2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)

Automating Personnel Rostering by Learning Constraints Using Tensors

Mohit Kumar
KU Leuven
mohit.kumar@cs.kuleuven.be

Stefano Teso
KU Leuven
stefano.teso@cs.kuleuven.be

Patrick De Causmaecker
KU Leuven
patrick.decausmaecker@kuleuven.be

Luc De Raedt
KU Leuven
luc.deraedt@cs.kuleuven.be

***Abstract*—Many problems in operations research require that constraints be specified in the model. Determining right constraints is a hard and laborsome task. We propose an approach to automate this process using artificial intelligence and machine learning principles. We focus on personnel rostering and scheduling problems in which there are often past schedules available and show that it is possible to automatically learn constraints from such examples. To realize this, we adapted some techniques from the constraint programming community and extended them in order to cope with multidimensional examples. The method uses a tensor representation of the example, which helps in capturing the dimensionality as well as the structure of the example, and applies tensor operations to find the constraints that are satisfied by the example. The algorithm also identifies inherent clusters in the data and uses it as background knowledge to learn more detailed constraints. To evaluate the proposed algorithm, we used constraints from the Nurse Rostering Competition and generated solutions that satisfy these constraints; these solutions were then used as examples to learn constraints. Experiments demonstrate that the proposed algorithm is capable of producing human readable constraints that capture the underlying characteristics of the examples.**

***Index Terms*—Artificial Intelligence, Personnel Rostering, Constraint Learning, Inductive Learning, Machine Learning, Tensors**

I. Introduction

Constraints are pervasive in Operations Research (OR) applications such as scheduling and planning. Consider for instance the case of nurse rostering [1]. Hospitals usually generate a weekly schedule for their nurses based on constraints like "a nurse can work at most 5 days every week". As the number of nurses and the complexity of the constraints increases, generating the schedule manually becomes impractical. Hospitals can use optimization solvers (e.g. Gurobi) to produce the schedules automatically, but these solvers require modeling the constraints into an OR model which itself is a complicated task. Hiring a domain and modeling expert to manually design a model can be expensive and time consuming [2]. A tempting alternative is to employ constraint learning [3] to automatically induce the model from examples of past schedules. The learned model can then be used as-is to produce solutions in line with the constraints extracted from the example schedules, or serve as a first step to design the final model.

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models)". The authors are grateful to Samuel Kolb for interesting discussions related to constraint learning.

	Day$_1$			Day$_2$			Day$_3$		
	S$_1$	S$_2$	S$_3$	S$_1$	S$_2$	S$_3$	S$_1$	S$_2$	S$_3$
Nurse$_1$	1	0	0	0	1	0	0	1	0
Nurse$_2$	0	1	0	1	0	0	0	0	1
Nurse$_3$	0	0	1	0	0	0	1	0	0
Nurse$_4$	0	0	1	0	0	1	1	0	0

TABLE I: Example nurse schedule with three dimensions: four nurses, three days, and three shifts per day. Cells identified by the sub-tensor $\mathbf{X}[\texttt{Day}_1]$ have been highlighted by yellow and $\mathbf{X}[\texttt{Nurse}_1, \texttt{Day}_3]$ is highlighted by green. Best viewed in color.

There are a number of exisiting approaches that learn constraints from examples of known feasible (and infeasible) assignments of values to variables [3]. However, most of them require examples of negative (i.e., infeasible) configurations, which are often unavailable in real world applications. Furthermore, most of them are not designed to handle structured (e.g. multi-dimensional) variables and numerical quantities which are the norms in rostering problems [4]–[6]. To see what we mean, consider the nurse schedule shown in Table I. For each combination of nurse, day and shift, the value 1 indicates that a particular nurse worked in that particular shift of that day, while a 0 means the nurse did not work. It is easy to see that nurses, days, and shifts are independent of each other and behave like different dimensions, and the quantities of interest like the number of working days are numerical. While many recent constraint learning methods can deal with numerical data (e.g., [7]), most approaches are not designed to handle high-dimensional problems. We discuss these issues further in Section II.

To amend this situation, we propose learning *COnstraint UsiNg TensORs* (COUNT-OR), a novel constraint learning approach that uses tensors—that is, n-dimensional matrices [8]—for capturing the structure and dimensionality of the schedules. COUNT-OR can easily deal with numerical information and most importantly relies only on feasible solutions. Given a set of past schedules, the goal of COUNT-OR is to extract constraints like "the minimum number of employees working each day is at least 3". In order to learn the constraints, COUNT-OR enumerates and extracts all meaningful slices (that is, parts) of the input schedules, aggregates them through tensor operations, and then computes bounds for the aggregates to generate candidate constraints. Some simple filtering strategies

2375-0197/19/$31.00 ©2019 IEEE
DOI 10.1109/ICTAI.2019.00102

IEEE computer society

are then applied to prune irrelevant and trivially satisfied candidates.

Our key contributions are:

1) A tensor representation of rostering models that allows us to reason over real-world personnel rostering instances and constraints.
2) A novel constraint learning algorithm, COUNT-OR, which uses tensor extraction and aggregation operations to acquire constraints from example schedules.
3) An extension of COUNT-OR to learn additional background knowledge (e.g., nurse skill levels) and use it to learn more detailed constraints (e.g., Minimum number of high skilled nurses required in a shift).
4) An empirical evaluation of COUNT-OR on real-world nurse rostering models taken from Nurse Rostering Competition[1] (INRC-II [9]), showing that our approach can recover models that behave correctly in a handful of seconds.

The paper is structured as follows. In the next section we briefly overview existing approaches to constraint learning. We present our learning method in Section III and evaluate it empirically on real-world nurse rostering problems in Section IV. We conclude with some final remarks and hints at promising extensions.

