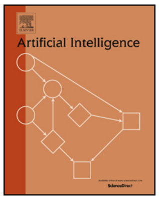
Artificial Intelligence 244 (2017) 188–216

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Relational linear programming

Kristian Kersting[a,*], Martin Mladenov[a], Pavel Tokmakov[b]

[a] CS Department, TU Dortmund University, Germany
[b] LEAR-INRIA Rhone-Alpes, Montbonnot, France

ARTICLE INFO

Article history:
Received in revised form 25 June 2015
Accepted 28 June 2015
Available online 2 July 2015

Keywords:
Machine learning
Optimization
Relational logic
Statistical relational learning
Linear programming
Symmetry
(Fractional) automorphism
Color-refinement
Lifted probabilistic inference
Lifted linear programming
Equitable partitions
Orbit partitions

ABSTRACT

We propose relational linear programming, a simple framework for combining linear programs (LPs) and logic programs. A relational linear program (RLP) is a declarative LP template defining the objective and the constraints through the logical concepts of objects, relations, and quantified variables. This allows one to express the LP objective and constraints relationally for a varying number of individuals and relations among them without enumerating them. Together with a logical knowledge base, effectively a logic program consisting of logical facts and rules, it induces a ground LP. This ground LP is solved using lifted linear programming. That is, symmetries within the ground LP are employed to reduce its dimensionality, if possible, and the reduced program is solved using any off-the-shelf LP solver. In contrast to mainstream LP template languages such as AMPL, which features a mixture of declarative and imperative programming styles, RLP's relational nature allows a more intuitive representation of optimization problems, in particular over relational domains. We illustrate this empirically by experiments on approximate inference in Markov logic networks using LP relaxations, on solving Markov decision processes, and on collective inference using LP support vector machines.

1. Introduction

Modern social and technological trends result in an enormous increase in the amount of accessible data, with a significant portion of the resources being interrelated in a complex way and having inherent uncertainty. Such data, which we may refer to as relational data, arises for instance in social network and media mining, natural language processing, open information extraction, the web, bioinformatics, and robotics, among others. Making this data amenable to computing machinery typically yields substantial social and/or business value. Therefore, it is not surprising that probabilistic logical languages[1] are currently provoking much new AI research with tremendous theoretical and practical implications. By combining aspects of logic and probabilities—a dream of AI dating back to at least the late 1980's when Nils Nilsson introduced the term probabilistic logics [5]—they help to effectively manage both complex interactions and uncertainty in the data.

However, instead of looking at AI through the glasses of probabilities over possible worlds, we may also approach it using optimization. That is, we have a preference relation, i.e., some objective function over possible worlds, and we want a best possible world according to the preference. Consider for example a typical data analyst solving a machine learning problem for a given dataset. She selects a model for the underlying phenomenon to be learned (choosing a learning bias), formats

* Corresponding author.
E-mail addresses: kristian.kersting@cs.tu-dortmund.de (K. Kersting), martin.mladenov@cs.tu-dortmund.de (M. Mladenov), pavel.tokmakov@inria.fr (P. Tokmakov).

[1] We refer to e.g. [1–4] and references therein for overviews.

http://dx.doi.org/10.1016/j.artint.2015.06.009
0004-3702/© 2015 Elsevier B.V. All rights reserved.

the raw data according to the chosen model, tunes the model parameters by minimizing some objective function induced by the data and the model assumptions, and may iterate the last step as part of model selection and validation. This is an instance of the declarative "**Model** + **Solver**" paradigm that was and is prevalent in AI [6], natural language processing [7], machine learning [8], and data mining [9]: instead of outlining how a solution should be computed, we specify what the problem is in terms some high-level modeling language and solve it using general solvers.

Unfortunately, however, today's solvers for mathematical programs typically require that the program is presented in solver-readable form and/or offer only some very restricted modeling environment. For example, a solver may require that a set of linear constraints be presented as a system of linear inequalities $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$. This can create severe difficulties for the user. First of all, the process of turning the intuition that defines the model "on paper" into a solver-readable form could be quite cumbersome. Consider the following example from graph isomorphism [10]. Given two graphs G and H, the LP formulation introduces a variable for every possible partial function mapping k vertices of G to k vertices in H. It is not a trivial task to come up with a convenient linear indexing of the variables, let alone expressing the resulting inequalities $\boldsymbol{A}\mathbf{x} \leq \mathbf{b}$. It requires the user to produce and maintain complicated matrix generation code, which can be tedious and error-prone. Moreover, the reusability of such specialized code is limited, as relatively minor modifications of the equations could require large modifications of the code (for example, the user decides to switch from having variables over sets of vertices to variables over tuples of vertices). Ideally, one would like to separate the rules that generate the problem from the problem instance itself. Finally, solver-readable forms are inherently propositional. By design they cannot model domains with a variable number of individuals and relations among them without enumerating all of them. As already mentioned, however, many AI tasks and domains are best modeled in terms of individuals and relations. Agents must deal with heterogeneous information of all types. Even more important, they must often build models before they know what individuals are in the domain and, therefore, before they know what variables exist. Hence modeling should facilitate the formulation of abstract, general knowledge.

To overcome these downsides and triggered by the success of probabilistic logical languages, we show that optimization is liftable to the relational level too. Specifically, we lift linear programs—the most tractable, best understood, and widely used in practice fragment of mathematical programs—by introducing **relational linear programs** (RLPs). They are declarative LP templates defined through the logical concepts of individuals, relations, and quantified variables, and allow the user to express LP objectives and constraints for a varying number of individuals without enumerating them. For instance, the following code

```
subject to {vertex(X),not source(X),not target(X)}: outflow(X)-inflow(X)=0;
```

encodes the conservation of flows for all vertices of a graph that are not source or target. Together with a logical knowledge base (LogKB) referring to the individuals and relations (effectively a logical program consisting of logical facts and rules), it induces a ground LP that can be solved using any LP solver. However, since RLPs consist of templates that are instantiated several times to construct a ground linear model, they are also likely to induce ground models that exhibit symmetries, and we will demonstrate how to detect and exploit them. As our main technical contribution, we introduce **lifted linear programming** (LLP). It detects symmetries in a linear program in quasilinear time and eliminates them by reparametrizing the original linear program into a smaller one sharing optimal solutions that can be computed using any LP solver.

Both contributions together result in **relational linear programming**, best summarized as

$$\big((\mathrm{LP} + \mathrm{Logic}) - \mathrm{Symmetry}\big) + \mathrm{Solver}.$$

The user describes a relational problem in a high-level, relational LP modeling language and—given a logical knowledge base (LogKB) encoding some individuals or rather data—the system automatically compiles a symmetry-reduced LP that in turn can be solved using any off-the-shelf LP solver.

Our empirical illustrations on several AI tasks such as computing optimal value-functions of Markov decision processes [11], approximate inference within Markov logic networks [12] using LP relaxations, and collective classification [13] demonstrate that relational linear programming can

- ease the process of turning the "modeler's form", i.e., the form in which the modeler understands a problem or actually a class of problems (since we can now deal with a varying number of individuals and relations among them), into a solver-readable form, and
- considerably reduce the time spent to compute solutions.

The present paper is a significant extension of a previously published conference paper [14]. It provides a much more concise development of LLP and the first coherent view on relational linear programming as a novel and promising way for scaling AI by developing and showcasing the first modeling language for LPs based on logic programming. One of the advantages of the language is the closeness of its syntax to the mathematical notation of LP problems while supporting language elements from logic programming such as individuals, relations, and quantified variables. This allows for a very concise and readable relational definition of linear optimization problems in a general, declarative fashion; the (relational) algebraic formulation of a model does not contain any hints how to process it.

We proceed as follows. We start off by reviewing linear programming and existing LP template languages in Section 2. Then, in Section 3, we introduce relational linear programming, both the syntax and the semantics. Afterwards, Section 4 shows how to detect and exploit symmetries in linear programs. Section 5 illustrates relational linear programming empirically on several examples from machine learning and AI. Before concluding, we finally touch upon related work and directions for future work.

2. Linear programming

A linear program (LP) [15] is an optimization problem that can be expressed in the following general form:

$$\begin{aligned} \text{minimize}_{\mathbf{x}\in\mathbb{R}^n} \quad & \langle \mathbf{c}, \mathbf{x}\rangle \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} \le \mathbf{b} \\ & \boldsymbol{G}\mathbf{x} = \mathbf{h}\,, \end{aligned}$$

where $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ and $\boldsymbol{G} \in \mathbb{R}^{p\times n}$ are matrices, $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{h}$ are real vectors of dimension m, n, and p respectively, and $\langle\cdot,\cdot\rangle$ denotes the scalar product of two vectors. Note that the equality constraints can be reformulated as inequality constraints to yield an LP in a form containing only inequalities,

$$\begin{aligned} \text{minimize}_{\mathbf{x}\in\mathbb{R}^n} \quad & \langle \mathbf{c}, \mathbf{x}\rangle \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} \le \mathbf{b}\,, \end{aligned} \tag{1}$$

which we represent by the triplet $L = (\boldsymbol{A}, \mathbf{b}, \mathbf{c})$.

LPs have found a wide application in the fields of operations research, where they are applied to problems like multi-commodity flow and optimal resource allocation, and combinatorial optimization, where they provide the basis for many approximation algorithms for hard problems such as TSP. They have also found their way to machine learning and AI. For instance, they have been used for classification [16–18], for structured prediction [19], as subroutines in collective classification approaches [20,21], for efficient approximate MAP inference (see e.g. [22]) for finding the optimal policy of Markov decision problems [23,24], for inverse reinforcement learning [25], and clustering and dimensionality reduction [26,27].

While LP models often look intuitive on paper, applying them in practice presents a challenge. The main issue is the following gap:

The solver-readable form of an LP—the L-triplet representation as shown above—is not the form that is natural for users.

Furthermore, the matrix $\boldsymbol{A}$ is typically sparse, i.e., having mostly 0-entries. Consequently, modeling any real world problem in the solver-friendly L-form can be quite error prone and time consuming. This might explain why LPs are typically specified in a more declarative, abstract way that we will discuss next.

2.1. Declarative modeling languages for linear programs

Linear programs are often presented in a declarative, abstract way, which separates the general structure of a problem at hand from the actual instance. As an example, consider maximizing the total flow in a network [28]. The problem is, given a finite directed graph $G(V, E)$ in which every edge $(u, v) \in E$ has a non-negative capacity $c(u, v)$, and two vertices s and t called source and target, to maximize a function $f : V \times V \to \mathbb{R}$, called flow. In order to be a valid flow, f must be subjected to the following constraints: $f(u, v) \le c(u, v)$, $f(u, v) \ge 0$, and

$$\sum\nolimits_{w\in V/\{s,t\}} f(u,w) = \sum\nolimits_{w\in V/\{s,t\}} f(w,u)\,.$$

The latter constraint asserts that incoming flow equals to outgoing flow for internal vertices, that is, no "goods" are created or destroyed in nodes that are not the source or the sink. To build the flow LP for a given network, we first introduce variables to represent flow over each edge of the network. Then, we formulate the capacity constraints and conservation constraints depending on the network structure. Finally, the objective function of the LP is the yields the value of the total flow in the network.

As one can see, the general definition of the flow LP can be separated from the specification of the network at hand and, in turn, be applied to different networks. Such a declarative, abstract modeling is realized using algebraic modeling languages (see Section 6 for references). They simplify LP definitions by allowing one to use algebraic notation instead of matrices to define the objective and the constraints through parameters whose domains are defined in a separate file, thus enabling model/instance separation.

Let us have a look at the L-form. Starting from (1), algebraic modeling languages typically make the involved arithmetic expressions explicit:

$$\begin{aligned} \text{minimize}_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \sum\nolimits_{j\in P} c_j x_j \\ \text{subject to} \quad & \sum\nolimits_{j\in P} a_{ij} x_j \le b_i \quad \text{for each } i \in K\,, \end{aligned} \tag{2}$$

```
set P;                         #column dimension of A
set K;                         #row dimension of A
param a {j in P, i in K};      #provided as input
param c {j in P};              #provided as input
param b {i in K};              #provided as input
var x {j in P};                #determined by the solver

#the objective
minimize : sum {j in P} c[j] * x[j];
#the constraints
subject to sum {j in P} a[i, j]*x[j] <= b[i];
```

Fig. 1. AMPL declaration scheme for a linear program in "set form" as shown in (2).

```
set VERTEX;                                                     #vertices
set EDGES within (VERTEX diff sink) cross (VERTEX diff source); #edges

param source symbolic in VERTEX;                         #entrance to the graph
param sink symbolic in VERTEX, <> source;                #exit from the graph
param cap EDGES >= 0;                                  #flow capacities

var Flow {(i,j) in EDGES} >= 0, <= cap[i,j];             #flows

maximize: sum {(source,j) in EDGES} Flow[source,j];      #objective

subject to {k in VERTEX diff {source,sink}}:             #conservation of flow
  sum {(i,k) in EDGES} Flow[i,k] = sum {(k,j) in EDGES} Flow[k,j];
```

Fig. 2. AMPL specification for a general flow linear program. The vertices and edges as well as the corresponding flow capacities are provided in separate files. The flows of the edges are declared to be determined by the LP solver.

where the sets P and K, as well as the corresponding non-zero entries of vectors $\mathbf{c}$, $\mathbf{b}$ and matrix $\boldsymbol{A}$ are defined in a separate file. This simplified representation is then automatically translated to the matrix format and fed to a solver that the user prefers.

To code the LP in this "set form", several mathematical programming modeling languages have been proposed. According to NEOS solver statistics,[2] AMPL is the most popular one. We only briefly review the basic AMPL concepts. For more details, we refer to [29,30].

Based on the "set form", an LP can be written in AMPL as shown in Fig. 1. In principle, an AMPL program consists of one objective (a line starting with a **maximize** or **minimize** keyword), and a number of ground or indexed constraints (lines starting with a **subject to** keyword). If a constraint is indexed (i.e. the constraint in the example above is indexed by the set K), a ground constraint is generated for every combination of values of the indexing variables (in the example above, there is just one index variable in the constraint, hence a ground constraint is generated for every value in K). The keyword **set** declares a set name, whose members are provided in a separate file. The keyword **param** declares a parameter, which may be a single scalar value or a collection of values indexed by a set. Subscripts in algebraic notation are written in square brackets as in `b[i]` instead of b_i. The values to be determined by the solver are defined by the **var** keyword. The typical $\sum$ symbol is replaced by the **sum** keyword. The key element of the AMPL system are the so-called **indexing expressions**:

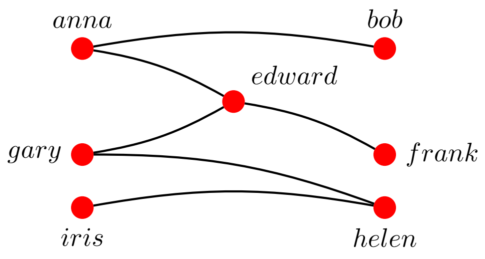
```
{j in P}.
```

In addition to being part of variable/parameter declaration, indexing expressions serve both as limits for sums and as indices for constraints. Finally, comments in AMPL start with the symbol #.

To illustrate AMPL, Fig. 2 shows the flow problems formulated in AMPL. The program starts with a definition of all sets, parameters and variables that appear in it. They are then used to define the objective and the constraints from the network flow problem, in particular the third one. The first two constraints are incorporated into the variable definition. As one can see, AMPL allows one to write down the problem description in a declarative way. It frees the user from engineering instance specific LPs while capturing the general properties of the problem class at hand.

Unfortunately, AMPL does not provide logically parametrized definitions for the arithmetic expressions and for the index sets. It is difficult to use the logical formula $\phi(\mathtt{X}) \equiv \exists \mathtt{Y}\ \mathtt{source(S)} \wedge \mathtt{edge(S, Y)} \wedge \mathtt{edge(Y, X)} \wedge \neg \mathtt{source(X)}$ to compactly code the set of all nodes that have a distance of two from the source node within a network. Relational linear programs (RLPs), which we will introduce now, feature exactly this. They allows us to keep AMPL's benefits that make it the number one choice for optimization experts and at the same time enable logical constructions that are established and effective tools for dealing with relational problems.

[2] http://www.neos-server.org/neos/report.html; accessed on April 19, 2014.



name	age	education	smokes
anna	27	uni	+
bob	22	college	−
edward	25	college	−
frank	30	uni	−
gary	45	college	+
helen	35	school	+
iris	50	school	−

Fig. 3. Example for collective inference. There are 7 people in a social network. Each person is described in terms of three attributes. The class label "cancer" is not shown.

3. Relational linear programs

The main idea is to parameterize AMPL's arithmetic expressions by logical variables and to replace AMPL's indexing expression by queries to a logical knowledge base. Before showing how to do this, let us briefly review logic programming. For more details we refer to [2,31,32].

3.1. Logic programs

A logic program is a set of clauses constructed using four types of symbols: constants, variables, functors, and predicates. Say we want to model the smoking habits of people. In addition to the usual "flat" data about attributes of people like age, education and smoking habits, we have access to the social network among the people, cf. Fig. 3. Formally speaking, we have that `attribute/2` and `friends/2` are **predicates** (with their *arity*, i.e., number of arguments listed explicitly). The symbols `anna, bob, edward, frank, gary, helen, iris` are **constants** and `X`, and `Y` are **variables**. All constants and variables are also **terms**. In addition, one can also have structured terms such as the **functor** `s(X)` of arity 1, which contains the **function symbol** `s` and the term `X`. **Atoms** are predicate symbols followed by the necessary number of terms, e.g., `friends(bob, anna)`, `nat(s(X))`, `attribute(X, passive)`, etc. **Literals** are atoms `nat(s(X))` (positive literal) and their negations `not nat(s(X))` (negative literals). We are now able to define the key concept of a **clause**. They are formulas of the form

$$\mathtt{A} :- \mathtt{B}_1, \ldots, \mathtt{B}_\mathtt{m}$$

where `A` (the head) and the $\mathtt{B}_\mathtt{j}$ (the body) are literals and all variables are understood to be universally quantified. For instance, the clause $c \equiv \mathtt{attribute(X, passive)} :- \mathtt{friends(X, Y)}, \mathtt{attribute(Y, smokes)}$ can be read as `X` has `attribute passive` if `X` and `Y` are `friends` and `Y` has the `attribute smokes`. Clauses with an empty body are **facts**. A **logic program** consists of a finite set of clauses. The set of variables in a term, atom, conjunction or clause E, is denoted as $\mathrm{Var}(E)$, e.g., $\mathrm{Var}(c) = \{\mathtt{X}, \mathtt{Y}\}$. A term, atom or clause E is **ground** when there is no variable occurring in E, i.e. $\mathrm{Var}(E) = \emptyset$.

