Learning Non-Linear Constraints using Tensors

Mohit Kumar[1], **Stefano Teso**[1], **Luc De Raedt**[1]

[1] KU Leuven

{mohit.kumar,stefano.teso,luc.deraedt}@cs.kuleuven.be

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.

1 Introduction

Constraints are pervasive in practical scheduling and rostering problems. For example, hospitals usually generate a weekly schedule for their nurses based on constraints like the maximum number of working days for a nurse. As the number of nurses and the complexity of the constraints increases, however, generating the schedule manually becomes impossible. Organizations may hire domain experts to manually model the constraints, but this is expensive and time consuming. A tempting alternative is to employ constraint learning [**?**] to automatically induce the constraints from examples of past schedules.

Unfortunately, existing constraint learners are not tailored for this setting. Classical approaches like Conacq [**?**] and Inductive Logic Programming tools [**?**] focus on logical variables only, while scheduling constraints often include numerical terms. Very few approaches can handle this case. TaCLe [**?**] focuses on 2-D tabular data (Excel spreadsheets), while schedules are inherently multi-dimensional. To see what we mean, consider the nurse schedule shown in Table **??**. For each combination of nurse, day and shift the value of 1 represents that the nurse worked in that particular shift of that day, while a 0 means the nurse didn't work. It is easy to see that nurses, days, and shifts are independent of each and behave like different dimensions. ModelSeeker [**?**] is the only method that can handle such multi-dimensional structures, but it is restricted to global constraints only.

To address this issue, we propose COnstraint UsiNg TensORs (NL-COUNTOR), a novel constraint learning approach that leverages tensors for capturing the inherent structure and dimensionality of the schedules. In order to learn the constraints, NL-COUNTOR extracts and enumerates all (meaningful) slices 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. When increasing the number of dimensions in the example, the rank of the tensor representing the example will increase accordingly but the proposed method NL-COUNTOR remains unchanged, so NL-COUNTOR can easily scale to large and complex schedules. The number of candidate sub-tensors however increases exponentially with the number of dimensions of $\mathbf{X}$.

We make the following key contributions: (**1**) A tensor representation of schedules and constraints appropriate for real-world personnel rostering problems. (**2**) A novel constraint learning algorithm, NL-COUNTOR, which uses tensor extraction and aggregation operations to learn the constraints hidden in the input schedules. (**3**) An empirical evaluation on real-world nurse rostering problems.

The paper is structured as follows. We present the method in Section 2, followed by evaluation on example instances in Section **??**. We conclude with some final remarks in Section **??**.

2 Method

Most of the constraints in the prevalant benchmark scheduling problems are non-linear in nature and share an inherent

structure. In this paper, we first define a mathematical structure to represent these constraints and then define an algorithm to learn these structures efficiently. Thus automating the rigourous task of desiging and coding constraints to reach the solution.

We introduce a running example before delving into more details. Consider a Nurse rostering model with the following variables.

$$\mathbf{X}_{n,s,d} = \begin{cases} 1, & \text{if nurse } n \text{ is allocated shift } s \text{ on day } d \\ 0, & \text{otherwise.} \end{cases}$$

$$\mathbf{H}_n = \begin{cases} 1, & \text{if nurse } n \text{ is high skilled.} \\ 0, & \text{otherwise.} \end{cases}$$

$$\mathbf{M}_n = \begin{cases} 1, & \text{if nurse } n \text{ is medium skilled.} \\ 0, & \text{otherwise.} \end{cases}$$

$$\mathbf{L}_n = \begin{cases} 1, & \text{if nurse } n \text{ is low skilled.} \\ 0, & \text{otherwise.} \end{cases}$$

$\mathbf{R}_{s,d}$ = Maximum number of high or medium skilled nurses required in shift s on day d

Considering these variables, a constraint like "the maximum number of high or medium skilled nurses working in a shift" can be represented as:

$$\sum_n \mathbf{X}_{n,s,d} \times \mathbf{H}_n + \sum_n \mathbf{X}_{n,s,d} \times \mathbf{M}_n \leq \mathbf{R}_{s,d} \quad \forall s, d \quad (1)$$

