Acquiring Integer Programs from Data

Mohit Kumar , **Stefano Teso** and **Luc De Raedt**
KU Leuven
{mohit.kumar,stefano.teso,luc.deraedt}@cs.kuleuven.be

Abstract

Integer programming (IP) is widely used within operations research to model and solve complex combinatorial problems such as personnel rostering and assignment problems. Modelling such problems is difficult for non-experts and expensive when hiring domain experts to perform the modelling. For many tasks, however, examples of working solutions are readily available. We propose ARNOLD, an approach that partially automates the modelling step by learning an integer program from example solutions. Contrary to existing alternatives, ARNOLD natively handles multi-dimensional quantities and non-linear operations, which are at the core of IP problems, and it only requires examples of feasible solution. The main challenge is to efficiently explore the space of possible programs. Our approach pairs a general-to-specific traversal strategy with a nested lexicographic ordering in order to prune large portions of the space of candidate constraints while avoiding visiting the same candidate multiple times. Our empirical evaluation shows that ARNOLD can acquire models for a number of realistic benchmark problems.

1 Introduction

Integer programming (IP) is a widespread framework for modelling and solving complex combinatorial problems such as scheduling, packing, routing, *etc.* [Nemhauser and Wolsey, 1989]. Real-world IP models consist of multi-dimensional decision variables (e.g. schedules) tied together by complex constraints. Designing integer programs requires technical know-how beyond the level of non-experts, and can be time-consuming and expensive. This hinders the adoption of IP.

In many applications, however, example solutions of reasonable quality are readily available. For instance, in nurse rostering, the hospital has access to past nurse schedules. In line with the previous work on constraint acquisition [De Raedt *et al.*, 2018; Bessiere *et al.*, 2017; Beldiceanu and Simonis, 2012], we propose to partially automate the modelling process by acquiring integer programs directly from working solutions, i.e., positive examples.

Existing approaches to constraint acquisition have not been designed to handle this setting. First, most of them are ill-equipped to deal with structured (e.g. multi-dimensional) variables and non-linear constraints, which are the norm in integer programming. Furthermore, most of them require examples of negative (i.e., infeasible) configurations, which are often unavailable in applications.

We contribute ARNOLD, for AcquiRing NOn-Linear moDels, a novel approach to learn integer programs from positive example solutions. ARNOLD relies on a tensor-based language for describing polynomial constraints between multi-dimensional quantities, which are common in integer programs. This representation enables handling several constraints (namely, one per element) at once. Our approach cleverly enumerates the potential IP constraints and collects those that are satisfied by the example solutions. This is guaranteed to produce a valid IP model, but it requires the evaluation of a large number of candidates. In line with approaches to inductive logic programming [De Raedt, 2008; De Raedt and Dehaspe, 1997] and graph mining [Jiang *et al.*, 2013], we use two strategies to manage the enumeration process: a general-to-specific search scheme that prunes away large portions of the search space, and a nested lexicographic ordering that avoids enumerating the same constraint twice.

Summarizing, our contributions include: (1) A constraint language for representing and manipulating typical integer programs, (2) ARNOLD, a novel algorithm for acquiring integer programs from examples of feasible (but not necessarily optimal) solutions, and (3) An extensive empirical analysis on a number of integer programs, showing that ARNOLD can acquire good quality programs from a handful of examples.

2 Learning Integer Programs

We aim to automatically acquire (polynomial) integer programs from positive examples. Since IP often relies on multi-dimensional quantities, we start by introducing the required notation.

As for notation, scalars x are written in lower-case, tensors $\mathbf{X}$ in bold upper-case, and sets $\mathcal{X}$ in calligraphic upper-case. Given a tensor $\mathbf{X}$, its elements are indicated as $\mathbf{X}_{i,j,k}$, its indices are referred by $\mathrm{index}(\mathbf{X})$, and the ranges of those indices as $\mathrm{range}(\mathbf{X})$. For instance, if $\mathbf{X} \in \{0,1\}^{3\times 5}$ has indices i and j, then $\mathrm{index}(\mathbf{X})$ is $\{i,j\}$ and $\mathrm{range}(\mathbf{X})$ is $\{1,\ldots,3\} \times \{1,\ldots,5\}$.

All operations and comparisons between tensors are implicitly element-wise. For instance, if $\mathbf{X}$ and $\mathbf{Z}$ have identical dimensions, then $(\mathbf{X} + \mathbf{Z})_{i,j} = \mathbf{X}_{i,j} + \mathbf{Z}_{i,j}$ for all i, j. In IP, tensor indices often represent semantically distinct entities, like objects 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, if $\text{index}(\mathbf{X}) = \{i, j\}$ and $\text{index}(\mathbf{Z}) = \{i, k, \ell\}$, then $\text{index}(\mathbf{XZ}) = \{i, j, k, \ell\}$ 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., $\mathbf{X} \leq \mathbf{Z}$ holds iff $\forall\, i, j, k, \ell \,.\, \mathbf{X}_{i,j} \leq \mathbf{Z}_{i,k,\ell}$.

2.1 Integer Programs

Let us introduce a toy integer program for nurse rostering [Burke *et al.*, 2004; Smet *et al.*, 2013].