A **substitution** $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$, e.g. $\{\mathtt{Y/anna}\}$, is an assignment of terms t_i to variables V_i. Applying a substitution θ to a term, atom or clause e yields the instantiated term, atom, or clause $e\theta$ where all occurrences of the variables V_i are simultaneously replaced by t_i, e.g. $c\{\mathtt{Y/anna}\}$ is $\mathtt{attribute(X, passive)} :- \mathtt{friends(X, anna)}, \mathtt{attribute(anna, smokes)}$. The **Herbrand base** of a logic program P, denoted as $\mathrm{hb}(P)$, is the set of all ground atoms constructed with the predicate, constant and function symbols in the alphabet of P. A **Herbrand interpretation** for a logic program P is a subset of $\mathrm{hb}(P)$. A Herbrand interpretation I is a **model** of a clause c if and only if for all substitutions θ such that $\mathrm{body}(c)\theta \subseteq I$ holds, it also holds that $\mathrm{head}(c)\theta \in I$. A clause c (logic program P) **entails** another clause c' (logic program P'), denoted as $c \models c'$ ($P \models P'$), if and only if, each model of c (P) is also a model of c' (P'). The **least Herbrand model** $\mathrm{LH}(P)$, which constitutes the semantics of the logic program P, consists of all facts $f \in \mathrm{hb}(P)$ such that P logically entails f, i.e. $P \models f$.

A **query** q is of the form $\mathtt{B}_1, \ldots, \mathtt{B}_\mathtt{m}$ where the $\mathtt{B}_\mathtt{j}$ are literals and all variables are understood to be existentially quantified. Given a logic program P, a correct answer to the query q is a substitution θ such as that $q\theta$ is entailed by P. That is, $q\theta$ is true in $\mathrm{LH}(P)$. The answer substitution θ is often computed using SLD-resolution. Finally, the **answer set** of a q is the set of all correct answers to q.

Logic programming is especially convenient for representing relational data such as our social network from Fig. 3. All one needs is the binary predicate `friends/2` to encode the edges in the social graph as well as the predicates `attribute(X, Attr)` to code the attributes (name, age, etc.) of the people in the social network.

3.2. Parametrizing linear programs using logic programs

Since our language can be seen as a logic programming variant of AMPL, we introduce its syntax in a contrast to the AMPL syntax.

A first important thing to notice is that AMPL mimics arithmetic notation in its syntax as much as possible. It operates on sets, intersections of sets and arithmetic expressions indexed with these sets. Our language for relational linear programming

```
 1 # declarations of LP predicates
 2 var flow/2, outflow/1, inflow/1;
 3 #objective
 4 maximize: sum {X in source(X)} {outflow(X)};
 5 #auxiliar variables capturing the outflow of nodes
 6 subject to forall{X in vertex(X)}:
 7    {outflow(X) = sum {Y in edge(X,Y)} {flow(X,Y)} };
 8 #auxiliar variables capturing the inflow of nodes
 9 subject to forall{Y in vertex(Y)}:
10    {inflow(Y) = sum {X in edge(X,Y)} {flow(X,Y)} };
11 #conservation of flow
12 subject to forall{X in vertex(X), not source(X), not sink(X)} :
13    {outflow(X) - inflow(X) = 0};
14 #capacity bound
15 subject to forall {X,Y in edge(X,Y)} : {cap(X,Y) - flow(X,Y) >= 0};
16 #no negative flows
17 subject to forall {X,Y in edge(X,Y)} : {flow(X,Y) >= 0};
```

Fig. 4. Relational linear program encoding the maximal flow problem. For details we refer to the main text.

effectively replaces these constructs with logical predicates, clauses, and queries to define the three main parts of an RLP: the objective template, the constraints template, and a logical knowledge base.

To illustrate this, let us reconsider the network flow problem as running example. An RLP for the flow example is shown in Fig. 4. It directly codes the flow constraints, concisely captures the essence of flow problems, and illustrates nicely that linear programming can be viewed as being highly relational in nature. Let us now discuss this program line by line.

3.2.1. *Logically parametrized algebraic expressions*

LP predicates define **logically parametrized sets of variables and parameters** in the LP. In our flow running example, `flow/2` captures for instance the flows between nodes. Sets that are explicitly defining domains in AMPL are discarded and parameter/variable domains are defined implicitly. In contrast to logic, (ground) LP atoms can take any numeric value, not just true or false. For instance the flow between the nodes is captured by `flow/2`, and the specific flow between node `f` and `t` could take the value 3.7, i.e., `flow(f, t) = 3.7`. Logically parameterized LP variables, **par-vars** for short, are values to be determined by the solver, and we follow AMPL's notation in lines 1–2:

```
1 # declarations of LP predicates
2 var flow/2, outflow/1, inflow/1;
```

The rest of the program specifies the objective (line 4) and the constraints (lines 5–17) of the flow problem. For this, we use logically parameterized algebraic expressions, or **par-expressions** in short.

Par-expressions are expressions of finite length of the form

$$\texttt{sum}\{\texttt{X}\ \texttt{in}\ \phi\}\ \underbrace{\{\psi_1\ \texttt{op}_1\ \psi_2\ \texttt{op}_2\ \ldots\ \texttt{op}_{n-1}\ \psi_n\}}_{=:\psi}$$

Here ψ_i are numeric constants, atoms or par-expressions, and the $\texttt{op}_j$ are arithmetic operators '$+$', '$-$', or '$\cdot$'. The term $\texttt{sum}\{\texttt{X in}\ \phi\}$, which is optional, essentially implements the AMPL aggregation **sum** but now indexed over the variables `X` that are a subset of the logical variables appearing in the logical query ϕ and in the LP atoms appearing in the ψ_i's; let atoms(ψ) denote this set. That is, the AMPL indexing expression `{j in P}` for the sum is turned into an indexing with respect to `X`. For those readers that a familiar with Prolog, essentially, one can think of this as calling the Prolog meta-predicate

$$\texttt{setof}\Big(\texttt{X}, (\phi, \text{atoms}(\psi)), \texttt{P}\Big)$$

treating atoms(ψ) as a conjunction over the involved atoms. This will produce the set `P` of all substitutions of the variables `X` (with any duplicate removed) such that the query $\phi \wedge \text{atoms}(\psi)$ is satisfied. In case, we are interested in multi-sets, i.e., to express counts, one may also use `findall/3`. This could be expressed using $\texttt{sum}\langle\phi\rangle$ instead of $\texttt{sum}\{\phi\}$. The `sum` aggregation and the involved par-expression is then evaluated over the resulting multidimensional index `P`. If no `sum` is provided, this will just be logical indexing for the evaluation of the par-expression $\psi_1\ \texttt{op}_1\ \psi_2\ \texttt{op}_2\ \ldots\ \texttt{op}_{n-1}\ \psi_n$.

Now, an **objective** is a par-expression without free logical variables. In our flow running example, the objective[3] reads e.g.

[3] For the sake of simplicity, we here assume that there are exactly one source and one sink vertex. If one wants to enforce this, we could simply add this as a logical constraint within the selection query, resulting in an empty objective if there are several source and sink notes. In AMPL, one would use additional `check` statements to express such restrictions, which cannot be expressed using simple inequalities.

```
3  #objective
4  maximize: sum {X in source(X)} outflow(X);
```

This says that we want to maximize the outflows for all source nodes. Note that we assume no free logical variables to avoid producing multiple and conflicting objectives.[4]

With par-expressions at hand, we can introduce **par-equalities**[5] and **par-inequalities** for specifying constraints. They are finite-length expression of the form

$$\phi_1 \ \mathtt{op}\ 0$$

where ϕ_1 is a par-expression and `op` denotes one of the operators '=', '<', '>', '<=', and '>='. Without loss of generality, we can assume the right hand side to be 0; if not we just move it to the left hand side by subtraction. We assume that all par-(in)equalities are all-quantified, which bounds all free variables not already bound by a `sum` statement. This is indicated by

$$\mathtt{forall}\{\mathtt{X}\ \mathtt{in}\ \phi\} : \{\psi\}$$

where `X` denotes all the free variables of the query ϕ and atoms(ψ). For each tuple in this indexed set, we then evaluate the par-(in)equality ψ. This way, constraints may lead to several ground instances.

In our flow running example, the constraints for the auxiliary LP predicate `outflow/2` is

```
5  #auxiliar variables capturing the outflow of nodes
6  subject to forall{X in vertex(X)}:
7     {outflow(X) = sum {Y in edge(X,Y)} {flow(X,Y)} };
```

Since the logical variable `Y` is bound by the summation for `outflow/1` and the logical variable `X` by the all-quantification, this says that there will be one equality expression as constraint per node summing over all flows of the outgoing edges. Likewise we can define the auxiliary LP predicate `inflow/2` summing over all ingoing edges:

```
8  #auxiliar variables capturing the inflow of nodes
9  subject to forall{Y in vertex(Y)}:
10    {inflow(Y) = sum {X in edge(X,Y)} {flow(X,Y)} };
```

The remaining constraints are defined in a similar fashion:

```
11 #conservation of flow
12 subject to forall{X in vertex(X), not source(X), not sink(X)} :
13    {outflow(X) - inflow(X) = 0};
14 #capacity bound
15 subject to forall {X,Y in edge(X,Y)} : {cap(X,Y) - flow(X,Y) >= 0};
16 #no negative flows
17 subject to forall {X,Y in edge(X,Y)} : {flow(X,Y) >= 0};
```

Already this simple flow example illustrates the power of RLPs. Since indexing expressions are logical queries, we can naturally express things that either look cumbersome in AMPL or even go beyond its capabilities. For instance the concept of **internal edge** (line 2 in Fig. 2), which is explicitly represented by a lengthy expression with sets intersections and differences in AMPL, is compactly represented using the logical query `vertex(X), not source(X), not target(X)` in the par-equations involving `in-/outflow` (line 12 in Fig. 4). As the reader may have noticed, several predicates such as `vertex/1` and `source/1` are not defined in the flow RLP. In such cases, we assume the predicate to be defined in a logical knowledge base LogKB that we will discuss next.

3.2.2. The logical knowledge base

Every predicate not defined to be a par-var is assumed to be a logically parameterized LP parameter, **par-param** for short, defined in a (possibly external[6]) logical knowledge base LogKB. We here use Prolog but assume that each query from the RLP produces a finite set of answers, i.e., ground substitutions of its logical variables rendering it to be true.

Fig. 5 illustrates an instance for the maximal flow problem. It can be expressed using the following LogKB:

```
cap(s,a) = 4.  cap(s,b) = 2.  cap(a,c) = 3.  cap(b,c) = 2.
cap(b,d) = 3.  cap(c,b) = 1.  cap(b,t) = 2.  cap(d,t) = 4.

edge(X,Y) :- cap(X,Y).
```

[4] Investigating free variables in the objective that in turn are bound by a leading `forall` statement is an attractive avenue for future work as it allows one to produce several LPs with shared constraints and to embed relational linear programs back into logic programming. Doing so goes beyond the scope of the present paper.

[5] We will sometimes also call them par-equations.

[6] Depending on the use case, some predicates may be defined as part of the LP, while others may be defined in an external database.

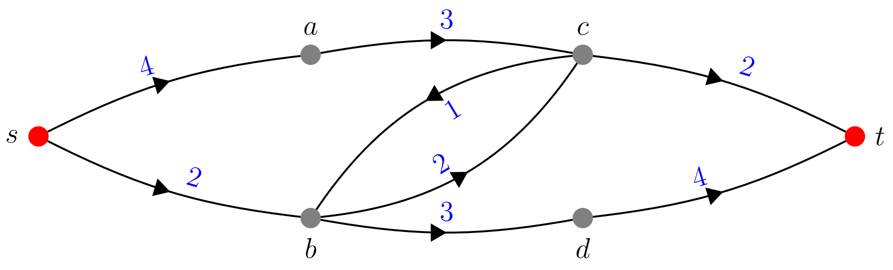


Fig. 5. A graph for a particular instance of the flow problem. The node s denotes the source, and t the target. The numbers associated with the edges are the flow capacities.

```
vertex(X) :- edge(X,_).
vertex(X) :- edge(_,X).

source(s). target(t).
```

where `cap(s, a) = 4` is short-hand notation for `cap(s,a,4)` and `cap(X, Y)` for `cap(X,Y,_)`, where we use an anonymized variable[7] '_'. The predicates `edge`/2 and `vertex`/1 are logical predicates taking the values 1 or 0 (corresponding to true and false) within the RLP. For instance `edge(X,Y)` is true for `{X/s,Y/a}`, `{X/a,Y/s}`, `{X/s,Y/b}`, `{X/b,Y/s}`, ... within the LogKB and, hence, they take the value 1 in the RLP. Finally, indeed, querying for a vertex could result in a multi-set as answer, since the definition of `vertex/1` tests for a vertex as the source and the target of an edge. However, recall that a logical indexing statement `{...}` removes any duplicates first.

Putting everything together, a **relational mathematical program** can be defined as follows.

Definition 1 *(Relational mathematical program).* A relational mathematical program consists of (1) a finite set of par-var declarations of predicates, (2) one par-expression to define the objective, and (3) a finite set of par-(in)equalities to define the constraints. Everything not explicitly defined is assumed to be a par-param defined in an external LogKB.

Finally, **relational linear programs (RLPs)** are relational mathematical programs where all involved par-expressions are **affine**. Affine par-expressions can be written (after potentially simplifying them) in the form $\sum_{i=1}^{n} \phi_i \cdot \psi_i$ where the ϕ_is are known quantities (LP parameters) and the ψ_is are LP atoms.

Let us now turn towards the semantics of relational linear programs.

3.2.3. The semantics and grounding

Together with a logical knowledge base LogKB, a relational linear program (RLP) induces a linear program (LP) as long as the logical queries in the RLP have finite answer sets:

Theorem 2. *An RLP together with a LogKB (such that the logical queries in the RLP have finite answer sets) induces a linear program (LP).*

Proof. Since there is only one objective and a finite set of constraints, it is sufficient to show that each of them induces only a finite set of ground instances of finite length. We will do so by induction over the depth of their syntax tree. Let us consider a par-equality constraint; all other cases follow the same principles:

$$\texttt{forall}\{\texttt{X in } \phi\} : \{\psi_1 \; \texttt{op}_1 \; \psi_2 \; \texttt{op}_2 \; \ldots \; \texttt{op}_{n-1} \; \psi_n \; = 0\} \, .$$

Assume $n = 1$. There are three cases. The statement is trivially true if ψ_1 is a constant (although this case may not really make sense for LPs). If ψ_1 is a parameter atom, its logical variables are bound by the logical query ϕ or by the LogKB. By assumption there are only finitely many groundings. Finally, if ψ_1 is an LP atom, its logical variables are bound by the logical query ϕ. Again, the answer set of ϕ is assumed to be finite. In both 'atomic' cases, this will generate a finite set of ground equality constraints; one for each answer of the free variables of ψ_1 unified with ϕ.

From this, by induction, it follows that any combination of $n > 1$ many LP atoms and constants using arithmetic operators induces only a finite set of ground instances.

What remains are `sum` statements. If we prove them to be 'finite', the theorem would follow. So assume that the par-equation involves `sum` statements. The first step we do is turning the par equation into a kind of **prenex normal form**. In logic, a formula is in prenex normal form if it is written as a string of quantifiers followed by a quantifier-free part. For the par-equation at hand, this means that one can be write it as a string of a single `forall` statement, followed by

[7] They can also be used withing the LP part of RLPs.

Algorithm 1: Grounding a relational linear program R.

```
   Input: RLP R together with a LogKB K
   Output: Ground LP L consisting of ground (AMPL) statements
1  Set L to the empty LP;
2  "Flatten" par-(in)equalities and the objective into prenex normal form by inlining definitions (see Section 3.2.4) and simplifying brackets;
3  for each par-(in)equality and the objective do
4      for each sum-aggreation and each separate atom do
5          Query the LogKB in order to obtain a grounding or a set of groundings if we are dealing with a constraint which involves an indexing
           expression;
6      Concatenate results of queries evaluation for each grounding to form a ground (in)equality resp. objective;
7      Add the ground (in)equality resp. objective to L;
8  return L
```

Algorithm 2: Relational linear programming.

```
1  Specify a relational linear program (RLP) R;
2  Given a logical knowledge base LogKB, ground R into an LP L, e.g. using Algorithm 1;
3  Solve L using any LP solver (after transforming it into the solver's input form);
```

a string of (zero or more) `sum` statements followed by a `sum`-free part, which is correspondingly simplified respectively rewritten. Actually, the block of `sum` statements can be merged after suitable renaming of the involved logical variables. Hence, without loss of generality, we can assume a single `sum`{`Y in` ϕ'}{ψ} string for some par-expression ψ of length n, which does not include any other `sum` statement.

