Acquiring Non-Linear Operations Research Models

ID XXXX

Abstract

Stefano: integer programming? Operations Research (OR) models often involve non-linear objectives and constraints over multi-dimensional decision variables, e.g., personnel schedules or packing configurations. Designing such models is difficult for non-experts and expensive when domain experts are asked to do it, however examples of working solutions are often available. We propose ARNOLD, an approach that partially automates the modelling step by acquiring a non-linear satisfaction or optimization model from examples of feasible solutions. The main challenge is to efficiently explore the space of possible models. Our approach leverages a general-to-specific traversal of the space of candidate constraints that allows to prune large chunks of the search space. The empirical results on real-world problem instances show that ARNOLD can acquire both hard and soft constraints that can be readily used in practice.

1 Introduction

Operations Research (OR) is pervasive. The constraints and objective functions found in real-world problems are often non-linear in nature and involve high-dimensional constants and variables. Consider for instance **Stefano: scheduling example**. As the complexity of the model increases, generating the schedule manually becomes difficult, especially for non-experts. 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 solutions.

Existing approaches do not deal with this setting. Most of them require both positive and negative examples, while in OR negative examples are hard to come by. In addition, few are designed to deal with multi-dimensional quantities, namely ModelSeeker [**?**] and CountOR [**?**]. However, ModelSeeker is restricted to global constraints only, and fails to learn any asymmetric or local constraints, which are the bread and butter of OR models. On the other hand, CountOR exploits tensors to represent solutions to OR problems and acquire constraints over them, but focuses on a specific class of bounding-box constraints that are common in scheduling problems.

To bridge this gap, we contribute a novel method, ARNOLD (for AcquiRing NOn-Linear moDels), that acquires constraints from feasible solutions of OR problems. It employs a generate-and-test strategy, and an efficient enumeration procedure for generating the candidate non-linear constraints. ARNOLD also support learning soft-constraints.

Summarizing, our contributions are:

- A language for capturing non-linear constraints over multidimensional quantities.
- ARNOLD, a novel algorithm for acquiring non-linear hard and soft constraints from examples of (possibly sub-optimal) feasible solutions.
- An extensive empirical analysis on a number of realistic OR problems.

2 Learning Non-linear OR Problems

In the present paper we consider the problem of automatically acquiring OR models from data. Since these models often rely on tensors and tensor operations, we start off by introducing the required notation.

Notation. Scalars x, y, z are written in lower-case, and multidimensional tensors $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ in bold upper-case. The elements of a tensor $\mathbf{X}$ are indicated as $\mathbf{X}_{i,j,k}$. Given a tensor $\mathbf{X}$, $\dim(\mathbf{X})$ refers to its indices and $\mathrm{ran}(\mathbf{X})$ to the ranges of those indices. For instance, if $\mathbf{X} \in \{0,1\}^{3\times 5}$ has indices i and j, then $\dim(\mathbf{X}) = \{i, j\}$ and $\mathrm{ran}(\mathbf{X}) = \{1,\dots,3\}\times\{1,\dots,5\}$.

Sums and products between tensors are element-wise. For instance, letting $\mathbf{X}$ and $\mathbf{Z}$ have identical shapes, $(\mathbf{X}+\mathbf{Z})_{i,j} = \mathbf{X}_{i,j} + \mathbf{Z}_{i,j}$ for all i, j. In OR tasks, tensor indices often represent semantically distinct entities like items to be packed or employees to be scheduled. For this reason, when performing operations across tensors we implicitly *match indices with the same name*. For example, letting $\dim(\mathbf{X}) = \{i, j\}$ and $\dim(\mathbf{Z}) = \{i, k, \ell\}$, then $\dim(\mathbf{XZ}) = \{i, j, k, \ell\}$ and $(\mathbf{XZ})_{i,j,k,\ell} = \mathbf{X}_{i,j}\mathbf{Z}_{i,k,\ell}$. A tensor satisfies a condition if *all* of its elements satisfy that condition, e.g., $\mathbf{X} \leq \mathbf{Z}$ is equivalent to $\forall\, i, j, k, \ell \,.\, \mathbf{X}_{i,j} \leq \mathbf{Z}_{i,k,\ell}$. Finally, the indicator function $\mathbb{I}\{cond\}$ evaluates to 1 if $cond$ is true and to 0 otherwise.

