Automating Personnel Rostering by Learning Constraints Using Tensors

Mohit Kumar, Stefano Teso, Patrick De Causmaecker, Luc De Raedt

KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract

Many problems in operations research require that constraints be specified in the model. Determining the right constraints is a hard and laborsome task. We propose an approach to automate this process using artificial intelligence and machine learning principles. So far there has been only little work on learning constraints within the operations research community. 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 we have 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. 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.

Keywords: Constraint Learning, Inductive Learning, Artificial Intelligence, Machine Learning, Tensor, Personnel Rostering

2010 MSC: 68T05, 90C90, 97M40

1. Introduction

Constraints are pervasive in Operations Research 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 "the maximum number of working days for a nurse ≤ 5 per week". As the number of nurses and the complexity of the constraints increases, generating the schedule manually becomes impractical. Hospitals leverage Operations Research (OR) tools to produce the schedules automatically, but modeling the constraints themselves is a complicated task. Obtaining a realistic model of the rostering task often requires to hire domain experts to manually design the constraints and debug the resulting model. This step 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 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.

There are a number of artificial intelligence approaches that learn constraints from examples of known feasible (and infeasible) variable assignments [3]. However, rostering problems—like many other OR applications—possess two features that existing approaches can not handle. First, OR problems usually involve numerical quantities, for instance the number of shifts worked or the number of workers assigned to a task. Classical learning approaches like Conacq [4] and Inductive Logic Programming [5] focus on Boolean variables only, and it is unclear how to introduce numerical terms in this context. Second, rostering problems are inherently *multi-dimensional* [6]. To see what we mean, consider the nurse schedule shown in Table 1 (top). For each combination of nurse, day and shift, the value of 1 indicates that the 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 and behave like different dimensions. While some recent constraint learning approaches can deal with numerical data (e.g., [7]), they are not designed to handle more than two dimensions. Please see Section 2 for a more detailed discussion.

We propose learning *COnstraint UsiNg TensORs* (COUNT-OR), a novel constraint

learning approach that leverages tensors[1] for capturing the inherent structure and dimensionality of the schedules, and that can easily deal with numerical information. Given a set of past schedules, the goal of COUNT-OR is to extract numerical 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 schedule(s), aggregates them through tensor operations, and then computes bounds for the aggregates to generate candidate numerical constraints. Some simple filtering strategies are applied to prune irrelevant and trivially satisfied candidates.

Our key contributions are the following:

1. A tensor representation of schedules and constraints that allows to easily represent and reason over real-world personnel rostering instances.
2. A novel constraint learning algorithm, COUNT-OR, which uses tensor extraction and aggregation operations to learn the constraints hidden in the input schedules.
3. An extension of COUNT-OR to cases where additional background knowledge (e.g., nurse skill levels) is available.
4. An empirical evaluation of the learning accuracy and run-time of COUNT-OR on real-world nurse rostering models taken from Nurse Rostering Competition[2] (INRC-II [9]), showing that our approach can indeed 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 3 and evaluate it empirically on real-world nurse rostering problems in Section 4. We conclude with some final remarks and hints at promising extensions.

[1] In this paper, tensors refer to n-dimensional generalizations of matrices, as normally done in AI [8]. These are not to be confused with tensor (fields), as used in physics.

[2] URL: `http://mobiz.vives.be/inrc2/`

2. Related Work

Constraint learning is a core artificial intelligence and machine learning task. Here we briefly review the basic concepts and discuss those existing methods that are more relevant to the OR domain. For a more general overview of the topic, see [3] and references therein.