Now, by our induction base, ψ itself would result in a finite set of ground arithmetic expressions as long as its free variables have been bound. This, however, is the case since by definition all free variables of ψ are bound by the LogKB, by the query ϕ' of the `sum` statement or by the logical query ϕ of the leading `forall` statement. Moreover, again by assumption, the answer set of the leading `forall` statement is finite. For each of its answers, we connect the resulting ground arithmetic expressions for ψ (each of them are of finite length) by "+"-operators. Doing so captures the semantics of the `sum` statement. This produces a finite set of arithmetic expressions of finite lengths, which proves the theorem. □

The proof of Theorem 2 is constructive and can be turned into a general algorithm for grounding the RLP. We turn all par-(in)equalities and the objective into prenex normal form. Viewing them essentially as logical formulas, we can now ground them inside-out with respect to the `forall` and `sum` statements. This can be done using any Prolog engine implementing a meta-interpreter calling e.g. `findall` and `setof` in the internal loop. A simple version of this is summarized in Algorithm 1.

In some cases, as also in some of our experiments, it can be more efficient to use a relational database management system (RDBMS) for grounding a RLP in a forward-chaining way, treating the queries in par-expressions as SQL queries. Niu, Ré, Doan and Shavlik [33] have demonstrated this to be beneficial for inference in Markov logic networks (MLNs) [12]. We essentially follow the same strategy for grounding RLPs. PostgreSQL was used in our experiments, which allows to perform arithmetic computations and string concatenations inside an SQL query, so we are able to get sums of LP variables with corresponding coefficients directly from a query. As a result, the only post processing needed is the concatenation of these strings. Our grounding implementation takes comparable time to Niu et al.'s TUFFY system for MLNs with comparable number of ground predicates to be generated, which is a state-of-the-art performance on this task at the moment.

3.2.4. *Relational linear programming and simplifications*

To summarize what we have so far, **relational linear programming** works as summarized in Algorithm 2. We encode the general problem structure of the optimization problem at hand as a relational linear program (line 1). Then we specify a problem instance using an external logical knowledge base LogKB (line 2). Finally, we ground the relational linear program for the problem instance using Algorithm 1 (line 3).

For illustration, reconsider the flow instance as depicted in Fig. 5. Together with the relational flow LP in Fig. 4, it induces the following ground linear program in AMPL notation (for the sake of readability, only some of the groundings are shown):

```
var flow(s,a), flow(s,b), ..., inflow(a), ..., outflow(a), ...
maximize: flow(s,a) + flow(s,b);
subject to outflow(a) = flow(a,c);
subject to outflow(b) = flow(b,c) + flow(b,d);
...
subject to inflow(a) = flow(s,a);
subject to inflow(b) = flow(s,b) + flow(c,b);
...
subject to outflow(a) - inflow(a) = 0;
...
subject to 4 - flow(s, a) >= 0;
subject to 2 - flow(s, c) >= 0;
...
```

```
 1 var flow/2;              #the flow along edges is determined by the solver
 2
 3 outflow(X) = sum {edge(X, Y)} flow(X, Y);                #outflow of a node
 4 inflow(Y)  = sum {edge(X, Y)} flow(X, Y);                 #inflow of a node
 5
 6 maximize: sum {source(X)} outflow(X);                            #objective
 7
 8 subject to {vertex(X), not source(X), not target(X)}:#conservation of flow
 9   outflow(X) - inflow(X) = 0;
10 subject to {edge(X, Y)}: cap(X, Y) - flow(X, Y) >= 0;      #capacity bound
11 subject to {edge(X, Y)}: flow(X, Y) >= 0;               #no negative flows
```

Fig. 6. Simplified relational encoding of the relational flow LP. Compared to the basic variant in Fig. 4, the forall statements are implicit and there are inline definitions in lines 3 and 4.

```
subject to flow(s, a) >= 0;
...
```

This illustrates another benefit of relational linear programming. While formulating LPs in canonical form as input to LP solvers requires explicit value enumeration for the $\boldsymbol{A}$ matrix and the $\mathbf{b}$ and $\mathbf{c}$ vectors, relational linear programming avoids this explicit value enumeration, exploiting the existence of domain objects, relations over these objects, and the ability to express objectives and constraints using quantification. We can just change the LogKB to induce a different flow LP; we do not have to touch the "LP logic" anymore. However, we can make relational linear programming even more concise. Although auxiliary LP predicates such as `inflow/1` and `inflow/1` indeed increase the readability of relational LPs, they may also unnecessarily blow up the induced LP. Both definitions could have been directly "substituted" in the conservation of flow constraint:

```
#conservation of flow
subject to forall {X in vertex(X), not source(X), not target(X)} :
  sum {edge(X, Y)} {flow(X, Y)} - sum {edge(X, Y)} {flow(X, Y)} = 0;
```

This avoids creating the $2n$ auxiliary LP variables and constraints, where n is the number of nodes in the graph, but also sacrifices the readability of the RLP. To keep the balance between readability and size complexity of the induced LP, we will therefore sometimes make use of what we call "**inline definitions**". Inline definitions appear before the objective and do not involve `forall` statements. There is just a single atom on the left hand side of an equality followed by a par-expression. The idea is that they tell the "compiler" to substitute the right hand side of the definition inline in the objective and constraints by performing inline expansion, i.e., by inserting the right hand side code there by pattern matching thereby saving the overhead of using auxiliary LP variables. Furthermore, we will often drop the `X in` parts if `X` consists of all variables of the query anyhow. Finally, we will often drop the brackets as long as the scope of logical variables is clear and make the all-quantification using forall statements only implicitly. That is, we drop the forall statements; a $\{\psi\}$ statement corresponds to a forall statement with all variables appearing the query ψ used for indexing. All simplifications are illustrated in the flow RLP shown in Fig. 6.

However, relational linear programming is not just about concise modeling. One can also efficiently detect and exploit symmetries within the induced LPs and, in turn, speed up solving them. How to do this will be shown next.

4. Exploiting symmetries for dimension reduction of LPs

As we have already mentioned in the introduction, one of the features of many relational models is that they can produce model instances with a lot of symmetries. These symmetries in turn can be exploited to perform inference at a "lifted" level, i.e., at the level of groups of variables. For probabilistic relational models, this lifted inference can yield dramatic speed-ups, since one reasons about the groups of indistinguishable variables as a whole, instead of treating them individually. Triggered by this success, we will now show that linear programming is liftable, too. To this end, we start off with a high-level discussion of our approach, connecting it to lifted probabilistic inference, followed by the technical results necessary to support our approach.

4.1. Symmetry detection and compression

In developing our compression method, we are motivated by a conceptual paradigm that has been successful for other inference algorithms such as lifted belief propagation (BP) [34–36], a message-passing algorithm for approximate inference in Markov random fields (MRFs). From a high-level perspective, one can summarize this paradigm as follows (the more precise definitions will be given later in this section). We start off with a standard inference algorithm and look for conditions (in terms of the specific model being solved) that would make a set of variables behave identically. That is, we look for features of the model that apriori guarantee that a set of variables would maintain equal values throughout the solution process, or at least in the final solution. This identical behavior is what we refer to as symmetry, while the variables subjected to it are referred to as indistinguishable. To take advantage of the discovered symmetry, we finally replace the

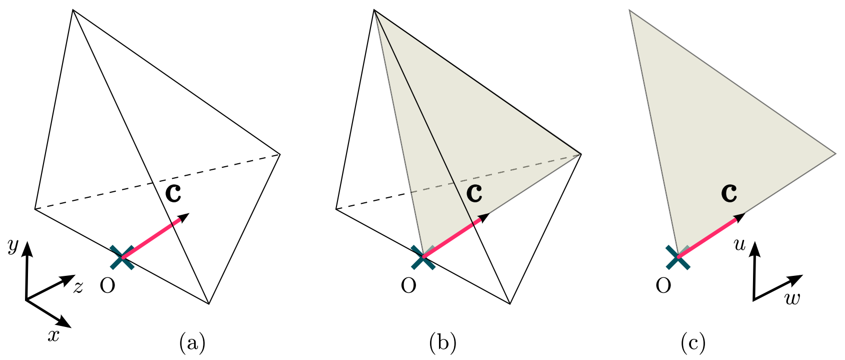


Fig. 7. Using symmetry to speed up linear programming: (a) the feasible region of an LP and the objective vector (in pink); (b) the span of the fractional automorphism of the LP (grey); (c) the lifted (compressed) LP is obtained by projecting the feasible region onto the span of the fractional automorphism. (Best viewed in color.)

sets of indistinguishable variables by a representative (also called lifted, or super-) variable. We then reformulate the model carefully (in general, it may also be necessary to modify the inference algorithm as well) so that a solution of the original model can be read off from the reformulated one. If the reformulated model is of smaller size, there is a good chance to solve the problem at hand faster.

Intuitively, to increase the chance of lower end-to-end solution times, the method for detecting symmetries should be considerably faster than the actual solver; otherwise we may end up spending more time detecting symmetries than we gain from compressing the model. In the following we will present such an efficient approach, more specifically, a quasi-linear time approach for lifting linear programs. For the technical discussion we first introduce our notion of symmetry: **equitable partitions** of (the variables and constraints of) a linear program and their algebraic representations, **fractional automorphisms**. To justify that equitable partitions are indeed a form of symmetry in the sense above, we prove that LPs admitting nontrivial equitable partitions also admit optimal solutions where all variables belonging to an equivalence class are equal. We then show how to reparametrize an LP given an equitable partition by replacing a set of indistinguishable variables by a single supervariable representing their sum. Unlike lifted BP, where the compressed model is not an MRF anymore, here we actually get a compressed LP, the solutions of which can be transfered back to solutions of the original LP. The compressed LP can be solved by any off-the-shelf LP solver. Finally, we will discuss symmetry detection, i.e., the computation of equitable partitions of LPs.

The steps outlined above (detecting symmetry, reparametrizing the LP, solving the reparametrized LP and recovering the original solution) constitute Lifted Linear Programming. While we will make use of mostly combinatorial and algebraic concepts along the way, these operations have a geometrical interpretation as well, as illustrated in Fig. 7. Essentially, we identify a lower-dimensional subspace that intersects the feasible region in such a way that an optimal solution is contained within that intersection. To arrive at a smaller LP, we project onto that subspace, and transfer back by computing the inverse image under this projection.

We now dive into the technical discussion of Lifted Linear Programming.

4.2. Equitable partitions and fractional automorphisms

Let $L = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ be an LP with $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, that is, with m constraints and n variables. In the following, we aim to partition the variables and constraints into mutually-exclusive classes, which are indistinguishable. Thus, we define a partition of the LP to be the set $\mathcal{P} = \{P_1, \ldots, P_p; Q_1, \ldots, Q_q\}$, where the sets $\cup_{i=1}^{p} P_i = \{1, \ldots, n\}$, $P_i \cap P_j = \emptyset$, partition the variables, and the sets $\cup_{i=1}^{q} Q_i = \{1, \ldots, m\}$, $Q_j \cap Q_j = \emptyset$, partition the constraints of the LP into (equivalence) classes. Hence, we also require that $P_i \cap Q_j = \emptyset$ for all appropriate i, j.

We say that a partition $\mathcal{P} = \{P_1, \ldots, P_p; Q_1, \ldots, Q_q\}$ of $L = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ is **equitable** if the following conditions hold.

i. For any two variables $i,\ j$ in the same class P, $\boldsymbol{c}_i = \boldsymbol{c}_j$. For any two constraints $i,\ j$ in the same class Q, $\boldsymbol{b}_i = \boldsymbol{b}_j$;
ii. For any two variables $i,\ j$ in the same class P, and for any constraint class Q and real number c:

$$|\{k \in Q \mid \boldsymbol{A}_{ki} = c\}| = |\{l \in Q \mid \boldsymbol{A}_{lj} = c\}|\ .$$

Analogously, for any two constraints $i,\ j$ in the same class Q, and for any constraint class P and real number c:

$$|\{k \in P \mid \boldsymbol{A}_{ik} = c\}| = |\{l \in P \mid \boldsymbol{A}_{jl} = c\}|\ .$$

In other words, if we fix any class of constraints Q, then the number of constraints in Q where a variable $i \in P$ participates with a coefficient of c should be equal for all other $j \in P$. The same should hold for any two equivalent constraints and any class of variables.

It is clear that every LP admits at least one equitable partition, namely the discrete (or trivial) one, where every constraint and variable is in its own singleton class. Whether the LP admits coarser ones depends on its structure. In general, we would like to partition LPs as coarsely as possible in order to gain the highest potential speed ups. How to do that efficiently is discussed later on in Section 4.4. Moreover, equitable partitions of LPs generalize equitable partitions for graphs [37]. To make the connection explicit, we consider an equitable partition of a matrix $\boldsymbol{M}$ to be an equitable partition of the LP $(\boldsymbol{M}, \mathbf{0}, \mathbf{0})$, whereas an equitable partition of a bipartite graph G is the equitable partition of its bipartite adjacency matrix.

Let us illustrate equitable partitions using the following didactic LP

```
var p/1;

maximize: sum{gadget(X)} p(X);

subject to sum{widget(X)} p(X) + sum{gadget(X)} p(X) <= 1;
subject to {widget(X)}: widget(X) <= 0;
subject to sum{widget(X)} p(X) - sum{gadget(X)} p(X) <= -1;
```

together with the LogKB (recall that logical atoms are assumed to evaluate to 0 and 1 within an RLP):

```
widget(x).
widget(y).
gadget(z).
```

If we ground this RLP and convert it to dual form (as in (1)), we obtain the following linear program $L^0 = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$:

$$\begin{array}{ll} \text{minimize}_{[x,y,z]^T \in \mathbb{R}^3} & 0x + 0y + 1z \\ \text{subject to} & \begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \le \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}, \end{array}$$

where for brevity we have substituted $\mathtt{p}(x)$, $\mathtt{p}(y)$, $\mathtt{p}(z)$ by x, y, z respectively. We claim now that the partition $\mathcal{P}^0 = \{\{1,2\},\{3\};\{1\}\{2,3\}\{4\}\}$—meaning that x is equivalent to y but not to z and the second and third constraint are equivalent, but not the first and the fourth—is an equitable partition of L^0. Let us verify that. First, we have that $\boldsymbol{c}_1 = \boldsymbol{c}_2 \neq \boldsymbol{c}_3$ and $\boldsymbol{b}_2 = \boldsymbol{b}_3 \neq \boldsymbol{b}_1 \neq \boldsymbol{b}_4$ (condition (i)). We now need to verify condition (ii). For the sake of illustration we will do this only for the class $P = \{1,2\}$. Consider $Q = \{1\}$. Since Q is a singleton, condition (ii) reduces to whether $\boldsymbol{A}_{11} = \boldsymbol{A}_{12} = c$, which happens to be the case in our example, as we have only $c = 1$. The same holds for $Q = \{4\}$. For $Q = \{2,3\}$, we need to check the condition for $c = 0$ and $c = -1$. We have

$$|\{i \in Q \mid \boldsymbol{A}_{i1} = 0\}| = |\{3\}| = |\{2\}| = |\{j \in Q \mid \boldsymbol{A}_{j2} = 0\}$$

as well as

$$|\{i \in Q \mid \boldsymbol{A}_{i1} = -1\}| = |\{2\}| = |\{3\}| = |\{j \in Q \mid \boldsymbol{A}_{j2} = -1\}\,.$$

One can easily complete the counting argument for all other pairs of classes.

So far, we have introduced equitable partitions of linear programs. In order to show that LP variables grouped together by an equitable partition assume equal values in some optimal solution of the LP, we have to introduce one more bit of mathematical machinery, which makes the interplay between equitable partitions apparent: the notion of **fractional automorphism** due to Ramana, Scheinerman and Ullmann [38].[8]

Let $\mathbf{M}$ be an $m \times n$ real matrix. A **fractional automorphism** of $\mathbf{M}$ is a pair of doubly stochastic (meaning that the entries are non-negative, and every row and column sum to one) matrices $\boldsymbol{X}_P, \boldsymbol{X}_Q$ such that

$$\boldsymbol{M}\boldsymbol{X}_P = \boldsymbol{X}_Q \boldsymbol{M}.$$

The following theorem establishes the correspondence between equitable partitions and fractional automorphisms.

Theorem 3. *(See [38,39].) Let $\boldsymbol{M}$ be a rectangular real matrix. Then:*

i) if $\mathcal{P} = \{P_1, \ldots, P_p; Q_1, \ldots, Q_q\}$ is an equitable partition of $\boldsymbol{M}$ (i.e., of the LP $(\boldsymbol{M}, \mathbf{0}, \mathbf{0})$), then the matrices $\boldsymbol{X}_P, \boldsymbol{X}_Q$, having entries

[8] Note that our definition is slightly modified from the original one in [38], since we are interested in rectangular matrices (equivalent to weighted bipartite graphs in [38]).

$$(\boldsymbol{X}_P)_{ij} = \begin{cases} 1/|P| & \text{if both vertices } i,\ j \text{ are in the same } P \in \mathcal{U}, \\ 0 & \text{otherwise,} \end{cases}$$
$$(\boldsymbol{X}_Q)_{ij} = \begin{cases} 1/|Q| & \text{if both vertices } i,\ j \text{ are in the same } Q \in \mathcal{U}, \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

is a fractional automorphism of $\boldsymbol{M}$.

ii) *conversely, let* $\boldsymbol{X}_P, \boldsymbol{X}_Q$ *be a fractional automorphism of the matrix* $\boldsymbol{M}$. *Then the partition* $\mathcal{P}$, *where columns* i, j *belong to the same* $P \in \mathcal{P}$ *if and only if at least one of* $(\boldsymbol{X}_P)_{ij}$ *and* $(\boldsymbol{X}_P)_{ji}$ *is greater than* 0, *respectively rows* i, j *belong to a class* Q *if* $(\boldsymbol{X}_Q)_{ij}$ *or* $(\boldsymbol{X}_Q)_{ji}$ *is greater than* 0, *is an equitable partition of* $\boldsymbol{M}$.