Example. *Consider a small hospital with five nurses. The goal is to find a seven day schedule, where every day has three shifts, such that a minimum number of at-least-medium skilled nurses are always available. Let* $\boldsymbol{X} \in \{0,1\}^{5\times 7\times 3}$ *be a decision variable such that* $\boldsymbol{X}_{n,d,s}$ *encodes whether nurse* n *works on shift* s *of day* d*,* $\boldsymbol{V} \in \{0,1\}^5$ *(resp.* $\boldsymbol{M}$*,* $\boldsymbol{L}$*) indicate which nurses are very skilled (resp. medium or low skilled), and* $\boldsymbol{R} \in \mathbb{N}^{7\times 3}$ *be the minimum number of skilled nurses required in each shift. Then, an integer program for this problem can be written as:*

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

This example captures some important aspects of typical integer programs: **a**) The decision variables (i.e. the quantities determined by the IP solver, like $\mathbf{X}$ above) and the constants (all the other quantities, such as $\mathbf{R}$, $\mathbf{V}$, $\mathbf{M}$, $\mathbf{L}$) are *non-negative integer tensors* of arbitrary dimensions. We use $\mathcal{V}$ and $\mathcal{C}$ to indicate the set of tensor variables and constants, respectively, and $\mathcal{T} = \mathcal{V} \cup \mathcal{C}$ to indicate all the tensors appearing in the program. We stress that the constants $\mathcal{C}$ (e.g., the skill levels) are often known beforehand. **b**) The constraints only include sums, products, and comparisons among tensors; in other words, they are *polynomial inequalities*. The caveat is that the variables and the value of the polynomial can be multi-dimensional. **c**) The constraints are *non-linear*, that is, they include products among tensors (constants and decision variables alike). This holds for both linear and non-linear integer programs. **d**) This is a *satisfaction* problem. Satisficing problems, where the goal is to find a configuration that is "good enough" according to some objective function, are subsumed as a special case, as discussed below.

For these reasons, we focus on IP programs $\mathcal{P}$ of the form:

$$\text{find } \mathcal{V} \quad \text{s.t. } g_j(\mathcal{V}) \leq \mathbf{Z}_j \qquad j = 1, 2, \ldots, m$$

where $g_1, \ldots, g_m$ are polynomials of $\mathcal{V}$ with coefficients in $\mathcal{C}$, and $\mathbf{Z}_1, \ldots, \mathbf{Z}_m$ are constants in $\mathcal{C}$. Such integer programs can be viewed as sets of polynomial constraints. Given $\mathcal{P}$, a *solution* is a value assignment to the decision variables in $\mathcal{V}$ that satisfies all the constraints in $\mathcal{P}$. Abusing notation, we write value assignments as $\mathcal{V}$ too. We use $\mathcal{P} \models \mathcal{V}$ to indicate that a value assignment $\mathcal{V}$ is feasible with respect to $\mathcal{P}$, that is:

$$\mathcal{V} \models \mathcal{P} \iff \forall j = 1, \ldots, m \,.\, g_j(\mathcal{V}) \leq \mathbf{Z}_j$$

We will also say that $\mathcal{V}$ satisfies P. The set of solutions of $\mathcal{P}$ is written $\text{Sol}(\mathcal{P}) = \{\mathcal{V} : \mathcal{V} \models \mathcal{P}\}$.

Observe that this setup also captures *satisficing* scenarios, where an (integer polynomial) objective $f(\mathcal{V})$ is given and the goal is to find solutions satisfying $f(\mathcal{V}) \geq \theta$, for some threshold θ. More generally, pure 0–1 integer programming is extremely expressive, as it subsumes propositional logic.

2.2 Problem Statement

Our aim is to simplify the modelling step by acquiring a program $\mathcal{P}_L$ from a set of examples of feasible solutions. This can be formalized as follows:

Definition (Integer Program Learning)**.** *Given a set of tensor constants* $\mathcal{C}$*, tensor decision variables* $\mathcal{V}$*, and feasible solutions* $\mathcal{D} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ *taken from some unknown integer program, find an integer program* $\mathcal{P}_L$ *that is satisfied by all examples, i.e.,* $\forall i \,.\, \mathcal{V}_i \models \mathcal{P}_L$*.*

The estimated model can be used to produce new schedules that are as good or better than the example ones. Of course, the learned program can be validated or extended by domains experts, for instance to include corner cases that were not observed in the data. At the same time, the domain experts could already have specified constraints that need to be part of the model $\mathcal{P}_L$.

Notice that we do not assume the input examples to be optimal, which would be very hard to guarantee in practice. Rather, we make the more realistic assumption that they represent desirable solutions, e.g., past schedules that were observed to work well in practice. For instance, existing schedules are often the result of a laborious process of trial-and-error, where the administration of a hospital tried out different alternatives and retained the ones that worked well enough. Of course, in general, the higher the quality of the examples, the better the solutions of the learned program.

3 Learning Integer Programs with ARNOLD

We propose ARNOLD (for AcquiRing NOn-Linear moDels), an approach for learning integer programs that acquires polynomial constraints from examples of feasible solutions. In order to do so, ARNOLD relies on a language for expressing and manipulating the constraints, which we introduce next.