The most basic form of constraints are propositional logic formulae, or *concepts*. To see the connection between formulae and constraints, note that formulae can be used to define the feasibility of interpretations (truth value assignments to all the domain variables): an interpretation is feasible if it does satisfy the formula and infeasible otherwise. Indeed, 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 prespecified 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 infeasible 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 [5, 11]. 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 [4] follow the same setup as concept learning, but are 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 integers). These approaches maintain a candidate model and iteratively refine it to account for the examples that are inconsistent with it. The main issue of these approaches is computational, as checking whether an example is inconsistent requires one to invoke a constraint solver. Since the number of variables (e.g. nurse assignments in a roster) increases exponentially with the number of dimensions in the data (roughly # of days $\times$ # of shifts $\times$ # of nurses), and since checking satisfaction is in general NP-hard, these approaches do not scale to realistically-sized rostering

instances. In contrast, COUNT-OR does not require costly satisfiability checks.

There exists a class of learning approaches that can potentially deal with high-dimensional numerical domains [12, 13]. 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, just like the methods of Biessere *et al.*, these approaches also require to invoke a satisfaction (or optimization) oracle, and thus do not scale to larger learning settings. Other recent works on constraint learning (e.g. [7]) can also deal with numerical constraints, but are designed specifically for tabular data.

To the best of our knowledge, the only other constraint learning approach that uses a tensor representation is ModelSeeker [14], which aims at learning *global* constraints from examples. In order to do so, ModelSeeker 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 based on similar ideas, 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.

3. Method

In this paper we consider a very practical learning problem. We assume that a hospital has access to a set of working past schedules compiled by hand, e.g., by the administration staff or by the head nurse. The problem we address can be stated as follows: **given** a set of past schedules, **find** a rostering model that satisfies all of them. The learned model can then be used to automatically produce new schedules that satisfy the same constraints observed to hold in the examples. Of course, extra constraints—for instance, contractual obligations about corner cases that rarely occur in practice—can be added to the learned model, and domain experts can be asked to improve or validate it. The aim of COUNT-OR is to induce a working model that can be either used as is or further improved, as to simplify the modeling process.

Before detailing our technical contribution, we make some important observations.

First of all, in contrast with many learning approaches, we do not make strong assumptions on the distribution of the examples. Indeed, many learning algorithms assume that the examples are independent and identically distributed. While this assumption greatly simplifies the design and analysis of the algorithms, it is also often violated in practice. This is also the case in constraint learning for nurse rostering. We merely assume that 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 a laborious process of trial-and-error, where the head nurse or administrative staff tried out different alternatives and kept the ones that worked best. For the same reason, we also expect the examples to be sub-optimal (according to any given optimality criterion). Of course, since COUNT-OR induces a model that mimics the past schedules, the higher the desirability of the provided examples, the better the schedules generated by the learned model.

We also note that the example schedules likely reflect contractual obligations and other regulations. Since these are always known in advance, it may seem pointless to learn a rostering model from examples, since it might only capture well-known constraints. This is not the case, for three reasons. First, even though all employees should in princple know and understand the contract, it may be nevertheless difficult for non-experts to encode the contract into a working OR model. COUNT-OR takes care of this automatically. Second, we expect the examples to reflect hard constraints not appearing in the contract. For instance, the hospital may only need two specialized workers (e.g., physiotherapists) at a time, because there are only a limited number of facilities for them. Knowing that it holds may avoid generating infeasible (and costly) schedules where more than two physiotherapists are allocated. This constraint can be easily acquired by COUNT-OR. Third, and most importantly, real-world schedules might *not* respect contractual obligations or other regulations. Using COUNT-OR with such "imperfect" examples allows to capture real trends occurring in the working place, find out existing practices that violate the contract, or—by including contractual obligations into the learned model—build a model that blends known-working practices and strict regulations.

3.1. High-level overview

Now we proceed at giving a high-level overview of the proposed learning algorithm. More specifically, COUNT-OR consists of four steps:

1. **Tensorization**. In the first step, each input schedule is mapped to a tensor. The elements of this tensor represent, in the simplest case, which employees worked in which shift. This step is described in detail in Section 3.2.
2. **Aggregation**. Next, the algorithm enumerates all sub-tensors and summarizes them by applying one or more aggregation operations. This way it obtains several quantities of interest, for instance, the number of employees each day or the number of working days for each employee. See Section 3.3.
3. **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 3.5.
4. **Filtering**. Finally, trivially satisfied candidates are filtered out to produce a set of consistent constraints. These can be readily fed to any constraint solver to generate new schedules consistent with the examples. See Section 3.6.

