurse Rostering	100%	18%	100%	100%	100%	100%

Table 3 Performance of COUNT-CP across different problems and different training sizes. The results are shown only for test instances.

Q2. Do models learned by COUNT-CP generalize to unseen instances?

By measuring the precision and recall of models learned by COUNT-CP for unseen test instances (see Table 3), we can observe that the performance is similar to the performance seen on training instances: Learning from just one example per training instance, our approach obtains 100% precision – even for unseen instances – and given 5 examples, COUNT-CP achieves 100% recall, as well.

In this paper, we argued for the need to introduce symbolic bounds in constraints in order for them to be able to generalize to unseen instances. We evaluate this claim qualitatively by comparing the scores obtained by COUNT-CP on the *Nurse Rostering* problem with a modified version that simply keeps numeric bounds. As expected, we see that the learned model cannot generalize well to unseen instances with different staffing requirements (see Table 4).

Training Size	1		5		10	
	Precision	Recall	Precision	Recall	Precision	Recall
COUNT-CP	100%	18%	100%	100%	100%	100%
Naive version	0%	100%	0%	100%	0%	100%

Table 4 Comparison of COUNT-CP against a naive version which learns numerical bounds instead of symbolic expressions.

Q3. How does the performance change with the size of the training set?

The change in performance across different training sizes is shown in Table 2 and Table 3. When we use more training examples, COUNT-CP learns less tight bounds, which in turn would lead to improved recall, and that is exactly what we observe in the results as well. In most cases we learn perfect model with just one example, and in cases where this is not the case (last two rows in both tables), the performance improves as the size of the training set increases.

Problem	Time Taken (in seconds)
Sudoku	2.5
Magic Square	58.3
N-Queens	8.5
Graph Coloring	22.6
Nurse Rostering	328.3

Table 5 COUNT-CP learns all problems in less than a minute except nurse rostering where it takes close to 5 minutes.

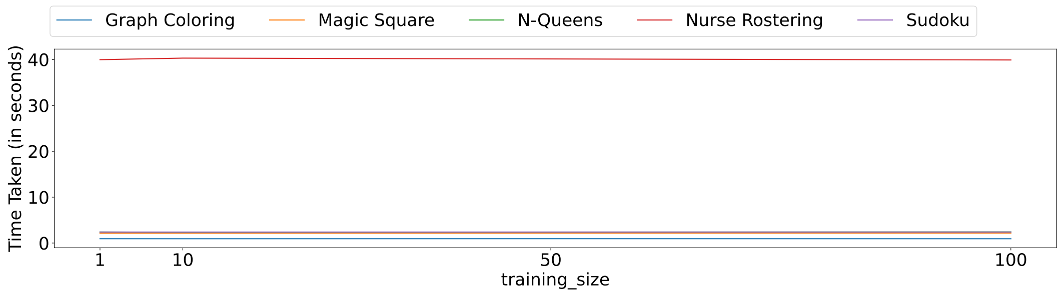

Figure 1 The learning time of COUNT-CP remains consistent when increasing the number of training examples

Q4. How fast is COUNT-CP and how does the run-time scale with the number of training examples?

COUNT-CP learns most problems in a less than a minute, except for the *Nurse Rostering* problem, for which it requires close to 5 minutes (see Table 5). Considering the time taken by experts to model a problem and the fact that once learnt, these models can be used to solve problems of different sizes, we can characterize our learning time as lightning fast in comparison. The run-time depends mainly on the number of decision variables and since COUNT-CP enumerates all pairs of variables, the run-time increases quadratically with the number of decision variables. Here, again, the ability to learn models from small instances and apply them to much larger instances makes COUNT-CP useful in practice.

COUNT-CP scales linearly with the number of training examples, however, by evaluating candidate expressions efficiently using vectorized operations, the impact of the number of training instances is negligible in most cases (see Figure 1). This, again, is good news, as it allows the user to provide large number of training examples to learn more accurate models, while avoiding long learning times.

Q5. How effective is the filtering step in COUNT-CP?

In Subsection 3.5, we discussed the importance of filtering out useless and overly restrictive constraints. Unnecessary constraints make the learned models less interpretable and slower to solve. Filtering out these constraints, however, has a cost: It significantly increases the learning time by adding overhead for every single local constraint learned.