3.1 The Constraint Language

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

$$\text{term}_{\mathcal{I},\mathcal{X}} = \textstyle\sum_{(i_1,i_2,\ldots)\in \text{range}(\mathcal{I})} \prod_{\mathbf{X}\in\mathcal{X}} \mathbf{X}_{i_1,i_2,\ldots} \tag{2}$$

where $\mathcal{X} \subseteq \mathcal{T}$ is a set of the decision variables and constants appearing in the product, both of which can be a tensor. For example, in $\sum_{d,s} \mathbf{V}_n \mathbf{X}_{n,d,s}$, $\mathbf{V}_n$ is a 1-d constant while $\mathbf{X}_{n,d,s}$ is a 3-d decision variable. $\mathcal{I}$ represents the set of indices being summed over. For consistency, the indices $\mathcal{I}$ are required to actually appear in the tensors in $\mathcal{X}$, that is, $\mathcal{I} \subseteq \bigcup_{\mathbf{X}\in\mathcal{X}} \text{index}(\mathbf{X})$. Notice that terms are themselves multi-dimensional quantities. In particular, the indices of a term are the indices that appear in its tensors but are not summed

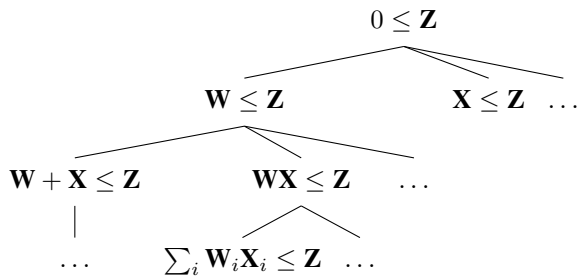

Figure 1: Recursive refinement of a most-general constraint $(0 \leq \mathbf{Z}) \in \rho(\top)$. The levels illustrate three specialization schemes: adding a term, elongating a product and adding an index. $\mathbf{W}$, $\mathbf{X}$ and $\mathbf{Z}$ have values greater than or equal to 1 and have single index i.

over, namely $\text{index}\,(\text{term}_{\mathcal{I},\mathcal{X}}) = \bigcup_{\mathbf{X}\in\mathcal{X}} \text{index}\,(\mathbf{X}) \setminus \mathcal{I}$. In the above term $\mathcal{I}$ is $\{d, s\}$, which is indeed a subset of $\{n, d, s\}$, and the term's indices are $\{n, d, s\} \setminus \{d, s\} = \{n\}$.
The constraints we consider have the form:

$$\textstyle\sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k} \leq \mathbf{Z} \tag{3}$$

As in Eq. 1, the left-hand side is a sum of terms. The sum implicitly iterates over all terms appearing in the constraint. To be a *well-formed* constraint, all terms in the sum must have the same dimension. For instance, if $\text{index}\,(\mathbf{A}) = \text{index}\,(\mathbf{X}) = \{n, d, s\}$ and $\text{index}\,(\mathbf{B}) = \text{index}\,(\mathbf{Y}) = \{n, d\}$, then $\sum_{d,s} \mathbf{A}_n \mathbf{X}_{n,d,s} + \sum_d \mathbf{B}_d \mathbf{Y}_{n,d}$ is well-formed, because both terms have dimension $\{n\}$. The right hand side is a single tensor constant $\mathbf{Z} \in \mathcal{C}$.

Importantly, since tensor comparisons are element-wise, all of the indices that are not summed over, namely $\text{index}\left(\text{term}_{\mathcal{I}^1,\mathcal{X}^1}\right) \cup \text{index}\,(\mathbf{Z})$, are implicitly universally quantified. For example in Eq. 1, the constraint is universally quantified over $\{d, s\}$ because $\text{index}\,(\sum_n \mathbf{V}_n \mathbf{X}_{n,d,s}) \cup \text{index}\,(\mathbf{R}_{d,s}) = \{d, s\}$. Notice that the indices of the expression on the left hand side need not be identical to those on the right. For instance, the constraint $\mathbf{X} \leq \mathbf{Z}$ could have indices $\{s, n\}$ for $\mathbf{X}$ and $\{s\}$ for $\mathbf{Z}$, as the constraint is universally quantified over s and n; in other words, this constraint states that for all s and n, $\mathbf{X}_{s,n} \leq \mathbf{Z}_n$.

3.2 The Arnold Algorithm

Given a set of feasible solutions $\mathcal{D} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ and a bias including additional "utility" tensors (such as the all-zero and all-one tensors), ARNOLD acquires an integer program $\mathcal{P}_L$ using a general-to-specific search strategy that is reminiscent of inductive programming, graph mining and constraint acquisition [De Raedt and Dehaspe, 1997; De Raedt, 2008; Jiang *et al.*, 2013; De Raedt *et al.*, 2018].

Letting p be the (user-specified) maximum number of tensors in a term and s the maximum number of terms in a constraint, ARNOLD cleverly searches the space of all syntactically correct constraints up to the given complexity and keeps the ones that are compatible with all examples. It is easy to see that the number of potential terms is exponential in p and that the number of candidate constraints is even larger. The main challenge is thus to avoid enumerating as many candidates as possible. ARNOLD adopts two strategies to prune the search space: 1) Enumerating the candidate constraints in a general-to-specific fashion, which allows it to prune away large parts of the search space; and 2) Using a canonical form and a nested lexicographic ordering to avoid enumerating the same constraint twice.