In particular, we require part i) of Theorem 3 since encoding equitable partitions of LPs as fractional automorphisms will provide us with insights into the geometrical aspects of lifting and will be an essential tool for proving its soundness. Before we continue with our review of the mathematical background for lifted linear programming, we make one additional observation.

Proposition 4. *Let* $\mathcal{P}$ *be an equitable partition of* $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ *and* $\boldsymbol{X}_P, \boldsymbol{X}_Q$ *be the matrices according to* (3). *Then,* $\boldsymbol{c}^T\boldsymbol{X}_P = \boldsymbol{c}^T$ *and* $\boldsymbol{X}_Q\boldsymbol{b} = \boldsymbol{b}$.

Proof. The proof follows directly from the fact that we group together elements of an LP only if their corresponding $\boldsymbol{c}$ or $\boldsymbol{b}$ entries are equal. □

Finally, we note that any matrix/graph partition (equitable or not) can be represented by a doubly stochastic matrix using (3). However, keep in mind that the resulting matrix, which is called partition matrix, will not be a fractional automorphism unless the partition is equitable. In any case, partition matrices have a useful property that will later on allow us to reduce the number of constraints and variables of a linear program. More precisely:

Proposition 5. *(See [40].) Let* $\boldsymbol{X}$ *be a doubly stochastic matrix produced by some partition* $\mathcal{P}$ *according to* (3). *Then* $\boldsymbol{X} = \widetilde{\boldsymbol{B}}\widetilde{\boldsymbol{B}}^T$ *with*

$$\widetilde{\mathbf{B}}_{iP} = \begin{cases} \frac{1}{\sqrt{|P|}} & \text{if element } i \text{ belongs to part } P, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Before starting to develop lifted linear programming, let us illustrate fractional automorphisms and partition matrices. Reconsider the equitable partition $\mathcal{P}^0 = \{\{1,2\},\{3\};\{1\}\{2,3\}\{4\}\}$ of the LP L^0 in our running example. The fractional automorphism of L^0 induced by (3) is

$$\boldsymbol{X}_P = \begin{bmatrix} .5 & .5 & 0 \\ .5 & .5 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \widetilde{\mathbf{B}}\widetilde{\mathbf{B}}^T$$

and

$$\boldsymbol{X}_Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & .5 & .5 & 0 \\ 0 & .5 & .5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \widetilde{\mathbf{C}}\widetilde{\mathbf{C}}^T .$$

Moreover, Proposition 4 holds since

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} .5 & .5 & 0 \\ .5 & .5 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{X}_P} = \begin{bmatrix} 1 & 1 & 1 \\ -.5 & -.5 & 0 \\ -.5 & -.5 & 0 \\ 1 & 1 & -1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & .5 & .5 & 0 \\ 0 & .5 & .5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{X}_Q} \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{bmatrix}}_{\boldsymbol{A}} .$$

It is easily verified that $\boldsymbol{c}^T\boldsymbol{X}_P = \boldsymbol{c}^T$ and $\boldsymbol{X}_Q\boldsymbol{b} = \boldsymbol{b}$ hold as well.

4.3. Lifted linear programming

We are now ready to establish lifted linear programs. We split the argument in two parts: first, we will show that if an LP L admits an optimal solution $\boldsymbol{x}^*$, then it also admits a solution $\boldsymbol{X}_P\boldsymbol{x}^*$. Note that if $\boldsymbol{X}_P$ is a doubly stochastic matrix, every entry in $\boldsymbol{X}_P\boldsymbol{x}^*$ is the average value of all the entries in its equivalence class. Hence, as claimed, variables that are

Algorithm 3: Lifted linear programming.

```
Input: An inequality-constrained LP, $L = (\mathbf{A}, \mathbf{b}, \mathbf{c})$
Output: $\mathbf{x}^* = \operatorname{argmin}_{\{\mathbf{x}|\mathbf{A}\mathbf{x}\le\mathbf{b}\}} \mathbf{c}^T\mathbf{x}$
1 Compute an equitable partition of $L$ (see Section 4.4).;
2 Read off the characteristic matrix $\widetilde{\mathbf{B}}_P$;
3 Compress $L$ to $(\mathbf{A}\widetilde{\mathbf{B}}_P, \mathbf{b}, \widetilde{\mathbf{B}}_P^T\mathbf{c})$ and remove redundant constraints;
4 Obtain compressed solution $\boldsymbol{y}^*$ using any standard LP solver;
5 return $\mathbf{x}^* = \widetilde{\mathbf{B}}_P\mathbf{y}^*$;
```

indistinguishable according to the equitable partition behave identically. Now, if we add the constraint $\exists \boldsymbol{y} \in \mathbb{R}^n : \boldsymbol{x} = \boldsymbol{X}_P\boldsymbol{y}$, in other words $\boldsymbol{x} \in \operatorname{span}(\boldsymbol{X}_P)$, to the linear program, we will not cut away the optimum. The second claim is that instead of adding $\exists \boldsymbol{y} \in \mathbb{R}^n : \boldsymbol{x} = \boldsymbol{X}_P\boldsymbol{y}$, we can achieve this restriction by projecting the entire LP into the span of $\boldsymbol{X}_P$. Unless $\mathcal{P}$ is the discrete partition with all singleton classes, $\boldsymbol{X}_P$ will not be full rank, thus the resulting LP will have fewer variables. One can verify that the rank of $\boldsymbol{X}_P$ is the number of P-classes of $\mathcal{P}$. So, we project to a low-dimensional space, solve the LP there, and then recover the high-dimensional solution via simple matrix multiplication. This idea is exactly what was illustrated in Fig. 7.

Let us now state the main result on lifted linear programming:

Theorem 6. *Let $L = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ be a linear program and $(\boldsymbol{X}_P, \boldsymbol{X}_Q)$ be a fractional automorphism of L. Then, it holds that if $\boldsymbol{x}$ is a feasible in L, then $\boldsymbol{X}_P\boldsymbol{x}$ is feasible as well and both have the same objective value. As a consequence, if $\boldsymbol{x}^*$ is an optimal solution, $\boldsymbol{X}_P\boldsymbol{x}^*$ is optimal as well.*

Proof. Let $\boldsymbol{x}$ be feasible in $L = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$, i.e. $\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}$. Observe that left multiplication of the system by a doubly stochastic matrix preserves the direction of inequalities. More precisely

$$\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b} \Rightarrow \boldsymbol{S}\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{S}\boldsymbol{b}\,,$$

for any doubly stochastic[9] $\boldsymbol{S}$. Now, we left-multiply the system by $\boldsymbol{X}_Q$

$$\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b} \Rightarrow \boldsymbol{X}_Q\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{X}_Q\boldsymbol{b} = \boldsymbol{A}\boldsymbol{X}_P\boldsymbol{x} \le \boldsymbol{b},$$

since $\boldsymbol{X}_Q\boldsymbol{A} = \boldsymbol{A}\boldsymbol{X}_P$ and $\boldsymbol{X}_Q\boldsymbol{b} = \boldsymbol{b}$. This proves the first part of the Theorem. Finally, observe that $\boldsymbol{c}^T(\boldsymbol{X}_P\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x}$ as $\boldsymbol{c}^T\boldsymbol{X}_P = \boldsymbol{c}^T$. This proves the second part of the theorem. □

We have thus shown that if we add the constraint $\boldsymbol{x} \in \operatorname{span}(\boldsymbol{X}_P)$ to L, we can still find a solution of the same quality as in the original program. How does this help reducing the dimensionality of the LP?

To answer this, we observe that the constraint $\boldsymbol{x} \in \operatorname{span}(\boldsymbol{X}_P)$ can be implemented implicitly, through reparametrization. That is, instead of adding it to $L = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ explicitly, we take the LP $L' = (\boldsymbol{A}\boldsymbol{X}_P, \boldsymbol{b}, \boldsymbol{X}_P^T\boldsymbol{c})$. Now, recall that $\boldsymbol{X}_P$ was generated by an equitable partition, and it can be factorized as $X_P = \widetilde{\boldsymbol{B}}_P\widetilde{\boldsymbol{B}}_P^T$ where $\widetilde{\boldsymbol{B}}_P$ is the normalized incidence matrix of $\{P_1, \ldots, P_p\} \subset \mathcal{U}$ as in (4). Note that the span of $\boldsymbol{X}_P = \widetilde{\mathbf{B}}_P\widetilde{\mathbf{B}}_P^T$ is equivalent (in the vector space isomorphism sense) to the column space of $\widetilde{\mathbf{B}}_P$. That is, every $\boldsymbol{x} \in \mathbb{R}^n$ with $\boldsymbol{x} \in \operatorname{span}(\boldsymbol{X}_P)$ can be expressed as $\boldsymbol{x} = \widetilde{\mathbf{B}}_P\boldsymbol{y}$ for some $y \in \mathbb{R}^p$ and conversely $\widetilde{\mathbf{B}}_P\boldsymbol{y} \in \operatorname{span}(\boldsymbol{X}_P)$ for all $\boldsymbol{y} \in \mathbb{R}^p$. Hence, we can replace $L' = (\boldsymbol{A}\boldsymbol{X}_P, \boldsymbol{b}, \boldsymbol{X}_P^T\boldsymbol{c})$ with the equivalent $L'' = (\boldsymbol{A}\widetilde{\mathbf{B}}_P, \boldsymbol{b}, \widetilde{\mathbf{B}}_P^T\boldsymbol{c})$. Since this is now a problem in $p \le n$ variables, i.e., of (potentially) reduced dimension, a speed-up of solving the original LP is possible. Finally, by the above, if $\boldsymbol{y}^*$ is an optimal solution of L'', $\widetilde{\mathbf{B}}_P\boldsymbol{y}$ is an optimum solution of L.

Moreover, please observe that after compressing all equivalent variables, the equivalent constraints (i.e., the Q-classes of the partition) become redundant. That is, the rows of $\boldsymbol{A}\widetilde{\mathbf{B}}_P$ whose indices were grouped together by the equitable partition of the original LP become identical vectors. Thus, one can achieve additional compression by retaining only one constraint per constraint equivalence class.

Overall, this proves that the **lifted linear programming** approach as summarized in Algorithm 3 is sound. Given an LP, we find an equitable partition (line 1). Since the number of classes directly determines the size of the lifted LP, we seek to compute the coarsest such partition; how to do this is the topic of the next subsection. Then, we read off the characteristic matrix as per (4) (line 2). Finally, we solve the lifted LP (line 3) and "unlift" the lifted solution to a solution of the original LP (line 4). Applying this lifted linear programming to LPs induced by RLPs, we can revise **relational linear programming** as summarized in Algorithm 4.

Before showing how to compute the coarsest equitable partition, we illustrate how the lifted LP looks like for our running example L^0. First, we compute $\boldsymbol{A}\widetilde{\boldsymbol{B}}_P$ as

[9] To see why this is the case, consider the real numbers $a_1 \le b_1$ and $a_2 \le b_2$. For any positive s_1, s_2, we have $s_1a_1 + s_2a_2 \le s_1b_1 + s_2b_2$. We generalize this by induction to any finite number of variables and apply to $\boldsymbol{S}(\boldsymbol{A}\boldsymbol{x})_i = \sum_j \boldsymbol{S}_{ij}(\boldsymbol{A}\boldsymbol{x})_j \le \sum_j \boldsymbol{S}_{ij}\boldsymbol{b}_j = (\boldsymbol{S}\boldsymbol{b})_i$, as $\boldsymbol{S}$ is positive and $(\boldsymbol{A}\boldsymbol{x})_j \le \boldsymbol{b}_j$ by assumption.

```
Algorithm 4: Relational linear programming revised using lifted linear programming.
1 Specify an RLP $R$;
2 Given a LogKB, ground $R$ into an LP $L$, e.g. using Algorithm 1;
3 Solve $L$ using lifted linear programming as described in Algorithm 3;
```

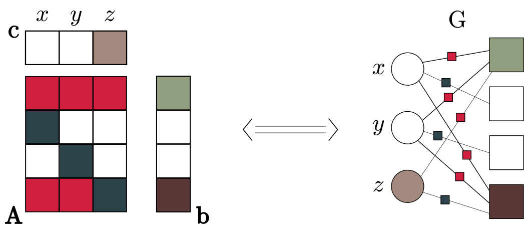


Fig. 8. Construction of the coefficient graph G_L of our running example L^0. On the left-hand side, the coloring of the LP is shown. This turns into the colored coefficient graph shown on the right-hand side. (Best viewed in color.)

$$\boldsymbol{A}\widetilde{\boldsymbol{B}}_P = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{\sqrt{2}} & 1 \\ -\frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & 0 \\ \frac{2}{\sqrt{2}} & -1 \end{bmatrix}.$$

As noted above, the equivalent constraints 2 and 3 have become redundant, so we drop them. If we recompute the $\boldsymbol{c}$ and $\boldsymbol{b}$ vector accordingly, our lifted LP becomes

$$\begin{aligned} &\text{minimize}_{[\mathfrak{x},z]^T\in\mathbb{R}^2} && 0\mathfrak{x} + 1z \\ &\text{subject to} && \begin{bmatrix} 1 & 1 \\ -\frac{1}{\sqrt{2}} & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \mathfrak{x} \\ z \end{bmatrix} \le \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \end{aligned}$$

having replaced $x + y$ by the supervariable $\mathfrak{x}$.

4.4. Computing fractional automorphisms

So far, we have developed a method for speeding up solving LPs given an equitable partition of their variables and constraints. However, we left open the question of how to find such partitions. Now, we will make up for this. That is, we will now deal with the computational aspect of lifting.

In graph theory, equitable partitions are well-studied generalizations of orbit partitions [37]—partitions induced by the action of a (sub)group of automorphisms of a graph. That is, an orbit partition is an equitable partition, but not vice versa. While computing the orbit partition of a graph is a hard (GI-complete, to be precise) problem, computing the **coarsest equitable partition (CEP)** of a graph can be done using an algorithm called **color refinement** (also known as **naive vertex classification**). It is a very simple, yet extremely useful algorithmic routine for graph isomorphism testing. It classifies the vertices by iteratively refining a coloring—the colors encode the (equivalence) classes—of the vertices as follows. Initially, all vertices have the same color. Then in each step of the iteration, two vertices that currently have the same color get different colors if for some color c they have a different number of neighbors of color c. The process stops if no further refinement is achieved, resulting in a stable coloring of the graph. If one carefully chooses the order in which the resulting classes are refined, Berkholz, Bonsma and Grohe [41] have shown that color refinement can be realized in time $\mathcal{O}(|E| + |V|)\log(|V|)$, i.e. quasilinear in the size of the graph with nodes V and edges E. Automorphism group computation tools such as Nauty and Saucy make use of this fact and employ the coarsest equitable partition as a heuristic in orbit computation. As a result, there are highly efficient (both theoretically and empirically) implementations of color refinement.

To compute the coarsest equitable partition of an LP, we make use of these tools by converting the LP CEP problem to a colored graph CEP problem. Hence, we need a graphical representation of $L = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$, which we call the **coefficient graph** of L (see Fig. 8), G_L. To construct G_L, we add a vertex to G_L for every of the m constraints and n variables of L. Then, we connect a constraint vertex i and variable vertex j *if and only if* $\mathbf{A}_{ij} \neq 0$. Furthermore, we assign colors to the edges $\{i, j\}$ in such a way that $color(\{i, j\}) = color(\{u, v\}) \iff \mathbf{A}_{ij} = \mathbf{A}_{uv}$. Finally, to ensure that $\mathbf{c}$ and $\mathbf{b}$ are preserved by any automorphism we find, we color the vertices in a similar manner, i.e., for row vertices i, j $color(i) = color(j) \iff \boldsymbol{b}_i = \boldsymbol{b}_j$ and $color(u) = color(v) \iff \mathbf{c}_u = \mathbf{c}_v$ for column vertices. We must also choose the colors in a way that no pair of row and column vertices share the same color; this is always possible.

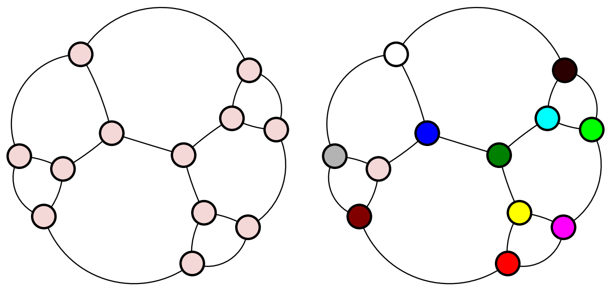


Fig. 9. The Frucht graph with 12 nodes. The colors indicate the resulting node partitions using color refinement (the coarsest equitable partition, left) and using automorphisms (the orbit partition, right). (Best viewed in color.)

One can verify that an equitable partition of G_L yields an equitable partition of L. From the complexity of color refinement and the fact that our graphical construction grows only linearly with the size of the LP, the following theorem holds:

Theorem 7. *The coarsest (i.e., the one yielding the most compression) equitable partition of an LP is computable in quasilinear time in terms of the number of variables and the size of the constraints, i.e., of the non-zero entries of* $\boldsymbol{A}$.

Before illustrating relational linear programming empirically, we would like to provide some additional remarks on lifted linear programming.

4.5. Discussion of lifted linear programming

First, indeed lifted linear programming can be applied to any LP, not just those generated by RLP. At first sight, one may consider them to be orthogonal. However, this is not the case. One should rather think about lifted linear programming as the "assembly language" of (relational) linear programming, since RLPs can be grounded and solved as ordinary LPs. Going beyond that, the relational specification of an LP can be used to compute equitable partitions faster, ideally even without grounding. Such an approach was demonstrated to be beneficial by Apsel, Kersting and Mladenov [42] for relational MAP inference. Under certain restrictions on the language, this could be generalized to RLPs. Another example for language-induced symmetries are the renaming groups due to Bui, Huynh and Riedel [43]. While these approaches make heavy use of relational representations, they are still based on the basic fact that the underlying ground problem is liftable. This is the perspective we choose to adapt for RLPs in the present work. The interaction between languages and equitable partitions is a very promising avenue for future research.