II. Related Work

COUNT-OR is a unique approach to learn constraints, in the sense that it handles numerical data, utilizes the structure and dimensionality of the data, and needs only feasible solutions to discover local constraints in the form of inequalities.

The most basic form of constraints are propositional logic formulae, or *concepts*. Learning concepts from examples has a long history in AI [10]. In the simplest case, concept learning assumes that there exists a hidden, target concept belonging to some pre-specified class (for instance the class of conjunctive formulae with clauses of some predefined length). Given a set of examples, that is, assignments of values to variables labeled as feasible or unfeasible with respect to the target concept, the goal is to retrieve the hidden concept from the observed examples. This framework has been extended to first-order formulae under the name of inductive logic programming [11], [12]. While relevant, these classical approaches are designed to deal with Boolean variables only; it is unclear how to extend these approaches to learn OR models, since these often include numerical variables and constraints on them. More recent approaches can learn constraint programs with numerical terms. For instance, the works of Bessiere and colleagues [13] follow the same setup as concept learning, but is designed to learn arbitrary constraint satisfaction models. These can include both numerical variables and constraints on them (for example $x \leq y$ or $x \neq y$, where x and y are integer variables). These approaches maintain a candidate model and iteratively refine it to account for the examples that are inconsistent with it. These approaches, however, are not designed for tensor data and do not take advantage of the multi-dimensional structure imposed by it. In particular, they do not natively support complex tensor operations like slicing and continguous one/zero checks. Further, it does not support acquisition of background knowledge like nurse skill. COUNT-OR, on the other hand, is specifically meant for tensor data and does automatically acquire background knowledge if needed, as detailed later on.

There exists a class of learning approaches that can potentially deal with high-dimensional numerical domains [14], [15]. The advantage of these works is that they can learn *soft* constraints, i.e., constraints that can be violated, and whose "degree of satisfaction" is taken into account by the objective function. However these approaches require to invoke a satisfaction (or optimization) oracle, and thus do not scale to larger learning settings. TaCLe [7] can also deal with numerical variables, but is designed specifically for tabular data.

Other recent work include ESOCCS [16] which is a heuristic approach based on evolutionary optimization. It starts from positive examples only but uses density estimation to sample negatives, thus converting the positive-only setting to binary classification [15]. In addition, it can learn weighted constraints. (Their strategy could be adapted to learn IP objective functions, but we leave this to future work.) Contrary to COUNT-OR, ESOCCS expects the terms to be enumerated upfront, which is impractical in IP, because the number of terms can be huge. Furthermore, it does not exploit structure on the set of variables.

To the best of our knowledge, the only other constraint learning approaches that uses a tensor representation are ModelSeeker [17] and ARNOLD [18]. ModelSeeker aims at learning *global* constraints from examples. In order to do so, it takes a vector of integer values as input and looks for patterns holding in different rearrangements of the vector in the form of matrices (i.e. bi-dimensional tensors). While apparently similar, COUNT-OR focuses on discovering *local* constraints, which are much more common in OR problems. It is of course conceivable to combine the two approaches if global constraints do appear in the particular application domain. ARNOLD learns integer programs with polynomial inequalities using a tensor representations of the variables and constants, but differs from COUNT-OR in two ways. First, ARNOLD assumes all the constants (including the bounds) to be given as an input. COUNT-OR on the other hand learns the bounds from the schedules. Second, ARNOLD cannot extract slices from tensors, so this must be done manually if needed, while COUNT-OR enumerates all relevant slices automatically.

III. Learning Constraints with COUNT-OR

The aim of COUNT-OR is to simplify the modeling process by inducing a working model that can be either used as-is to produce new schedules or further improved. More formally, the problem we address can be stated as follows: **given** a set of past schedules, **find** a rostering model M_L satisfied by all of them.

[1] http://mobiz.vives.be/inrc2/

First, we assume the examples to represent *desirable* schedules, i.e., schedules that were observed to work well enough in practice. This is reasonable, because the example schedules are likely the result of trial-and-error where only the best schedules were kept. For the same reason, we also expect the examples to not be optimal in any sense of the word. Of course, the learned model mimics the example schedules, and so the higher the desirability of the provided examples, the better the schedules generated by the learned model.

Since the example schedules likely reflect contractual obligations (and other regulations) that are always known in advance, it may seem pointless to learn a rostering model from examples. This is not the case. Indeed, even though in principle all employees should know and understand the contract, they may be unable to encode it into a working model. COUNT-OR takes care of this automatically. In addition, the examples will often reflect constraints not appearing in the contract: for instance, the hospital may only need two physiotherapists at any time, because there are only a limited number of facilities for them. Modelling this fact avoids generating infeasible and costly schedules. Most importantly, real-world schedules may not respect the contractual obligations. Such "imperfect" examples capture real trends occurring in the working place and, by including contractual obligations into the learned model, allow to build models that blends regulations with known-working practices.

While soft constraints appear frequently in OR models, it is true that hard constraints reflect practical limitations, regulations and contractual obligations, which lie at the core of most applications. For this reason, COUNT-OR focuses on learning hard (feasibility) constraints only.

A. High-level overview

At a high-level, COUNT-OR consists of four steps:

Tensorization: Each input schedule is converted to a tensor whose elements represent, in the simplest case, which employees worked in which days and shifts, as detailed in Section III-B.

Aggregation: The algorithm enumerates all sub-tensors and summarizes them by applying one or more aggregation operations. This way, it obtains several quantities of interest, e.g., the number of employees each day or the number of working days for each employee. See Section III-C.