Problem statement. Let us start by introducing a toy model for nurse rostering [**?**; **?**], a prototypical OR problem.

Example. *Consider a nurse rostering problem over five nurses, seven days, and three shifts per day, where the goal is to find a schedule such that, in every shift, a minimum number of at-least-medium skilled nurses are available.*

Letting $\boldsymbol{V}_n$, $\boldsymbol{M}_n$, *and* $\boldsymbol{L}_n \in \{0,1\}$ *indicate whether a nurse* $n \in \{1,\dots,5\}$ *is very skilled, medium skilled, or low skilled, the model can be written as:*

$$\begin{aligned} &\text{find } \boldsymbol{X} \\ &\text{s.t. } \textstyle\sum_n \boldsymbol{V}_n \boldsymbol{X}_{n,d,s} + \sum_n \boldsymbol{M}_n \boldsymbol{X}_{n,d,s} \geq \boldsymbol{R}_{d,s} \quad \forall s,d \quad (1) \\ &\quad \textstyle\sum_{d,s} \boldsymbol{X}_{n,d,s} \geq 10 \quad \forall n \quad (2) \end{aligned}$$

Here the decision variables $\boldsymbol{X}_{n,d,s} \in \{0,1\}$ *encode whether nurse* n *works on shift* s *of day* d, *while* $\boldsymbol{R} \in \mathbb{N}^{7\times 3}$ *is the minimum number of skilled nurses required in every shift.*

Letting $\boldsymbol{C}_s \in \mathbb{R}$ *be the pay of a nurse in a particular shift, we can also consider minimizing the total cost of the schedule subject to the same constraint:*

$$\min_{\boldsymbol{X}} \textstyle\sum_{n,d,s} \boldsymbol{C}_n \boldsymbol{X}_{n,d,s} \quad (3)$$

This example captures some important aspects of common OR problems. First, the model relies on a number of *tensors* of different dimensions, including decision variables $\mathcal{V} = \{\mathbf{X}\}$ and constants $\mathcal{C} = \{\mathbf{R}, \mathbf{V}, \mathbf{M}, \mathbf{L}\}$. Second, it is clear that the LHS of the constraints only include sums and products among tensors in $\mathcal{T} = \mathcal{V} \cup \mathcal{C}$, i.e., they are *polynomials*. **Stefano: they can give multi-dimensional outputs though**

Many models used in nurse rostering, course timetabling, home care scheduling, project scheduling, and other OR tasks **Stefano: @Mohit: add refs** adhere to the same format. All of them can be written as:

$$\begin{aligned} &\min_{\mathcal{V}} f(\mathcal{T}) \qquad (\text{or find } \mathcal{V}) \\ &\text{s.t. } g^c(\mathcal{T}) \leq \mathbf{Z}^c \qquad\qquad c = 1,2,\dots \end{aligned}$$

where f and $\{g^c\}$ are polynomials of $\mathcal{T}$. Notice that equality constraints $g^c(\mathcal{T}) = 0$ can always be rewritten as $g^c(\mathcal{T}) \leq 0 \wedge -g^c(\mathcal{T}) \leq 0$.

A *solution* to such a problem is a value assignment to all the decision variables, which, with a slight abuse of notation, we also indicate as $\mathcal{V}$. Given an OR problem M in this form, we write $M \models \mathcal{V}$ to indicate that the assignment $\mathcal{V}$ is feasible and $\mathrm{Sol}(M) = \{\mathcal{V} : M \models \mathcal{V}\}$ for the set of solutions.