Second, recall that our compression method works with any equitable partition, not only the coarsest one (which happens to be efficiently computable). As mentioned, an equitable partition of a graph—the orbit partition—can be constructed out of its automorphism group, i.e., the set of all possible ways in which we can rename the vertices and get the same graph back. The orbit partition groups two vertices whenever there exists an automorphism that maps one to the other. Applying this partitioning method to coefficient graphs of linear programs and using the corresponding fractional automorphism is equivalent to previous theoretical well-known results in solving linear programs under symmetry (see e.g. [44] and references therein). Note that there are major benefits for using the color refinement partition instead of the orbit partition for linear programs:

- The color refinement partition is at least as coarse as the orbit partition as shown in [37]. To illustrate this, consider the so-called Frucht graph as shown in Fig. 9. Suppose we turn this graph into a linear program by introducing constraint nodes along the edges and coloring everything with the same color. The Frucht graph has two extreme properties with respect to equitable partitions: 1) it is asymmetric, meaning that the orbit partition is trivial having one vertex per equivalence class; 2) it is regular (every vertex has degree 3); as one can easily verify, in this case the coarsest equitable partition consists of a single class! Due to these two properties, the orbit partition yields no compression on the Frucht graph, whereas the coarsest equitable resp. color refinement partition produces an LP with a single variable.
- The color refinement partition can be computed in quasilinear time, yet current tools for orbit partition enumeration may have significantly worse running times: although computing orbit-partitions may indeed be practical in a number of cases, computing them is a GI-complete problem. Thus, by using color refinement we achieve strict gains in both compression and efficiency compared to using orbits.

Finally, the color refinement algorithm is essentially the lifting algorithm of lifted BP as formulated by Kersting, Ahmadi and Natarajan [35]. Yet, there are remarkable differences between lifted BP and lifted LPs. For example, after lifting, a lifted MRF is no longer an MRF in the classical sense since one needs a special data structure to keep track of counts of variables and factors. Hence, one also has to introduce a modified message-passing algorithm. In contrast, a lifted LP is a linear program

```
var value/1;              #value function to be determined by the LP solver

maximize: sum{reward(S,_)} value(S);          #best values for all states

#encoding of discounted Bellman optimality as in inequality (5)
subject to forall {S, T in transProb(S,T,_)}: value(S) <= reward(S,A) +
    gamma*sum{transProb(S,T,A)} transProb(S,T,A)*value(T);
```

Fig. 10. An RLP for computing the value function `value/1` of a Markov decision process. There is a finite set of states and actions, and the agent receives a reward `reward(S,A)` for performing and action `A` in state `S`, specified in a LogKB.

and no specialized solver is necessary. Moreover, even though the solution found by lifted BP is identical to the one of BP, it need not be a true solution of the inference task (computing the single-node marginals) since BP approximates the task. Thus, the exactness of lifted BP for inference is limited by that of BP. On the other hand, a lifted LP always recovers an exact optimal solution of the corresponding LP. No approximation is involved.

Let us now turn towards illustrating empirically relational linear programming using several AI tasks.

5. Illustrations of relational linear programming

Our intention here is to investigate empirically the viability of the ideas and concepts of relational linear programming through the following questions:

(Q1) Can important AI tasks be encoded in a concise and readable relational way using RLPs?
(Q2) Are there (R)LPs that can be solved more efficiently using lifting?
(Q3) Does relational linear programming enable a programming approach to AI tasks facilitating the construction of more sophisticated models from simpler ones by adding par-constraints?
(Q4) If lifted linear programming is beneficial, can the benefits be observed for different LP solvers or are they bound to a particular solver?
(Q5) Is the numerical accuracy of the solver unaffected by lifting?

If all question can be answered affirmatively, relational linear programming has the potential to make linear models faster to write and easier to understand, to reduce the development time and cost to encourage experimentation, and in turn to reduce the level of expertise necessary to build AI applications. Consequently, our primary focus is not to achieve the best performance by using advanced models. Instead we will focus on basic models.

We have implemented a prototype system of relational linear programming, and illustrate the relational modeling of several AI tasks: computing the value function of Markov decision processes, performing MAP inference in Markov logic networks via an LP relaxation and performing collective transductive classification using LP support vector machines.

5.1. Lifted linear programming for solving Markov decision processes

Our first application for illustrating relational linear programing is the computation of the value function of Markov Decision Problems (MDPs). The LP formulation of this task is as follows [45]:

$$\begin{aligned} &\text{maximize}_{\mathbf{v}} \quad \mathbf{1}^T\mathbf{v}, \\ &\text{subject to} \quad \boldsymbol{v}_i \le c_i^k + \gamma \sum\nolimits_{j\in\Omega_S} p_{ij}^k \boldsymbol{v}_j \quad \forall i \in \Omega_S,\ \forall k \in K, \end{aligned} \tag{5}$$

where $\boldsymbol{v}_i$ is the value of state i from the set of all states Ω_S, c_i^k is the reward that the agent receives when carrying out action $k \in K$, and p_{ij}^k is the probability of transferring from state i to state j by taking action k. γ is a discounting factor. The corresponding RLP is given in Fig. 10. Since it abstracts away the states and rewards—they are defined in the LogKB—it extracts the essence of computing value functions of MDPs. Given a LogKB, a ground LP is automatically created instead of coding the LP by hand for each problem instance again and again as in vanilla linear programming. This answers question **(Q1)** affirmatively.

The MDP instance that we used is the well-known Gridworld (see e.g. [46]). The gridworld problem consists of an agent navigating within a grid of $n \times n$ states. Every state has an associated reward $R(s)$. Typically there is one or several states with high rewards (considered the goals) whereas the other states have zero or negative associated rewards. We considered an instance of gridworld with a single goal state in the upper-right corner with a reward of 100. The reward of all other states was set to −1. The scheme for the LogKB is as follows:

```
reward(state(n-1,n),right)=100.
reward(state(n,n-1),up)=100.
reward(state(X,Y),_)=-1 :- X>0, X<n-1, Y>0, Y<n-1.
```

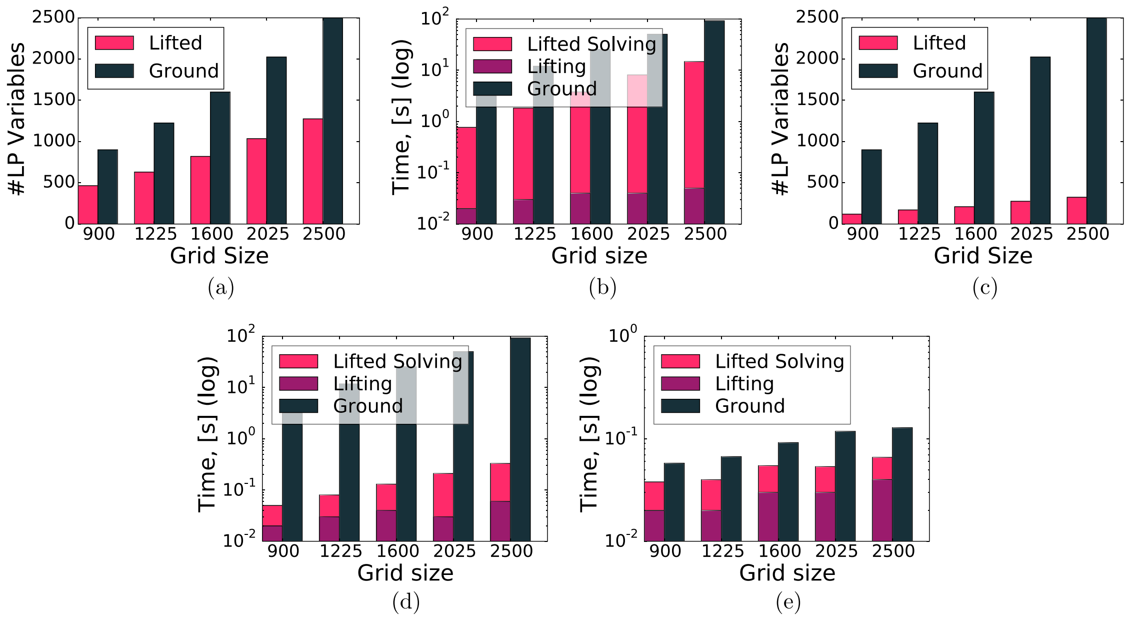


Fig. 11. Experimental results of relational linear programming for solving Markov decision processes (MDP). (a) Number of variables in the ground and lifted LPs in the basic gridworld. (b) Measured times on the basic gridworld MDP in log scale. (c) Number of variables in the gridworld with additional symmetry. (d) Measured times on the gridworld with additional symmetry in log scale. (e) Measured times on the gridworld with additional symmetry in sparse form, again in log scale. *Lifted solving* denotes end-to-end time, while *lifting* indicates the portion of that time spent on lifting. (Best viewed in color.)

We solved the resulting set of LPs using CVXOPT on a 3.2 GHz Core i7, 32 GB RAM workstation running Ubuntu 12. As summarized in Fig. 11(a), the MDPS-LPs can be compiled to about half of the original size. Furthermore, Fig. 11(b) shows that already this compression leads to improved running time. In both figures, we broke down the measured total time for solving the LP into the time spent on lifting and solving respectively.

We then introduced additional symmetries by putting a goal in every corner of the grid. As expected, this gave more room for compression, which further improved efficiency as reflected in Figs. 11(c) and 11(d).

These results affirmatively answer question **(Q2)** and strongly support that **(Q1)** can be answered affirmatively as well. However, the examples that we have considered so far are quite sparse in their structure. Thus, one might wonder whether the demonstrated benefit is achieved only because we are solving a sparse problem in dense form. To address this we converted the MDP problem to a sparse representation for our further experiments. As one can see in Fig. 11(e) lifting still resulted in an improvement of size as well as running time. Therefore, we can conclude that lifting an LP is beneficial regardless of whether the problem is sparse or dense, thus one might view symmetry as a dimension orthogonal to sparsity. Remarkably, the results follow closely what has been achieved with MDP-specific symmetry finding and model minimization approaches [47–49].

5.2. *Programming MAP-LP inference in Markov logic networks (MLNs)*

MLNs [12] are a prominent probabilistic relational model using weighted logical rules such as 0.75 $\texttt{smokes(X)} \Rightarrow \texttt{cancer(X)}$. Given a set of constants, an MLN induces a Markov random field (MRF) with a node for each ground atom and a clique for every ground formula. We here focus on MAP (maximum a posteriori) inference where we want to find a most likely joint assignment to all the random variables. A common approach to approximate MAP inference in MRFs is based on LPs [22]. Let us now briefly review this approach.

Suppose we are presented with a propositional MRF with binary random variables $X = \{x_1, \ldots, x_n\}$ and factors $F = \{(\theta_f, x_f)\}_f$, where each θ_f is a function (having no 0-values) over a subset of random variables $x_f \subseteq X$. As LP variables, we introduce variable beliefs μ_i over the states of each random variable x_i (e.g. $\mu_i(x_i = 0)$, $\mu_i(x_i = 1)$ are LP variables) and joint beliefs μ_f over all joint configuration of each subset x_f (e.g. $\mu_f(x_i = 0, x_j = 1, x_k = 0)$, etc.). The essence of the MAP-LP approach is to constrain the joint beliefs to be consistent with the variable beliefs under marginalization, e.g., $\mu_f(x_i = 0, x_j = 0) + \mu_f(x_i = 0, x_j = 1) = \mu_i(x_i = 0)$. More precisely, the MAP-LP is defined as follows

$$\begin{aligned} \text{maximize}_{\boldsymbol{\mu} \geq 0} \quad & \textstyle\sum_f \theta_f(x_f)\mu_f(x_f) \\ \text{subject to} \quad & \textstyle\sum_{x_i} \mu_i(x_i) = 1 \qquad \forall x_i \in X, \\ & \textstyle\sum_{x_f \setminus x_i} \mu_f(x_f) = \mu_i(x_i) \quad \forall x_f, \forall x_i \in x_f\ , \end{aligned} \tag{6}$$

and if we were to solve this LP as an integer LP, the variable beliefs would give us the exact MAP assignment. However, without integrality constraints, this is an approximation.

Using RLPs, we can compactly encode the MAP-LP for MLNs. To see this, consider the friends-and-smokers MLN [12] consisting of two rules. The first rule says that smoking can cause cancer 0.75 $\mathtt{smokes(X)} \Rightarrow \mathtt{cancer(X)}$, and the second implies that if two people are friends then they are likely to have the same smoking habits 0.75 $\mathtt{friends(X,Y)} \wedge \mathtt{smokes(X)} \Rightarrow \mathtt{smokes(Y)}$. Since both MLNs and RLPs are based on logic programming, we expect to be able to define the MAP-LP in a first-order fashion, incorporating the MLN semantics directly into (6). In order to understand how this works, let us discuss some key features. From the MLN semantics we know the following: (1) we can generate the set of random variables by grounding out $\mathtt{smokes(X)}$, $\mathtt{cancer(X)}$, and $\mathtt{friends(X,Y)}$ for every person in the domain. (2) We can generate the set of factors by grounding out the above two rules. Every ground rule is a factor over the ground atoms involved in it. In the friends-and-smokers example, we have pairwise and ternary rules (involving 2 resp. 3 ground atoms).

Thus, the set of MAP-LP variables is parametrized in terms of the predicates and clauses of the MLN, and we directly get a set of logically parameterized LP variables. They are: atom beliefs, $\mathtt{m(smokes(X)=t)}$, $\mathtt{m(smokes(X)=f)}$; pairwise factor beliefs,

$$\mathtt{m(smokes(X)=f, cancer(X)=f)}, \quad \mathtt{m(smokes(X)=t, cancer(X)=f)}\,,$$
$$\mathtt{m(smokes(X)=f, cancer(X)=t)}, \quad \mathtt{m(smokes(X)=t, cancer(X)=t)}\,,$$

induced by the rule $\mathtt{smokes(X)} \Rightarrow \mathtt{cancer(X)}$; and finally ternary beliefs like $\mathtt{m(friends(X,Y)=f, smokes(X)=t, smokes(Y)=f)}$ induced by the third rule (we will not list all joint configurations here). It is now easy to see that the marginalization constraints are logically parametrized as well. For example, we have constraints such as

$$\sum\nolimits_{\mathtt{cancer(X)}\in\{\mathtt{t,f}\}} \mathtt{m(smokes(X), cancer(X))} = \mathtt{m(smokes(X))}\,.$$

Finally, the objective weight of a joint belief variable follows from the MLN semantics as well: it is the weight of the rule if the joint configuration satisfies the rule and 0 otherwise. E.g., the weight of $\mathtt{m(smokes(anna)=t, cancer(anna)=f)}$ is 0 because this truth assignment is not a model of $\mathtt{smokes(anna)} \Rightarrow \mathtt{cancer(anna)}$. In contrast, the beliefs over the remaining three configurations have a weight of 0.75. Consequently, one can encode the objective weights in a predicate `w` given in the LogKB.

The resulting RLP is shown in Fig. 12. If we were to ground out this RLP, we would get exactly the LP from (6) applied to the ground MRF induced by the friends-and-smokers MLN. Note how we have abstracted away the names of the predicates. As a result, this RLP can be used with **any ternary MLN**. Changes in evidence, constants, or MLN rules are confined entirely to the LogKB, which for the friends-and-smokers MLN looks as follows:

```
person(anna).  person(bob).   ...    #the people in the social network
value(f).  value(t).                 #an atom could be true or false
#encoding of the MLN clauses and their weights
#smoking causes cancer
w(smokes(X), cancer(X), t, f) = 0 :- person(X).
w(smokes(X), cancer(X), V1, V2) = 0.75 :-
                   person(X), value(V1), value(V2).
#friends have similar smoking habits
w(friends(X, Y), smokes(X), smokes(Y), t, t, t) = 0.75 :-
                   person(X), person(Y).
w(friends(X, Y), smokes(X), smokes(Y), t, f, f) = 0.75 :-
                   person(X), person(Y).
w(friends(X, Y), smokes(X), smokes(Y), V1, V2, V3) = 0 :-
                   person(X), person(Y), value(V1), value(V2), value(V3).
#evidence
m(smokes(gary), t). ... m(smokes(helen), t).   #known smokers
m(friends(anna, bob), t). ... m(friends(helen,iris), t). #known friendships
```

Here, `w` represents the objective value assigned to the joint belief of the atoms in the first two (resp. three) arguments having the values of the latter two (resp. three) arguments. Please keep our short-hand notation in mind: `w(smokes(X), cancer(X), t, f) = 0` stands for `w(smokes(X), cancer(X), t, f, 0)`.

With the RLP in Fig. 12, we have seamlessly merged together MLN and MAP-LP semantics in one compact RLP and retained the freedom to change the MLN rules, domain constants and evidence easily within the RLP itself. This illustrates the modeling power of RLP in relational domains and presents a strong argument that **(Q1)** can be answered affirmatively.

Let us now turn towards investigating **(Q2)**. As shown in previous works, inference in graphical models can be dramatically sped-up using lifted inference. Thus, it is natural to expect that the symmetries in graphical models, which can be exploited by standard lifted inference techniques, will also be reflected in the corresponding MAP-(R)LP. To verify whether this is indeed the case we induced MRFs of varying size from a friends-and-smokers MLN by varying the number of people from 50 to 300.