Bound computation: Numerical bounds for these quantities are then computed by taking the minimum and maximum across all sub-tensors, producing a number of candidate constraints of the form "the minimum number of employees each day is 4". See Section III-E.

Filtering: Trivially satisfied candidates are filtered out to produce a set of consistent constraints. These can readily be fed to any constraint solver to generate new schedules consistent with the examples. See Section III-F.

While introducing the algorithm, we will assume to be given a single example schedule and no background knowledge, for clarity, and lift these restrictions later on. In particular, Section III-G discusses the simple modifications required to handle multiple examples, while Sections III-I and III-J detail how to incorporate background knowledge into the learning process and acquiring the background knowledge from data, respectively.

Days (front slice; Nurses × Shifts)

Nurses	Shifts				
	1	0	0	0	0
	0	1	0	0	1
	0	0	1	0	0
	0	0	1	1	0

TABLE II: Tensor representation of the schedule in Table I

Before proceeding, we define some common terms to avoid ambiguities due to differences between OR and AI terminologies. The term *problem* refers to the inference problem of producing feasible schedules for a particular rostering instance, while *model* indicates the model used for obtaining said schedules. This paper is about the *learning problem* of inducing a model from a set of past schedules, referred to as *input schedules* or *examples*.

B. Mapping Schedules to Tensors

Consider the nurse schedule in Table I. Formally, let $D = \{D_1, D_2, \ldots, D_n\}$ be the *dimensions* and D_i the set of distinct values for the ith dimension. In our example schedule there are 3 dimensions, namely $D = \{\texttt{Nurses}, \texttt{Days}, \texttt{Shifts}\}$, where $\texttt{Nurses} = \{\texttt{Nurse}_1, \ldots, \texttt{Nurse}_4\}$ and similarly for $\texttt{Days}$ and $\texttt{Shifts}$.

A schedule in this format can be readily represented by a rank-n tensor, denoted $\mathbf{X}$, where $n = |D|$ is the number of dimensions in the schedule. The shape of the tensor reflects the number of distinct values for each dimension. In our example, the schedule has 3 dimensions, so it can be represented by a tensor of rank 3 with shape $[4, 3, 3]$ (as shown in Table II). The elements of the tensor are identified by a particular (nurse, day, shift) combination. For instance, $\mathbf{X}[\texttt{Nurse}_2, \texttt{Day}_1, \texttt{Shift}_3]$ shows whether $\texttt{Nurse}_2$ worked on $\texttt{Day}_1$ in $\texttt{Shift}_3$.

Before proceeding, we introduce some required notation. We write $\mathbf{X}[d_i]$ to indicate the sub-tensor obtained by fixing the ith dimension of $\mathbf{X}$ to $d_i \in D_i$. For example, $\mathbf{X}[\texttt{Day}_1]$ extracts the working schedule of all nurses for the three shifts on $\texttt{Day}_1$ (highlighted by yellow in Table I). Fixing multiple dimensions is allowed. For instance, $\mathbf{X}[\texttt{Nurse}_1, \texttt{Day}_1]$ extracts the sub-tensor $[1, 0, 0]$, where the elements refer to the different shifts for $\texttt{Nurse}_1$ on $\texttt{Day}_1$ (cf. Table I, Green). We will make extensive use of the Cartesian product, defined as $D_1 \otimes D_2 = \{(d_1, d_2) \mid d_1 \in D_1, d_2 \in D_2\}$. For any choice of dimensions $D' \subseteq D$, we use the shorthand $\bigotimes(D')$ to indicate the Cartesian product of the dimensions in D'.

C. Enumeration and Aggregation

We are now ready to introduce the first two aggregation functions, Nonzero and Sum, which play a central role in our

algorithm. Given an input tensor $\mathbf{X}$, $\text{Nonzero}(\mathbf{X}, D')$ reduces it to an aggregate tensor $\mathbf{Y}$ indexed by $e \in \bigotimes(D')$, by checking for each e whether there exists at least one non-zero element in the sub-tensor $\mathbf{X}[e]$. More formally, letting $I(c)$ be the indicator function, the output of Nonzero satisfies $\mathbf{Y}[e] = I(\mathbf{X}[e] \neq \mathbf{0})$ for every $e \in \bigotimes(D')$. The $\text{Sum}(\mathbf{X}, D')$ function is defined analogously, except that $\mathbf{Y}$ is obtained by summing up all the elements of $\mathbf{X}[e]$. These functions allow us to capture many quantities of interest. For instance, checking whether $\texttt{Nurse}_i$ works in *any* shift of $\texttt{Day}_j$ can be accomplished by applying the Nonzero function with $D' = \{\texttt{Nurses}, \texttt{Days}\}$, so that $\mathbf{Y}$ is:

$$\mathbf{Y}[\texttt{Nurse}_i, \texttt{Day}_j] = I(\mathbf{X}[\texttt{Nurse}_i, \texttt{Day}_j] \neq \mathbf{0})$$

Similarly, the number of shifts worked by $\texttt{Nurse}_i$ on $\texttt{Day}_j$ can be retrieved by applying the Sum function with $D' = \{\texttt{Nurses}, \texttt{Days}\}$, so that $\mathbf{Y}$ is:

$$\mathbf{Y}[\texttt{Nurse}_i, \texttt{Day}_j] = \text{sum of values in } \mathbf{X}[\texttt{Nurse}_i, \texttt{Day}_j]$$

By varying D', these functions produce other relevant quantities, such as the number of working employees for each day, the number of working days for each employee, *etc*.