Algorithm 1 The ARNOLD search algorithm.

```
1: procedure ARNOLD(D: dataset)
2:     return REFINE(⊤, D)
3: procedure REFINE(c: constraint, D)
4:     P_c ← ∅
5:     if D ⊨ c then
6:         P_c ← {c}
7:         for c' ∈ ρ(c) do
8:             P_c ← P_c ∪ REFINE(c', D)
9:     return P_c
```

3.3 General-to-specific Search

Given two constraints c and c', we say that c is more specific than c' (and that c' is more general than c), written $c \models c'$, if and only if all value assignments $\mathcal{V}$ that satisfy c also satisfy c'. Intuitively, specializations impose more strong conditions in the left hand side of the constraint. For instance, if $\mathbf{P}$ is non-negative, then $(\mathbf{X} + \mathbf{P} \leq \mathbf{Z}) \models (\mathbf{X} \leq \mathbf{Z})$ holds for any $\mathbf{X}$, $\mathbf{Z}$ of compatible sizes. Enumerating constraints in a general-to-specific order is convenient because, whenever a constraint c is inconsistent with the examples $\mathcal{D}$, all of the more specific constraints c' are also inconsistent and can be efficiently pruned. This is in line with logical and relational learning, where the generality relation coincides with logical entailment [De Raedt, 2008].

As in that line of work, ARNOLD performs general-to-specific search by recursively specializing constraints using a *refinement operator* as shown in Fig 1. A refinement operator ρ maps a constraint c (or a set of constraints) to a set of more specific constraints $\rho(c)$. A complete general-to-specific search then starts from the maximally general constraint (denoted by $\top$), and recursively applies the refinement operator ρ, while imposing some pruning.The recursive application of a refinement operator ρ is denoted by ρ^n if it is applied recursively n times. Furthermore, $\rho^*(c) = \rho(c) \cup \rho^2(c) \cup \ldots$

ARNOLD follows this schema faithfully, see Algorithm 1. In particular, ARNOLD uses an *ad hoc* refinement operator for generating polynomial integer programming constraints, discussed next. In order to simplify the presentation, we temporarily assume all tensors to be non-negative, i.e., for all $\mathbf{X} \in \mathcal{T}$, $\mathbf{X} \geq 0$. Notice that, in this case, the most general constraints are simply $\rho(\top) = \{0 \leq \mathbf{Z} : \mathbf{Z} \in \mathcal{T}\}$.

3.4 The Refinement Operator

We are now ready to define our refinement operator. In this definition, we distinguish *good* tensors satisfying $\mathbf{G} \geq 1$, *bad* tensors satisfying $0 \leq \mathbf{B} \leq 1$, and *ugly* tensors $\mathbf{U}$, which are neither good nor bad. All the other unqualified (but still positive) tensors will be denoted by $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$. Let c be a well-formed constraint $\sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k} \leq \mathbf{Z}$ and $\text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell}$ be one of its terms. All specializations $c' \in \rho(c)$ are obtained by

increasing the left hand side of c through any of the following operations:

a) *Adding a term to the left hand side.* For every non-negative tensor $\mathbf{P} \in \mathcal{T}$, adding it as a term:

$$\sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k} + \mathbf{P} \geq \sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k}$$

b) *Adding an index to a term.* For any non-negative term $\text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell} \geq 0$, adding another index i to $\mathcal{I}^\ell$, that is:

$$\text{term}_{\mathcal{I}^\ell \cup \{i\},\mathcal{X}^\ell} \geq \text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell}$$

For instance, consider a 2×2 non-negative tensor tensor $\mathbf{P}$ with indices $\{i, j\}$. Then, the sum $\sum_{i,j} \mathbf{P}_{i,j} = \mathbf{P}_{1,1} + \ldots + \mathbf{P}_{1,2}$ is always (element-wise) greater than or equal to the partial sums $\sum_i \mathbf{P}_{i,j} = (\mathbf{P}_{1,1} + \mathbf{P}_{2,1}, \mathbf{P}_{1,2} + \mathbf{P}_{2,2})$ and $\sum_j \mathbf{P}_{i,j} = (\mathbf{P}_{1,1} + \mathbf{P}_{1,2}, \mathbf{P}_{2,1} + \mathbf{P}_{2,2})$.

c) *Adding a tensor to a product.* For any non-negative term $\text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell} \geq 0$, adding any good tensor $\mathbf{G} \in \mathcal{T}$ to a product:

$$\text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell \cup \{\mathbf{G}\}} \geq \text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell}$$

It is easy to verify that, for every c' above, $c' \models c$.

Notice that none of the above operations changes the right hand side of c: doing so would amount to replacing the tensor $\mathbf{Z}$ with some other tensor. However, all valid alternatives are already implicitly enumerated when contructing $\rho(\top)$.