The results of the experiments are summarized in Figs. 13(a) and (b). As Fig. 13(a) shows, the number of LP variables is significantly reduced. Not only is the linear program reduced, but due to the fact that the lifting is carried out only once, we also measure a considerable decrease in running time as depicted in Fig. 13(b). Note that the time for the lifted experiment includes the time needed to compile the LP. This affirmatively answers **(Q2)**.

```
var m/2;  #single node, pairwise, and
var m/4;  #triplewise beliefs
var m/6;  #of configurations to be determined by the solver
#value of the MAP assignment
score = sum{w(P, V)} w(P, V) * m(P, V) +
    sum{w(P1, P2, V1, V2)} w(P1, P2, V1, V2) * m(P1, P2, V1, V2) +
    sum{w(P1, P2, P3, V1, V2, V3)} w(P1, P2, P3, V1, V2, V3) *
    m(P1, P2, P3, V1, V2, V3);

#marginalization of pairwise beliefs
marginalize(P1, P2, V1) = sum{w(P2, V2)} m(P1, P2, V1, V2);
...
#marginalization of ternary beliefs
marginalize(P1, P2, P3, V1) = sum{w(P3, V3), w(P2, V2)}
    m(P1, P2, P3, V1, V2, V3);
...
maximize: score;                         #find assignment with largest value
subject to forall {P in w(P, _)}:
        sum {w(P, V)} m(P, V) = 1;            #atom beliefs sum to one
#pairwise consistency constraints
subject to forall {P1, P2, V1 in w(P1, P2, V1, _)}:
        marginalize(P1, P2, V1) =  m(P1, V1);
...
#ternary consistency constraints
subject to forall {P1, P2, P3, V1 in w(P1, P2, P3, V1, _, _)}:
        marginalize(P1, P2, P3, V1) = m(P1, V1);
...
```

Fig. 12. RLP encoding the MAP-LP for the friends-and-smokers MLNs as shown in (6). The last two constraints as well as the last two aggregates have symmetric copies that have been omitted (this redundancy is necessary, since logic predicates are not symmetric).

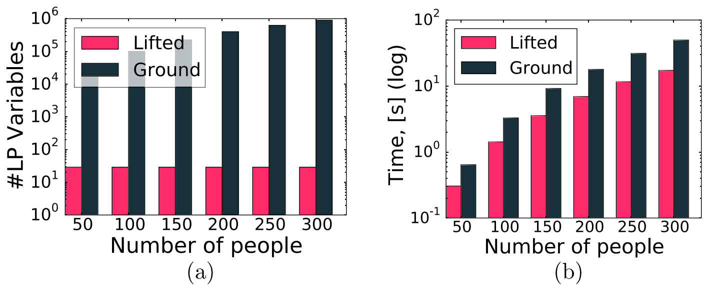


Fig. 13. Experimental results of relational linear programming for MAP-LP inference within Markov logic networks. (a) Number of variables in the lifted and ground LPs. (b) Time for solving the ground LP vs. time for lifting and solving. (Best viewed in color.)

5.3. Programming collective classification using LP-SVM

Networks have become ubiquitous, and often we are interested in how objects in these networks influence each other. Consequently, collective classification has received a lot of attention [50–53,12,13]. It refers to the task of jointly classifying a set of inter-related objects. It exploits the fact that inter-related objects often share a lot of similarities. For example, in citation networks there are dependencies among the topics of a papers' references, and in social networks people who are in close contact tend to have similar interests. Using these dependencies allows collective classification methods to outperform methods that assume object independence [50].

Despite being successful, most of the research in collective classification so far has focused on generative models, at least as part of the column generation approach to solving quadratic program formulations of the collective classification task. We here illustrate that relational linear programming could provide a first step towards a principled large-margin approach. Specifically, we introduce a transductive[10] collective SVM based on RLPs. In essence, to classify unlabeled objects, we will not only consider their attributes, but also their relationship to the labeled ones as part of the model.

We start off by reviewing the vanilla LP approach to SVMs and then show how RLPs can be used to program a transductive collective classifier.

Support vector machines (SVMs) [54] are the most widely used model for discriminative classification at the moment. The hypothesis space of an SVM is the space of affine models represented by coefficients (weights) of a vector orthogonal to a hyperplane, and an intercept. In the soft-margin version of SVM, training strives to strike a balance between width of the margin between the examples and the separating hyperplane, and the number of examples that fall within or on the wrong

[10] Transductive inference is a mode of direct reasoning from observed to unobserved instances without an inductive hypothesis finding problem [54].

```
var slack/1;                              #the slacks
var weight/1;                             #the slope of the hyperplane
var b/0;                                  #the intercept of the hyperplane
var r/0;                                  #margin

slacks=sum{label(I)} slack(I);                                  #total slack
innerProd(I)=sum{attribute(_,J)} weight(J)*attribute(I,J);      #hyperplane

#find the largest margin. Here C encodes a trade-off parameter
minimize: -r + C * slacks;

#examples should be on the correct side of the hyperplane
subject to forall{I in label(I)}:
        label(I)*(innerProd(I) + b) + slack(I) >= r;
#weights are between -1 and 1
subject to forall {J in attribute(_, J)}: -1 <= weight(J) <= 1;
subject to r >= 0;          #the margin is positive
subject to forall {I in label(I)}: slack(I) >= 0;  #slacks are positive
```

Fig. 14. A linear programming SVM encoded as an RLP. Note, for convenience, we now use a `minimize` instead of a `maximize` statement.

side of the margin (misclassifications). Maximization of the margin is achieved by penalizing squared Euclidean norm of the weight vector and traditionally posed as a quadratic optimization problem (QP).

Zhou et al. [16] have shown that the same problem can be modeled as an LP when replacing the squared Euclidean norm by the infinity norm without incurring a major loss in generalization performance. The LP suggested by Zhou et al. is as follows, and we refer to [16] for more details:

$$\begin{aligned} &\text{minimize}_{\xi_i, r\in\mathbb{R}} && -r + C\sum\nolimits_{i=1}^{l} \xi_i \\ &\text{subject to} && y_i(\mathbf{w}\mathbf{x}_i + b) \geq r - \xi_i, \\ & && -1 \leq \mathbf{w}_i \leq 1\,, \\ & && \xi_i \geq 0,\ r \geq 0\,. \end{aligned}$$

This LP-SVM can readily be applied to classify papers in the Cora dataset [55]. The Cora dataset consists of 2708 scientific publications classified into one of seven classes. The citation network consists of 5429 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of a corresponding word from a dictionary. The dictionary consists of 1433 unique words. We turned this problem into a binary classification problem by taking the largest of the 7 classes as a positive class and merging the other 6 into a negative class.

The RLP in Fig. 14 encodes the vanilla LP-SVM. In this non-collective setting, we ignored the citation information and classified documents based on 0/1 word features using the following LogKB:

```
C = 0.021.                # regularization parameter
attribute(31336, 119).  attribute(31336, 126).  ...
label(17798) = -1.  label(10531) = 1.  ...
```

Here the arguments of `attribute/2` are the indices of a document and a word present in a document respectively. Words that are not present in a document, i.e., have value zero in the original dataset, are not specified. Again, this answers **(Q1)** affirmatively, since the RLP stays fixed for different datasets and only the LogKB changes.

We now show how to transform this vanilla (R)LP-SVM model into a transductive collective one (TC-RLP-SVM) by making only a slight modification to the RLP.

Indeed, there are a number of ways to do this, and exploring them is an interesting avenue for future work. Since our primary focus is not to achieve the best performance but to illustrate the ease of the relational mathematical programming approach, we chose the following, rather basic approach. We add constraints which favor that unlabeled instances have the same label as their labeled neighbors. To account for contradicting examples, we introduce slack variables for these constraints and add them to the objective with a separate penalty parameter. This results in the TC-RLP-SVM model shown in Fig. 15. Here, the new predicate `pred/2` denotes the predicted label for unlabeled instances. The LogKB gets two new predicates:

```
C(1) = 0.0021.  C(2) = 0.0031.

cite(89547, 1132385).  cite(89547, 1152379).  ...
query(1128959).  query(16008).  ...
```

The `cite/2` predicate encodes citation information, and the `query/1` predicate marks unlabeled instances whose labels are to be inferred. We notice that the parameters in the objective play a different role in the TC-RLP-SVM. In the vanilla case a parameter has to be carefully chosen in the training phase, but then prediction is done using the learned weight vector only. In the transductive setting the linear model, which is at the heart of RLP-SVM, plays a role of a medium between

```
var pred/1;       #predicted label for unlabeled instances
var slack/1;      #the slacks
var coslack/2;      #slack between neighboring instances
var weight/1;     #the slope of the hyperplane
var b/0;          #the intercept of the hyperplane
var r/0;          #margin

slack   = sum{label(I)} slack(I);
coslack = sum{cite(I1,I2),label(I1),query(I2)} slack(I1,I2)
                + sum{cite(I1,I2),label(I2),query(I1)} slack(I1,I2)

#find the largest margin. Here the C's encode trade-off parameters
minimize: -r + C(1) * slack + C(2) * coslack;

subject to forall {I in query(I)}: pred(I) = innerProd(I) + b;
#related instances should have the same labels.
subject to forall {I1, I2 in cite(I1, I2), label(I1), query(I2)}:
    label(I1) * pred(I2) +  slack(I1, I2) >= r;
#the symmetric case
subject to forall {I1, I2 in cite(I1, I2), label(I2), query(I1)}:
    label(I2) * pred(I1) + slack(I1, I2) >= r;

#examples should be on the correct side of the hyperplane
subject to forall {I in label(I)}:
        label(I)*(innerProd(I) + b) + slack(I) >= r;
#weights are between -1 and 1
subject to forall {J in attribute(_, J)}: -1 <= weight(J) <= 1;
subject to : r >= 0;          #the margin is positive
subject to forall {I in label(I)}: slack(I) >= 0;    #slacks are positive
```

Fig. 15. An RLP-SVM model for collective inference in a transductive setting.

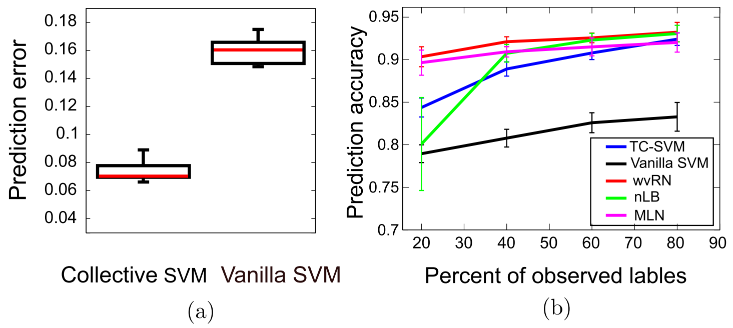


Fig. 16. Experimental results of linear programming support vector machines. (a) TC-RLP-SVM versus vanilla RLP-SVM on the Cora dataset; (b) Experimental results of evaluation of different collective classification methods on the Cora dataset. (Best viewed in color.)

labeled and unlabeled instances. The weights are tuned for every new problem instance (recall the point of transductive inference is not to learn an intermediate predictive model, but to do inference directly). In this case the objective parameters become more important. The optimal value of the parameters depends on the data and hence they have to be tuned during the transductive inference phase as well using for instance cross-validation.

To investigate the benefit of the relational mathematical programming approach to collective inference, we compared the performance of the vanilla RLP-SVM to the TC-RLP-SVM—the same RLP-SVM with few additional relational constraints—on the task of paper topic classification using the Cora dataset. The experiment protocol was as follows. We first randomly split the dataset into a training set A, a validation set C, and a test set B in proportion 70/15/15. The validation set was used to select the parameters of the TC-RLP-SVM in a 5-fold cross-validation fashion. That is, we split the validation set into 5 subsets C_i of equal size. On these sets we selected the parameter using a grid search for each C_i on an $A \cup (C \setminus C_i)$ labeled and $B \cup C_i$ unlabeled examples, computing the prediction error on C_i and averaging it over all C_is. We then evaluated the selected parameters on the test set B whose labels were never revealed in training. We repeated this experiment 5 times (one for each C_i) for the TC-RLP-SVM. For consistency, we followed the same protocol with RLP-SVM, except that the set $B \cup C_i$ did not appear during training at all as the RLP-SVM has no use for unlabeled examples. That is, we selected parameters by training on $A \cup (C \setminus C_i)$ and evaluating on C_i. The selected parameters were then evaluated on the test set B. The results are summarized in Fig. 16(a). As one can see, the vanilla RLP-SVM achieved a prediction error of $16 \pm 1\%$. The TC-RLP-SVM achieved $7.5 \pm 1\%$. A paired t-test ($p = 0.05$) revealed that the difference in mean is significant.

Although best performance was not our goal, the performance is quite encouraging. Consequently, we conducted a more detailed study of the TC-RLP-SVM performance. We applied three state-of-the-art collective classification approaches to the Cora dataset, evaluated them on exactly the same splits of the data end compared their performance to that of our model.

The first two methods are part of NetKit [56]—a toolkit for classification in networked data. Among the many available options we chose the two methods that performed best in the original study [56]. The simplest classifier available in the toolkit, the weight-voted relational neighbor classifier (wvRN), turned out to also be one of the two best in combination with relaxation labeling as a collective inference procedure. In its basic setting (which is the one being used in the experiments to follow) it simply predicts a probability of each label for an instance as an average probability of this label for all the neighbors of the instance in the network:

$$P(x_i = c|N_i) = Z^{-1}\sum\nolimits_{v_j \in N_i} P(x_j = c|N_j)$$

where N_i is a set of neighbors of a node v_i and Z a normalization constant. The second model we compared to is the link-based classifier [57] (nLB). This classifier creates a feature vector for a node by aggregating the labels of neighboring nodes, and then uses logistic regression to build a discriminative model based on these feature vectors. The learned model is then applied to estimate $P(x_i = c|N_i)$. Various aggregation methods can be used in this setting, including existence, the mode, and value counts. The latter method with normalized value counts has been shown to have the best performance and is thus used in our study. It is also combined with relaxation labeling for collective inference. For a fair comparisons to the TC-RLP-SVM, both methods are also allowed to use instance-level information (word vectors) by training a logistic regression model on the observed instances and using its predictions as priors for the unobserved ones. Finally, a simple network-only MLN was used as the last baseline. It consists of a single rule

```
category(v0, c), cites(v0, v1) => category(v1, c)
```

with an additional restriction that every document can have only one category. For probabilistic inference we used Tuffy [33].

The experiment protocol was follows: we had 4 settings in which we kept 20, 40, 60 and 80% of the labels in the training set. For each setting 10 random splits were generated, with labeled instances selected uniformly at random. Each method was then evaluated on the same random splits. We report average errors for each setting and method. To once again illustrate the benefits of using relational information, the Vanilla (LP-)SVM, which uses instance-level information only, is included into the evaluation as well. The results are summarized in Fig. 16(b).

The performances of the NetKit methods correspond to those originally reported by Macskassy and Provost [56]. The nLB classifier does not outperform the simpler wvRN even with 80% of the labeled examples. The MLN classifier, which ignores the instance-level information, shows a very similar performance to the wvRN but is consistently a little bit worse. Note that already with 40% of the labeled examples, all the classifiers, except for the Vanilla (LP-)SVM, achieve accuracies of at least 85%. In fact, even the highest performance achieved by the Vanilla (LP-)SVM (83% accuracy) is significantly lower. This confirms that utilizing relational information can significantly boost the classification performance. However, one can also see that the Vanilla (LP-)SVM can significantly be improved by just adding few additional constraints. More precisely, the TC-(LP-)SVM outperforms the Vanilla (LP-)SVM in all cases and catches up quickly with the NetKit methods. The slightly lower performance in general and in particular for lower percent of observed labels is likely due to the fact that we used a very basic model that does not propagate label information through the citation network.

In any case, the empirical results clearly show that collective inference can be tackled using RLPs and, hence, provide another affirmative answer to **(Q1)**. That the performance gain was achieved by simply reprogramming the Vanilla (LP-)SVM highlights the power of relational linear programming and answers **(Q3)** affirmatively. Keep in mind that our goal was not to obtain a novel and better collective model, but just to demonstrate the possibilities opened up by relational linear programming.

5.4. Lifted LP support vector machines

To close the loop, we finally investigated the liftability of RLP-SVMs. Although not surprising from a theoretical perspective—we have already shown that any LP that contains symmetries is liftable—this constitutes the very first symmetry-aware (LP-)SVM solver and is an encouraging sign that lifting goes beyond probabilistic inference: lifted statistical machine learning in general is not insurmountable.

To investigate symmetries in relational SVM classification, we solved the LPs produced by the parameter selection phase of TC-SVM across different observed label percentages in a lifted fashion. More precisely, following the experimental protocol of the last experiment, we started with four cases: 20%, 40%, 60% and 80% observed labels. For each case, we limited ourselves to evaluating 4 regularization parameter pairs (`(C(1), C(2))` in line 13 of the RLP in Fig. 15, also referred to as (c_1, c_2) for brevity). These four pairs consist of the optimal parameters, and 2×, 10× and 1000× the optimal parameters. The motivation behind this choice is to represent different parameter scales during the grid search. As in the previous experiment, for each parameter evaluation we did a 5-fold cross-validation. Finally, to encompass a wider variety of training sets, we repeated this process across 10 random $A/B/C$ splits. Thus, our dataset consists of $4 \times 4 \times 5 \times 10 = 800$ LPs in total.