OR models like the above are not easy to write by hand **Stefano: refs**. Our aim is to simplify the modeling process by learning a working model that can be either used as-is or improved upon. This is reasonable whenever it is easy to get accesso to example solutions. For instance, in nurse rostering, the hospital often has access to a set of known-working past schedules, e.g., compiled by hand by the head nurse.

Our learning problem can be stated as follows:

Definition 1 (Non-linear constraint learning)**.** *Given a dataset of feasible solutions* $\{\mathcal{V}_1,\dots,\mathcal{V}_n\}$ *sampled from some hidden non-linear optimization or satisfaction problem containing constants* $\mathcal{C}$, *find a set of polynomial constraints that cover all of the examples.*

The estimated model can be used to automatically produce new schedules that are as good or better than the provided examples. Of course, prior to deployment, the model can be validated or extended by domains experts.

Notice that we merely assume the examples to represent desirable solutions, i.e., schedules that were observed to work well enough in practice. In many applications this can be safely assumed. For instance, in nurse rostering the past schedules are likely the result of a laborious process of trial-and-error, where the hospital tried out different alternatives and kept the best ones. For the same reason, we also expect the examples—which are generated by hand—to be sub-optimal. Of course, the higher the desirability of the provided examples, the better the schedules generated by the learned model.

3 Method

The goal of ARNOLD is to acquire polynomial-like constraints like Eq. 1–2 from examples of feasible solutions. At a high-level, this is accomplished through a generate-and-test strategy that relies on a non-redundant enumeration of the candidate constraints. In the following, we introduce our constraint language, and then proceed to discuss our learning strategy.

Constraint language. The basic elements of our constraints are *terms*, that is, expressions like $\sum_{d,s} \mathbf{V}_n \mathbf{X}_{n,d,s}$ in Eq. 1. Formally, a term has the form:

$$\mathrm{term}_{\mathcal{I},\mathcal{X}} = \pm \textstyle\sum_{\mathbf{i} \in \mathrm{ran}(\mathcal{I})} \prod_{\mathbf{X} \in \mathcal{X}} \mathbf{X}_{\mathbf{i}}$$

where $\mathcal{X} \subseteq \mathcal{T}$ is the set of tensors being multiplied together and $\mathcal{I}$ is the set of indices being summed over. We require the indices $\mathcal{I}$ to actually appear in the tensors in $\mathcal{X}$, that is, $\mathcal{I} \subseteq \bigcup_{\mathbf{X} \in \mathcal{X}} \dim \mathbf{X}$.

Stefano: @Mohit: integrate → If we enumerate through all possible constraints in $\mathcal{C}$, we can notice that it considers each tensor in the positive as well as negative term. Therefore if we encounter any negative tensor we can multiply it with -1 to make it positive without losing any constraint that can be represented by Eq 4. Based on this property and our assumption, moving forward we can safely assume that all tensors are positive without loss of any generality. **Stefano: end integrate**

To be precise, let $\mathcal{D}$ be the set of all dimensions appearing in all of the tensors; then $\mathcal{I} \subseteq \mathcal{D}$. The dimension of the term is simply the set of dimensions that appear in the various tensors in the term but not summed over, that is $\dim(\mathrm{term}_{\mathcal{I},\mathcal{X}}) = \bigcup_{\mathcal{X} \in \mathbf{X}} \dim(\mathcal{X}) \setminus \mathcal{I}$. In the above example, $\mathcal{I} = \{d,s\}$, which is a subset of $\mathcal{D} = \{n,d,s\}$, and the dimension of the term is $\{n,d,s\} \setminus \{d,s\} = \{n\}$.