Each term in the left hand side of Eq 1 can be represented by:

$$\bigotimes_{\mathbf{S}}(\overline{\mathbf{X}}, \overline{\mathbf{M}}) = \sum_{\mathbf{S}} \prod_{\mathbf{X},\mathbf{M} \in \overline{\mathbf{X}} \odot \overline{\mathbf{M}}} \mathbf{X}_{\mathbf{M}} \quad (2)$$

Here, $\odot$ is an element wise concatenation, for example: **Mohit: needs fixing**

$$\{X_1, X_2\} \odot \{M_1, M_2\} = \{(X_1, M_1), (X_2, M_2)\}$$

Taking $\overline{\mathbf{X}} = \{\mathbf{X}_{n,s,d}, \mathbf{H}_n\}$, $\overline{\mathbf{M}} = \{\{n, s, d\}, \{n\}\}$ and $\mathbf{S} = \{n\}$ will give us the first term in Eq 1, i.e. $\bigotimes_{\mathbf{S}}(\overline{\mathbf{X}}, \overline{\mathbf{M}}) = \sum_n \mathbf{X}_{n,s,d} \times \mathbf{H}_n$.

Let $\mathbb{X}$ represents the set of all tensors, $\mathbf{P}(\mathbb{X})$ represents the power set of $\mathbb{X}$ and $\mathbf{E} \subseteq \mathbf{P}(\mathbb{X})$, then the following inequality captures the type of constraint we saw in Eq 1.

$$\sum_{\overline{\mathbf{X}}_i \in \mathbf{E}} \bigotimes_{\mathbf{S}_i}(\overline{\mathbf{X}}_i, \overline{\mathbf{M}}_i) \leq \mathbf{Z}_M$$

$$\forall \bigcup_i (\bigcup_{\mathbf{M} \in \overline{\mathbf{M}}_i} \mathbf{M}/\mathbf{S}_i) \bigcup M \quad (3)$$

Each constraint in Eq 3 can be uniquely identified by the following parameters:

- $\mathbf{E}, \overline{\mathbf{M}}_i, \mathbf{S}_i, \mathbf{Z}, \mathbf{M}$

For example, choosing the following assignments will give us the constraint represented by Eq 1

$$E = \{(X, H), (X, M)\}$$
$$Z = R$$
$$\overline{M}_{(X,H)} = \overline{M}_{(X,M)} = \{(n, s, d), (n)\}$$
$$S_{(X,H)} = S_{(X,M)} = \{n\}$$
$$M = \{s, d\}$$

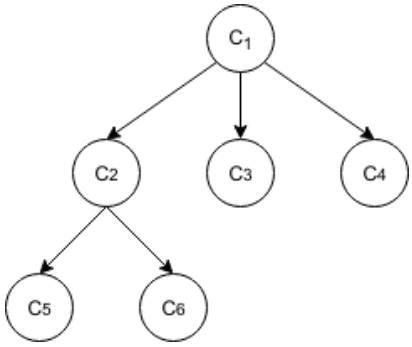


Figure 1: Constraint Dependency Tree: The root represents the most general constraint and as we keep going down the constraints become more specific.

By enumerating through the possible values of these variables, we can enumerate through all the possible constraints. However, that is not a feasible solution, as the number of combinations for this structure is huge and enumerating through all the possibilities will not scale for the real world problems. In this paper we define a two step solution to tackle this problem:

2.1 Signatures

Signatures are a pro active measure to refine the set of constraints to be enumerated. We define the following conditions which must be satisfied by a constraint in order for it to be a valid constraint. Let N_y represents the dimension set of $\mathbf{Y}$

