International Journal of Approximate Reasoning 106 (2019) 32–50

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar

Speeding up parameter and rule learning for acyclic probabilistic logic programs ☆

Francisco H.O. Vieira de Faria [a], Arthur Colombini Gusmão [a], Glauber De Bona [a], Denis Deratani Mauá [b], Fabio Gagliardi Cozman [a,*]

[a] Escola Politécnica, Universidade de São Paulo, Brazil
[b] Instituto de Matemática e Estatística, Universidade de São Paulo, Brazil

ARTICLE INFO

Article history:
Received 19 March 2018
Received in revised form 23 October 2018
Accepted 14 December 2018
Available online 21 December 2018

Keywords:
Probabilistic logic programming
Expectation-Maximization algorithm
Rule learning

ABSTRACT

This paper introduces techniques that speed-up parameter and rule learning for acyclic probabilistic logic programs. We focus on maximum likelihood estimation of parameters, and show that significant improvements can be obtained by efficiently handling probabilistic rules. We then move to structure learning, where we learn sets of rules, by introducing an algorithm that greatly simplifies exact score-based learning. Experiments demonstrate that our methods can produce orders of magnitude speed-ups over the state-of-art in parameter and rule learning.

1. Introduction

There is a myriad of ways to combine logic and probability, so as to reason about relational structures coupled with uncertainty. A few formalisms that pursue this combination are relational Bayesian networks [1] and probabilistic relational models [2], Markov logic networks [3], probabilistic logic programming [13] and other probabilistic programming languages [5], as well as a variety of probabilistic logics [6,7]. In this paper we focus on probabilistic logic programming; that is, on languages that add probabilities to logic programming.

One can find various proposals on how to introduce probabilities in logic programs [2,4,8–14]. A particularly popular semantics for probabilistic logic programming is due to Sato, and usually referred to as the *distribution semantics* [12,15]. The idea is that we have rules, as in logic programming, such as

$$\mathsf{cares}(X, Y) :\!-\ \mathsf{person}(X), \mathsf{person}(Y), \mathsf{neighbor}(X, Y).$$

and probabilistic facts such as

$$0.8 :: \mathsf{neighbor}(X, Y)., \tag{1}$$

meaning that the probability that any X and Y are neighbors is 0.8. Probabilistic facts are assumed independent, and induce a probability measure over logic programs, consequently inducing a probability measure over interpretations of atoms.

☆ This paper is part of the Virtual special issue on 4th Workshop on Probabilistic Logic Programming, Edited by Christian Theil Have and Riccardo Zese.

* Corresponding author.
E-mail address: fgcozman@usp.br (F.G. Cozman).

https://doi.org/10.1016/j.ijar.2018.12.012
0888-613X/

Expression (1) is written in the syntax of ProbLog, a popular and freely available package for probabilistic logic programming that follows Sato's distribution semantics [16]. ProbLog also allows for "probabilistic rules"; for instance, one can state that person(X) and person(Y) yield a proof for cares(X, Y) with probability 0.8, by writing:

$$0.8 :: \mathsf{cares}(X, Y) :- \mathsf{person}(X), \mathsf{person}(Y).$$

ProbLog implements state-of-art inference and learning algorithms for probabilistic logic programs. A closely related package is ProbFOIL, that implements rule learning (restricted to a single head) in the context of Sato's distribution semantics [17]. Both packages resort to some of the best techniques available in the literature; however, they require significant computational power, even on datasets that are small compared to currently available data resources.

If we restrict ourselves to acyclic probabilistic logic programs, we focus on a language that is closely related to Bayesian networks, with the added benefit that it can handle relational modeling, and the potential benefit that it may require less parameters to encode some probability distributions — thus offering a language that may be more adept at avoiding overfitting. One advantage of acyclic programs is that they can be learned via well-tested techniques usually employed to learn Bayesian networks. Alas, we see in practice that much more computational effort is needed to learn an acyclic probabilistic logic program than an apparently equivalent Bayesian network. For instance, we find that even complete databases are usually processed by the Expectation-Maximization algorithm (EM), due to the insertion of latent random variables associated with probabilistic rules. Thus the added expressivity of probabilistic logic programs seems to impose a hefty computational cost.

The goal of this paper is to show that we can greatly simplify existing algorithms for parameter and rule learning in acyclic logic programs, often obtaining orders of magnitude speed-ups. We introduce a variety of techniques that move us toward the boundary of what can be done with reasonable computational resources when learning a probabilistic logic program. We show empirically that substantial gains can be attained, sometimes even by application of relatively simple ideas.[1]

We start by investigating parameter learning; that is, we assume that a set of facts and rules is given, and we learn probabilities attached to them. We focus on maximum likelihood, and we show that EM-style iterations can be often avoided; in particular they can be avoided altogether when the input data has no missing values. We then examine rule learning, emphasizing score-based methods (in particular, minimum description length). We focus on exact methods; that is, we want to find a probabilistic logic program that does maximize the score of interest. We show that many operations that are needed in this optimization problem can be solved in closed-form, while other operations can be just discarded. We specialize our contributions to the case where each rule is restricted to have at most two literals in its body; this is admitedly a restricted class of programs, but a class that can encode all "noisy" Boolean circuits, serving well as a starting point for more ambitious future work.

Section 2 reviews some relevant terminology and notation. In Section 3 we review the algorithms in the ProbLog package, as they capture the state-of-art in the topic. We present novel techniques for parameter learning in Section 4, and then move to rule learning in Section 5. Experiments are described in Section 6, together with analysis that reveals the benefits of our techniques.

2. Background

Most probabilistic logic programs (PLPs) can be viewed as logic programs containing facts annotated with probabilities. We follow ProbLog's syntax in our presentation [16,21]. ProbLog's semantics is directly based on Sato's distribution semantics, and it can be used to convey the main semantic conventions adopted by various languages and packages.

2.1. Syntax

Consider a vocabulary with logical variables $X, Y, \ldots$, predicate symbols $r, s, \ldots$ and constants $a, b, \ldots$. Each predicate symbol has an associated arity. An *atom* is an expression of the form $r(t_1, t_2, \ldots, t_m)$ where r is a predicate symbol with arity m, and each t_i is a *term*, which is either a constant or a logical variable. An atom is *ground* if it has no logical variables. An atomic proposition is a 0-arity predicate symbol. Clearly a proposition is also a ground atom. A *literal* is an atom, say A, possibly preceded by **not** (that is, **not** A). A *(deterministic) rule* is an expression of the form

$$H :- B_1, \ldots, B_n.,$$

where the *head* H is an atom and each *subgoal* B_i is a literal, with $B_1, \ldots, B_n$ being the rule's *body*. A fact is a rule with empty body; instead of writing $H :-$ **.**, we simply write H**.** If an atom is in the head of some rule, it is said to be a *derived* atom. A set of rules is a *normal logic program*.

[1] This paper collects material from three previous publications [18–20], with additional experimental validation.

A *grounding* is a function taking logical variables and returning constants. The *grounding* of a rule is a *ground rule* obtained by applying the same grounding to each atom. The grounding of a program is the propositional program obtained by applying every possible grounding to each rule and fact, using only the constants in the program.

For a given logic program, the *dependency graph* contains its ground atoms as nodes and an arc from B to H if there is a rule grounding with B in the body, possibly negated, and H in the head. A logic program is *acyclic* if its dependency graph is acyclic.

A ProbLog program consists of a normal logic program together with *probabilistic facts*. Each probabilistic fact is written as

$$\theta :: F.,$$

where $\theta \in [0, 1]$ is a real number and F is an atom. Similarly to rules, a probabilistic fact can be grounded to form a set of ground probabilistic facts.

A pair $\langle \mathbf{P}, \mathbf{PF} \rangle$, where $\mathbf{P}$ is a normal logic program and $\mathbf{PF}$ is a set of probabilistic facts, is a probabilistic logic program. We always assume that probabilistic atoms are not derived in $\mathbf{P}$; we also assume that a probabilistic fact does not unify with another probabilistic fact.

The class of normal logic programs where each rule has at most k atoms in the body is often denoted $\text{LP}(k)$ [22]; we analogously write $\text{PLP}(k)$ to denote the class of PLPs where each rule has at most k literals in the body.

2.2. Semantics

The semantics of a probabilistic logic program is defined through the semantics of its grounding, so we can focus on the semantics of propositional programs. ProbLog's semantics is inspired by the standard semantics of Prolog, appropriately modified to take into account the probabilistic framework.

First, it is convenient to view a ground atom A in a program as a random variable taking values in $\{0, 1\}$ (respectively standing for false and true). We write $P \models \{A = 1\}$ (resp., $P \models \{A = 0\}$) iff A (resp., **not** A) is entailed by the logic program P; and similarly $P \models \{Q = q\}$ when Q is a set of atoms.