For the sake of clarity, while describing these steps we assume that the algorithm is working with a *single* example schedule. Clearly, this is not in general the case, and our algorithm is well-equipped to deal with several input schedules. The simple modifications required to handle this case are described in Section 3.7. Finally, in Section 3.9 we will discuss how the learning process can be extended to find more fine-grained constraints by incorporating additional background knowledge, e.g., nurse skills.

Notation. There is some overlap between OR and AI terminology. In order to prevent any ambiguity, we adopt the following convention. When unqualified, the term *problem* is used to refer to the inference problem of producing feasible schedules for a particular rostering instance, and *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. These past schedules will be interchangeably referred to as *input schedules* or simply *examples*.

	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

	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

	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 1: Top: example nurse schedule with three dimensions: four nurses, three days, and three shifts per day. Bottom left: same schedule as above, but the cells identified by the sub-tensor $\mathbf{X}[\texttt{Day}_1]$ have been highlighted. Bottom right: same, except that $\mathbf{X}[\texttt{Nurse}_1, \texttt{Day}_1]$ is highlighted. Best viewed in color.

3.2. Mapping Schedules to Tensors

Consider the example nurse schedule in Table 1 (top). Here the rows represent different nurses and the columns represent shifts on different days. A value of 1 means that the corresponding nurse worked in that shift on that particular day, while 0 means that the nurse did not work. This is a very natural representation for schedules. More formally, let $D = \{D_1, D_2, \ldots, D_n\}$ be the scheduling *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 2). 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 indi-

	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

$\Rightarrow$

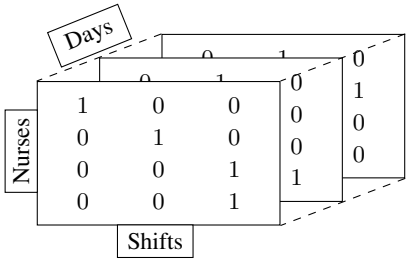

Table 2: Tensor representation of a schedule

cate 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 in Table 1, bottom left). 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 $\texttt{Nurse}_1$ in $\texttt{Day}_1$ (cf. Table 1, bottom right). 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'.

3.3. 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 $\mathbf{X}[e]$, that is, $\mathbf{Y}[e] =$ sum of values in $\mathbf{X}[e]$. 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:

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

For instance, for $M = \{\texttt{Nurses}\}$ and $S = \{\texttt{Days}\}$, 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 that the tensor output by Count($\mathbf{X}, M, S$), indexed by the values in S, encodes the total number of distinct employees working on different days: $\mathbf{Z} = [4 \;\; 3 \;\; 4]$.

3.4. 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 holidays for an employee" or "the 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', 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.

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., $|D \setminus D'| = 1$.

3.5. 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 and for each quantity of interest (e.g. Sum or Count), first it calculates Count($\mathbf{X}, M, S$), and from there it computes the empirical lower and upper bounds as:

$$\bot^{\text{Count}}_{M,S} = \min_e \left(\text{Count}(\mathbf{X}, M, S)\right)[e], \qquad \top^{\text{Count}}_{M,S} = \max_e \left(\text{Count}(\mathbf{X}, M, S)\right)[e]$$

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 3 (among others).

Once the upper and lower bound of some quantity (e.g. 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 the previous equation $\mathbf{X}$ is the tensor *constant* representing the input schedule. In contrast, in this equation $\mathbf{V}$ is a tensor *variable* whose value is being constrained.

M	S	Count($\mathbf{X}$, M, S)
{Days}	{Nurses}	# of working days / Nurse
{Days, Shifts}	{Nurses}	# of working shifts / Nurse
{Nurses}	{Days}	# of distinct employees / 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 / nurse
{Nurses}	{Days, Shifts}	# of nurses / shift each day

Table 3: A selection of quantities of interest representable by the Count function for different choices of M and S.