3.5 Dealing with Bad Tensors

For bad tensors, which have elements between 0 and 1, cases (a) and (b) are still valid, but case (c) is not. Indeed, adding a bad tensor to a product reduces the value of the LHS, rather than increasing it. The refinement operator has to be extended with:

d) *Removing a bad tensor from a product.* Removing a bad tensor $\mathbf{B}$ from any term $\text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell} \geq 0$:

$$\text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell \setminus \{\mathbf{B}\}} \geq \text{term}_{\mathcal{I}^\ell,\mathcal{X}^\ell}$$

Now, two desirable properties of refinement operators in our context are: *completeness*, i.e., $\rho^*(\top)$ corresponds to the set of all well-formed constraints; and *non-redundancy*, in that for every well-formed constraint c there is exactly one sequence of constraints $c_1, \ldots, c_n$ such that $c_1 \in \rho(\top), c_2 \in \rho(c_1), \ldots, c \in \rho(c_n)$. To ensure that our refinement operator is complete, it should always increase the left hand side of c by the tiniest possible amount. When bad tensors are considered, the minimal increase is obtained by multiplying together $p-1$ bad tensors, namley $\mathbf{B}_1 \cdot \ldots \cdot \mathbf{B}_{p-1}$. Consequently, case (a) has to be revised as follows:

a') *Adding a non-negative term to the LHS.* For every non-negative tensor $\mathbf{P} \in \mathcal{T}$, adding $\mathbf{P} \prod_{i=1}^{p-1} \mathbf{B}_i$, that is:

$$\sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k} + \mathbf{P} \prod_{i=1}^{p-1} \mathbf{B}_i \geq \sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k}$$

Of course, in the absence of bad tensors (a) and (a') coincide.

3.6 Dealing with Ugly Tensors

Unlike good or bad tensors, multiplying a term with an ugly tensor neither specializes nor generalizes the constraint. Therefore, the general to specific search fails for this specific case; that is why ARNOLD uses a more elaborate strategy to automatically enumerate them. For each constraint generated during general to specific search, ARNOLD generates all possible constraints by multiplying an ugly tensor to any term in the left hand side and adds this constraint in the enumeration process. Although this increases the time taken to enumerate all constraints, it ensures the completeness property of the refinement operator.

3.7 Nested Lexicographic Ordering

Notice that our refinement operator ρ is not optimal, as distinct constraints c_1 and c_2 can have the same specialization c'. For instance, if $\mathcal{T} = \{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$, then $\mathbf{X} + \mathbf{Y} \leq \mathbf{Z}$ can be obtained by specializing either $\mathbf{X} \leq \mathbf{Z}$ or $\mathbf{Y} \leq \mathbf{Z}$. In this case, c' and all of its descendants would be enumerated twice.

To avoid this situation, we define a nested lexicographic ordering over *tensors*, *indices*, *terms* and *operations* used for refinement in the following way: (1) $\rho(c)$ can only use operations of higher or equal lexical rank compared to the operations used to reach c from a most general constraint; (2) $\rho(c)$ can only modify a term of higher or equal lexical rank compared to the terms modified in c; and (3) $\rho(c)$ can only add or remove tensors (resp. indices) from a term that have higher or equal lexical order compared to the tensors (resp. indices) added or removed from that term previously.

For tensors and indices we used standard alphabetical order. For terms, the one added last has the highest order, so that ρ can modify it. For operations, using the wrong order can make enumeration skip over some constraints. For example, in an ordering where (b) $\preceq$ (a'), after adding a term we can not sum it over any index. To ensure the completeness of the refinement operator we define the following ordering: (a') $\preceq$ (d) $\preceq$ (c) $\preceq$ (b). The intuition is that: (a') $\preceq$ (d), because a bad tensors must be added to a term before it can be removed; (d) $\preceq$ (c), since terms added by (a') have p tensors, we allow (d) to remove them before allowing (c) to add more; (c) $\preceq$ (b), since operators preceding (b) make changes to a term, we allow (b) to add indexes to any of these modified terms.

3.8 Dealing with Negative Tensors

A side-effect of only allowing non-negative tensors is that constraints like $\sum_{i,j} X_{i,j} \geq \mathbf{Z}$, or equivalently $-\sum_{i,j} X_{i,j} \leq -\mathbf{Z}$, can not be represented. This can be fixed by allowing both non-positive and non-negative tensors. Doing so amounts to factorizing the sign out of the tensors themselves and into the terms and constraints in Eq. 2–3, thus updating them to:

$$\begin{aligned} \text{term}_{\mathcal{I},\mathcal{X}} &= \pm \textstyle\sum_{(i_1,i_2,\ldots)\in \text{range}(\mathcal{I})} \prod_{\mathbf{X}\in\mathcal{X}} \mathbf{X}_{i_1,i_2,\ldots} \\ \textstyle\sum_k \text{term}_{\mathcal{I}^k,\mathcal{X}^k} &\leq \pm \mathbf{Z} \end{aligned}$$

In doing so, we can safely assume that all tensors in $\mathcal{T}$ are non-negative, w.l.o.g. The tensors which have both positive and negative values are hard to deal with but are also very infrequent. We did not encounter any such tensor (Table 2) and thus leave it for the future work. The refinement operator must be adapted accordingly. Most importantly, $\rho(\top)$ now contains all constraints with the most negative possible left hand side, i.e., the sum of the most negative terms that can be built with the tensors in $\mathcal{T}$. We add four more operations in