Let $T = \langle \mathbf{P}, \mathbf{PF} \rangle$ be a probabilistic program, and let $\{\theta_i :: F_i | 1 \leq i \leq n\}$ be the grounding of $\mathbf{PF}$, for $\theta_1, \ldots, \theta_n \in [0, 1]$ and a set of ground facts $\mathbf{F} = \{F_1, \ldots, F_n\}$. The semantics of T is given by a probability distribution over normal logic programs, such that

$$\mathbb{P}_{\mathbf{PF}}(\mathbf{P} \cup \mathbf{F}') = \prod_{F_i \in \mathbf{F}'} \theta_i \prod_{F_i \in \mathbf{F} \setminus \mathbf{F}'} (1 - \theta_i),$$

where $\mathbf{F}'$ (referred to as a *total choice*) is any subset of $\mathbf{F}$. Then we define the probability that a given set of ground atoms $Q = \{Q_1, \ldots, Q_n\}$ has respectively truth values $q = \langle q_1, \ldots, q_n \rangle \in \{0, 1\}^n$ as follows:

$$\mathbb{P}_T(Q = q) = \sum_{\mathbf{P} \cup \mathbf{F}' \models \{Q = q\}} \mathbb{P}_{\mathbf{PF}}(\mathbf{P} \cup \mathbf{F}').$$

Given some observations $\{E = e\}$, where E is a set of ground atoms and e is a set of observed truth values, the conditional probability $\mathbb{P}_T(Q = q | E = e)$ is, as usual, $\mathbb{P}_T(Q = q, E = e)/\mathbb{P}_T(E = e)$, provided $\{E = e\}$ has positive probability. If Q is a set of random variables, $\mathbb{P}_T(Q)$ denotes the corresponding probability distribution.

Example 1. *Consider the following probabilistic logic program T:*

```
0.2 :: burglary.
0.3 :: fire.
alarm :- burglary.
alarm :- fire.
```

Here we have $\mathbf{P} = \{\text{alarm} :- \text{burglary}., \text{alarm} :- \text{fire}.\}$ *and four total choices* $\mathbf{F}' \subseteq \{\text{burglary}, \text{fire}\}$. *Suppose we want to compute* $\mathbb{P}_T(\text{alarm} = 1)$. *We must consider* $\mathbb{P}_{\mathbf{PF}}(\mathbf{P} \cup \mathbf{F}')$ *for any* $\mathbf{F}'$ *such that* $\mathbf{P} \cup \mathbf{F}' \models \{\text{alarm} = 1\}$ *— which are the non-empty* $\mathbf{F}'$ *in this case. We obtain* $\mathbb{P}_T(\text{alarm} = 1) = 0.2 \times 0.3 + 0.2 \times 0.7 + 0.8 \times 0.3 = 0.44$. □

2.3. Probabilistic rules

A probabilistic rule is obtained by annotating a deterministic rule with a number $p \in [0, 1]$, as follows:

$$p :: H :- B_1, \ldots, B_n.,$$

where H and B_i are as before. We now extend the previous probabilistic logic programs to include probabilistic rules. That is, now a probabilistic logic program is a pair $T = \langle \mathbf{P}, \mathbf{PR} \rangle$, where $\mathbf{P}$ is a normal logic program and $\mathbf{PR}$ is a set of probabilistic rules, some of which may be probabilistic facts.

Grounding the probabilistic rules, one obtains a set $\{\theta_i :: R_i | 1 \leq i \leq n\}$, where $\mathbf{R} = \{R_1, \ldots, R_n\}$ is a set of ground (deterministic) rules. Then the probability of a total choice $\mathbf{R}' \subseteq \mathbf{R}$ entails the probability of a logic program

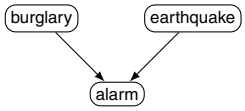
$$\mathbb{P}_{\mathbf{PR}}(\mathbf{P} \cup \mathbf{R}') = \prod_{R_i \in \mathbf{R}'} \theta_i \prod_{R_i \in \mathbf{R} \setminus \mathbf{R}'} (1 - \theta_i).$$

For a set of ground atoms Q and corresponding truth values q, we have:

$$\mathbb{P}_T(Q = q) = \sum_{\mathbf{P} \cup \mathbf{R}' \models \{Q = q\}} \mathbb{P}_{\mathbf{PR}}(\mathbf{P} \cup \mathbf{R}'). \tag{2}$$

Example 2. *Here is a propositional* PLP *with probabilistic rules, with the corresponding dependency graph presented on the right:*

0.1 :: burglary.
0.2 :: earthquake**.**
0.9 :: alarm :— burglary, earthquake**.**
0.8 :: alarm :— burglary, **not** earthquake**.**
0.7 :: alarm :— **not** burglary, earthquake**.**
0.1 :: alarm :— **not** burglary, **not** earthquake**.**



This is clearly an acyclic propositional PLP. *We have* $\mathbb{P}(\text{alarm}) = 0.1 \times 0.2 \times 0.9 + 0.1 \times 0.8 \times 0.8 + 0.9 \times 0.2 \times 0.7 + 0.9 \times 0.8 \times 0.1 = 0.28$. □

A probabilistic rule $p :: H :- B_1, \ldots, B_n$**.** is equivalent to a pair formed by a deterministic rule $H :- B_1, \ldots, B_n, A$**.** and a probabilistic fact $p :: A$**.**, where A is an auxiliary atom that does not appear anywhere else, with the same logical variables as H [16]. One might look at probabilistic rules as syntactic sugar, even though, as we see later, there are benefits in avoiding these auxiliary probabilistic facts.

2.4. Bayesian networks and acyclic probabilistic logic programs

There is a close relation between acyclic propositional PLPs and Bayesian networks, as we explore in our contributed methods. Recall that a Bayesian network consists of a directed acyclic graph where each node is a random variable, and a probability joint distribution over the same random variables, such that the distribution satisfies the following Markov condition: a variable X is independent of its nondescendants nonparents given its parents [23]. The parents of a variable X, denoted by $\text{PA}[X]$, are the nodes/variables that point to X. For any Bayesian network, its directed acyclic graph is referred to as its "structure".

In this paper every random variable has finitely many values (indeed all of them are binary). When a conditional probability distribution over random variables with finitely many values is encoded using a table, this *conditional probability table* is often referred to as a *CPT*. Typically Bayesian networks are specified by CPTs. Each CPT contains the values of $\mathbb{P}(X_i = x_{ij} | \text{PA}[X_i] = \pi_{ik})$ for each variable X_i, each value x_{ij}, and each configuration π_{ik} of the parents of X_i.

Any Bayesian network over binary variables can be specified using acyclic propositional PLPs [11,24]. For instance, Example 2 specifies a Bayesian network using probabilistic facts and rules, in essence enumerating all entries of relevant CPTs. Acyclic propositional PLPs can offer a much more concise specification than a CPT-based Bayesian network, as rules can encode deterministic effects. For instance, here is a compact specification for a NoisyOr gate [23] that could be used in a version of Example 2:

0.3 :: alarm :— burglary**.**
0.6 :: alarm :— earthquake**.**

Conversely, any acyclic propositional PLP can be interpreted as a Bayesian network. This should be clear from Example 2: the Bayesian network described by the PLP has the structure given by the dependency graph, and the parameters of the network are just the probabilities associated with probabilistic facts and rules, plus some zeros and ones encoding the minimal model semantics of normal logic programming.

3. Parameter learning in ProbLog

The *structure* of a probabilistic logic program $T = \langle \mathbf{P}, \mathbf{PR} \rangle$ is the normal logic program $\mathbf{P}$ and the deterministic rules (and facts) $R_1, \ldots, R_n$ that are annotated with probabilities (without the probabilities themselves). The set of parameters that are associated with rules $R_1, \ldots, R_n$ are denoted respectively by $\theta_1, \ldots, \theta_n$; that is, the probabilistic rules are $\{\theta_1 :: R_1, \ldots, \theta_n :: R_n\}$. Denote by Θ the tuple containing all parameters needed to specify T. Also, denote by T_Θ the result of annotating the structure of T with parameters $\theta_1, \ldots, \theta_n$.

Denote by $\mathbf{At}(T)$ the set of all ground atoms that can be generated by grounding the structure of T. An *observation* $\{E = e\}$ consists of a set of ground atoms $E \subseteq \mathbf{At}(T)$ together with their corresponding truth values $e \in \{0,1\}^{|E|}$. A *dataset* is a set of observations that are assumed independent; we implicitly assume that the dataset has positive probability for some selection of rules and parameters.