We introduce one more function, Count, which combines Nonzero and Sum to express even more quantities of interest. Let M and S be two disjoint, non-empty subsets of D. Count is defined as:

$$\text{Count}(\mathbf{X}, M, S) = \text{Sum}(\text{Nonzero}(\mathbf{X}, M \cup S), S) \quad (1)$$

For instance, if $M = \{\texttt{Nurses}\}$ and $S = \{\texttt{Days}\}$, $\text{Nonzero}(\mathbf{X}, M \cup S)$ returns a rank 2 tensor $\mathbf{Y}$ of shape [4, 3] over the 4 nurses and the 3 days, where $\mathbf{Y}[\texttt{Nurse}_i, \texttt{Day}_j]$ encodes whether the ith nurse worked in any shift on the jth day. Count then applies Sum to this tensor to obtain a 1-d tensor, $\mathbf{Z} = \text{Sum}(\mathbf{Y}, S)$, of shape [3] over the three days, where $\mathbf{Z}[\texttt{Day}_i]$ encodes the total number of nurses working on the ith day. The end result is a tensor indexed by the values in S, showing the total number of distinct employees working on different days: $\mathbf{Z} = [4 \;\; 3 \;\; 4]$.

D. Dealing with other constraints

By definition, the Sum and Nonzero ignore the order in which zeros and ones appear in the input tensor. Order, however, is essential for capturing quantities like "the minimum (or maximum) number of consecutive working days (or shifts) for an employee". To deal with these, we introduce four more aggregation functions: MinConsZero, MinConsOne, MaxConsZero, and MaxConsOne, described next.

Given an input tensor $\mathbf{X}$ and a subset of dimensions D', $\text{MaxConsOne}(\mathbf{X}, D')$ outputs a tensor $\mathbf{Y}$ of rank $|D'|$ by taking the maximum number of consecutive ones in $\mathbf{X}[e]$, where $e \in \bigotimes(D')$. Similarly, MinConsOne computes the minimum number of consecutive ones. These two functions produce an upper and lower bound, respectively. The two other functions, MaxConsZero and MinConsZero work analogously, but for consecutive zeros.

M, S	$\text{Count}(\mathbf{X}, M, S)$
{Days} , {Nurses}	# of working days per Nurse
{Days, Shifts} , {Nurses}	# of working shifts per Nurse
{Nurses} , {Days}	# of distinct employees per day
{Shifts} , {Days}	# of shifts for each day with at least one nurse working
{Shifts} , {Nurses, Days}	# of working shifts per day per nurse
{Days} , {Nurses, Shifts}	# of working days for a shift per nurse
{Nurses} , {Days, Shifts}	# of nurses per shift each day

TABLE III: A selection of quantities of interest representable by the Count function for different choices of M and S.

To see how these work, consider the case $M = \{\texttt{Days}\}$ and $S = \{\texttt{Nurses}\}$: replacing Sum with MaxConsOne in Eq. 1 allows us to compute number of maximum consecutive working days for each nurse.

Note that when $\mathbf{X}[e]$ is a tensor of rank ≥ 1 (i.e., when it is not a vector), talking about consecutive ones does not make much sense. So for these four functions we only consider the cases where $\mathbf{X}[e]$ is a 1-d tensor, i.e., $|D \setminus D'| = 1$.

E. Computing the bounds

After enumerating all the quantities of interest, COUNT-OR computes their minima and maxima to produce candidate constraints capturing their variation. So for each combination of M and S, first it calculates $\text{Count}(\mathbf{X}, M, S)$, and from there it computes the empirical lower and upper bounds as:

$$\bot^{\text{Count}}_{M,S} = \min_e \, (\text{Count}(\mathbf{X}, M, S))[e] \quad (2)$$

$$\top^{\text{Count}}_{M,S} = \max_e \, (\text{Count}(\mathbf{X}, M, S))[e] \quad (3)$$

Here the index e iterates over all elements of the result of Count, that is, $e \in \bigotimes S$. When applied to our example schedule, our algorithm would produce the lower and upper bounds $\bot^{\text{Count}}_{M,S}, \top^{\text{Count}}_{M,S}$ for each quantity of interest listed in Table III (among others).

Once the upper and lower bound of some quantity (e.g. $\text{Count}(\mathbf{X}, M, S)$) are known, we can immediately turn them into a constraint regulating the feasible values of that quantity. The resulting constraint is:

$$\bot^{\text{Count}}_{M,S} \leq \text{Count}(\mathbf{V}, M, S) \leq \top^{\text{Count}}_{M,S}$$

Note that in equations 2 and 3 $\mathbf{X}$ is the tensor *constant* representing the input schedule. In contrast, in this equation $\mathbf{V}$ is a tensor *variable* whose value is being constrained.

F. Dealing with irrelevant constraints

COUNT-OR considers all possible combinations of M and S, which might lead to some trivially satisfied or meaningless constraints. For example, learning that "the minimum number

of working days for a nurse is 0" does not give any information: the number of working days is always non-negative, hence the learned constraint is trivially satisfied.

Depending on the application domain, some combinations of M and S might also lead to syntactically sound, but meaningless, constraints. For example, if $M = \{\texttt{Shifts}\}$ and $S = \{\texttt{Days}\}$, the count we get represents "the number of shifts for each day where there was at least one nurse working", which does not make much sense. COUNT-OR can be instructed not to enumerate sub-tensors for known meaningless choices of M and S. This can be partially automated by observing some common patterns followed by such constraints. Specifically, for constraints learned using the ordered aggregation functions (MinConsZero, MinConsOne, etc.). For instance, ordering of Nurses is not important, as all the nurses are identical in the example above, so finding consecutive values of ones or zeros across Nurses does not make any sense. More formally, for a constraint, if $D_i \in M$ such that ordering of values in D_i is not important, then we neglect the constraint learned using ordered aggregation functions.