Problem $\mathcal{P}_*$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{C}\rvert$	Largest	$\#G$	$\#B$	$\#U$
knapsack	1	4	5×1	4	0	1
shipping	1	4	4×3	4	0	1
social golfers	1	5	9×4	6	0	0
assignment	2	4	5×4	5	1	0
capital budget	2	5	4×1	6	0	1
stuckey assignment	2	5	5×5	6	1	0
schedule	3	5	7×1	8	0	0
rostering	3	5	7×3	8	0	0
curriculum	3	7	46×8	9	1	0
scheduling bratko	5	6	1×1	10	1	0

Table 2: Properties of the problems used in the experiments, including (left to right): number of variables, number of constants, size of the largest tensor, and number of good/bad/ugly tensors.

ρ to deal with negative terms. Each of these are analogous to the operations defined above. For example, similar to (a), removing a negative term from the left hand side of a constraint c produces a more specific constraint. The complete list is: e) *Removing a non-positive term from the LHS.* f) *Removing an index from a non-positive term.* g) *Removing a good tensor from a non-positive term.* h) *Multiplying a bad tensor to a non-positive term.* We define an ordering for these added operations which is just the opposite of the order defined for the analogous operations earlier. For example, previously it made sense to add a term before adding an index to that term. Similarly, now it makes sense to remove an index from a term before removing the term. The complete ordering of operations used is given by: (f) $\preceq$ (g) $\preceq$ (h) $\preceq$ (e) $\preceq$ (a') $\preceq$ (d) $\preceq$ (c) $\preceq$ (b). Notice that the operations for non-positive terms precedes the ones for non-negative terms because the LHS in $\rho(\top)$ starts with all negative terms.

We ensure that ρ produces well formed constraints by allowing it to only add terms compatible with sum, i.e., dimensions of terms must be same. Thus, ρ is complete as long as only good and bad tensors are present, and optimal for positive terms and good tensors. The constraints with ugly tensors are enumerated separately as discussed above.

4 Empirical Analysis

We addressed the following research questions: **Q1**) Does ARNOLD acquire accurate integer programs? **Q2**) Is ARNOLD efficient in practice? **Q3**) Do pruning and nested lexicographic ordering reduce the runtime of ARNOLD? To this end, we used ARNOLD for learning 10 satisfaction/satisficing MiniZinc [Nethercote *et al.*, 2007] benchmark integer programs[1], detailed in Table 2. To ensure that learning is challenging enough, we chose programs with ≥ 5 tensors.

For each program $\mathcal{P}_*$, first we sampled examples of feasible solutions, and then checked how well ARNOLD could recover it. Sampling independent solutions of integer programs is non-trivial, so we enumerated 10,000 (correlated) solutions using the Gecode solver [Schulte *et al.*, 2006] and then resorted to reservoir sampling [Tillé, 2011] to obtain a subset of 125 solutions. Next, we split the dataset into five folds of 25 solutions each and fed ARNOLD with $n \in \{1, 2, 10, 25\}$ random solutions from one fold as training set, while using the union of the other four folds for performance evaluation. For each learned program $\mathcal{P}_L$, we measured its recall and precision with respect to the hidden program $\mathcal{P}_*$, namely Pr $= |\text{Sol}(\mathcal{P}_*) \cap \text{Sol}(\mathcal{P}_L)|/|\text{Sol}(\mathcal{P}_L)|$ and Rc $= |\text{Sol}(\mathcal{P}_*) \cap \text{Sol}(\mathcal{P}_L)|/|\text{Sol}(\mathcal{P}_*)|$. Exact computation of these quantities is not trivial, so they were estimated using sampling: for the recall, the four test folds were used as samples, while for precision 1,000 solutions were sampled anew from $\mathcal{P}_L$. The procedure was repeated as in 5-fold cross-validation.

Table 1 shows the average precision and recall of the programs learned by ARNOLD using different target complexity (s, p), and number of examples n. The parameters used were $(s, p) = (1, 1), (1, 2), (1, 3), (2, 1), (3, 1)$, which are large enough to capture the majority of the benchmark problems, and $n = 1, 5, 10, 25$. The performance changes monotonically with increasing n, so the results for $n = 5, 10$ are not reported. Blanks indicate time-outs, i.e., $>$ 6h per fold.

In general, increasing s or p increases precision—as more complex constraints are captured by the learned program—but it tends to decrease recall if not enough examples are available. In other words, too complex programs may overfit, as expected. However, if n is large enough, then ARNOLD achieves high performance: in 8 out of 10 programs, the precision and recall both surpass 90%. For instance, when p increases to 2, captial budget's precision jumps to 100%, and similarly, schedule can be learned perfectly if $s \geq 2$. Notice that these results are obtained using $n = 25$ examples at most. Further, the behavior with $n = 10$ examples is essentially the same (data not reported). The two remaining programs (social golfers and rostering) represent the satisfaction problem in a slightly incompatible manner (namely, using equality checks over numerical variables). In principle, a simple change in representation would make them learnable. We plan to automate this step in future work. This allows us to answer **Q1**: ARNOLD does acquire good IP programs so long as they are representable in its language, as expected.