We first computed the compression ratio as the ratio of the size (number of constraints and variables) of the lifted LP and the size of the ground LP for each of the LPs. Fig. 17(a) breaks down the ratios by percentage of observed labels. As one can see, even with the constraints induced by the random subgraphs of Cora corresponding to the random label deletions,

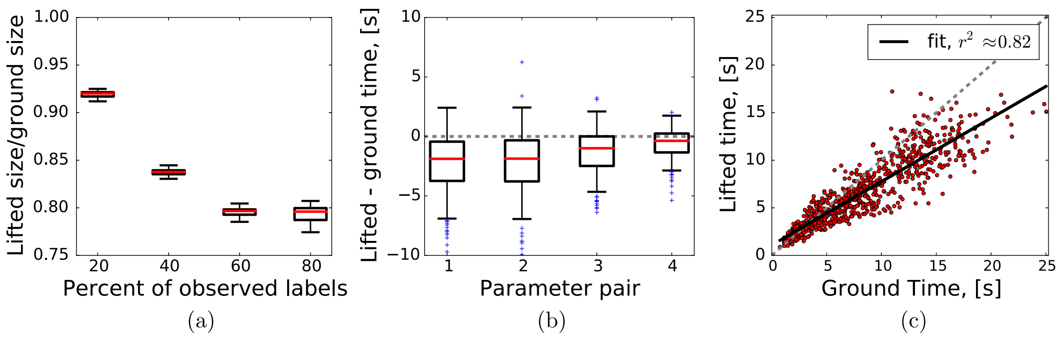


Fig. 17. Speeding-up TC-SVM with lifted linear programming. All experiments were performed with Gurobi; (a) Compression ratios: ratios of lifted model size (number of variables plus number of constraints) to ground model size across the random CV-splits used in parameter selection for a fixed percentage of observed labels; (b) net speed-up (compression + solving the lifted LP in Gurobi vs. solving the ground LP in Gurobi) for solving with Gurobi across the random CV-splits for 4 different (c_1, c_2) parameter pairs; (c) scatter plot of ground vs. lifted time across all LPs used in the experiment. (Best viewed in color.)

a non-negligible amount of symmetry is present in the problem. Moreover, on average, the compression rate increases as more labels are introduced. On first sight this seems to be counterintuitive since adding information is typically considered to break symmetries. However, since the RLP employs information only between pairs of observed and unobserved instances, the lower the number of such pairs, the more symmetry is there actually. That is, as we increase the percent of observed labels, the number of such pairs is likely to decrease.

Next, we solved all LPs using Gurobi (version 6.0), the most popular LP solver in 2012 according to NEOS statistics[11] with its default settings. We used the same machine as in the MDP experiments. Fig. 17(b) summarizes the net gain per LP of lifted solving (including the time for symmetry detection and compression) as opposed to solving the ground model across the different parameter pairs. One can see that even with Gurobi's sophisticated pre-solving heuristics, lifting yields a considerable speed-up, although there are a few LPs where solving the lifted model was slower. Across all LPs, however, lifting the LPs gained us a total of 1290.83 seconds compared to the ground solver. Fig. 17(c) shows a scatterplot of the lifted vs. ground solving time when solving with Gurobi, together with the fitted regression model. As can be seen, for this particular set of LPs, the correlation coefficient between lifted and ground time is on the order of the amount of symmetry present in the problem. These results not only provide an affirmative answer to **(Q2)** but also present strong evidence that speed-ups can indeed be observed across different solvers, even in state-of-the-art commercial packages. Hence, **(Q4)** can also be answered affirmatively.

Finally, to investigate **(Q5)**, we recorded the relative difference of the quality of the lifted and the ground solutions, i.e.

$$\Delta = \frac{\max(\boldsymbol{c}^T \boldsymbol{x}^*_{\text{Lifted}}, \boldsymbol{c}^T \boldsymbol{x}^*_{\text{Ground}})}{\min(\boldsymbol{c}^T \boldsymbol{x}^*_{\text{Lifted}}, \boldsymbol{c}^T \boldsymbol{x}^*_{\text{Ground}})} - 1 ,$$

in all experiments involving lifting. For all experiments with CVXOPT, we ran the solver with a convergence threshold of 10^{-7}. For the MDP experiments, we observed Δ's between 10^{-6} and 10^{-9}. For MAP, the Δ's were in the range 10^{-7} to 10^{-8}. With Gurobi, in all experiments Δ was in the range 10^{-13} to 10^{-14}. Gurobi automatically set its convergence threshold to 10^{-8}. While one can imagine the existence of pathological cases where lifting influences the numerical performance of the solver, this was not observed in our experiments. This provides an affirmative answer to question **(Q5)**.

Taking all results together, the experimental illustrations clearly show that all five questions **(Q1)**–**(Q5)** can be answered affirmatively.

6. Related work

Relational linear programming as introduced here is related to two lines of research, namely rich languages for mathematical programming and exploiting symmetries for AI tasks.

6.1. Languages for mathematical programming

Several expressive modeling languages for mathematical programming have been proposed. Examples of popular ones are AMPL [29,30], GAMS [58], AIMMS [59], and Xpress-Mosel [60], and more can be found in general surveys [61–63]. These modeling languages are mixtures of declarative and imperative programming styles using sets of objects to index multidimensional parameters and LP variables. Employing essentially "for-in" statements, this indexing is used to define

[11] http://zverovich.net/2013/01/01/neos-statistics-for-2012.html. In 2013, it is still the far more popular than CPLEX although non-linear solvers were more in focus, see http://zverovich.net/2014/01/02/neos-statistics-for-2013.html. Both webpages were queried on Nov. 29, 2014.

objectives and constraints in an abstract model, which separates the declaration of a model from the data used to generate a specific model instance. Although this indeed has much in common with relational linear programming, there are also fundamental differences. Consider e.g. AMPL. As stated by Fourer, Gay, and Kernighan [30, page 122], the "current version of AMPL does not support a full-fledged 'logical' type of parameter that would stand for only the values true and false". Instead it promotes the use parameters of type binary together with logical operators such as "if-then-else" and in particular "for-all" and "exists". However, this requires one to also encode what is false, and these operators are intended to work with parameters, not LP variables as featured in relational linear programming. To do this in AMPL, one would have to introduce some binary LP variables with corresponding constraints that encode the "exists" and "for-all" operators. This, however, again blows up the model since we would need to also represent what is false. In contrast, by unification across different queries, relational linear programming connects different "sets" of LP variables encoded as atoms using logic programming proof techniques.

As an alternative, since index sets and data tables are closely related to the attributes and relations of relational database systems, there have also been proposals for "feeding" a linear program directly from relational database systems [64–66] as well as AMPL's interface with relational databases to define sets and parameters. However, this takes the "logic" out of the LP modeling language. Moreover, logic programming—which allows one to use e.g. complex terms and to define recursive predicates, where we do not know how often we have to union two "tables"—was not considered and the resulting approaches do not provide a syntax close to the mathematical notation of linear and logic programs.

This also holds for Roth and Yih [67], who presented an abstract ILP model for global inference in natural language processing (NLP) tasks. The idea is to use an object-oriented entity-relationship model in the background and to index the sum-statements in ILPs over the members of object and relation classes. This is closer in spirit to relational linear programming but e.g. unification and negation are not supported when defining constraints. Generally, motivated by the already mentioned "Model + Solver" paradigm, NLP witnesses a growing need for relational mathematical modeling languages [68–72]. Consider e.g. the subsequent work on global inference by Clarke and Lapata [69]. They quantify constraints using statements such as "$\forall i, j, k \,:\, x_j \wedge x_k$ conjoined by x_i" saying if two head words are conjoined in the source sentence, then add some algebraic constraints over variables indexed by i, j, and k. Although akin to a relational specification, no general relational modeling language for (I)LPs has been presented. The approaches are rather ad-hoc systems that "ground" the specific constraints by looping over some specialized code in some high-level programming language such as C/C++. For solving relational Markov Decision Processes, Sanner and Boutilier [24] used approximate linear programming with relational constraints represented via case statements. This approach scaled to problems of previously prohibitive size by avoiding grounding and is indeed close in spirit to relational programming. However, Sanner and Boutilier did not introduce a relational modeling language for arbitrary LPs.

Thus, following Klabjan, Fourer and Ma [73, unpublished], one can argue that there is a need for a mathematical programming solution with built-in language constructs that not only facilitates natural algebraic modeling but also provides integrated capabilities with logic programming. Klabjan et al. show that the basic functionalities and requirements of mathematical programming languages such as AMPL can be realized in any Datalog-like logic programming language. This is probably the closest in spirit to relational linear programming. However, there are significant differences. First, RLPs directly extend the syntax of mathematical programming languages with logic programming concepts and, hence, stay close to the mathematical notation of LPs. Moreover, Datalog is not expressive enough to capture the entire AI spectrum of problems we would like to tackle using RLPs. For instance, it disallows complex terms as arguments of predicates—e.g., $p(1, 2)$ is admissible but not $p(f(1), 2)$—making it hard to work with complex data structures such as graphs, imposing certain stratification restrictions on the use of negation and recursion, and only allowing range restricted variables, i.e. each variable in the head of a rule must also appear in a not negated clause in the premise of this rule. Consider dealing with LP problems on graphs. Indeed, with Datalog one could encode this using predicates encoding the nodes and edges of the graph. However, there are several other ways to encode graphs using complex terms, and which form is the best certainly depends on the LP application at hand. For instance, if the number of nodes is important and not provided by the user (the graph is generated within the program) then the simple predicate-based representation would require meta-predicates to compute the number of nodes. In a list-based representation, however, we can just compute the length of the list. Moreover, complex terms allow—as illustrated in the present paper—for a concise specification of MAP-LP inference in Markov logic networks. Hence, we advocate Prolog for similar reasons as Eisner and Filardo do in their AI language DYNA [74].

Recently, Mattingley and Boyd [75] have introduced CVXGEN, a software tool that takes a high level description of a convex optimization problem family, and automatically generates custom C code that compiles into a reliable, high speed solver for the problem family. CVXGEN features indexed parameters and variables but no logical way for declaring the objective and the constraints while referring to them. Generally, there is a tendency to embed mathematical programming languages into imperative high-level programming language such as Python [76,77], Java, C/C++ and Matlab, among others.[12] In particular Diamond, Chu and Boyd's CVXPY [77] enables an object-oriented approach to constructing optimization problems and notes that such an object-oriented approach is simpler and more flexible than the traditional method of constructing problems by embedding information in matrices. In contrast, the goal of relational linear programming is to put logic programming into optimization. The same holds for Rizzolo and Roth's Learning Bayes Java (LBJ) [78], which grew out of the above mentioned

[12] All major solver suites such as GUROBI and CPLEX feature APIs for this. We refer to the corresponding manuals.

research on global inference in NLP. LBJ combines ideas from optimization, first order logic, and object-oriented programming for compiling complex models into ILPs. In particular, LBJ provides a convenient syntax for specifying the interactions between Java functions as arbitrary first-order-logical formulas. That is, it does not aim at putting logic programming into optimization. Moreover, the only two predicates in the constraints are equality and inequality, although their arguments may be arbitrary Java expressions. Finally, Gordon, Hong, and Dudík [79,80] developed first-order programming (FOP) that combines the strength of mixed-integer LPs and first-order logic. In contrast to the present paper, however, they focused on first-order logical reasoning and not on specifying arbitrary linear programs in a relational way.

6.2. *Exploiting symmetries for solving AI tasks*

Another distinguishing feature of relational linear programming compared to all the modeling approaches mentioned so far is that none of them has considered how to detect and exploit symmetries in arbitrary LPs. Indeed, there are symmetry-breaking approaches for (mixed–)integer programming (MIPs) [81] that are also featured by commercial solvers such as CPLEX [82] and GUROBI [83]. The dominant paradigm is to add symmetry breaking inequalities, similarly to what has been done for SAT and CSP [84]. Alternatively one can prune the search space to eliminate symmetric solutions (see e.g. [81] for a survey). In contrast, lifted linear programming reduces the dimensionality of the LP at hand; the LP is compressed. To do so, one takes advantage of convexity and projects the LP into the fixed space of its symmetry group [44]. The projections we investigated in the present paper are similar in spirit. Until recently, however, discussions were mostly concentrated on the case where the symmetry group of the LP consists of permutations [85]. In such cases the problem of computing the symmetry group of the LP can be reduced to computing the colored automorphisms of a “coefficient” graph connected with the linear program [86,81]. Moreover, the reduction of the LP in this case essentially consists of mapping variables to their orbits. Our approach subsumes this method, as we replace the orbits with a coarser equivalence relation which, in contrast to the orbits, is computable in quasilinear time. Going beyond permutations, Bödi and Herr [44] extend the scope of symmetry, showing that any invertible linear map, which preserves the feasible region and objective of the LP, may be used to speed-up solving. While this setting offers more compression, the symmetry detection problem becomes even more difficult.

Finally, lifted linear programming as introduced here has been proven already beneficial. Lifted (I)LP-MAP inference approaches for (relational) graphical models based on (relaxed) graph automorphisms and variants have been explored in several ways [43,87–89,42], which go beyond the scope of the present paper.

7. Future work

The indent of our paper has been to introduce and explore the basic idea and concepts of relational linear programming. Consequently, there is significant additional work to be done. More work is needed to extended the language presented with the concepts of modules and name spaces, allowing one to build libraries of relational programs, as well as combining it with Mattingley and Boyd's [75] CVXGEN to automatically generate custom C code that compiles into a reliable, high speed solver for the problem family at hand. The framework should also be extended to other mathematical programs such as (mixed) integer LPs, quadratic programs, and semi-definite programs, among others. Column generation and cutting plane approaches to lifted linear programming should also be explored. Very recently, e.g., Lu and Boutilier [90] presented a value-directed compression technique for propositional assignment LP. They dynamically segment the individuals into blocks using a form of column generation, constructing groups of individuals who can provably be treated identically in the optimal assignment solution. Together with the declarative nature of the resulting relational mathematical programming approach to AI, one should investigate program analysis approaches to automate problem decomposition at a lifted level. If symmetries could be detected and exploited efficiently in other mathematical programs, too, this would put general symmetry-aware machine learning and AI even more into reach.

In general, we have only started to explore the interaction of lifting and solving linear programs, and mathematical programs in general. Since lifting changes the geometry of the linear program, it could have an impact on the various heuristics employed in modern solvers for efficient solving linear programs. This interaction should be explored and it should be investigated whether it may even cancel the benefits of lifting. Ground and lifted linear programs may also require different convergence thresholds in order to maintain the same quality of solutions. While we did not observe any surprising behavior in our experiments, exploring these issues is an important and interesting avenue for future work.

The most attractive immediate avenue, however, is to explore relational linear programming within other AI and machine learning tasks. First of all, the novel collective classification approach should be rigorously evaluated and compared to other approaches on a number of other benchmark datasets. Other attractive avenues are the exploration of the symmetry-aware SVMs outlined in the present paper within other learning setting, relational dimensionality reduction via LP-SVMs [91], novel relational boosting approaches via linear programs [92], and developing relational and lifted solvers for computing optimal Stackelberg strategies in two-player normal-form game [93], among others. Lifting should also be explored within recent probabilistic CSP proposals. For instance, Sraswat et al. [94] very recently sketched the probabilistic constraint programming language C10. It is based on the concurrent constraint programming framework and implemented on top of X10, a language for scale-out computation. Lifting may scale C10 even more. Along similar lines, one should explore lifting consensus optimization [95].

One should also push the programming view on relational machine learning tasks. As an example consider kernels for classifying graphs. Graph classification is a very important task in bioinformatics [96], natural language processing [97] and many other fields. Kernelized SVM is often the method of choice. The inner products in SVM can be replaced by functions (kernels) that effectively represent inner products in higher dimensional space. This inner product view on one hand makes SVM a non-linear classifier but also allows one to deal with structured objects such as graphs e.g. using convolution kernels [98]. Convolution kernels introduced the idea that kernels could be built to work with discrete data structures interactively from kernels for smaller composite parts. RLPs suggests to view them as a programming task. Within LogKB we define the parts and a generalized sum over products—a generalized convolution—is realized within RLP. Similarly, many other graph kernels known could be realized. Walks [99], cyclic patterns [100] and shortest-paths [101] are examples of substructures considered so far. All these concepts are naturally representable as a logic programs in Prolog. E.g., `shortestPath(A, B, G, Paths)` computes in `Path` the shortest path between nodes `A` and `B` in graph `G`. One can program the shortest path and use it to define a convolution kernel in an RLP SVM as follows:

```
k(G1, G2) = sum{vertex(G1,V1), vertex(G1,V2),
  shortestPath(V1,V2,G1,L1),  vertex(G2,V3), vertex(G2,V4),
  shortestPath(V3,V4,G2,L2)}
             simple_k(V1,V2,L1,V3,V4,L2).
simple_k(V1,V2,L1,V3,V4,L2) = ...
```

where `simple_k` is any kernel on vertices and lengths. In this way, relational mathematical programming suggests a novel programming view on graph kernels that integrates the kernel programming with the mathematical program into one declarative model.

8. Conclusion

We have introduced relational linear programming, a simple framework combining linear and logic programming. Its main building blocks are relational linear programs (RLPs). They are compact LP templates defining the objective and the constraints through the logical concepts of individuals, relations, and quantified variables. This contrasts with mainstream LP template languages such as AMPL, which mixes imperative and linear programming, and allows a more intuitive representation of optimization problems over relational domains where we have to reason about a varying number of objects and relations among them, without enumerating them. Inference in RLPs is performed by lifted linear programming. That is, symmetries within the ground linear program are employed to reduce its dimensionality, if possible, and the reduced program is solved using any off-the-shelf linear program solver. This significantly extends the scope of lifted inference since it paves the way for lifted LP solvers for linear assignment, allocation and many other AI task that can be solved using LPs. Empirical results on approximate inference in Markov logic networks using LP relaxations, on solving Markov decision processes, and on collective inference using relational LP support vector machines illustrated the promise of relational linear programming.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. K.K. and M.M. would also like to thank Babak Ahmadi for his fruitful collaboration on the earlier version of lifted linear programming as well as Martin Grohe and Aziz Erkal Selman for their fruitful collaboration on color-refinement. P.T. was with the CS Department of the University of Bonn, Germany, when working on relational linear programming. K.K. and M.M. were with Fraunhofer IAIS when working on the earlier version of lifted linear programming. The research presented was partly supported by the Fraunhofer ATTRACT fellowship STREAM, by the EC under contract number FP-248258-First-MM, by the German-Israeli Foundation for Scientific Research and Development, 1180-218.6/2011, and by the German Science Foundation (DFG), KE 1686/2-1, as part of the Coordination Project SPP 1527.