Second, we filter out trivially satisfied candidate constraints as follows. For the upper bound, if it holds that $\top^{\text{Count}}_{M,S} = |\bigotimes(S)|$ then we discard this upper bound because $|\bigotimes(S)|$ is the maximum possible value $\text{Count}(\mathbf{X}, M, S)$ can take, so this bound is trivial. For example, when $M = \{\texttt{Nurses}\}$ and $S = \{\texttt{Days}, \texttt{Shifts}\}$, if we learn that $\top^{\text{Count}}_{M,S} = |\bigotimes(S)| = 21$, it translates to maximum number of working shifts being 21 in a week, which is an obvious bound, so we drop this constraint. Similarly, for the lower bound, if $\bot^{\text{Count}}_{M,S} = 0$ we drop the constraint as it is trivial. These same two rules hold when using the ordered aggregation functions instead of Sum.

These simple filtering rules are surprisingly effective in practice. In our experiments with real-world scheduling problems, we noticed that they get rid of most of the irrelevant constraints without having to tell explicitly the algorithm which constraints to ignore (we have reported the numbers in Section IV). This is because the hidden, target problem does not constraint all possible combinations of dimensions and aggregation operations, and therefore the learned bounds match the "trivial", unconstrained minima and maxima of the quantities of interest. Our filtering stage detects these occurrences and gets rid of the superfluous constraints learned in these cases.

G. Handling multiple input schedules

If we have multiple input schedules, we can use them to learn more precise bounds for the constraints as follows. For each given input schedule, we first learn the bounds for all the possible combinations of M and S and aggregation operators using our algorithm. Then we aggregate the bounds themselves by taking a minimum over the lower bounds and a maximum over the upper bounds. For example, if we learn that "the number of working days ≥ 3" for one input schedule and "the number of working days ≥ 2" in the other schedule, our final constraint would be "the number of working days ≥ 2" because it is more general and satisfies both the input schedules. Then based on the filtering rules defined in the previous section we discard some of learned constraints.

Algorithm 1 The COUNT-OR algorithm. $\mathbf{X}$ is the input schedule in tensor form, D is the set of dimensions of $\mathbf{X}$, and O is the set of dimensions of $\mathbf{X}$ for which ordering does matter.

```
1:  procedure FINDCONSTRAINTS(X, D, O)
2:      M_L ← ∅
3:      for M, S ∈ ENUMERATESPLITS(D) do
4:          B ← ∅
5:          C ← Count(X, M, S)
6:          B ← B ∪ {Count(V, M, S) ≥ min_e C[e]}
7:          B ← B ∪ {Count(V, M, S) ≤ max_e C[e]}
8:          if |M| = 1 and M ∩ O ≠ ∅ then
9:              C ← MinConsZeroCount(X, M, S)
10:             B ← B ∪ {MinConsZeroCount(V, M, S) ≥ min_e C[e]}
11:             C ← MaxConsZeroCount(X, M, S)
12:             B ← B ∪ {MaxConsZeroCount(V, M, S) ≤ max_e C[e]}
13:             C ← MinConsOneCount(X, M, S)
14:             B ← B ∪ {MinConsOneCount(V, M, S) ≥ min_e C[e]}
15:             C ← MaxConsOneCount(X, M, S)
16:             B ← B ∪ {MaxConsOneCount(V, M, S) ≤ max_e C[e]}
17:         M_L ← M_L ∪ FILTER(B, 0, |⊗S|)
18:     return M_L
```

H. The algorithm

Now we have everything together to formally define our learning algorithm. The pseudo-code is listed in Algorithm 1.

Inputs given to the algorithm are $\mathbf{X}$ (data tensor), D (set of dimensions for $\mathbf{X}$) and optionally O (a subset of D for which ordering matters). In line 2, the learned model is initialized to an empty set. Then all possible combinations of M and S is enumerated by calling ENUMERATESPLITS. For each M and S, the maximum and minimum of $\text{Count}(\mathbf{X}, M, S)$ is calculated and converted into constraints, and added to the learned model (lines 5–7). As discussed in Section III-F, COUNT-OR checks whether it makes sense to call the ordered aggregation operators in line 8. If conditions are satisfied, it computes the ordered counts MaxConsOneCount, MinConsOneCount, MaxConsZeroCount, and MinConsZeroCount by replacing Sum in Eq. 1 by the ordered aggregation functions MaxConsOne, MinConsOne, MaxConsZero, and MinConsZero respectively, then it compute their bounds, and updates the learned model (lines 9–16). At line 17 it computes the minimum and maximum possible values that the bound can take (0 and $|\bigotimes S|$, respectively) and pass it to the FILTER procedure, implemented as per Section III-F, to discard all constraints that are trivially satisfied. Finally, it returns the filtered learned model at line 18.

I. *Exploiting background knowledge*

So far we have assumed that the learner has only access to the input schedules. In order to learn more fine-grained constraints, now we consider additional background knowledge in the form of *partitions* of entities. For instance, COUNT-OR may be given a partition of days into work days / holidays, or of nurses based on their competences. The idea is that different categories might follow different constraints, as is the case with constraints like "the maximum number of highly skilled nurses in each shift is at most 2".

To learn such constraints, we encode the background knowledge into a set of rank 2 tensors. For instance, given a categorization of nurses as high / low skilled, we create two rank 2 tensors, one for high skilled nurses and one for low skilled ones. These tensors are constructed in such a way that multiplying them with the input tensor **X** filters out the data unrelated to the value that the tensor represents. For example, multiplying **X** with the tensor corresponding to the High Skilled Nurses would filter out the data related to the Low Skilled Nurses. Once we get the data, we can apply our constraint learner defined in Section III-H to learn the specific constraints related to the specific value of the dimension.