Understandably, the number of learned constraints increases with $|\mathcal{V}|$, $|\mathcal{C}|$, s, and p, while it decreases as more examples are given. With $n = 25$ examples, ARNOLD learns 50–250 (avg. 160) constraints for $s = 1$, $p = 3$, and about 300–1000 (avg. 600) for $s = 3$, $p = 1$. This number can be substantially reduced by pruning entailed constraints. Runtime also increases along the same dimensions, but only linearly in the number of examples. In the most complex task ($s = 3$ and $p = 1$), ARNOLD takes 20m–3.5h (avg. 1.5h) to complete with $n = 25$ examples. Although it seems that the scalability is limited, but the learned models can be re-used multiple times, so it makes sense to use ARNOLD for larger IPs as well. Question **Q2** can thus be answered affirmatively.

To answer **Q3**, we compared ARNOLD to two naive versions without general-to-specific pruning or nested lexicographic ordering, respectively. Since the naive algorithms take much longer to complete than ARNOLD, we compared them on knapsack, the simplest benchmark program (cf. Table 2). Notice that the naive algorithms—by construction—have the same recall and precision of ARNOLD, so these are

[1]From github.com/MiniZinc/benchmarks and from hakank.org/minizinc. Our code is available at github.com/mohitKULeuven/arnold

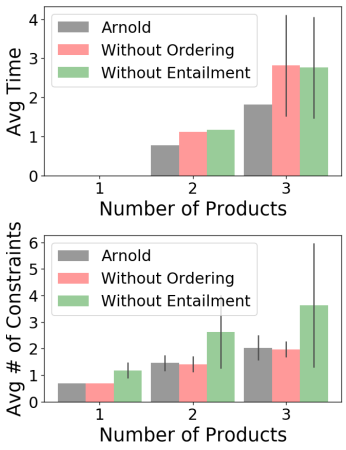

Figure 2: Effect of general-to-specific pruning and nested lexicographic ordering. Top: average runtime (and variance) for $s = 1$ and $p = 1, 2, 3$ (logarithmic scale - base 10). Bottom: average number of acquired constraints (logarithmic scale). (Best viewed in color.)

Problem $\mathcal{P}_*$	n	$s, p = 1, 1$	$s, p = 1, 2$	$s, p = 1, 3$	$s, p = 2, 1$	$s, p = 3, 1$
knapsack	1	0.70, 0.12	0.54, **1.00**	0.49, **1.00**	0.20, 0.20	0.11, 0.20
	25	**0.99**, 0.09	**0.97**, **1.00**	**0.96**, **1.00**	**0.96**, 0.30	**0.92**, 0.32
shipping	1	**1.00**, **1.00**	**1.00**, **1.00**	0.80, **1.00**	**1.00**, **1.00**	**1.00**, **1.00**
	25	**1.00**, **1.00**	**1.00**, **1.00**	**1.00**, **1.00**	**1.00**, **1.00**	**1.00**, **1.00**
assignment	1	0.49, 0.03	0.19, **1.00**	0.11, **1.00**	0.16, 0.06	0.06, 0.10
	25	**0.98**, 0.01	**0.98**, **1.00**	**0.98**, **1.00**	**0.98**, 0.03	**0.97**, 0.03
capital-budget	1	0.36, 0.01	0.06, **1.00**	0.01, **1.00**	0.03, 0.04	0.01, 0.10
	25	**0.96**, 0.01	**0.95**, **1.00**	**0.91**, **1.00**	**0.92**, 0.00	**0.91**, 0.01
stuckey-assignment	1	0.35, 0.03	0.10, **1.00**	0.10, **1.00**	0.24, 0.76	0.13, 0.03
	25	**1.00**, 0.01	**0.98**, **1.00**	**0.98**, **1.00**	**0.98**, 0.09	•
schedule	1	**1.00**, 0.00	**0.95**, 0.00	0.20, 0.72	**1.00**, **1.00**	**1.00**, **1.00**
	25	**1.00**, 0.00	**0.98**, 0.00	**0.97**, 0.00	**1.00**, **1.00**	•
curriculum	1	0.82, 0.00	0.27, **1.00**	0.08, **1.00**	0.02, 0.00	•
	25	**1.00**, 0.00	**0.96**, **1.00**	**0.96**, **1.00**	0.93, 0.00	•
scheduling-bratko	1	0.02, 0.04	0.04, 0.40	0.00, 0.50	0.00, **1.00**	0.00, **1.00**
	25	**1.00**, 0.00	**0.92**, 0.00	**0.91**, 0.00	**0.90**, **1.00**	•
social-golfers	1	**1.00**, 0.00	0.71, 0.00	0.68, 0.00	**0.91**, 0.00	**0.91**, 0.00
	25	**1.00**, 0.00	**1.00**, 0.00	**0.98**, 0.00	**0.98**, 0.00	**0.98**, 0.00
rostering	1	**1.00**, 0.00	0.87, 0.00	0.65, 0.00	**1.00**, 0.00	**1.00**, 0.00
	25	**1.00**, 0.00	**1.00**, 0.00	**0.99**, 0.00	**1.00**, 0.00	•

Table 1: Cells report the average (recall, precision) of the program learned by ARNOLD; n, s, and p are as in the main text. Bold indicates values above while 90%, • represents time-outs.

not reported. The cross-validated runtime and number of acquired constraints for $n \in \{1, 25\}$, $s = 1$ and $p \in \{1, 2, 3\}$ are shown in Figure 2. As we increase the complexity of the learned model, ARNOLD becomes much faster compared to both the naïve algorithms: in the most complex setting, namely $p = 3$, ARNOLD is more than 8 times faster than the alternatives. Switching off pruning also increases the number of constraints learned, by up to 40 times for $p = 3$.