The constraints we consider are naturally expressed by sums of terms, e.g., $\sum_{d,s} \mathbf{A}_n \mathbf{X}_{n,d,s} + \sum_d \mathbf{B}_d \mathbf{Y}_{n,d}$. The general form is:

$$g_c(\mathcal{T}) = \textstyle\sum_k \mathrm{term}_{\mathcal{I}^k,\mathcal{X}^k} \leq \pm \mathbf{Z}^c \qquad \forall \mathbf{j} \in \mathcal{J} \quad (4)$$

where $\mathcal{J} = \dim\big(\mathrm{term}_{\mathcal{I}^1,\mathcal{X}^1}\big) \cup \dim(\mathbf{Z}^c)$. In order for the outer sum to make sense, we require all terms to have the same dimension; as shown by Eq. 4, these remaining indices $\mathbf{j}$ are implicitly quantified over. Later on, we will generalize this to include negative terms as well. **Stefano: at some point explain that all tensors are actually taken to be positive**

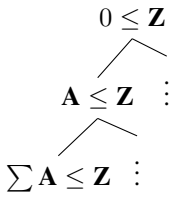

Figure 1: Example constraint tree. The root represents the most general constraint and as we keep going down the constraints become more specific. **Stefano: @Stefano: make it match the example; @Mohit: add simple bad/ugly version.**

Finally, a model M is just a set of constraints in this format. Later on we will also consider objective functions.

Learning the constraints. Now we have everything together to introduce ARNOLD. In order to learn a constraint theory, we first have to limit its complexity. ARNOLD accepts two parameters: the maximum number of terms $|M|$ in the model, as well as the maximum lenght of the products in any term, i.e., $\max_k |\mathcal{X}^k|$.

We use a generate-and-test approach. Namely, we enumerate through all the constraints and check which ones are satisfied by the data. However, the number of candidates is exponential in the number of tensors $|\mathcal{T}|$: **Stefano: @Mohit: quantify the approx. numbre of candidate constraints.** Thus, naïve enumeration does not scale to the real-world problems. The main challenge is therefore to avoid explicitly enumerating as many candidates as possible. In the following, we introduce our solution.

Enumerating the constraints. Before explaining our enumeration strategy, we temporarily make the simplifying assumption that all of the tensors are positives.

In order to enumerate the constraints, we visit the search space in a top-down manner, starting from the most general constraint(s) toward more and more specific constraints. Consider the constraint entailment tree shown in Fig. 1. Each node in this tree represents a constraint c, i.e., $g^c(\mathcal{T}) \leq \mathbf{Z}^c$. A directed edge from node c to node c' means that the child node entails the parent, i.e., $c' \models c$. For example, in Fig. 1, $\mathbf{A} \leq \mathbf{Z}$ entails $\mathbf{A} + \mathbf{B} \leq \mathbf{Z}$, which holds so long as all the quantities in the tensors are positive. Whenever a node (aka constraint) c is not satisfied, none of its children will be satisfied either. This means that, by visiting all nodes in a top-down manner, as soon as we find an unsatisfied node, we can immediately prune away all of its descendants. For instance, in Fig. 1, the enumeration algorithm would start by checking the validity of the root. If it is not satisfied, then it does not need to check any other constraints. On the other hand, if it is not, then we check each of its children one by one and repeat the process.

Now, ARNOLD builds the constraint forest as follows:

- **Generate Roots:** To generate roots these steps are followed: (1) Pick a tensor for $\mathbf{Z}$ then (2) Create all possible configuration for LHS such that all the signatures are satisfied and the value is minimum. There can be multiple configurations that satisfy these properties. For example, consider the nurse rostering example defined above, if the signature specifies m=2 and n=2, some of the possible configuration for LHS would be: $-\sum_{n,s,d} \mathbf{H}_n \mathbf{X}_{n,s,d} - \sum_{n,s,d} \mathbf{M}_n \mathbf{X}_{n,s,d}$, $-\sum_n \mathbf{H}_n \mathbf{H}_n - \sum_{n,s,d} \mathbf{M}_n \mathbf{X}_{n,s,d}$
- For every root, recursively check its trees. **Visit the trees:** Thereafter children are generated for each root by keeping $\mathbf{Z}$ fixed and increasing the value for LHS by using operations and conditions on tensor that guarantees the entailment property.