Now if we have a structure and a dataset, we may wish to estimate the needed parameters Θ; this is the problem of *parameter learning* as addressed by ProbLog [16]. Following ProbLog, we are interested in maximum likelihood estimates; that is, we have:

- **Input:** (i) a structure consisting of a normal logic program **P** and a set of rules/facts $\{R_1, \ldots, R_n\}$ to be annotated with corresponding parameters $\Theta = \langle \theta_1, \ldots, \theta_n \rangle$, and (ii) a dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$;
- **Output:** the maximum likelihood estimates $\hat{\Theta} = \langle \hat{\theta}_1, \ldots, \hat{\theta}_n \rangle$:

$$\hat{\Theta} = \arg\max_{\Theta} \mathbb{P}_{T_\Theta}(D) = \arg\max_{\Theta} \prod_{i=1}^{m} \mathbb{P}_{T_\Theta}(E_i = e_i)\,.$$

Each observation $\{E_j = e_j\}$ in the input dataset is a *training example*. When an observation $\{E = e\}$ is such that $E = \mathbf{At}(T_\Theta)$, we say it is a *complete observation*. A dataset D is complete if every one of its observations is complete.

Suppose first that we have a structure with a normal logic program **P** and a set of facts **PF**, $F_1, \ldots, F_n$, to be annotated with corresponding parameters in Θ, and we additionally have a complete dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$. Let $\{F_{ij} \mid 1 \leq j \leq n_i\}$ be the set of possible groundings of F_i, for each $1 \leq i \leq n$. As no probabilistic fact unifies with the head of a rule or with another probabilistic fact, the probability of F_{ij} being true is exactly the probability associated with the probabilistic fact $\theta_i :: F_i\mathbf{.}$ in **PF**:

$$\mathbb{P}_T(F_{ij} = 1) = \theta_i.$$

Then the maximum-likelihood parameters $\hat{\theta} = \langle \hat{\theta}_1, \ldots, \hat{\theta}_n \rangle$ are given by

$$\hat{\theta}_i = \frac{1}{mn_i} \sum_{k=1}^{m} \sum_{j=1}^{n_i} \delta_{i,j}^k, \text{ where } \delta_{i,j}^k = \begin{cases} 1 & \text{if } F_{ij} \in E_k; \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The normalization factor mn_i is the number of groundings of the probabilistic atom F_i observed through the whole dataset D. In the propositional case, $n_i = 1$ for every $1 \leq i \leq n$. For a relational program we often have $m = 1$ and $n_i \gg 1$, the latter guaranteeing many observable groundings for any probabilistic fact $\theta_i :: F_i$.

Closed-form solutions for maximum likelihood estimates cannot be found in general when some atoms are not observed in the training dataset. The Expectation-Maximization algorithm (EM) [25] has been the main tool to learn parameters with missing data. The idea behind EM is to iterate two steps: in the E-step, the algorithm uses parameters θ_i^t to compute the probability of each missing ground fact given observations; in the M-step, the algorithm maximizes the expected log-likelihood using the probabilities from the E-step, thus arriving at parameters θ_i^{t+1}.

ProbLog's parameter learning implementation, called LFI-ProbLog, takes as input a normal logic program **P**, a set of facts $\{F_1, \ldots, F_n\}$ and a dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$ that may not be complete. As before, F_{ij} is a grounding of fact F_i, and $T_\Theta = \langle \mathbf{P}, \{\theta_1 :: F_1, \ldots, \theta_n :: F_n\} \rangle$ is the probabilistic program produced by the parameters $\langle \theta_1, \ldots, \theta_n \rangle$. To begin, LFI-ProbLog sets each θ_i^0 randomly, and iterates as follows until $\mathbb{P}_{T_\Theta}(D)$ changes less than a threshold:

E-step for $1 \leq i \leq n, 1 \leq j \leq n_i, 1 \leq k \leq m$, set $\delta_{i,j}^k = \mathbb{P}_{T_{\Theta^t}}(F_{ij}{=}1 | E_k{=}e_k)$;

M-step for each $1 \leq i \leq n$, $\theta_i^{t+1} = \frac{1}{mn_i} \sum_{k=1}^{m} \sum_{j=1}^{n_i} \delta_{i,j}^k$.

When F_{ij} is in E_k, $\mathbb{P}_{T_\Theta}(F_{ij} = 1 | E_k = e_k)$ in the E-step is simply 1 or 0, and there is no need for inference. In all other cases ProbLog must compute $\mathbb{P}_{T_\Theta^t}(F_{ij} = 1 | E_k = e_k)$, although some optimizations are possible. For instance, ProbLog can detect when F_{ij} is independent from E_k, thus avoiding various inference computations [16].

Only probabilistic facts have been considered in this section so far. When probabilistic rules are allowed, ProbLog converts each probabilistic rule $\theta :: H :\!- B_1, \ldots, B_n\mathbf{.}$ into a deterministic rule $H :\!- B_1, \ldots, B_n, A\mathbf{.}$ and an auxiliary probabilistic fact $\theta :: A\mathbf{.}$, for some A that does not appear anywhere else. Such a translation is semantically correct (Section 2.3). Note that this translation introduces a latent random variable for each grounding of the auxiliary atom A, hence the input dataset is always incomplete with respect to the model actually handled by LFI-ProbLog. Thus any PLP whose structure contains a probabilistic rule must be learned by EM in ProbLog's approach.

4. Speeding-up parameter learning

Consider first that one wishes to learn the parameters of an acyclic PLP that contains probabilistic rules, using a complete dataset as training data. The structure of the PLP is given, so we must only estimate the parameters.

If we were to learn a Bayesian network with analogous structure, but specified with CPTs, we would need estimates for each probability $\vartheta_{ijk} = \mathbb{P}(X_i = x_{ij} | \text{PA}[X_i] = \pi_{ik})$. Given a complete dataset, the maximum likelihood estimates for such probabilities can be computed in closed-form [26]: $\hat{\vartheta}_{ijk} = N_{ijk}/N_{ik}$, where N_{ijk} is the number of times that $\{X_i = x_{ij}, \text{PA}[X_i] = \pi_{ik}\}$ is observed in the training dataset, and N_{ik} is the number of times that $\{\text{PA}[X_i] = \pi_{ik}\}$ is observed in the training dataset.

Unfortunately, even for an acyclic and propositional PLP we must maximize likelihood functions that may be significantly more complex than the ones that apply to CPT-based Bayesian networks. For instance, suppose our input structure consists of three probabilistic rules:

$$\begin{aligned} &\theta_1 :: A_3. \\ &\theta_2 :: A_3 :- A_1, \textbf{not } A_2. \\ &\theta_3 :: A_3 :- \textbf{not } A_1, \textbf{not } A_2. \end{aligned} \tag{4}$$

in addition to two probabilistic facts:

$$0.3 :: A_1. \qquad 0.5 :: A_2..$$

The three probabilistic rules induce the following CPT:

A_1	A_2	$\mathbb{P}(A_3 = 1 \mid A_1, A_2)$
0	0	$\theta_1 + \theta_3 - \theta_1\theta_3$
0	1	θ_1
1	0	$\theta_1 + \theta_2 - \theta_1\theta_2$
1	1	θ_1

Now suppose we have a complete dataset with N independent observations of A_1, A_2 and A_3, and $N_{a_1a_2a_3}$ denotes the number of times we observe $\{A_1 = a_1, A_2 = a_2, A_3 = a_3\}$. The likelihood is proportional to

$$(\theta_{1;3})^{N_{001}}(1-\theta_{1;3})^{N_{000}}(\theta_{1;2})^{N_{101}}(1-\theta_{1;2})^{N_{100}}\theta_1^{N_{011}+N_{111}}(1-\theta_1)^{N_{010}+N_{110}}, \tag{5}$$

where we adopt, here and in the remainder of the paper,

$$\theta_{i;j} := \theta_i + \theta_j - \theta_i\theta_j.$$

Expression (5) is not decomposable like the likelihood obtained for CPT-based Bayesian networks; one solution is to introduce latent variables and resort to EM, as done for instance by LFI-ProbLog.

However, it is in fact possible to maximize, *in closed-form*, Expression (5). By setting derivatives to zero and solving the resulting equations, we obtain:

$$\hat{\theta}_1 = \frac{K_2}{K_1 + K_2}, \quad \hat{\theta}_2 = \frac{K_1 N_{101} - K_2 N_{100}}{K_1(N_{101} + N_{100})}, \quad \hat{\theta}_3 = \frac{K_1 N_{001} - K_2 N_{000}}{K_1(N_{001} + N_{000})},$$

where $K_1 = N_{010} + N_{110}$ and $K_2 = N_{011} + N_{111}$.

Similarly, if we take the following structure, where a NoisyOr gate is encoded:

$$\begin{aligned} &\theta_1 :: A_1. \\ &\theta_2 :: A_2. \\ &\theta_3 :: A_3 :- A_1. \\ &\theta_4 :: A_3 :- A_2. \end{aligned} \tag{6}$$

and we collect a complete dataset, we obtain maximum likelihood estimates in (somewhat complicated) closed-form, as discussed in Appendix A.