We explain the construction of these dimension's value specific 2-d tensors through an example. Consider the data given in Table I alongside the following background knowledge: $\texttt{Nurse}_1$ and $\texttt{Nurse}_4$ are high skilled, while the other two nurses are low skilled. Let, $H = \{\texttt{Nurse}_1, \texttt{Nurse}_4\}$, be the list of High Skilled Nurses and $L = \{\texttt{Nurse}_2, \texttt{Nurse}_3\}$, be the list of Low Skilled Nurses. If $|\texttt{Nurses}| = n$ and $|H| = m$ then the shape of the tensor corresponding to High Skill will be $m \times n$, where for the ith row, all values will be zero except at the jth column such that $H_i = \texttt{Nurse}_j$. Using this construction we transform the given background knowledge into two tensors shown in Figure 1. Then the concept of tensor

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix} \qquad \begin{bmatrix}0 & 1 & 0 & 0\\0 & 0 & 1 & 0\end{bmatrix}$$

High Skilled Nurses — Low Skilled Nurses

Fig. 1: Converting background knowledge to a tensor.

multiplication is applied to get the filtered data that can give us more detailed constraints. For example, multiplying the tensor corresponding to high skilled nurses in Figure 1 with a small part of data gives the following result:

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix} \times$$

	Mon		
	S1	S2	S3
Nurse1	1	0	0
Nurse2	0	1	0
Nurse3	0	0	1
Nurse4	0	0	1

$\Rightarrow$

	Mon		
	S1	S2	S3
Nurse1	1	0	0
Nurse4	0	0	1

Now applying COUNT-OR on this data will generate constraints specific to High Skilled Nurses. We can do this for each category of nurses to learn specific constraints.

In the case where we have access to categorization of multiple dimensions, we can learn even more specific constraints. For example, if days are also categorized into weekdays and weekends, we can use the concepts discussed above recursively to learn constraints like "maximum number of high skilled nurses working on a weekday".

J. *Learning Background Knowledge*

In the previous section we explained how to exploit given categorization of variables. In this section we present a method to learn these categorization automatically. In Section III-C, we generate quantities of interest across each dimension, COUNT-OR uses these quantities as features for identifying clusters in the data. For example when $M = \{\texttt{Nurses}\}$, it considers all possible values for S and finds the quantities of interest for each nurse, such as, "number of working days in a week", "number of working shifts in a day", etc., These features are then used to calculate Hopkin's statistic [19] which quantifies the possibility of a cluster, if the value comes out to be higher than a certain threshold a clustering algorithm is applied to find the categorization in the data.

IV. EMPIRICAL ANALYSIS

We addressed the following research questions: **Q1**) Can COUNT-OR acquire the target model? **Q2**) How many input examples are needed to achieve a good accuracy? **Q3**) Is COUNT-OR efficient in practice? **Q4**) How COUNT-OR performs in the presence of background knowledge? **Q5**) Can COUNT-OR automatically identify the background knowledge? To answer these questions, COUNT-OR was used to recover several target scheduling models taken from the Second International Nurse Rostering Competition [9].

We run two sets of experiments in increasingly more complex settings. For each experiment, we first pick a target model M_T (that is, the set of constraints to be recovered) from the INRC-II benchmark instances and generate multiple solutions using it. These are then used as input schedules for COUNT-OR, which returns a learned model M_L. Finally, the target and learned model are compared to check whether the latter captures the essential features of the former.

We have divided the experiments at different levels. First we have two target models, one with no background knowledge, representing the examples of a single table with just the schedule, then there is another target model with the added background knowledge about nurses, categorizing them either as full time or part time employee. We further divide the experiments in two different scenarios. First, where M_T only includes constraints that can be represented using the tensor operations we defined. Second, where we add a few hard constraints which can not be represented using our definitions. Going further we evaluate COUNT-OR on three different scenarios for each constraint set discussed above by changing the number of nurses and bounds for the constraints, these variations are meant to simulate hospitals of different sizes: a small hospital (10 nurses, 28 days, 4 shifts), a medium sized hospital (31, 28, 4), and a large hospital (49, 28, 4). To