Algorithm 1 The enumeration algorithm.

```
1:  procedure BUILDTREE(V^1, ..., V^n: examples)
2:      M_L ← ∅
3:      for root ∈ GENERATEROOTS(data) do
4:          if ¬satSignat(root) ∨ satConstr(root) then
5:              M_L ← M_L ∪ ENUMERATE(root, ∅)
6:      return M_L

7:  procedure ENUMERATE(node, constraintList)
8:      if node is NULL or (satisfiessSignature(node)
        and not satisfiesConstraint(node)) then
9:          return constraintList
10:     children ← generateChildren(node)
11:     childrenSatisfaction = 0
12:     for child in children do
13:         if ¬satSignat(child) ∨ satConstr(child) then
14:             constraints = ENUMERATE(child)
15:             if constraints is not NULL then
16:                 childrenSatisfaction = 1
17:     if childrenSatisfaction = 0 then
18:         constraintList.append(root)
19:     return constraintList
```

The pseudo code of ARNOLD is given in Algorithm 1.

In the procedure $buildTree$, first all the possible root nodes are generated, for each root if it either does not satisfy the signatures or satisfies the constraint we enumerate the possible tree from that root node. When a node doesn't satisfy the signatures the constraint represented by it doesn't make sense and hence can't be verified but a child of such node might satisfy the signatures and the constraint represented as well. Hence, while enumerating the constraint entailment tree, children are generated for a node only if either it doesn't satisfy the signatures or satisfies the constraint.

In the $enumerate$ procedure we use a depth first search, for each node children are generated unless the node satisfies the signature but not the constraint, if none of it's children satisfy the constraint only then that node is included in the constraint list. There are two important functions used in this algorithm, one is $generateRoots$ and the other is $generateChildren$. We have already seen earlier how to generate roots, let us see now how to generate children.

The good, the bad, and the ugly. We have seen that the constraint tree is defined by

In Eq. 4 three operations were used: summations over terms, summations over dimensions and product over tensors.

All these operations have certain entailment properties based on some conditions being satisfied by the tensors used in the operation. For example, if $\mathbf{Y} \geq 0$, then:

$$(\mathbf{X} + \mathbf{Y} \leq \mathbf{Z}) \models (\mathbf{X} \leq \mathbf{Z})$$

that is, adding a positive tensor in LHS generates a child.

In order to discuss these entailment properties, we have to first distinguish tensors into three types:

- A tensor $\mathbf{X}$ is **good** iff $\iff \mathbf{X} \geq 1$.
- **Bad**: $\iff \mathbf{X} \leq 1$.
- **Ugly**: $\iff \mathbf{X} \not\leq 1 \wedge 1 \not\leq \mathbf{X}$.

The good tensors. If the examples used to learn constraints only have good tensors, then children can be obtained by applying one of the following operations:

G1 For a negative term k, shrinking $\mathcal{I}^k$, e.g.:

$$(-\sum_{\mathcal{I}'} \mathbf{X_i Y_i} \leq \mathbf{Z}) \models (-\sum_{\mathcal{I}} \mathbf{XY} \leq \mathbf{Z}) : \mathcal{I}' \subset \mathcal{I}$$

G2 Removing a tensor from $\mathcal{X}$ in a negative term, e.g.:

$$\left(-\sum_{D} \mathbf{X} \leq \mathbf{Z}\right) \models \left(-\sum_{D} \mathbf{X} \times \mathbf{Y} \leq \mathbf{Z}\right)$$

G3 remove a negative term with only one tensor summing over no dimensions

$$-\sum_{D} \mathbf{X} \leq \mathbf{Z} \models -\sum_{D} \mathbf{X} - \mathbf{Y} \leq \mathbf{Z}$$