Even though the likelihood of a complete dataset with respect to an acyclic PLP does not decompose parameter-wise in the same way as the likelihood of CPT-based Bayesian networks, the former likelihood does decompose as a product, each factor related to a "family" of atoms. That is, for dataset D, we have $\mathbb{P}_{T_\Theta}(D) = \prod_j \mathbb{P}_{T_\Theta}(d_j)$, where each d_j is a complete observation, and $\mathbb{P}_{T_\Theta}(d_j)$ in turn is the product of "local" probabilities $\mathbb{P}_{T_\Theta}(X_i | \text{PA}[X_i])$ that are induced by rules. Now each probability $\mathbb{P}_{T_\Theta}(X_i | \text{PA}[X_i])$ depends only on the head atom corresponding to X_i, and atoms that appear in the bodies of all rules sharing X_i as head. These latter atoms, denoted by $\text{PA}[X_i]$, are indeed the parents of the head atom in the program's dependency graph; the family of X_i is X_i itself and its parents. Thus we can produce closed-form solutions for maximum likelihood estimates in a piecemeal fashion, examining each family separately: parameters for distinct families can be obtained independently of each other.

The higher the number of probabilistic rules that share a head, the more complex the likelihood with respect to parameters associated with those rules. We cannot expect to find closed-form expressions for estimates as we increase the

size of rules and the number of rules sharing a head; in our implementation (described in Section 6) we have explicitly coded every closed-form solution for two or three rules that share a head, when those rules have two or three atoms in the body. We have run through all possible rule patterns, relying on a symbolic computation system[2] to look for closed-form solutions.

Even though this strategy is necessarily limited as we cannot find closed-form solutions for every possible structure, it is well known that Bayesian networks tend to have a relatively low average number of parents per variable [27], possibly because the elicitation of a densely connected structure is too difficult. It is often the case that simplifying model fragments, such as the NoisyOr gate, are employed in practice, thus emphasizing the importance of relatively simple local patterns. Thus it seems reasonable to spend effort fishing for closed-form solutions that handle relatively small rule patterns.

What can be done to handle rule patterns that defy closed-form maximization? In that case one must resort to numerical optimization; the insertion of "local" latent variables and EM is an option. Another option is to directly maximize the likelihood associated with each rule head that resists closed-form solution, for example by gradient ascent or similar techniques, as we describe later.

Thus, in short, there is no need to introduce latent variables, as done by LFI-ProbLog, when a complete dataset is available. As we show in Section 6, much can be gained by dispensing with EM when the input dataset is complete.

Now suppose that the input dataset D is not complete. We must then compute the likelihood by summing over the missing data, thus obtaining an expression that usually fails to yield closed-form maximizing estimates. In this case we resort to EM, but note that we need not insert latent variables to accommodate probabilistic rules. Suppose we have a probabilistic program $T_\Theta = \langle \mathbf{P}, \mathbf{PR} \rangle$ and an incomplete observation $\{E = e\}$. Let $Z_E = \mathbf{At}(T_\Theta) \setminus E$ be the set of unobserved ground atoms for a given observation $E = e$. For each complete observation $\{E = e, Z_E = z\}$, we can express the likelihood using parameters $\langle \theta_1, \ldots, \theta_n \rangle$ as explained previously in this section. Clearly, a probability distribution over Z_E induces an expected value for the log-likelihood. By starting with $\theta_i^0 = 0.5$ for each parameter θ_i, we can apply the EM algorithm as follows, iterating until some convergence criterion is met:

E-step given a set of parameters Θ^t, for each $\{E_k = e_k\} \in D$, compute $\mathbb{P}_{T_\Theta}(Z_{E_k} = z | E_k = e_k)$ for each $z \in \{0,1\}^{|Z_{E_k}|}$;
M-step find the set of parameters Θ^{t+1} that maximize the expected log-likelihood:

$$\sum_{k=1}^{m} \sum_{z \in \{0,1\}^{|Z_{E_k}|}} \mathbb{P}_{T_{\Theta^t}}\left(Z_{E_k} = z | E_k = e_k\right) \ln\left(\mathbb{P}_{T_{\Theta^{t+1}}}(E_k = e_k, Z_{E_k} = z)\right). \tag{7}$$

To compute each term $\mathbb{P}_{T_{\Theta^t}}(Z_{E_k} = z | E_k = e_k)$, we must run some inference algorithm; in our implementation we simply call ProbLog's inference service whenever necessary. In principle, for each $\{E_k = e_k\}$, each z would yield an inference in the E-step, but we can do much better. Let $\mathcal{F}(A_i)$ denote the family of a ground atom A_i. If $q \in \{0,1\}^{|Q|}$ is a vector of truth values associated to a set Q of ground atoms ($\{Q = q\}$), we use $q_{Q'} \in \{0,1\}^{|Q'|}$ to denote the vector of truth values corresponding to the subset $Q' \subseteq Q$. As $\mathbb{P}_{T_{\Theta^t}}(Z_{E_k} = z | E_k = e_k) = \mathbb{P}_{T_{\Theta^t}}(Z_{E_k} = z, E_k = e_k | E_k = e_k)$ and $\mathbb{P}_{T_{\Theta^t}}(Z_{E_k} = z, E_k = x | E_k = e_k) = 0$ for all $x \neq e_k$, the inner sum in Expression (7) can be rewritten as:

$$\sum_{q \in \{0,1\}^{|\mathbf{At}(T_\Theta)|}} \mathbb{P}_{T_{\Theta^t}}(\mathbf{At}(T_\Theta) = q | E_k = e_k) \ln\left(\mathbb{P}_{T_{\Theta^{t+1}}}(\mathbf{At}(T_\Theta) = q)\right). \tag{8}$$

Let $\mathbf{A}$ denote $\mathbf{At}(T_\Theta)$. The likelihood $\mathbb{P}_{T_{\Theta^{t+1}}}(\mathbf{At}(T_\Theta) = q)$ above can be factored through the families of the ground atoms A_i, as usual:

$$\mathbb{P}_{T_{\Theta^{t+1}}}(\mathbf{A} = q) = \prod_{A_i \in \mathbf{A}} \mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \mathrm{PA}[A_i] = q_{\mathrm{PA}[A_i]}). \tag{9}$$

Now we can rewrite Expression (8) as:

$$\sum_{q \in \{0,1\}^{|\mathbf{A}|}} \mathbb{P}_{T_{\Theta^t}}(\mathbf{A} = q | E_k = e_k) \sum_{A_i \in \mathbf{A}} \ln\left(\mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \mathrm{PA}[A_i] = q_{\mathrm{PA}[A_i]})\right). \tag{10}$$

The expression above is the sum of $2^{|\mathbf{A}|} \times |\mathbf{A}|$ terms, each in the form $\mathbb{P}_{T_{\Theta^t}}(\mathbf{A} = q | E_k = e_k) \ln\big(\mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \mathrm{PA}[A_i] = q_{\mathrm{PA}[A_i]})\big)$. For a fixed A_i and given $\langle q_{A_i}, q_{\mathrm{PA}[A_i]} \rangle = q_{\mathcal{F}(A_i)}$, we can group the $2^{|\mathbf{A}| - |\mathcal{F}(A_i)|}$ terms sharing the common factor $\ln\left(\mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \mathrm{PA}[A_i] = q_{\mathrm{PA}[A_i]})\right)$, yielding:

$$\ln\left(\mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \mathrm{PA}[A_i] = q_{\mathrm{PA}[A_i]})\right) \sum_{q' \in \{0,1\}^{|\mathbf{A}|} | q'_{\mathcal{F}(A_i)} = q_{\mathcal{F}(A_i)}} \mathbb{P}_{T_{\Theta^t}}(A = q' | E_k = e_k). \tag{11}$$

[2] We have used Mathematica (version 9) for those calculations.

As the latter sum equals $\mathbb{P}_{T_{\Theta^t}}(\mathcal{F}(A_i) = q_{\mathcal{F}(A_i)} | E_k = e_k)$, the expected log-likelihood, for each $\{E_k = e_k\}$, can be rewritten as:

$$\sum_{A_i \in \mathbf{At}(T_\Theta)} \sum_{q \in \{0,1\}^{|\mathcal{F}(A_i)|}} \left(\mathbb{P}_{T_{\Theta^t}}(\mathcal{F}(A_i) = q | E_k = e_k) \times \ln \left(\mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \text{PA}[A_i] = q_{\text{PA}[A_i]}) \right) \right). \tag{12}$$

Thus, assuming the observations are consistent with the model, we only need to make inferences $\mathbb{P}_{T_\Theta}(\mathcal{F}(A_i) = q | E_k = e_k)$ for those q such that $\mathbb{P}_{T_{\Theta^{t+1}}}(A_i = q_{A_i} | \text{PA}[A_i] = q_{\text{PA}[A_i]})$ is a (non-constant) function of the parameters, as the others can be ignored during the maximization.