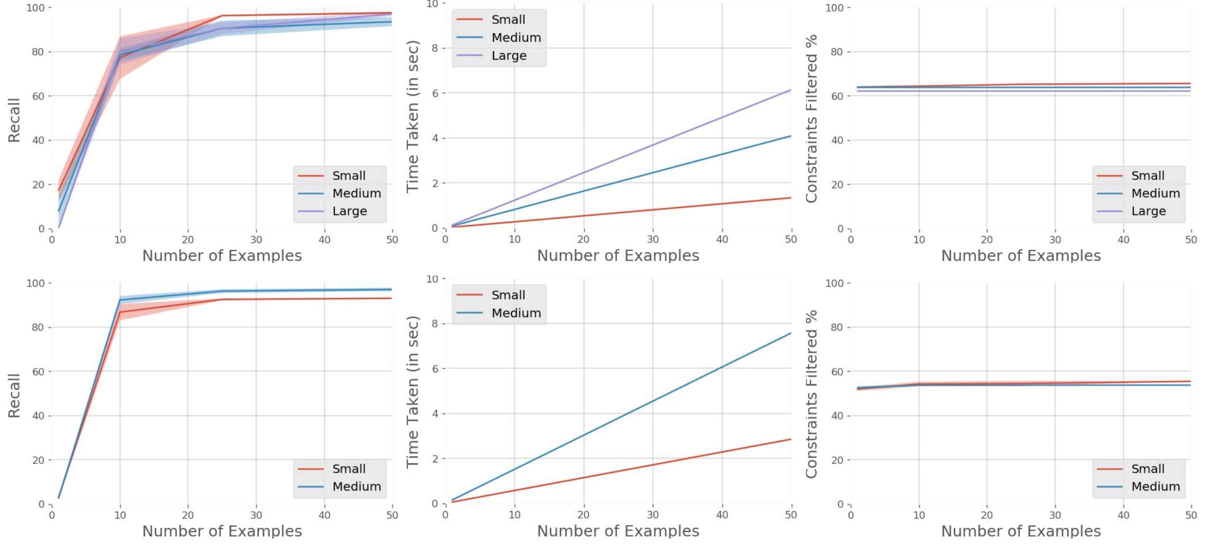

Fig. 2: Results for representable hard constraints with no background knowledge (top) and nurse skill information (bottom). From left to rigth: recall, runtime, number of constraints filtered.

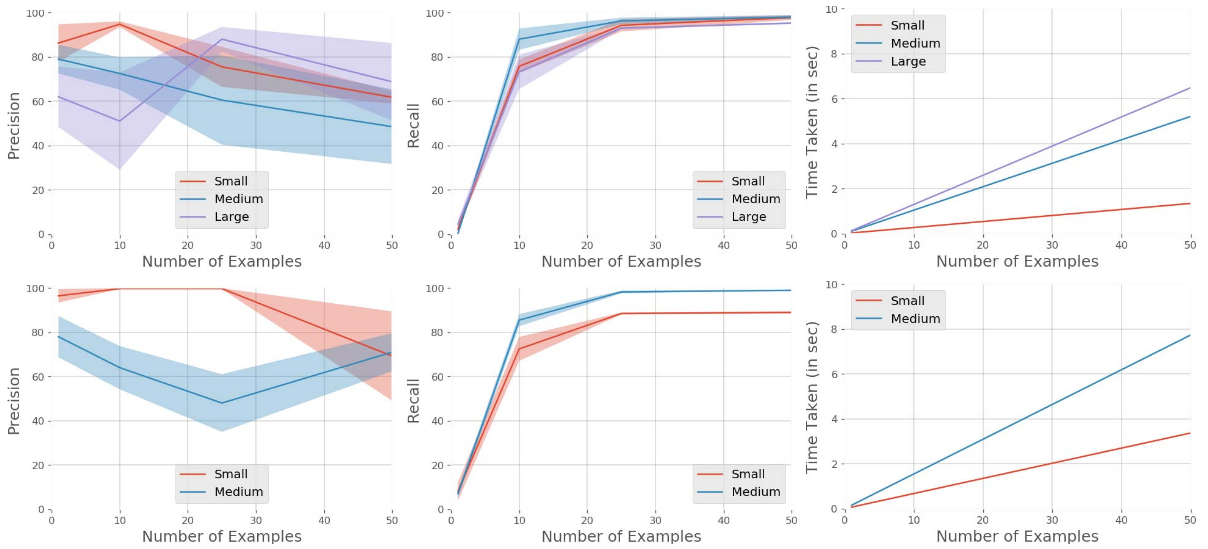

Fig. 3: Evaluation with added non-representable hard constraints. Top: No Background Knowledge, Bottom: With Background Knowledge. From left to rigth: precision, recall, runtime.

design these models, we used "sprint", "medium" and "large" example instances from the Nurse rostering competition [20], which represent models of realistic sizes.

For each model M_T, we used Gurobi to generate 10,000 solutions, then we split the dataset into five folds and fed COUNT-OR with $n \in \{1, 2, 10, 25\}$ random examples from one fold as training set, while using the union of the other four folds for performance evaluation. For each learned model M_L, we measured its recall and precision with respect to the hidden model M_T: $\mathrm{Pr} = |\mathrm{Sol}(M_T) \cap \mathrm{Sol}(M_L)|/|\mathrm{Sol}(M_L)|$ and $\mathrm{Rc} = |\mathrm{Sol}(M_T) \cap \mathrm{Sol}(M_L)|/|\mathrm{Sol}(M_T)|$. Here $\mathrm{Sol}(M)$ is simply the set of solutions of model M. Computing these quantities is not trivial, so we estimate them using sampling: for the recall, we generate 10,000 examples using M_T, and check how many of these examples satisfy M_L. The precision is computed analogously. The dependency of COUNT-OR on Gurobi restricted the experiments that we could do. For the models with background knowledge and simulating large hospital instance, Gurobi couldn't generate 10000 solutions in real time with the available resources. So we have removed just this instance from our experiments.

a) Evaluation for target model with only representable hard constraints: In this setting, all the constraints in the target model are representable by our constraint language. As

a consequence, COUNT-OR achieves 100% precision for all target models in this setting. The recall, reported in Figure 2 (top), however is not 100%. The x-axis is the number of input examples provided to COUNT-OR, while the y-axis is the average recall. We can see that when learning from just 1 example the recall is quite low, but as we increase the number of examples to 25 we achieve $\sim 95\%$ recall in both the cases: data with background knowledge and data without background knowledge. The time taken to learn the constraints, see Figure 2 (middle), increases linearly with the number of examples, and for 50 examples in the single table case, when the recall is $\sim 95\%$, the time taken on average is around 3 seconds while for the multiple table it takes on average 5 seconds to learn from 50 examples. A 100% precision in both the cases ensures that the solution generated by M_L will always satisfy all the constraints in M_T. In Figure 2 (right) we have reported the performance of our filtering algorithm, y-axis representing the percentage of irrelevant constraints filtered using filtering rules defined in section III-F. On an average we are able to filter around $\sim 60\%$ of the irrelevant and trivially satisfied constraints.