G4 add a positive term with only one tensor summing over no dimensions

$$\sum_{D} \mathbf{X} - \sum_{D'} \mathbf{Y} + \mathbf{X}' \leq \mathbf{Z} \models \sum_{D} \mathbf{X} - \sum_{D'} \mathbf{Y} \leq \mathbf{Z}$$

G5 multiply a tensor to a positive term

$$\sum_{D} \mathbf{X} \times \mathbf{X}' - \sum_{D'} \mathbf{Y} \leq \mathbf{Z} \models \sum_{D} \mathbf{X} - \sum_{D'} \mathbf{Y} \leq \mathbf{Z}$$

G6 add a dimension sum in a positive term

$$\sum_{D'} \mathbf{X} \times \mathbf{X}' \leq \mathbf{Z} \models \sum_{D} \mathbf{X} \times \mathbf{X}' \leq \mathbf{Z} : D \subseteq D'$$

The bad tensors. Including a bad tensor in the example changes the entailment properties of some of the operations used to generate children. Because the elements of a bad tensor are between 0 and 1, it changes the entailment property of products, while keeping the entailment properties of summations.

To generate children, earlier we used: i) multiplication of a positive term with a good tensor, and ii) removing a good tensor from a negative term. For bad tensors, it is the opposite: multiplication of a positive term with a bad tensor gives a parent as does the removal of a bad tensor from a negative term.

So we introduce two new operations with some small changes in the already existing ones:

- remove a dimension from the sum in a negative term
- remove a good tensor from the product in a negative term
- multiply a bad tensor to a negative term
- remove a negative term with only one tensor summing over no dimensions
- add a positive term with at max one good tensor multiplied with $n-1$ bad terms summing over no dimensions, where n is the maximum number of tensors in a term.
- remove a bad tensor from the product in a positive term
- multiply a good tensor to a positive term
- add a dimension sum in a positive term

The ugly tensors. Ugly tensors have at least one element less than one and at least one element greater than one. This means the multiplication operation now will neither give us a children nor a parent, although both the summation operations still follow the same entailment properties as the numbers are still non-negative.

So, to include the ugly tensors in our constraints we follow this approach: for each node consider every ugly version of it, by ugly version we mean take any term in the node and multiply it with an ugly tensor, for each ugly version if it satisfies the constraint enumerate a separate tree with that ugly version as the root node.

Ordering. The enumeration method described above can lead to a situation where multiple nodes have the same child, consider an example with three tensors $\mathcal{T} = [\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3]$ each of dimension $m * n$. Consider the following nodes: (1) $\mathbf{X}_1 \leq \mathbf{X}_3$ and (2) $\mathbf{X}_2 \leq \mathbf{X}_3$ for both these nodes one of the children would be: $\mathbf{X}_1 + \mathbf{X}_2 \leq \mathbf{X}_3$, hence while enumerating the constraint entailment tree we might encounter same constraints multiple times.

To break this symmetry we define a nested lexicographic ordering as follows:

- A lexicographic ordering is defined for the tensors, for instance we can define $\mathbf{X}_1 \prec \mathbf{X}_2 \prec \mathbf{X}_3$.
- Similarly we define lexicographic ordering of dimensions for each tensor. 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.

While enumerating the constraint entailment tree we maintain a list L for each node that contains the highest order tensor, dimension and operation used on the root node to reach that node. To generate a child for any node, one can only use a tensor, dimension or operation which has higher or equal order compared to the tensor, dimension and operation in L. This rule eliminates redundancy from the graph. In the example discussed above, now $\mathbf{X}_2 \leq \mathbf{X}_3$ cannot generate $\mathbf{X}_1 + \mathbf{X}_2 \leq \mathbf{X}_3$ because once we add $\mathbf{X}_2$ to the inequality we can't add $\mathbf{X}_1$ as $\mathbf{X}_1 \prec \mathbf{X}_2$

4 Empirical Analysis

We address the following research questions:

1. Can ARNOLD solve real-world problems?
2. Is ARNOLD efficient?

3. Are the models learned by ARNOLD usable?
4. Are the soft constraints learned by ARNOLD any good?

Learning hard constraints. We start off by studying the ability of ARNOLD to acquire meaningful non-linear satisfaction problems. In order to quantify its performance, we considered a number **Stefano: @Mohit: how many?** of MiniZinc [**?**] non-linear optimization problems taken from: http://mohit.add.url.