For instance, suppose that the ground atom H is the head of a single ground probabilistic rule $\theta :: H :- B_1, B_2$**.** There are eight possible values for q regarding H's family $\{H, B_1, B_2\}$. However, we have to consider only two of them, $q = \langle 0, 1, 1 \rangle$ and $q = \langle 1, 1, 1 \rangle$, for the other values of q yield either $\mathbb{P}_{T_{\Theta^{t+1}}}(H = q_H | \text{PA}[H] = q_{\text{PA}[H]}) = 1$ (e.g. $q = \langle 0, 0, 1 \rangle$) or $\mathbb{P}_{T_{\Theta^{t+1}}}(H = q_H | \text{PA}[H] = q_{\text{PA}[H]}) = 0$ (e.g. $q = \langle 1, 1, 0 \rangle$).

It is worth noting that, both with complete and incomplete data, our approach via direct maximization of likelihood does not require the assumption that probabilistic atoms cannot be derived from deterministic rules. If probabilistic atoms are in fact derived, we just have to change the likelihood to accommodate it.

5. Learning the structure of probabilistic programs

The techniques described in the previous section are quite effective in speeding up parameter learning, as we demonstrate later. It is only natural to ask whether some of these techniques, perhaps enlarged with other insights, can lead to gains in structure learning. In this section we focus on this question, concentrating on *exact* structure learning; that is, on structure learning that pursues exact maximization of likelihood. The goal is to learn, from data, both probabilistic rules/facts and their associated probabilities.

A common strategy in structure learning is to resort to techniques from Inductive Logic Programming [4,13,14]. Typically one runs a search over the space of rules, under the assumption that some training examples are "positive" and must receive high probability, while other training examples are "negative" and must receive low probability. Search schemes vary and are almost universally based on heuristic measures, to guarantee that large datasets can be processed [17,28–30]. These techniques attempt to maximize appropriate scores by local moves that stop when some criterion is met.

Another general strategy for rule learning is to maximize a score that quantifies the fit between model and data, usually combining likelihood and penalties for model complexity [31–34]. Here we follow this strategy: we look for structures and parameters that maximize scores imported from Bayesian network learning. Unlike previous work [31,33,34], we do not introduce unnecessary latent or auxiliary variables; this allow us to exploit the parameter learning techniques described in the previous section to efficiently compute optimal local scores (when data is complete and rule size is bounded).

We start by summarizing a few relevant ideas in score-based structure learning.

5.1. Score-based structure learning of Bayesian networks

A score $s(B, D)$ gets a Bayesian network structure B and a dataset D, and yields a number that indicates the fit between both. Assume that D is complete with N observations of all random variables of interest. Sensible scores balance the desire to maximize likelihood against the need to constrain the number of learned parameters (it is well-known that if one maximizes only the likelihood, then the densest networks are always obtained) [26]. One particularly popular score, that we adopt throughout the paper, is based on minimum description length:

$$s_{\text{MDL}}(B, D) = \text{LL}_D(B) - \frac{|B| \log N}{2}, \tag{13}$$

where $|B|$ is the number of parameters needed to specify the network and $\text{LL}_D(B)$ is the log-likelihood of D at the maximum likelihood estimates. The latter estimates are denoted $\hat{\Theta}^{B,D} = \arg\max_\Theta \mathbb{P}_\Theta(D|B)$. The MDL score is *decomposable* in that it is a sum of *local scores*, each one a function of a variable's family.

The current technology on structure learning of Bayesian networks can handle relatively large sets of random variables if the maximum number of parents (or family size) is small [35–38]. Most existing methods proceed in two steps: first calculate the local score for every possible family (pruning provable suboptimal families); then maximize the global score while avoiding cycles, usually by integer programming [35], heuristic search [38] or constraint programming [36]. When one deals with structure learning of Bayesian networks where conditional probability distributions are encoded by CPTs, then maximum likelihood estimates $\hat{\Theta}^{B,D}$ are obtained in closed form: they are, in fact, simply relative frequencies, as noted at the beginning of Section 4. Thus the calculation of the scores is not really taxing for CPT-based Bayesian networks; the real computational effort is spent running the global optimization step.

5.2. A score-based learning algorithm for acyclic propositional PLPs

We now focus on propositional settings, and the goal is to learn an acyclic probabilistic logic program from a complete dataset. As before, we move freely between atoms and random variables, as each atom defines a random variable with values 1 (true) and 0 (false). Thus the dataset contains truth values for propositions, and observed values for random variables.

Because the MDL score is decomposable as a product of functions, each depending on a family of random variables, we can globally maximize it by first maximizing each local score separately, and then running a global maximization step that selects a family for each variable. Thus we can follow the two-step scheme described in the previous section: first the local score is computed for each possible family, by locally maximizing likelihood, and then a global maximization step produces the whole PLP. Note that this scheme can be used even if the MDL score is enlarged with penalties that focus on single families (for instance, number of literals per rule or number of atoms shared by rules with identical head); exploring enhancements for the MDL score is a promising path for future work.

There is however a difference between this procedure and the analogous one for Bayesian network: within a family, we are not just optimizing for parameters, but also for the rule pattern itself. That is, within a family we must choose the rule pattern that best relates a head atom with its parents. For instance, suppose that we evaluate the family $\{A_1, A_2, A_3\}$, where A_1 is to be the head atom. Do we select the rule pattern conveyed by Expression (4), or a simple rule pattern

$$\theta :: A_1 :- A_2, A_3.,$$

thus relying on a single rule? Note that both rule patterns admit closed-form maximum likelihood estimates, so the problem is not the increased complexity of local likelihood expressions. Indeed we have already dealt with maximization of local likelihood when discussing parameter learning (Section 4), and that material applies here. Rather, the problem is that we have freedom in selecting how many parameters we use in each family, and how these parameters interact.

How many different rule sets must we consider? Suppose first that we have a family containing only the head A. Then there is a single rule, the probabilistic fact $\theta :: A.$. If we instead have a family containing the head A and the body proposition B, there are six other options to consider:

$\theta :: A :- B.$	$\theta :: A :-$ **not** $B.$	$\theta_1 :: A :- B.$ $\theta_2 :: A.$
$\theta_1 :: A :-$ **not** $B.$ $\theta_2 :: A.$	$\theta_1 :: A :- B.$ $\theta_2 :: A :-$ **not** $B.$	$\theta_1 :: A :- B.$ $\theta_2 :: A :-$ **not** $B.$ $\theta_3 :: A.$

Now suppose we have a head A with parents B and C. First, there are 9 possible rules where A is the head and no proposition other than B or C appears in the body.[3] Each one of the 2^9 subsets of these rules is a possible rule set for this family; however, 14 of these subsets do not mention either B or C. Thus there are $2^9 - 14 = 498$ new rule sets to evaluate.

We discuss the search for a local rule pattern in the next section. Once all rule patterns are selected and local scores are computed, the relevant families and their scores are delivered to the second step consisting of a global optimization. In our implementation we resort to the constraint-programming algorithm (CPBayes) by Van Beek and Hoffmann [36] to run the global optimization step, thus selecting families so as to have an acyclic PLP — a selection of families specifies a PLP, as each family is associated with the rule set that maximizes the local score. The CPBayes algorithm defines a set of constraints that must be satisfied in the Bayesian network learning problem, and seeks for an optimal solution based on a depth-first BnB search. When trying to expand a node in the search tree, two conditions are verified: (1) whether constraints are satisfied, and (2) whether a lower bound estimate of the cost does not exceed the current upper bound. The constraint model includes dominance constraints, symmetry-breaking constraints, cost-based pruning rules, and a global acyclicity constraint. We remark that other approaches for the global optimization could be used, and our contribution is certainly not due to our use of CPBayes in the global optimization step. Thus we do not dwell on this second step.

5.3. Computing the local score

In this section we address the main novel challenge posed by exact score-based learning of acyclic PLPs; namely, the computation of local scores. There are too many rule patterns to consider, and each rule pattern may require a specific maximizing expression (a challenge we have faced in Section 4). We now reduce the burden of searching through rule patterns by pruning rules during search according to three insights listed below; in doing so we are inspired by methods that prune structures in Bayesian network learning, but our techniques are entirely different from those [37].

First of all, we can easily discard rule sets that assign zero probability to some configuration observed in the dataset.

Insight 1. *Rule sets that are inconsistent with some observation can be ignored while maximizing the local score.*

For instance, observation $\{A_1 = 1, A_2 = 0\}$ eliminates from further analysis a rule pattern consisting of a single rule $\theta :: A_1 :- A_2.$.

[3] These 9 rules are: $\theta :: A.$, $\theta :: A :- B.$, $\theta :: A :-$ **not** $B.$, $\theta :: A :- C.$, $\theta :: A :-$ **not** $C.$, $\theta :: A :- B, C.$, $\theta :: A :- B,$ **not** $C.$, $\theta :: A :-$ **not** $B, C.$, $\theta :: A :-$ **not** $B,$ **not** $C.$.