3.6. Dealing with irrelevant constraints

To learn the candidate constraints, we consider all possible combinations of M and S and then apply aggregation functions to the input tensor. Therefore, no matter what the input looks like, we will always obtain candidates for each constraint represented by our aggregation operators. In this process we might acquire 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 at all: the number of working days is always non-negative, hence the learned constraint is trivially satisfied.

We might also learn some meaningless constraints. Depending on the application domain, some combinations of M and S may 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. The fact that such combinations should be avoided can be introduced as background knowledge into our algorithm. 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, MaxConsZero, and MaxConsOne) we know that in our examples ordering of Nurses is not important as all the nurses are identical, so finding consecutive values of ones or zeros across Nurses does not make any sense. So we use the following method to neglect such constraints. 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 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. Second, for the lower bound, if $\bot^{\text{Count}}_{M,S} = 0$ then again we drop this constraint as it is an obvious lower bound. These same two rules hold when using the ordered aggregation functions instead of Sum.

These simple filtering rules can be surprisingly effective in practice. In our experiments with real-world scheduling problems, we noticed that they get rid of most of the irrelevant constraints, we have reported the numbers in Section 4, without having to tell explicitly the algorithm which constraints to ignore. 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 "natural", 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.

3.7. 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.

3.8. The algorithm

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

The algorithm takes as input the data tensor $\mathbf{X}$, a set of dimensions D, and the set of dimensions O for which ordering *does* matter (as per Section 3.6). First we initialize the learned model to the empty set at line 2. Then we obtain all the possible values of M and S by invoking the EnumerateSplits procedure and iterate over them. Then, for each M and S, we take the maximum and minimum of Count$(\mathbf{X}, M, S)$, convert the obtained bounds into constraints, and add the latter to the learned model (lines 5–7). Next, at line 8 we check whether it makes sense to consider the ordered aggregation operators for the given M and S. This is only the case if M contains only one dimension because ordering in multiple dimensions does not make much sense and also M must not contain dimensions for which ordering is not defined (as discussed in Section 3.6). If both conditions are satisfied, we compute 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 we compute their bounds, and update the learned model (lines 9–16). At line 17 we compute 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 3.6, to discard all constraints that are trivially satisfied. Finally, we return the filtered learned model at line 18.

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.

procedure FINDCONSTRAINTS($\mathbf{X}$, D, O)
  $\mathcal{M}_L \leftarrow \emptyset$
  for $M, S \in$ ENUMERATESPLITS(D) **do**
    $bounds \leftarrow \emptyset$
    $\mathbf{C} \leftarrow$ Count($\mathbf{X}, M, S$)
    $bounds \leftarrow bounds \cup \{\text{Count}(\mathbf{V}, M, S) \geq \min_e \mathbf{C}[e]\}$
    $bounds \leftarrow bounds \cup \{\text{Count}(\mathbf{V}, M, S) \leq \max_e \mathbf{C}[e]\}$
    if $|M| = 1$ and $M \cap O \neq \emptyset$ **then**
      $\mathbf{C} \leftarrow$ MinConsZeroCount($\mathbf{X}, M, S$)
      $bounds \leftarrow bounds \cup \{\text{MinConsZeroCount}(\mathbf{V}, M, S) \geq \min_e \mathbf{C}[e]\}$
      $\mathbf{C} \leftarrow$ MaxConsZeroCount($\mathbf{X}, M, S$)
      $bounds \leftarrow bounds \cup \{\text{MaxConsZeroCount}(\mathbf{V}, M, S) \leq \max_e \mathbf{C}[e]\}$
      $\mathbf{C} \leftarrow$ MinConsOneCount($\mathbf{X}, M, S$)
      $bounds \leftarrow bounds \cup \{\text{MinConsOneCount}(\mathbf{V}, M, S) \geq \min_e \mathbf{C}[e]\}$
      $\mathbf{C} \leftarrow$ MaxConsOneCount($\mathbf{X}, M, S$)
      $bounds \leftarrow bounds \cup \{\text{MaxConsOneCount}(\mathbf{V}, M, S) \leq \max_e \mathbf{C}[e]\}$
    $\mathcal{M}_L \leftarrow \mathcal{M}_L \cup$ FILTER($bounds, 0, |\bigotimes S|$)
  return $\mathcal{M}_L$