To answer Q5 and evaluate the effectiveness of the filtering step, we compare the total

[3] `https://github.com/CPMpy/cpmpy`

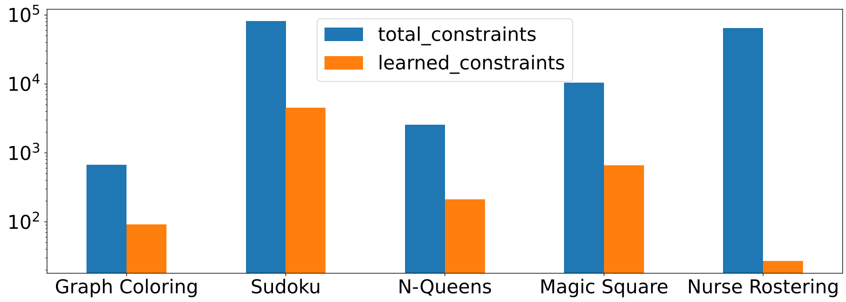

Figure 2 Filtering step in COUNT-CP leads to more than 96% reduction in the total number of learned constraints.

number of possible constraints produced by COUNT-CP with the constraints included in the learned model after the filtering step. Our experiments show that COUNT-CP is able to drastically reduce the number of constraints it outputs (see Figure 2). On average, COUNT-CP filters out 96.7% of the constraints, significantly improving the interpretability and solving time of learned models.

5 Conclusion

In this paper, we presented the novel constraint learner COUNT-CP, which uses simple grammars and a generate-and-aggregate approach to generate mathematical expressions, compute their bounds across training examples and group the learned constraints to obtain first-order constraints that can generalize to unseen instances. A symbolic expression fitting step is used to obtain symbolic bounds for expressions, making them instance-independent. Additionally, COUNT-CP uses an effective filtering step to remove useless and spurious constraints. We empirically evaluated our approach on a set of suitable benchmark problems. This evaluation showed that, indeed, COUNT-CP is able to learn compact, high quality models quickly. The learned models achieve high precision and recall, even when only trained on a handful of examples. Because the learned models contain first-order constraints and support bound expressions, these results also hold true for unseen instances. Finally, our simple interaction protocol allows users to provide relevant background knowledge without requiring any specialized knowledge about the underlying constraint language. We believe that the COUNT-CP architecture is a promising approach to constraint learning that can be further tuned to learn a wide range of constraint problems.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

2 Nicolas Beldiceanu and Helmut Simonis. A Model Seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, 2012.

3 Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El-Houssine Bouyakhf. Boosting constraint acquisition via generalization queries. In *ECAI*, pages 99–104, 2014.

4 Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace,

editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 123–137, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

5 Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *Machine Learning: ECML 2005*, pages 23–34, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

6 Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 50–55, 2007. URL: `http://ijcai.org/Proceedings/07/Papers/006.pdf`.

7 Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, Christian Bessiere, and El Houssine Bouyakhf. Detecting types of variables for generalization in constraint acquisition. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 413–420. IEEE, 2015.

8 Luc De Raedt and Sašo Džeroski. First-order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70(1-2):375–392, 1994.

9 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *The 18th workshop on Constraint Modelling and Reformulation at CP*, pages 1–8. ModRef, 2019.

10 Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. Learning constraints in spreadsheets and tabular data. *Mach. Learn.*, 106(9-10):1441–1468, 2017. `doi:10.1007/s10994-017-5640-x`.

11 Mohit Kumar, Stefano Teso, Patrick De Causmaecker, and Luc De Raedt. Automating personnel rostering by learning constraints using tensors. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 697–704, 2019. `doi:10.1109/ICTAI.2019.00102`.

12 Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 45–52. IEEE Computer Society, 2010. `doi:10.1109/ICTAI.2010.16`.

13 Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *Journal of Automated Reasoning*, 24(1):165–203, 2000. `doi:10.1023/A:1006326723002`.

14 Ralph Eric Mcgregor. *Automated Theorem Proving Using Sat.* PhD thesis, USA, 2011. AAI3471671.

15 Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 102–115, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

16 Tomasz Pawlak. Synthesis of mathematical programming models with one-class evolutionary strategies. *Swarm and Evolutionary Computation*, 44, 05 2018. `doi:10.1016/j.swevo.2018.04.007`.

17 Tomasz Pawlak and Krzysztof Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *European Journal of Operational Research*, 261, 02 2017. `doi:10.1016/j.ejor.2017.02.034`.

18 Tomasz Pawlak and Krzysztof Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *EJOR*, 2017.

19 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence - AAAI 18*, pages 7965–7970. AAAI Press, 2018.

20 Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Knowledge compilation meets uniform sampling. In *Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 11 2018.

21 L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, November 1984. `doi:10.1145/1968.1972`.