Second, and more importantly, suppose that we are learning rules with at most k literals in the body. Using 2^k rules, we can write a rule for each configuration of the parents: Example 2 illustrates this scenario. Note that for such "disjoint" rules, likelihood maximization is simple as it is the same as for usual CPT-based Bayesian networks. And because *any* CPT can be exactly built with such 2^k rules, any set of rules with more than 2^k rules cannot have higher likelihood, and thus cannot be optimal (as the penalty for the number of parameters increases). In fact, any other rule set with 2^k rules that are not disjoint can be also discarded; these sets can only produce the same likelihood, and will pay the same penalty on parameters, but they will be more difficult to handle.

Insight 2. *While maximizing the local score for a family with k parents, rule sets with size greater then 2^k can be ignored; furthermore, one needs to consider only one set with exactly 2^k rules.*

That is, we must only deal with rule sets with at most $2^k - 1$ rules, plus the one set of 2^k "disjoint" rules. Hence:

- If we have a family with $k = 1$, we only need to look at sets of one rule plus one set containing two disjoint rules; that is, we only have to consider:

$$\theta :: A :- B. \quad \Big| \quad \theta :: A :- \textbf{not } B. \quad \Big| \quad \begin{array}{c} \theta_1 :: A :- B. \\ \theta_2 :: A :- \textbf{not } B. \end{array}$$

  Note that the last of these three rule sets corresponds to a typical CPT, while the first two rule sets genuinely reduce the number of parameters in the model. Estimates that maximize likelihood are easily computed in all three cases.
- Now if we have a family with $k = 2$, say $\text{PA}[A_1] = \{A_2, A_3\}$, we only need to look at sets of up to three rules, plus one set containing four disjoint rules. Let $\mathcal{C}_i$ denote the set of rules with exactly i literals in the body. There are 4 sets consisting of one rule each, which are the four disjoint rules in $\mathcal{C}_2$. There are 30 sets consisting of two rules each: $4 \times 5 = 20$ sets containing 1 element from $\mathcal{C}_2$ and 1 from $\mathcal{C}_0 \cup \mathcal{C}_1$, 6 subsets of $\mathcal{C}_2$ and 4 sets with the form $\{\theta_1 :: A_1 :- (\textbf{not})\ A_2\textbf{.}, \theta_2 :: A_1 :- (\textbf{not})\ A_3\textbf{.}\}$. There are 82 sets consisting of three rules each: 4 subsets of $\mathcal{C}_2$, $6 \times 5 = 30$ sets containing exactly 2 elements from $\mathcal{C}_2$, $4 \times 10 = 40$ sets containing exactly one element from $\mathcal{C}_2$, 4 subsets of $\mathcal{C}_1$ and 4 sets with the form $\{\theta_1 :: A_1 :- (\textbf{not})\ A_2\textbf{.}, \theta_2 :: A_1 :- (\textbf{not})\ A_3\textbf{.}, \theta_3 :: A_1\textbf{.}\}$. Probability values may have nonlinear expressions as discussed in connection with Expression (4). And we still have a challenging optimization problem, where we must select a rule set out of many.

To address the difficulty mentioned in the previous sentence, we resort to a third insight: many of the rule sets obtained for $k = 2$ are actually restricted versions of a few patterns. As an example, consider Table 1. There we find four different sets of rules, some with two rules, and one with three rules. The shape of their likelihoods is the same, sans some renaming of parameters. Note in this particular example that the maximum likelihood of the first three sets of rules can always be attained by the likelihood of the last set of rules; consequently, it makes sense only to retain the last rule set (in this case the last pattern consists of disjoint rules that lead to simple maximum likelihood estimation, but more complex rule sets may be selected in general).

Insight 3. *During the maximization of the local score for a family $\mathcal{F}(A) = \{A\} \cup \text{PA}[A]$, a rule set T_Θ can be ignored if there is another rule set $T'_{\Theta'}$, with $|T'_{\Theta'}| \leq |T_\Theta|$, such that, for any value of Θ, there is a value for Θ' yielding $\mathbb{P}_{T'_{\Theta'}}(A \mid \text{PA}[A]) = \mathbb{P}_{T_\Theta}(A \mid \text{PA}[A])$ for all assignments to $\mathcal{F}(A)$.*

The equivalent patterns can be detected by inspection, by writing down the possible symbolic expressions and checking whether parameter renaming operations would make them equivalent.

By doing this additional pruning, we reach 27 distinct rule sets for $k = 2$ (see Appendix B); amongst them we must find a rule pattern that maximizes likelihood. To do so, the maximization techniques discussed in Section 4 can be employed.

We have discussed in detail the case $k = 2$; one can of course consider larger values of k, but the computational cost is sure to increase quickly. In our implementation and experiments we restricted ourselves to $k = 2$; this is a rather limited class but note that it includes all circuits with noisy two-input logical gates, a very general class. We denote by AP-PLP(2) the class of acyclic propositional probabilistic logic programs with at most two atoms in the body of rules. Larger rules and larger rule patterns should be addressed in future work, as well as extensions into the relational setting and into the missing data regime.

The procedure we have developed is summarized by Algorithm 1. It starts by generating, for each possible family (a variable and its candidate parents), the set of all possible rules, and then building its subsets (combinations of rules). These combinations are partitioned into patterns, according to their likelihood functions — Table 1 shows an example of a pattern. Within each pattern, only one combination with the minimum number of parameters (ensured lower score) need to be considered, and the others are pruned, as are those with zero likelihood. Parameters are then locally optimized for each of the combinations left. Each family is then associated with the combination of rules that gives it the highest score. Finally, a global score maximization algorithm is used to select the best family candidates.

Table 1
Likelihood patterns shared by several rule sets; the first column presents the rule set, and the following columns display the conditional probability $\mathbb{P}(A_1 \mid A_2, A_3)$ for the configurations of A_1, A_2, A_3.

Sets of rules	000	001	010	011	100	101	110	111
$\theta_1 :: A_1 :- A_2\bullet$ $\theta_2 :: A_1 :- A_2, A_3\bullet$ $\theta_3 :: A_1 :- A_2, \textbf{not } A_3\bullet$	1	1	$1-\theta_{1;3}$	$1-\theta_{1;2}$	0	0	$\theta_{1;3}$	$\theta_{1;2}$
$\theta_1 :: A_1 :- A_2\bullet$ $\theta_2 :: A_1 :- A_2, A_3\bullet$	1	1	$1-\theta_1$	$1-\theta_{1;2}$	0	0	θ_1	$\theta_{1;2}$
$\theta_1 :: A_1 :- A_2\bullet$ $\theta_3 :: A_1 :- A_2, \textbf{not } A_3\bullet$	1	1	$1-\theta_{1;3}$	$1-\theta_1$	0	0	$\theta_{1;3}$	θ_1
$\theta_2 :: A_1 :- A_2, A_3\bullet$ $\theta_3 :: A_1 :- A_2, \textbf{not } A_3\bullet$	1	1	$1-\theta_3$	$1-\theta_2$	0	0	θ_3	θ_2

Algorithm 1 Structure learning algorithm for acyclic propositional PLPs.

```
collect variables from dataset
for each family of variables in dataset do
    build all possible rules
    build all possible sets of rules
    gather rule sets into patterns
    for each pattern do
        prune combinations with ensured lower score
        prune combinations with zero likelihood
        for each combination left do
            if the likelihood maximization problem has an exact solution then
                calculate parameters
            else
                run numeric (exact or approximate) likelihood maximization
        calculate local scores
for each family do
    associate best rule set with family
call CPBayes algorithm to maximize global score
```

6. Experiments

Methods for parameter and for structure learning were tested separately. For parameter learning, we could compare our techniques with ProbLog's implementation, in terms of likelihood and running time. For structure learning, the experiments were aimed at validating the method and exploring the reduction in the number of parameters that PLPs may yield over Bayesian networks.

To simplify the text, we refer to our scheme for direct maximization of likelihood, as described in Section 4, as *PL-direct*.

6.1. Learning parameters

The goal of these experiments was to compare PL-direct with the LFI-ProbLog algorithm for varying dataset sizes and rates of missing data. To accomplish this we generated data from two different PLPs, one propositional and one relational, and used the data to learn parameters. To generate the datasets for the propositional PLP, we sampled atoms a number of times; for the relational PLP, we sampled atoms once, but we varied the number of constants in the program (hence the meaning of "dataset size" differs in the propositional and relational settings).