References

[1] L. Getoor, B. Taskar (Eds.), Introduction to Statistical Relational Learning, MIT Press, Cambridge, MA, 2007.
[2] L. De Raedt, Logical and Relational Learning, Springer, 2008.
[3] L. De Raedt, P. Frasconi, K. Kersting, S.H. Muggleton (Eds.), Probabilistic Inductive Logic Programming, Springer, 2008.
[4] L.D. Raedt, K. Kersting, Statistical relational learning, in: G.W.C. Sammut (Ed.), Encyclopedia of Machine Learning, Springer, Heidelberg, 2010, pp. 916–924.
[5] N. Nilsson, Probabilistic logic, Artif. Intell. 28 (1) (1986) 71–87.
[6] H. Geffner, Artificial intelligence: from programs to solvers, AI Commun. 27 (1) (2014) 45–51.
[7] A. Rush, M. Collins, A tutorial on dual decomposition and Lagrangian relaxation for inference in natural language processing, J. Artif. Intell. Res. 45 (2012) 305–362.
[8] S. Sra, S. Nowozin, S. Wright (Eds.), Optimization for Machine Learning, MIT Press, 2011.
[9] T. Guns, S. Nijssen, L. De Raedt, Itemset mining: a constraint programming perspective, Artif. Intell. 175 (12–13) (2011) 1951–1983.
[10] A. Atserias, E. Maneva, Sherali–Adams relaxations and indistinguishability in counting logics, SIAM J. Comput. 42 (1) (2013) 112–137.
[11] M. Littman, T. Dean, L. Pack Kaelbling, On the complexity of solving Markov decision problems, in: Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence, UAI-95, 1995, pp. 394–402.

[12] M. Richardson, P. Domingos, Markov logic networks, Mach. Learn. 62 (1–2) (2006) 107–136.
[13] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, T. Eliassi-Rad, Collective classification in network data, AI Mag. 29 (3) (2008) 93–106.
[14] M. Mladenov, B. Ahmadi, K. Kersting, Lifted linear programming, in: Proceedings of the 15th International Conference on Artificial Intelligence and Statistics, AISTATS, in: J. Mach. Learn. Res. Workshop Conf. Proc., vol. 22, 2012, pp. 788–797.
[15] G. Dantzig, M. Thapa, Linear Programming 2: Theory and Extensions, Springer, 2003.
[16] W. Zhou, L. Zhang, L. Jiao, Linear programming support vector machines, Pattern Recognit. 35 (12) (2002) 2927–2936.
[17] A. Demiriz, K.P. Bennett, J. Shawe-Taylor, Linear programming boosting via column generation, Mach. Learn. 46 (1–3) (2002) 225–254.
[18] K. Ataman, W. Street, Y. Zhang, Learning to rank by maximizing auc with linear programming, in: Proceedings of the International Joint Conference on Neural Networks, IJCNN, 2006, pp. 123–129.
[19] Z. Wang, J. Shawe-Taylor, Large-margin structured prediction via linear programming, in: Proceedings of the 12th International Conference on Artificial Intelligence and Statistics, AISTATS, 2009, pp. 599–606.
[20] T. Klein, U. Brefeld, T. Scheffer, Exact and approximate inference for annotating graphs with structural SVMs, in: Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, Part 1, ECML PKDD-08, 2008, pp. 611–623.
[21] M. Torkamani, D. Lowd, Convex adversarial collective classification, in: Proceedings of the 30th International Conference on Machine Learning, ICML, 2013, pp. 642–650.
[22] M.J. Wainwright, M.I. Jordan, Graphical models, exponential families, and variational inference, Found. Trends Mach. Learn. 1 (1–2) (2008) 1–305.
[23] U. Syed, M. Bowling, R.E. Schapire, Apprenticeship learning using linear programming, in: Proceedings of the 25th International Conference on Machine Learning, ICML, ACM, 2008, pp. 1032–1039.
[24] S. Sanner, C. Boutilier, Practical solution techniques for first-order MDPs, Artif. Intell. 173 (5–6) (2009) 748–788.
[25] A.Y. Ng, S.J. Russell, Algorithms for inverse reinforcement learning, in: Proceedings of the 17th International Conference on Machine Learning, ICML, 2000, pp. 663–670.
[26] N. Komodakis, N. Paragios, G. Tziritas, Clustering via LP-based stabilities, in: Proceedings of the 21st Conference on Neural Information Processing Systems, NIPS, 2008, pp. 865–872.
[27] M. Sandler, On the use of linear programming for unsupervised text classification, in: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, ACM, 2005, pp. 256–264.
[28] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice Hall, 1993.
[29] R. Fourer, D.M. Gay, B.W. Kernighan, AMPL: A Mathematical Programming Language, The Scientific Press, San Francisco, CA, 1993.
[30] R. Fourer, D. Gay, B. Kernighan, AMPL: A Modeling Language for Mathematical Programming, second edition, Duxbury Press/Brooks/Cole Publishing Company, 2002, http://ampl.com/resources/the-ampl-book/.
[31] J. Lloyd, Foundations of Logic Programming, Springer-Verlag, Berlin, 1987.
[32] P. Flach, Simply Logical – Intelligent Reasoning by Example, Wiley Professional Computing, Wiley, 1994.
[33] F. Niu, C. Ré, A. Doan, J. Shavlik, Tuffy: scaling up statistical inference in Markov logic networks using an RDBMS, Proc. VLDB Endow. 4 (6) (2011) 373–384.
[34] P. Singla, P. Domingos, Lifted first-order belief propagation, in: Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI, Chicago, IL, USA, 2008, pp. 1094–1099.
[35] K. Kersting, B. Ahmadi, S. Natarajan, Counting belief propagation, in: Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence, UAI, 2009.
[36] B. Ahmadi, K. Kersting, M. Mladenov, S. Natarajan, Exploiting symmetries for scaling loopy belief propagation and relational training, Mach. Learn. 92 (2013) 91–132.
[37] C. Godsil, G. Royle, Algebraic Graph Theory, Springer, 2001.
[38] M. Ramana, E. Scheinerman, D. Ullman, Fractional isomorphism of graphs, Discrete Math. 132 (1994) 247–265.
[39] M. Grohe, K. Kersting, M. Mladenov, E. Selman, Dimension reduction via colour refinement, in: Proceedings of the 22th Annual European Symposium, ESA-14, 2014, pp. 505–516.
[40] C. Godsil, Compact graphs and equitable partitions, Linear Algebra Appl. 255 (1997) 259–266.
[41] C. Berkholz, P. Bonsma, M. Grohe, Tight lower and upper bounds for the complexity of canonical colour refinement, in: Proceedings of the 21st Annual European Symposium on Algorithms, ESA, 2013, pp. 145–156.
[42] U. Apsel, K. Kersting, M. Mladenov, Lifting relational MAP-LPs using cluster signatures, in: Proceedings of the 28th AAAI Conference on Artificial Intelligence, AAAI, 2014.
[43] H. Bui, T. Huynh, S. Riedel, Automorphism groups of graphical models and lifted variational inference, in: Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence, UAI, 2013.
[44] R. Bödi, K. Herr, M. Joswig, Algorithms for highly symmetric linear and integer programs, Math. Program., Ser. A 137 (1–2) (2013) 65–90.
[45] M. Littman, T. Dean, L.P. Kaelbling, On the complexity of solving Markov decision problems, in: Proceedings of the 11th International Conference on Uncertainty in Artificial Intelligence, UAI, 1995, pp. 394–402.
[46] R. Sutton, A. Barto, Reinforcement Learning: An Introduction, The MIT Press, 1998.
[47] S. Narayanamurthy, B. Ravindran, On the hardness of finding symmetries in Markov decision processes, in: Proceedings of the 25th International Conference on Machine Learning, ICML, 2008, pp. 688–695.
[48] B. Ravindran, A. Barto, Symmetries and model minimization in Markov decision processes, Tech. Rep. 01-43, University of Massachusetts, Amherst, MA, USA, 2001.
[49] T. Dean, R. Givan, Model minimization in Markov decision processes, in: Proceedings of the 14th National Conference on Artificial Intelligence, AAAI, 1997, pp. 106–111.
[50] S. Chakrabarti, B. Dom, P. Indyk, Enhanced hypertext categorization using hyperlinks, in: Proceedings of ACM SIGMOD International Conference on Management of Data, SIGMOD, 1998, pp. 307–318.
[51] J. Neville, D. Jensen, Iterative classification in relational data, in: Proceedings of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data, 2000, pp. 13–20.
[52] J. Neville, D. Jensen, Collective classification with relational dependency networks, in: Proceedings of the 2nd International Workshop on Multi-Relational Data Mining, 2003, pp. 77–91.
[53] J. Neville, D. Jensen, Relational dependency networks, J. Mach. Learn. Res. 8 (2007) 653–692.
[54] V.N. Vapnik, Statistical Learning Theory, Adaptive and Learning Systems for Signal Processing, Communications and Control Series, John Wiley & Sons, New York, 1998, A Wiley-Interscience Publication.
[55] P. Sen, G.M. Namata, M. Bilgic, L. Getoor, B. Gallagher, T. Eliassi-Rad, Collective classification in network data, AI Mag. 29 (3) (2008) 93–106.
[56] S.A. Macskassy, F. Provost, Classification in networked data: a toolkit and a univariate case study, J. Mach. Learn. Res. 8 (2007) 935–983.
[57] Q. Lu, L. Getoor, Link-based classification, in: Proceedings of the 20th International Conference on Machine Learning, ICML, 2003, pp. 496–503.
[58] A. Brooke, D. Kendrick, A. Meeraus, GAMS: A User's Guide, The Scientific Press, Redwood City, CA, 1992.
[59] J. Bisschop, P. Lindberg, AIMMS the Modeling System, Paragon Decision Technology, 1993.

[60] T. Ciriani, Y. Colombani, S. Heipcke, Embedding optimisation algorithms with mosel, 4OR 1 (2) (2003) 155–167.
[61] C. Kuip, Algebraic languages for mathematical programming, Eur. J. Oper. Res. 67 (1993) 25–51.
[62] E. Fragniere, J. Gondzio, Optimization modeling languages, in: P. Pardalos, M. Resende (Eds.), Handbook of Applied Optimization, Oxford University Press, New York, 2002, pp. 993–1007.
[63] S. Wallace, W. Ziemba (Eds.), Applications of Stochastic Programming, SIAM, Philadelphia, 2005.
[64] G. Mitra, C. Luca, S. Moody, B. Kristjanssonl, Sets and indices in linear programming modelling and their integration with relational data models, Comput. Optim. Appl. 4 (1995) 263–283.
[65] A. Atamtürk, E. Johnson, J. Linderoth, M. Savelsbergh, A relational modeling system for linear and integer programming, Oper. Res. 48 (6) (2000) 846–857.
[66] R. Farrell, T. Maness, A relational database approach to a linear programming-based decision support system for production planning in secondary wood product manufacturing, Decis. Support Syst. 40 (2) (2005) 183–196.
[67] W. Yih, D. Roth, Global inference for entity and relation identification via a linear programming formulation, in: L. Getoor, B. Taskar (Eds.), An Introduction to Statistical Relational Learning, MIT Press, 2007.
[68] S. Riedel, J. Clarke, Incremental integer linear programming for non-projective dependency parsing, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP, 2006, pp. 129–137.
[69] J. Clarke, M. Lapata, Global inference for sentence compression: an integer linear programming approach, J. Artif. Intell. Res. 31 (2008) 399–429.
[70] A. Martins, N. Smith, E. Xing, Concise integer linear programming formulations for dependency parsing, in: Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics, ACL, 2009, pp. 342–350.
[71] S. Riedel, D. Smith, A. McCallum, Parse, price and cut—delayed column and row generation for graph based parsers, in: Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL, 2012, pp. 732–743.
[72] X. Cheng, D. Roth, Relational inference for wikification, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP, 2013, pp. 1787–1796.
[73] D. Kabjan, R. Fourer, J. Ma, Algebraic modeling in a deductive database language, http://dynresmanagement.com/uploads/3/3/2/9/3329212/datalog_modeling.pdf, 2009, presented at the 11th INFORMS Computing Society (ICS) Conference 2009, but not published.
[74] J. Eisner, N. Filardo, Dyna: extending datalog for modern AI, in: O. de Moor, G. Gottlob, T. Furche, A. Sellers (Eds.), Datalog Reloaded – 1st International Workshop, Datalog 2010, Oxford, UK, March 16–19, 2010, in: Lect. Notes Comput. Sci., vol. 6702, Springer, 2011, pp. 181–220, Revised Selected Papers.
[75] J. Mattingley, S. Boyd, CVXGEN: a code generator for embedded convex optimization, Optim. Eng. 12 (1) (2012) 1–27.
[76] W. Hart, J.-P. Watson, D. Woodruff, Pyomo: modeling and solving mathematical programs in Python, Math. Program. Comput. 3 (2011) 219–260.
[77] S. Diamond, E. Chu, S. Boyd, CVXPY: a Python-embedded modeling language for convex optimization, version 0.2, http://cvxpy.org/, May 2014.
[78] N. Rizzolo, D. Roth, Modeling discriminative global inference, in: Proceedings of the IEEE International Conference on Semantic Computing, ICSC, 2007, pp. 597–604.
[79] G. Gordon, S. Hong, M. Dudík, First-order mixed integer linear programming, in: Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence, UAI, 2009, pp. 213–222.
[80] E. Zawadzki, G. Gordon, A. Platzer, An instantiation-based theorem prover for first-order programming, in: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, AISTATS, vol. 15, 2011, pp. 855–863.
[81] F. Margot, Symmetry in integer linear programming, in: M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, L. Wolsey (Eds.), 50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art, Springer, 2010, pp. 1–40.
[82] T. Achterberg, R. Wunderling, Mixed integer programming: analysing 12 years of progress, in: M. Jünger, G. Reinelt (Eds.), Facets of Combinatorial Optimization: Festschrift for Martin Grötschel, Springer, 2002, pp. 449–481.
[83] Gurobi Optimization, Inc., Gurobi optimizer reference manual, http://www.gurobi.com, 2014.
[84] M. Sellmann, P. Van Hentenryck, Structural symmetry breaking, in: Proceedings of 19th International Joint Conference on Artificial Intelligence, IJCAI, 2005, pp. 298–303.
[85] R. Bödi, T. Grundhöfer, K. Herr, Symmetries of linear programs, Note Mat. 30 (1) (2010) 129–132.
[86] T. Berthold, M. Pfetsch, Detecting orbitopal symmetries, 2009.
[87] J. Noessner, M. Niepert, H. Stuckenschmidt Rockit, Exploiting parallelism and symmetry for map inference in statistical relational models, in: Proceedings of the 27th AAAI Conference on Artificial Intelligence, AAAI, 2013.
[88] M. Mladenov, A. Globerson, K. Kersting, Efficient lifting of MAP LP relaxations using *k*-locality, in: Proceedings of the 17th Int. Conf. on Artificial Intelligence and Statistics, AISTATS, in: J. Mach. Learn. Res. Workshop Conf. Proc., vol. 33, 2014.
[89] M. Mladenov, A. Globerson, K. Kersting, Lifted message passing as reparametrization of graphical models, in: Proceedings of the 30th Int. Conf. on Uncertainty in Artificial Intelligence, UAI, 2014.
[90] T. Lu, C. Boutilier, Value-directed compression of large-scale assignment problems, in: Proceedings of the 29th AAAI Conference on Artificial Intelligence, AAAI, 2015.
[91] J. Bi, K. Bennett, M. Embrechts, C. Breneman, M. Song, Dimensionality reduction via sparse support vector machines, J. Mach. Learn. Res. 3 (2003) 1229–1243.
[92] A. Demiriz, K. Bennett, J. Shawe-Taylor, Linear programming boosting via column generation, Mach. Learn. 46 (1–3) (2002) 225–254.
[93] V. Conitzer, T. Sandholm, Computing the optimal strategy to commit to, in: Proceedings 7th ACM Conference on Electronic Commerce, EC, 2006, pp. 82–90.
[94] V. Saraswat, V. Gupta, R. Jagadeesan, P. Panangaden, D. Precup, F. Rossi, P. Sen, Probabilistic constraint programming, in: Working Notes of the 2014 NIPS Workshop on Probabilistic Programming, 2014.
[95] S. Bach, M. Broecheler, L. Getoor, D. O'leary, Scaling MPE inference for constrained continuous Markov random fields with consensus optimization, in: Proceedings of the International Conference on Neural Information Processing Systems, NIPS, 2012, pp. 2654–2662.
[96] A. Airola, S. Pyysalo, J. Björne, T. Pahikkala, F. Ginter, T. Salakoski, All-paths graph kernel for protein–protein interaction extraction with evaluation of cross-corpus learning, BMC Bioinform. 9 (Suppl. 11) (2008) S2.
[97] J. Suzuki, T. Hirao, Y. Sasaki, E. Maeda, Hierarchical directed acyclic graph kernel: methods for structured natural language data, in: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics, ACL, 2003, pp. 32–39.
[98] D. Haussler, Convolution kernels on discrete structures, Tech. rep., UCSC-CRL-99-10, UC Santa Cruz, 1999.
[99] T. Gärtner, Exponential and geometric kernels for graphs, in: Working Notes of the NIPS Workshop on Unreal Data: Principles of Modeling Nonvectorial Data, 2002, pp. 49–58.
[100] T. Horváth, T. Gärtner, S. Wrobel, Cyclic pattern kernels for predictive graph mining, in: Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, 2004, pp. 158–167.
[101] K.M. Borgwardt, H.-P. Kriegel, Shortest-path kernels on graphs, in: Proceedings of the 5th IEEE International Conference on Data Mining, ICDM, 2005.