3.9. Exploiting background knowledge

So far we have assumed that all the information available to the learner is contained in the input schedules, but sometimes additional background knowledge may be available, for instance a categorization of days into work days and holidays or a categorization of nurses on the basis of their skills (as in Table 1). Clearly, different categories may be affected by different constraints. We introduce an extension of COUNT-OR that leverages this kind of background knowledge to learn more fine-grained constraints. For example, skill levels can be utilized to learn 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 categorization of nurses as high skilled or 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, the tensor corresponding to the High Skilled Nurses would be such that, when we multiply **X** with this constructed tensor, it filters out the data related to the Low Skilled Nurses. Once we get the specific data, we can apply our constraint learner defined in Section 3.8 to learn the specific constraints related to the specific value of the dimension, in this case the High Skilled Nurses.

Now, let us understand the construction of these dimension's value specific 2-d tensors through an example and then we will see how the tensor product works. Consider the data given in Table 1 alongside with the following background knowledge: $\texttt{Nurse}_1$ and $\texttt{Nurse}_4$ are high skilled, while the other two nurses are low skilled.

First we identify the number of distinct types of nurses, which in this case are 2, so we would need two 2-d tensors, one for the type High Skilled and the other for the 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.

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

High Skilled Nurses Low Skilled Nurses

Figure 1: Converting background knowledge to a tensor.

Once we convert the data into these tensors, we apply the concept of tensor multiplication to get the filtered data that can give us more detailed constraints. To understand

this, first let us see how tensor multiplication works, it is an extension of simple matrix multiplication to higher dimensions. In matrix multiplication, matrices are tensors of 2 dimensions, although for our algorithm, the matrices that we get from the background knowledge are 2 dimensional, but the input data, **X** can be a tensor of any dimensions, so let us see how we multiply two tensors of different dimensions through an example, let's say we want to multiply a rank-2 tensor A of shape [I, J] with a rank-3 tensor B of shape [J, K, L], we will get a rank-3 tensor C of shape [I, K , L] and the values in this new tensor is calculated based on the following formula:

$$C[i,k,l] = \sum_{j} A[i,j] \cdot B[j,k,l]$$

Let us see what we get when we apply this tensor product on our example. So, let's take the first tensor from Figure 1, which represents the high skilled nurses and let's multiply this to a small part of the data below:

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

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

$=$

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

So, when multiplied with the tensor representation of the high skilled nurses, we get a subset of the data with all the information for just the high skilled Nurses, now if we apply our constraint learner on this data we can learn all the constraints we learned earlier, but just specific to High Skilled Nurses. We can do this for each type of nurses to learn specific constraints.

In the example given above, we assumed that both type of nurses (high skilled and low skilled) have some specific skill set and cannot replace each other, but sometimes it can happen that a high skilled nurse has all the required skill set of a low skilled nurse. In this case, when creating matrix for the low skilled nurses, we include the high skilled

nurses as well. So, to make it more generic, if there is a ranking of nurses, such that a higher ranked nurse possesses all the skills of lower ranked nurses, when creating the multiplication matrix for a particular rank we include all the higher ranked nurses as well.

Other thing that we did not consider in the example above is background knowledge of multiple dimensions. In the case where we have access to categorization of multiple dimensions, we can learn even more specific constraints, for example, if we have another table categorizing weekdays and weekends, we can learn constraints like "maximum number of high skilled nurses working on a weekday".

4. Empirical Analysis

In this section we empirically address the following research questions:

Q1. Does the learned model produce schedules similar to those given by the target model?

Q2. How many example schedules does the learner need to achieve a good accuracy?