- $M_{l,y} \subseteq N_y \quad \forall l \in E \bigcup F$
- $S_l \subseteq \bigcup_{\mathbf{Y} \in l} M_{l,y}$ for all $l \in E \bigcup F$
- $\bigcup_{\mathbf{Y} \in l} M_{l,y}/S_l = \bigcup_{\mathbf{Y} \in l'} M_{l,y}/S_{l'} \quad \forall l, l' \in E \bigcup F$
- $\bigcup_{\mathbf{Y} \in l} M_{l,y}/S_l \subseteq M$ or $\bigcup_{\mathbf{Y} \in l} M_{l,y}/S_l \supseteq M \quad \forall l \in E \bigcup F$
- $\mathbf{Y} \neq \mathbf{Y}'$ for any $\mathbf{Y}, \mathbf{Y}' \in l \quad \forall l \in E \bigcup F$
- $|E| + |F| \leq 2$
- $|l| \leq 2 \quad \forall l \in E \bigcup F$

If we apply just the last two signatures to Eq 3, the structure reduces to:

$$\sum_S \mathbf{X}_{M_1} * \mathbf{Y}_{M_2} + \sum_{S'} \mathbf{X}'_{M'_1} * \mathbf{Y}'_{M'_2} \leq \mathbf{Z}_M \quad (4)$$

$$\sum_S \mathbf{X}_{M_1} * \mathbf{Y}_{M_2} + \sum_{S'} \mathbf{X}'_{M'_1} * \mathbf{Y}'_{M'_2} \geq \mathbf{Z}_M \quad (5)$$

$$\sum_S \mathbf{X}_{M_1} * \mathbf{Y}_{M_2} - \sum_{S'} \mathbf{X}'_{M'_1} * \mathbf{Y}'_{M'_2} \leq \mathbf{Z}_M \quad (6)$$

2.2 Intelligent Enumeration

This subsection presents an intelligent way to enumerate through constraints based on the concept of logical entailment. We define a constraint dependency tree as shown in Fig 1. Each node in this tree represents costraint and a directed edge between two nodes represent entailment. For example, in Fig 1, C_2 entails C_1 which means whenever C_2

is satisfied C_1 will also be satisfied, which also means if C_1 is not satisfied then C_2 is not satisfied either. There are two ways to enumerate using this tree:

Top Down: In the top down approach, we start with the most general constraint, if it's not satisfied then we don't need to check any of it's descendents. For example, in Fig 1, we start with checking the validity of C_1, if it is not satisfied we don't need to check any other constraints in the tree as none of them would be satisfied, and if it's not then we check each of it's children one by one and repeat the process.

Bottom Up: Bottom up works on the similar idea as the top down approach, here we start with the most specific constraint, in our case it would be C_5 or C_6, if it is satisfied we don't need to check any ancestors of that constraint as it will definitely be satisfied, but if it's not satisfied we check each parent one by one and repeat the process.

In this paper we are going to discuss top down approach, but we can easily transform the algorithm to consider the bottom up approach.

To build a constraint dependency tree, we define the entailment properties for the operations used while constructing a constraint using Eq 3: Summation over terms(S_t), summation over dimensions(S_d), product(P) and slicing(S). Except slicing all three operations can be used to define entailment:

- $X \not\leq Z \Rightarrow X + Y \not\leq Z \quad \forall Y \geq 0$
- $X_M \not\leq Z \Rightarrow X_{M'} \not\leq Z \quad \forall M' : M' \supset M \, and \, X \geq 0$
- $X \not\leq Z \Rightarrow X * Y \not\leq Z \quad \forall Y : Y \geq 1$
- $X + Y \not\leq Z \Rightarrow X \not\leq Z \quad \forall Y \leq 0$
- $X_M \not\leq Z \Rightarrow X_{M'} \not\leq Z \quad \forall M' : M' \subset M \, and \, X \leq 0$
- $X * Y \not\leq Z \Rightarrow X \not\leq Z \quad \forall Y : Y \leq -1$

To use these properties, we have to make some assumptions. We start by using the first three property to define a simple algorithm which gives an idea of how the algorithm works. Thereafter, we keep on relaxing the assumptions we made and explain how the algorithm changes for the revised structure to use the other properties given above. The first two properties can only be used if all the entries in the tensors are non-negative, while for the third one we need the values to be greater than or equal to 1. So for now we assume the values in each tensor to be greater than or equal to 1