b) Evaluation for target model with some added non-representable hard constraints: For the next set of experiment we added 20% constraints which can't be represented using the operations that we defined, so we expect the precision to drop in this case. In Figure 3 we can see that precision is still more than $\sim 80\%$ on average. Recall is more or less same as the previous case, reaching a level of $\sim 95\%$ when number of input examples is greater than 25. For the experiments with background knowledge we see a slight drop in recall compared to the previous case, still the average recall is $\sim 85\%$. When using 50 examples to learn the constraints, average time taken in this case is around 4 seconds for data without background knowledge and around 5 seconds for data with background knowledge. Adding non-representable hard constraints in the data doesn't make any visible changes to the performance of the filtering algorithm. We observe a similar graph as in the previous case with a filtering rate of around $\sim 60\%$.

c) Learning Background Knowledge: We used the setting with background knowledge, i.e, categorization of nurses as high or low skilled. We used a cutoff of 75% for Hopkin's Statistic to identify dimensions which might have clusters. COUNT-OR was able to identify the dimensions which might have cluster with 100% accuracy in all the cases. Further it was able to correctly cluster the identified dimensions with 100% accuracy for the small and medium case and 90% for the large case.

V. CONCLUSION

We introduced COUNT-OR, a novel constraint learning method tailored for nurse rostering problems. Given a set of example schedule, COUNT-OR computes many quantities of interest (e.g. the number of employees needed each day) by applying aggregation operators to a tensor representation of the schedules. COUNT-OR also supports background knowledge if available, and can acquire it automatically otherwise. Our empirical evaluation on benchmark rostering instances shows that COUNT-OR can easily and quickly recover the constraints appearing in real-world nurse rostering problems. Future work includes support for soft constraints, which can encode preferences among alternative feasible schedules, and extending the palette of aggregation operators in order to uncover more general quantities of interest.

REFERENCES

[1] P. Smet, P. De Causmaecker, B. Bilgin, and G. V. Berghe, "Nurse rostering: a complex example of personnel scheduling with perspectives," in *Automated Scheduling and Planning*. Springer, 2013, pp. 129–153.

[2] E. K. Burke, P. De Causmaecker, G. V. Berghe, and H. Van Landeghem, "The state of the art of nurse rostering," *Journal of scheduling*, vol. 7, no. 6, pp. 441–499, 2004.

[3] L. De Raedt, A. Passerini, and S. Teso, "Learning constraints from examples," in *Proceedings of AAAI'18*, 2018.

[4] P. De Causmaecker and G. V. Berghe, "A categorisation of nurse rostering problems," *Journal of Scheduling*, vol. 14, no. 1, pp. 3–16, 2011.

[5] J. H. Kingston, G. Post, and G. V. Berghe, *A Unified Nurse Rostering Model Based on XHSTT*, PATAT 2018: Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling, ISBN: 978-0-9929984-2-4, pages 81-96.

[6] J. H. Kingston, *Modelling History in Nurse Rostering*, PATAT 2018: Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling, ISBN: 978-0-9929984-2-4, pages 97-111.

[7] S. Kolb, S. Paramonov, T. Guns, and L. De Raedt, "Learning constraints in spreadsheets and tabular data," *Machine Learning*, vol. 106, no. 9-10, pp. 1441–1468, 2017.

[8] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[9] S. Ceschia, N. D. T. Thanh, P. D. Causmaecker, S. Haspeslagh, and A. Schaerf, "Second international nurse rostering competition (INRC-II) - problem description and rules -," *CoRR*, vol. abs/1501.04177, 2015. [Online]. Available: http://arxiv.org/abs/1501.04177

[10] L. G. Valiant, "A theory of the learnable," *Communications of the ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[11] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19, pp. 629–679, 1994.

[12] N. Lavrac and S. Dzeroski, "Inductive logic programming," in *WLP*. Springer, 1994, pp. 146–160.

[13] C. Bessiere, A. Daoudi, E. Hebrard, G. Katsirelos, N. Lazaar, Y. Mechqrane, N. Narodytska, C.-G. Quimper, and T. Walsh, "New approaches to constraint acquisition," in *Data mining and constraint programming*. Springer, 2016, pp. 51–76.

[14] S. Teso, R. Sebastiani, and A. Passerini, "Structured learning modulo theories," *Artificial Intelligence*, vol. 244, pp. 166–187, 2017.

[15] T. P. Pawlak and K. Krawiec, "Automatic synthesis of constraints from examples using mixed integer linear programming," *European Journal of Operational Research*, vol. 261, no. 3, pp. 1141–1157, 2017.

[16] T. P. Pawlak, "Synthesis of mathematical programming models with one-class evolutionary strategies," *Swarm and evolutionary computation*, vol. 44, pp. 335–348, 2019.

[17] N. Beldiceanu and H. Simonis, "A Model Seeker: extracting global constraint models from positive examples," 2012, pp. 141–157.

[18] M. Kumar, S. Teso, and L. De Raedt, "Acquiring integer programs from data," in *Proceedings of the Twenty-Eight International Joint Conference on Artificial Intelligence,{IJCAI-19}*. International Joint Conferences on Artificial Intelligence Organization, 2019.

[19] A. Banerjee and R. N. Dave, "Validating clusters using the hopkins statistic," in *2004 IEEE International Conference on Fuzzy Systems (IEEE Cat. No.04CH37542)*, vol. 1, July 2004, pp. 149–153 vol.1.

[20] "The Second International Nurse Rostering Competition website," https://bit.ly/2IyObjD, accessed: 2018-12.