Experiments were run using machines with identical processors at Amazon Web Services, and at a local server with 8 CPUs, 4.2 GHz, and 32 GB RAM. The implementation was coded in Python, using the optimization library scipy.optimize.[4] We tested two different algorithms for numeric optimization: (1) the Limited-memory BFGS (L-BFGS) algorithm and (2) the Basin-hopping algorithm [39]. The L-BFGS algorithm approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [40], an iterative method for solving unconstrained nonlinear optimization problems. The L-BFGS algorithm represents with a few vectors an approximation to the inverse Hessian matrix; this approach leads to a significant reduction on memory use. Nevertheless, it has a quite strong dependence on the initial guess. The Basin-hopping is a stochastic algorithm that usually provides a better approximation of the global maximum. The algorithm iteratively chooses an initial guess at random, proceeds to the local minimization and finally compares the new coordinates with the best ones found so far. This latter algorithm often required more time for convergence than L-BFGS, never finding a better likelihood value, and sometimes finding worse values; thus, even though we tested our methods with the Basin-hopping algorithms, we refrain from

[4] The implementation of PL-direct, together with datasets used in experiments, is publicly available at https://github.com/arthurcgusmao/asteroidea.

```
0.85 :: a1.          0.95 :: a3.          0.95 :: a2.          0.8 :: ll2.
0.8 :: pl2.          0.8 :: ll3.          0.8 :: pl3.
0.8 :: low_load :— ll1, pl1.              0.8 :: low_load :— ll2, pl2.
0.8 :: low_load :— ll3, pl3.              0.95 :: high_supply :— a2, a3.
0.95 :: high_supply :— a2, a4.            0.95 :: high_supply :— a3, a4.
0.7 :: ll1 :— emergency.                  0.7 :: pl1 :— emergency.
0.95 :: low_load :— high_load.            0.6 :: high_load :— ll2, ll3, pl2, pl3.
0.95 :: low_supply :— a1.                 1.0 :: low_supply :— high_supply.
0.98 :: emergency :— not a2, not a3.
0.75 :: a4 :— a2.                         0.75 :: a4 :— a3.
0.95 :: failure :— high_load, not high_supply.
0.95 :: failure :— low_load, not low_supply.
```

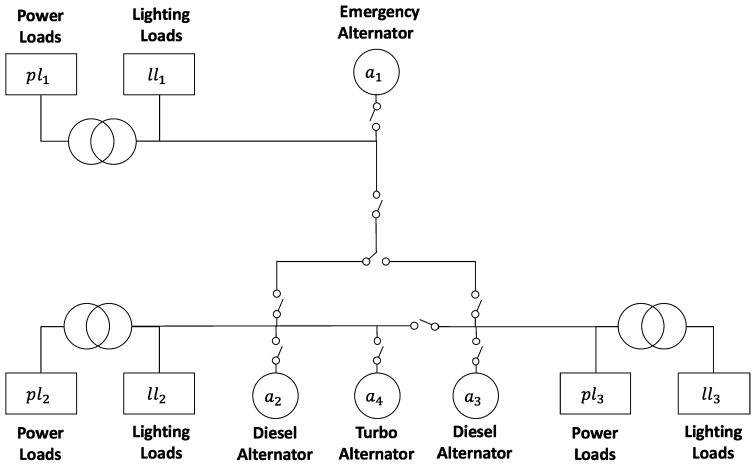


Fig. 1. A PLP encoding a ship's energy plant (top), and a diagram of the ship's energy plant (bottom).

```
0.3 :: fire(X) :— person(X).
0.4 :: burglary(X) :— person(X).
0.7 :: alarm(X) :— fire(X).
0.8 :: calls(X, Y) :— cares(X, Y), alarm(Y), not samePerson(X, Y).
0.9 :: alarm(X) :— burglary(X).
0.8 :: cares(X, Y) :— person(X), person(Y).
```

Fig. 2. A relational PLP with the alarm/earthquake problem.

extensively reporting on those tests. In particular, experiments reported in this section focus on L-BFGS (in the context of parameter learning the values reached by both methods were essentially identical, while for structure learning there were some differences as we indicate later in Table 3).

The propositional PLP we have used consists of 16 propositions, 17 probabilistic rules, and 7 probabilistic facts that encode a ship's energy plant. The model can be understood as an "almost" deterministic system, mostly specified through Boolean operators, together with probabilistic disturbances, as presented in Fig. 1. The model was built by the first author from a technical description in a specialized web site.[5] There are various kinds of elements: power loads, lighting loads, alternators, transformers and switches. Note that we must differentiate between the (demanded) power load and the (provided) power supply; when the load exceeds the supply, a failure ensues. Domain knowledge was translated as follows. First, when a block of lighting/power loads is operating, the load demanded from alternators is low. Second, to provide a high supply, at least two alternators must be operating (and one alternator is left for emergencies). Third, when the system provides (demands) a high supply (load), then a low supply (load) can also be provided (demanded); also, if the required power load is larger than the power supply, a failure is to be expected. And if the second and third blocks of lighting/power loads are operating, a high load is needed. Also, during an emergency the first block of lighting/power loads must be available; an emergency appears if two specific alternators do not operate. Finally, if one of the diesel alternators operate, then probably the turbo alternator operates as well.

For the relational datasets a variation of the alarm/earthquake problem was used; the PLP appears in Fig. 2.

[5] http://www.machineryspaces.com/emergency-power-supply.html.

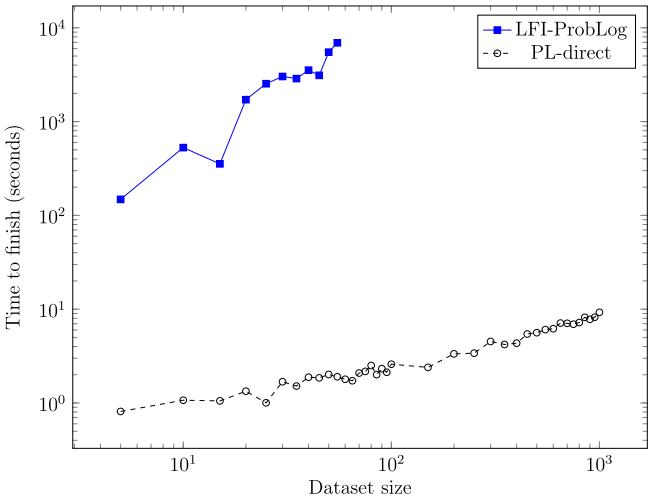


Fig. 3. Time to learn parameters (note log-scale!) in the propositional dataset with complete data. Both algorithms reached similar log-likelihood values. Dataset size corresponds to the number of observations for all atoms.

Table 2
Comparison of wall-clock time (in seconds) for PL-direct with the use of closed-form solutions, PL-direct without closed-form solutions, and LFI-ProbLog, in the alarm/earthquake problem, for various population sizes.

Population size	5	10	15	20
PL-direct	0.19	0.79	2.81	8.06
PL-direct, no closed-form	0.29	0.88	2.90	8.10
LFI-ProbLog	1.07	16.29	62.06	206.06

6.1.1. Complete data

Our first experiment tested the performance of PL-direct and LFI-ProbLog in complete datasets. Remember that PL-direct does not require EM when the data are complete, hence a significant difference between PL-direct and LFI-ProbLog is to be expected. In this experiment, ProbLog was set to stop whenever the log-likelihood improvement dropped below 10^{-3}. Fig. 3 shows the learning time for both algorithms for the propositional case (note the log-scale!), where both algorithms reached similar log-likelihood values. Indeed, PL-direct outperforms ProbLog by orders of magnitude. We limited the learning time to 120 minutes (after that period we aborted the run if the algorithm was still iterating), so in this experiment we only have ProbLog results for datasets with at most 55 observations. The results from the larger dataset that ProbLog could handle shows that it was more than 3600 times slower than PL-direct. Significant performance differences can also be seen for the relational case with no missing data, as can be seen in Fig. 4 (a).

There are two sources of speed-up here: the use of closed-form solutions whenever possible, and the use of direct optimization methods whenever closed-form solutions cannot be found. The relative effect of these two techniques clearly depends on the particular problem: if a model is such that many fixed sets of rules yield closed-form solutions, the overall running time is bound to be short, with closed-form expressions guaranteeing most of the speed up. However if many sets of rules cannot be tackled by closed-form solutions, then the speed-up comes from the optimization method.

In order to understand the main source of the speed up — whether it came from avoiding the latent variables or adding closed form solutions on top of that — in a particular problem, we performed another experiment using the alarm/earthquake problem.[6] We set up our implementation to always run the optimization procedure for all families, regardless of whether they have a closed-form solution or not. Table 2 conveys some interesting results, where each numeric value is the average of eight runs. The vast majority of the gain is due to removing the latent variables employed in ProbLog (as they introduce a significant computational burden); some gains, small but not insignificant, are due to the use of closed-form solutions.

6.1.2. Incomplete data

The effect of missing data was investigated by randomly discarding individual observations. Each test was run eight times, each time with a new independently sampled dataset; hence each datapoint in the following figures corresponds to the average computation time of eight runs; error bars indicate the standard deviation. In all tests, LFI-ProbLog was run

[6] We thank a reviewer for suggestions that led to this experiment.