To use the bottom up approach, we first define what the most specific constraints (the root nodes) look like and then define the properties a child node has to satisfy. A node is represented by a set of variables which can uniquely define a constraint : $(E, \overline{\mathbf{M}}_i, S_i, Z, \mathbf{M})$. Based on our assumptions and signatures the most general nodes only have to satisfy one property, which is $E = \phi$. Given two nodes $\{E, \overline{\mathbf{M}}_i, S_i, Z, \mathbf{M}\}$ and $\{E', \overline{\mathbf{M}}_i', S_i', Z', \mathbf{M}'\}$. The later will be a child of the former if and only if at least one of the following properties is satisfied keeping everything else unchanged.

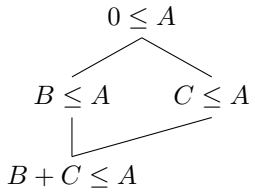
- $E' \supseteq E$, increase the summation over terms.
- $\exists \, \overline{\mathbf{X}}', \overline{\mathbf{X}} \in E', E \quad s.t. \, \overline{\mathbf{X}}' \supseteq \overline{\mathbf{X}}$, increase the product
- $\exists \, i \quad s.t. \, S_i' \supseteq S_i$, increase the summation over dimensions.

2.3 Total Order

With the tree structure defined above we can go ahead and start searching starting from the most general layer and moving down layer by layer. The only problem is that multiple nodes can have the same child. For example, consider 3 tensors A, B and C each of dimension $m * n$. There are multiple possibilities for the most general node:

$$0 \leq A,\, 0 \leq B,\, 0 \leq C,\, 0 \leq A_{m,n},\, 0 \leq B_{m,n},\, 0 \leq C_{m,n},$$
$$0 \leq A_m,\, 0 \leq B_m,\, 0 \leq C_m,\, 0 \leq A_n,\, 0 \leq B_n,\, 0 \leq C_n$$

Consider a small part of the lattice using the first node:



We can see that multiple nodes can take us to the same children. To break this symmetry we define an ordering that will ensure that a node can have only one parent.

- A random ordering is defined for the tensors, for instance we can define $A \prec B \prec C$.
- Similarly we define the ordering of dimensions for each tensor, with the first dimension preceding the second and so on. In our example it would be $m \prec n$ for each tensor.
- The final ordering is defined on the operations performed to reach the child node: Summation over terms $\prec$ Product over terms $\prec$ Summation over dimensions

By using these orderings when generating a child for a node we can remove the redundancy. For each node we have a list (L) of highest order tensor, dimension and operation used on the root node to reach that node. For our example above, $L(B + C \leq A) = \{C, \text{summation over terms}\}$ To generate a child for this node, one can only use a tensor, dimension or operation which has higher or equal order than the highest order tensor, dimension and operation in L. This rule eliminates redundancy from the graph. For example in the figure above, now $C \leq A$ cannot generate $B + C \leq A$ because once we add C to the inequality we can't add B as $B \prec C$

More Complex Setting

Let us now see what happens if we remove the first assumption which was "mod of the numbers in a tensor is all greater than or equal to one". In this case we divide our tensor set into three categories: (1) Where mod of each number is greater than or equal to 1, denoted by P_1, (2) Where mod of each number is less than or equal to 1, denoted by P_2, (3) Where mod of some numbers are less than 1 and some are greater than 1, denoted by P_3.

Taking a product with any tensor in P_1 will give us a child node, while multiplying with a tensor in P_2 will give us a parent node and a product with a tensor in P_3 will give us a sibling. So whenever we add a new term to a node, it must start with the product of all tensors in P_2 and then removing

tensors one by one in the precedence order defined over tensors will give us the child nodes. So, we split the operation used above to reach the child node by splitting the product into two different ones for this setting:

- $\exists\, \overline{\mathbf{X}}', \overline{\mathbf{X}} \in E', E \quad s.t. \, \overline{\mathbf{X}}' \supseteq \overline{\mathbf{X}}$ and $\overline{\mathbf{X}}' \backslash \overline{\mathbf{X}} \in P_1$, increase the product by multiplying with a good tensor.
- $\exists\, \overline{\mathbf{X}}', \overline{\mathbf{X}} \in E', E \quad s.t. \, \overline{\mathbf{X}}' \subseteq \overline{\mathbf{X}}$ and $\overline{\mathbf{X}} \backslash \overline{\mathbf{X}}' \in P_2$, increase the product by removing a bad tensor.