Q3. Does the algorithm scale to real world target models?

Q4. How does the algorithm perform when including background knowledge?

To this end, we apply COUNT-OR to recover several target scheduling models taken from the Second International Nurse Rostering Competition [9] which we describe in the next paragraph. In particular, we run two sets of experiments where COUNT-OR is asked to learn constraints in increasingly more complex settings. For each experiment, we first pick a target model $\mathcal{M}_T$ (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 an input schedules for COUNT-OR, which returns a learned model $\mathcal{M}_L$. Finally, the target and learned model are compared to check whether the latter captures the essential features of the former. Of course, in real-world use cases $\mathcal{M}_T$ is unknown. Here we assume to have access to the true target model $\mathcal{M}_T$ for benchmarking purposes only.

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, representing the examples of multiple tables. For both these models, 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 some 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 design these models, we used "sprint", "medium" and "large" example instances from the Nurse rostering competition[3], which represent models of realistic sizes.

To quantify the quality of the learned model for each of these experiments we follow these steps: First we pick the corresponding target model M_T, then use the Gurobi solver to generate 10,000 solutions for M_T, and use increasingly larger random subsets of this sample as input to COUNT-OR to learn the constraints. Then we evaluate the model learned by the algorithm, M_L, by measuring its precision and recall with respect to the target model M_T:

$$\text{pr} = \frac{|\text{Sol}(M_T) \cap \text{Sol}(M_L)|}{|\text{Sol}(M_L)|} \quad \text{rc} = \frac{|\text{Sol}(M_T) \cap \text{Sol}(M_L)|}{|\text{Sol}(M_T)|}$$

Here $\text{Sol}(M) = \{x : M \models x\}$ is simply the set of solutions of model M. Computing these quantities is not trivial, so we estimate them using sampling. To compute 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. For our experiments, higher precision means more solutions generated by the learned model satisfy all the constraints given in the target model M_T, while a high recall simply means that the learned model is able to capture most of the solution space of the target model.

[3] URL: `https://bit.ly/2IyObjD`

We can notice that COUNT-OR has a dependency on a constraint solver, as it only learns the constraints and then passes it on to a solver to generate the solutions, for the experiments in this paper we used Gurobi as a solver. This dependency on the solver restricted the experiments that we could do. For the models with background knowledge and simulating large hospital instance, Gurobi wasn't able to generate 10000 solutions in real time with the available resources. So we have removed just this instance from our experiments.