For each MiniZinc model M_T, first we sample a dataset $\mathcal{D}$ of 10,000 feasible solutions (using the procedure below) and split it into 5 folds of 2,000 examples each. Next, we train our method on progressively larger subsets of **Stefano: @Mohit: WRITEME**. and use these as examples for ARNOLD, . This procedure is repeated on gradually larger subsets of $\mathcal{D}$ of size 100 and $1,000$, thus obtaining a set of learned model M_L.

Sampling feasible configurations from a high-dimensional space constrained by non-linear functions is a very non-trivial problem. In practice, we enumerate a large number of (likely correlated) solutions using the Gecode MiniZinc backend [**?**] and obtain an IID subset using reservoir sampling.

We measured the precision and recall of the learned model M_L w.r.t. the true underlying model M_T, namely:

$$\text{Pr} = \frac{|\text{Sol}(M_T) \cap \text{Sol}(M_L)|}{|\text{Sol}(M_L)|} \quad \text{Rc} = \frac{|\text{Sol}(M_T) \cap \text{Sol}(M_L)|}{|\text{Sol}(M_T)|}$$

Computing these quantities is not trivial, so we estimate them using rejection sampling **Stefano: @Mohit: please complete**.

The average performance for increasingly complex models (with $s = 1, 2, 3$ and $p = 1, 2, 3$) are shown in Figure **??**. **Stefano: @Mohit: add summary figure or table**.

Learning soft constraints. **Stefano: @Mohit: WRITEME**

5 Related Work

Constraint learning is a core artificial intelligence and machine learning task. For an overview, see [**?**] and references therein.

Stefano: @Stefano: shuffle this section!

More recent approaches can learn constraint programs with numerical terms. For instance, the works of Bessiere and colleagues Bessiere et al. (2016) 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. 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, not to mention additional variables specifying qualifications and availability, to name only a few) increases exponentially with the number of dimensions in the data (roughly # of days # of shifts # of nurses), and since checking satisfaction is in general NP-hard, these approaches do not scale to realistically-sized rostering instances. In contrast, C OUNT -OR does not require costly satisfiability checks.

There exists a class of learning approaches that can potentially deal with highdimensional numerical domains Teso et al. (2017); Pawlak & Krawiec (2017). 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. Kolb et al. (2017)) can also deal with numerical variables, 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 Beldiceanu & Simonis (2012), 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 apparently similar, C OUNT -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.

Stefano: +CountOR

6 Conclusion

Stefano: WRITEME

Acknowledgements

The authors thank Paolo Dragone for help with the experiments. 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).

References

[Beldiceanu and Simonis, 2012] Nicolas Beldiceanu and Helmut Simonis. A Model Seeker: extracting global constraint models from positive examples. pages 141–157, 2012.

[Burke *et al.*, 2004] Edmund K Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The state of the art of nurse rostering. *Journal of scheduling*, 7(6):441–499, 2004.

[De Raedt *et al.*, 2018] Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings of AAAI'18*, 2018.

[Kumar *et al.*, 2018] Mohit Kumar, Stefano Teso, and Luc De Raedt. Constraint learning using tensors. In *EURO*, 2018.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J

Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[Schulte *et al.*,] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode.

[Smet *et al.*, 2013] Pieter Smet, Patrick De Causmaecker, Burak Bilgin, and Greet Vanden Berghe. Nurse rostering: a complex example of personnel scheduling with perspectives. In *Automated Scheduling and Planning*, pages 129–153. Springer, 2013.