5 Related Work

Learning IPs from examples has not received a lot of attention. Existing approaches include ESOCCS [Pawlak, 2019] and CountOR [Kumar *et al.*, 2018]. ESOCCS is a heuristic approach based on evolutionary optimization. It converts positive-only learning to binary classification by sampling negatives from an estimated density. ESOCCS can learn weighted constraints. Its strategy could be adapted to learn IP objective functions; we leave this to future work. However, contrary to ARNOLD, ESOCCS expects the terms to be enumerated upfront, which is impractical in IP as the number of terms can be huge. Furthermore, it does not exploit structure of the set of variables. On the contrary, CountOR uses tensors to represent and acquire IPs, but it focuses on a restricted class of constraints capturing ranges of quantities of interest, which are common in scheduling tasks. In contrast, ARNOLD acquires arbitrary polynomial constraints, which are much more expressive.

ARNOLD is based on general principles from constraint acquisition [De Raedt *et al.*, 2018; Bessiere *et al.*, 2017] and inductive programming [De Raedt, 2008]. Constraint acquisition has been studied, for instance, by [Bessiere *et al.*, 2017] and [Beldiceanu and Simonis, 2012], which both focus on constraint programming problems rather than integer programs. One advantage of the IP formulation is that it naturally ties together sets of variables into tensors, allowing to easily deal with high-dimensional constraints. [Bessiere *et al.*, 2017] employ a bi-directional search for finding a constraint satisfaction problem. Their approach does not exploit structure on the variables. Recent extensions [Arcangioli *et al.*, 2016; Tsouros *et al.*, 2018] also neglect multi-dimensional structures. The most related approach is the ModelSeeker [Beldiceanu and Simonis, 2012], which learns constraint programs from positives only by folding the data into tensors of different shapes. However, it can only deal with a single input vector. ARNOLD instead discovers polynomial terms and constraints and can deal with multiple input tensors.

ARNOLD's algorithm is reminiscent of the clausal discovery engine [De Raedt and Dehaspe, 1997], which searches for a set of clauses (i.e., logical constraints) that hold in databases. While the algorithm and the use of refinement are similar, ARNOLD searches a completely different space of constraints. On the other hand, IPs can also be used to represent (propositional) logic constraints, and therefore it could be interesting to use ARNOLD to learn sets of purely logical constraints. ARNOLD is also be related to systems that find equations in data [Todorovski and Dzeroski, 1997; Lloyd *et al.*, 2014]. The difference with ARNOLD is that these systems focus on continuous distributions and on equations rather than on integer programs and inequalities.

6 Conclusion

We presented ARNOLD, an approach for learning integer programs from examples of feasbile solutions. ARNOLD cleverly enumerates non-linear inequalities using a general-to-specific search (based on a novel refinement operator) and a nested lexicographic ordering. Crucially, ARNOLD exploits multi-dimensional numerical tensors and handles non-linear operations among them. Experiments show that ARNOLD can accurately acquire programs from a handful of examples.

Acknowledgements

The authors thank Paolo Dragone for help with setting up 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

[Arcangioli *et al.*, 2016] Robin Arcangioli, Christian Bessiere, and Nadjib Lazaar. Multiple constraint aquisition. In *IJCAI: International Joint Conference on Artificial Intelligence*, pages 698–704, 2016.

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

[Bessiere *et al.*, 2017] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

[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 and Dehaspe, 1997] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.

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

[De Raedt, 2008] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[Jiang *et al.*, 2013] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.

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

[Lloyd *et al.*, 2014] James Robert Lloyd, David Duvenaud, Roger Grosse, Joshua Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *Twenty-eighth AAAI conference on artificial intelligence*, 2014.

[Nemhauser and Wolsey, 1989] G. L. Nemhauser and L. A. Wolsey. Optimization. chapter Integer Programming, pages 447–527. Elsevier North-Holland, Inc., New York, NY, USA, 1989.

[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.

[Pawlak, 2019] Tomasz P Pawlak. Synthesis of mathematical programming models with one-class evolutionary strategies. *Swarm and evolutionary computation*, 44:335–348, 2019.

[Schulte *et al.*, 2006] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and Programming with Gecode. 2006.

[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.

[Tillé, 2011] Yves Tillé. *Sampling algorithms*. Springer, 2011.

[Todorovski and Dzeroski, 1997] Ljupco Todorovski and Saso Dzeroski. Declarative bias in equation discovery. In *ICML*, pages 376–384, 1997.

[Tsouros *et al.*, 2018] Dimosthenis C Tsouros, Kostas Stergiou, and Panagiotis G Sarigiannidis. Efficient methods for constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 373–388. Springer, 2018.