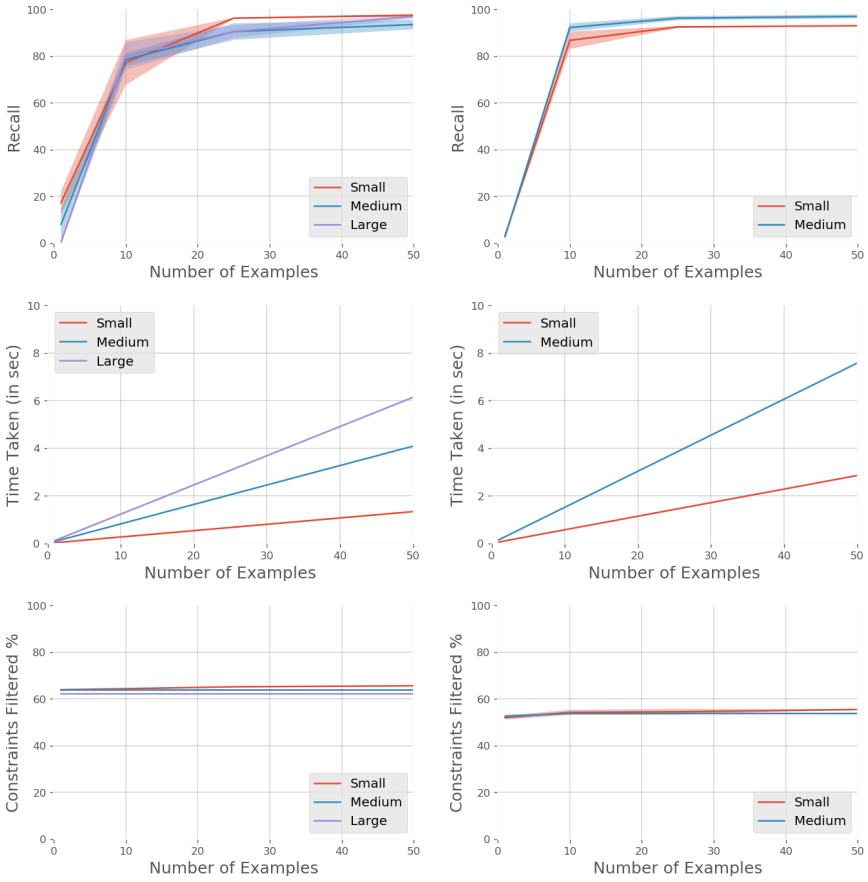

Figure 2: Evaluation for representable hard constraints. Left: No Background Knowledge, Right: With Background Knowledge

Evaluation for target model with only representable hard constraints. Recall that, in this setting, all the constraints in the target model are representable by our constraint language. As a consequence, COUNT-OR always achieves 100% precision for all target models in this setting, this follows from the fact that we can learn all the constraints of these target models, it is just that the bounds we learn might be more constrained. The recall, reported in Figure 2 (top), is more interesting. The x-axis is the number of input examples provided to COUNT-OR, while the y-axis is the average recall computed over 5 different subsets of examples. 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 (bottom) we have reported the performance of our filtering algorithm. In this graph, the y-axis represents the percentage of irrelevant constraints filtered using filtering rules defined in section 3.6. On an average we are able to filter around $\sim 60\%$ of the irrelevant and trivially satisfied constraints.

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 in this case is not 100% but still we get more than $\sim 80\%$ precision on average. Recall is more or less same as the previous case, reaching a level of $\sim 95\%$ for number of examples 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 of non-representable hard constraints in the data doesn't make any visible changes to the performance of the