The new precedence relationship among the operators is: Summation over terms $\prec$ Removing a bad tensor $\prec$ Multiplying a good tensor $\prec$ Summation over dimensions

2.4 Generalization

Before discussing the removal of the second assumption let us consider a generalization of the structure defined in Eq 3. The main idea behind this generalization is to be able to use negative tensors and subtractions. If we take Eq 3 and allow the possibility of taking negation of a tensor we get the following generalization:

$$\sum_{\overline{\mathbf{X}}_i \in \mathbf{E}} \bigotimes_{\mathbf{S}_i} (\overline{\mathbf{X}}_i, \overline{\mathbf{M}}_i) - \sum_{\overline{\mathbf{X}}'_i \in \mathbf{E}'} \bigotimes_{\mathbf{S}'_i} (\overline{\mathbf{X}}'_i, \overline{\mathbf{M}}'_i) \leq \mathbf{Z}_M \tag{7}$$

$$\mathbf{Z}_M \leq \sum_{\overline{\mathbf{X}}_i \in \mathbf{E}} \bigotimes_{\mathbf{S}_i} (\overline{\mathbf{X}}_i, \overline{\mathbf{M}}_i) - \sum_{\overline{\mathbf{X}}'_i \in \mathbf{E}'} \bigotimes_{\mathbf{S}'_i} (\overline{\mathbf{X}}'_i, \overline{\mathbf{M}}'_i) \tag{8}$$

For now let us focus on learning constraints defined by the structure in Eq 7, we will update our algorithm later to include Eq 8. If a tensor $\mathbf{X}$ is in any element of $\mathbf{E}$ it's being considered as is while if it belongs to any element of $\mathbf{E}$' it's negation is being considered. $\mathbf{X}$ can also belong to both $\mathbf{E}$ and $\mathbf{E}$', in which case it's being considered in both the states positive and it's negation. Now let's say we have a tensor in our data set with all numbers non-positive, then we can take a negation and make it non-negative and use this transformed tensor in our algorithm without any loss of information because the enumeration is going to consider both the tensor and it's negation as explained above. So we can relax one more assumption we had, we assumed that each tensor must have all numbers non-negative, but now we can also have tensors that have all numbers non-positive. To do the intelligent enumeration, we will have to re-define the concept of most general node and the rules to go to a child node. The most general node will have the following property now:

- $E = \phi$
- $|E'| = 2$
- $|\overline{\mathbf{X}}'| = 2 \quad \forall \overline{\mathbf{X}}' \in E'$
- $S'_i = \bigcup_{\mathbf{M} \in \overline{M}'_i} M \quad \forall \overline{M}'_i \in E'$

2.5 Bringing it all together

We will use the bottom up approach, so we start with the most specific constraint and if it's satisfied we consider all it's children and if not we remove it and all it's descendants, we will define a total ordering on the set of constraints to remove redundancy when enumerating as multiple nodes can have same children. To define ordering we are going to separate Eq 3: the positive one and the negative one.
A positive term can be made more general by doing one of the following:

- $|E'| > |E|$, adding more positive terms
- $\exists l' \in E \quad s.t. \quad |l'| < |l|$, adding more products
- $\exists S'_l \in E \quad s.t. \quad S'_l \subset S_l$, summing over more dimensions

A negative term can be made more general by doing one of the following:

- $|F'| < |F|$, removing negative terms
- $\exists l' \in F \quad s.t. \quad |l'| > |l|$, removing products
- $\exists S'_l \in F \quad s.t. \quad S'_l \supseteq S_l$, summing over less dimensions

We define a node n' to be a child of another node n if and only if either n' has a positive node that is child of another positive node in n or n' has a negative node that is child of another negative node in n but never both.