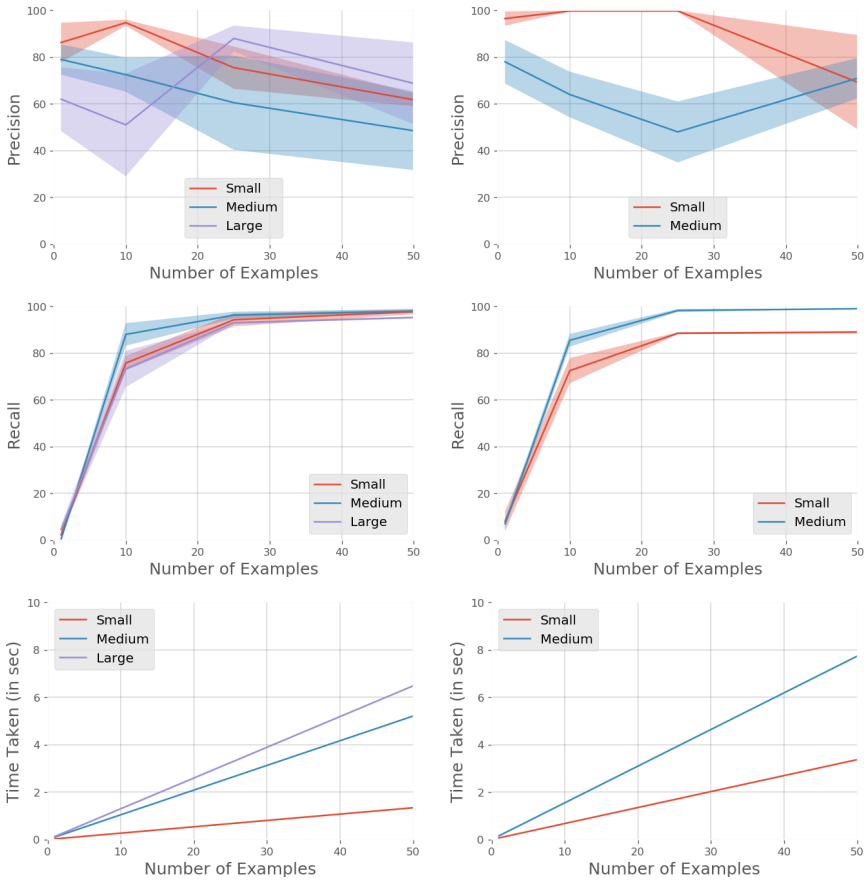

Figure 3: Evaluation for added non-representable hard constraints. Left:No Background Knowledge, Right:With Background Knowledge

filtering algorithm. We observe a similar graph as in the previous case with a filtering rate of around $\sim 60\%$.

5. Conclusion

We proposed a novel constraint learning approach, named COUNT-OR, specifically tailored for personnel rostering problems. Given examples of past schedules, COUNT-OR leverages tensors and operations on them to uncover the constraints hidden in the examples. The key idea is that many quantities of interest, like maximum working

hours or the number of employees needed each day, can be readily computed by applying aggregation operators to a tensor representation of the example schedules. This naturally takes into account the intrinsic structure and dimensionality of the scheduling problem. We then extended our algorithm to introduce support for background knowledge, using tensor products to incorporate this information into the learning problem. 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.

Our approach is simple and easily extensible, and as such opens the door to more fine-grained constraint learning techniques. The most prominent extension involves support for soft constraints, which can encode preferences among alternative feasible solutions. For instance, in some cases the example schedules might be annotated by their desirability: schedules that did work out well or not (according to the administration, head nurse, or employees) could be labeled as such. This extra information would allow to learn an objective function that favors more desirable schedules. Another promising direction is to extend the tensor language of COUNT-OR to capture more diverse OR models, for instance, by introducing tensor aggregators intended for specific applications or domains, such as sport scheduling and transport problems. For example, the indicator function used in the Nonzero function can be made more general: if we have multiple tables of background information, let's say a categorization of `Days`into weekdays and weekends and a categorization into holidays and working days, then we can change the indicator function to only consider the rows where the day is either a holiday or a weekend. This would allow us to learn constraints like "the maximum number of nurses required on a holiday OR a weekend ≤ 3". We could handle all sorts of combinations. We will explore these research directions in future work.

6. Acknowledgements

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agree-

ment No [694980] SYNTH: Synthesising Inductive Data Models)". The authors are grateful to Samuel Kolb for interesting discussions related to constraint learning.

References

[1] P. Smet, P. De Causmaecker, B. Bilgin, 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, H. Van Landeghem, The state of the art of nurse rostering, Journal of scheduling 7 (6) (2004) 441–499.

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

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

[5] S. Muggleton, L. De Raedt, Inductive logic programming: Theory and methods, The Journal of Logic Programming 19 (1994) 629–679.

[6] P. De Causmaecker, G. V. Berghe, A categorisation of nurse rostering problems, Journal of Scheduling 14 (1) (2011) 3–16.

[7] S. Kolb, S. Paramonov, T. Guns, L. De Raedt, Learning constraints in spreadsheets and tabular data, Machine Learning 106 (9-10) (2017) 1441–1468.

[8] T. G. Kolda, B. W. Bader, Tensor decompositions and applications, SIAM review 51 (3) (2009) 455–500.

[9] C. Sara, et al., The second international nurse rostering competition, in: Proceedings of PATAT'14, 2014, pp. 554–556.

[10] L. G. Valiant, A theory of the learnable, Communications of the ACM 27 (11) (1984) 1134–1142.

[11] N. Lavrac, S. Dzeroski, Inductive logic programming, in: WLP, Springer, 1994, pp. 146–160.

[12] S. Teso, R. Sebastiani, A. Passerini, Structured learning modulo theories, Artificial Intelligence 244 (2017) 166–187.

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

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