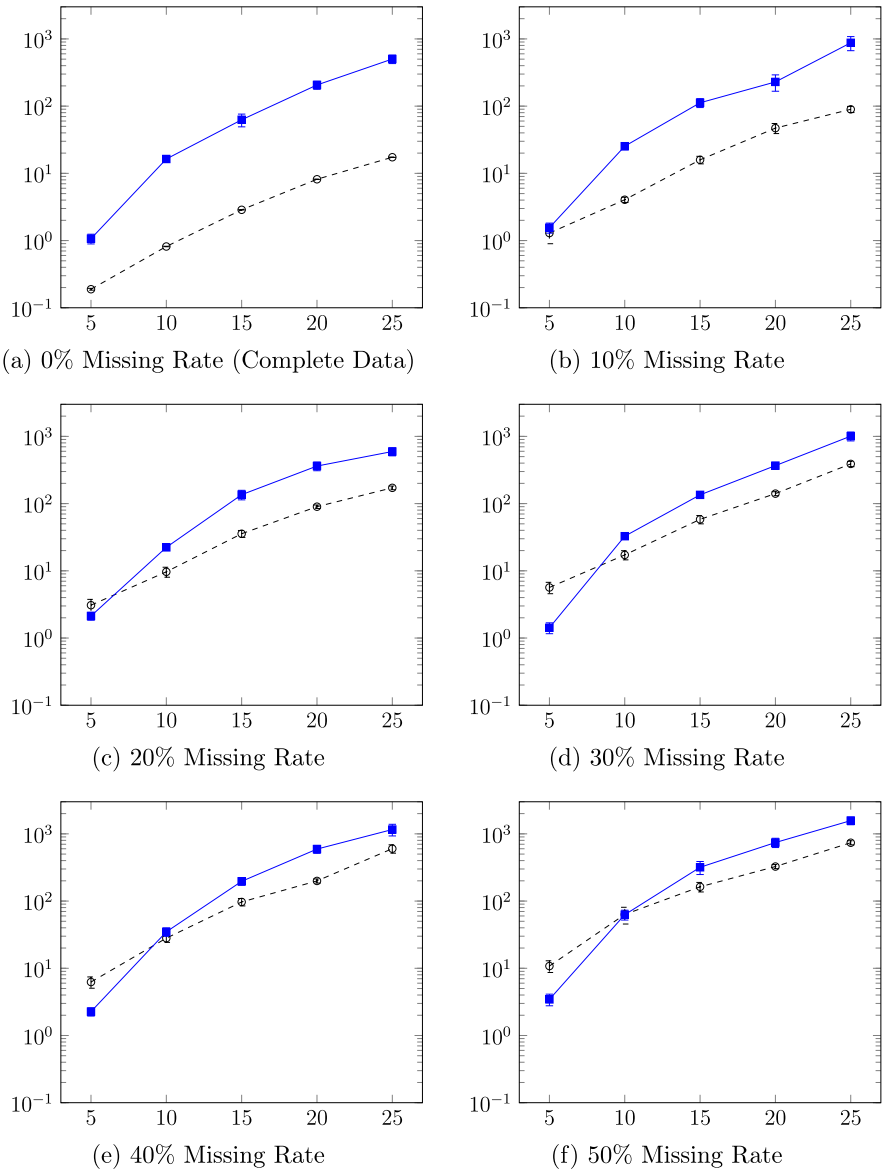


Fig. 4. Average time to learn parameters (log-scale) of eight independently sampled relational datasets (for each different missing data rate, new data samples were generated). Solid squares were generated by ProbLog; empty circles were generated by PL-direct. The number of constants in the program appears in the horizontal axis; this number is akin to "dataset size". The error bars represent the standard deviation of the mean.

until the log-likelihood variation between subsequent iterations was smaller than an arbitrary value (10^{-3}); then PL-direct was run until it reached an equal or better log-likelihood value than ProbLog. We had to limit ourselves to 25 constants because that is the limit for ProbLog (larger problems lead to numerical problems in ProbLog).

From Fig. 4 we can see that PL-direct tends to surpass LFI-ProbLog as the size of the dataset increases (graphs in that figure were obtained with the relational PLP). Overall, PL-direct outperformed ProbLog in the vast majority of cases, only displaying worse computation time averages for very small datasets (with 5 constants) that are subject to high missing rates (above 20%). Fig. 5 offers a better assessment on how the missing rate affects the performance of both algorithms. The more missing data, the more numeric computation is done by PL-direct, and the more PL-direct behaves as LFI-ProbLog with regard to computation time.

6.2. Learning structure

To validate our methods for structure learning, we have implemented Algorithm 1, and tested it with a number of datasets. Our goal here was to examine whether the algorithm actually produces sensible PLPs with less parameters than corresponding CPT-based Bayesian networks.

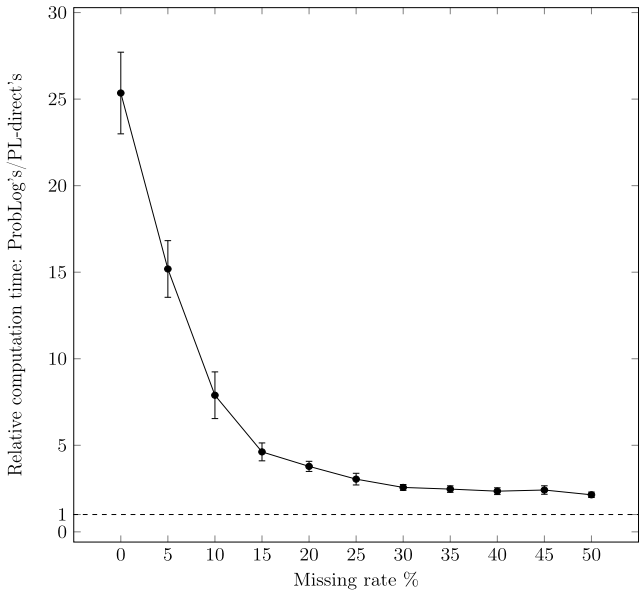


Fig. 5. ProbLog's learning time relative to PL-direct's, for similar log-likelihood values. Only datasets (sampled from the relational PLP) with the size of 15, 20 and 25 constants were considered, as larger datasets cannot be handled by ProbLog. The datapoints represent the average relative learning time of ProbLog of twenty-four runs (the sum of eight different runs for each dataset size). The error bars represent the respective standard error of the mean.

Table 3
Binary adder experiment results. The symbol $|\Theta|$ denotes the number of learned parameters.

L-BFGS					
# Instances		CPT-based		Any combination of rules	
Training	Testing	Log-Likelihood	$\|\Theta\|$	Log-Likelihood	$\|\Theta\|$
30	10000	−317590.82	61	−190592.57	48
60	10000	−282860.54	63	−211535.40	51
90	10000	−231096.56	61	−200086.83	48
120	10000	−281571.16	61	−197029.87	51
250	10000	−244887.02	65	−251489.19	51
500	10000	−228617.04	66	−217608.84	52
1000	10000	−188236.10	80	−177049.65	62
Basin-hopping					
# Instances		CPT-based		Any combination of rules	
Training	Testing	Log-Likelihood	$\|\Theta\|$	Log-Likelihood	$\|\Theta\|$
30	10000	−344580.91	61	−190652.02	48
1000	10000	−188236.10	80	−177049.52	62

The algorithm was implemented in Python and experiments were performed on a Unix Machine with Intel core i5 (2.7 GHz) processor and 8 GB 1867 MHz DDR3 SDRAM. For local optimization of the likelihood scores, in the cases where that was needed, we again compared (1) Limited-memory BFGS (L-BFGS) and (2) Basin-hopping.

We tested our implementations for three different applications described in the remainder of this section. Each test compares PLPs learned with Algorithm 1 and CPT-based Bayesian networks learned by exact score maximization.

Binary adder This first test aimed at learning an AP-PLP(2) to represent a (simulated) faulty Boolean circuit, designed to add two 4-bit numbers. To generate binary datasets for training and testing, we took a number of pairs of randomly selected 4-bit numbers; each pair was processed by an adder circuit with 18 logic gates (XOR, OR, AND). Each gate was associated with a 1% probability of failure. This probability of failure indicates how often gates should output random values, instead of the values expected. Training datasets contain 30, 60, 90, 120, 250, 500 and 1000 examples and the testing dataset contains 10000 examples. Examples contain 24 observations: 8 entry bits and 18 output values, one for each gate. For the optimization steps, we used the L-BFGS algorithm. A few tests were repeated using Basin-hopping optimization algorithm, which is much more time-consuming, but no improvements were observed in terms of log-likelihood and number of parameters. Results obtained are listed in Table 3.

Table 4
Heart Diagnosis experiment results.

L-BFGS			
Complete CPT		Any combination of rules	
Log-Likelihood	# Parameters	Log-Likelihood	# Parameters
−1281.78	63	−1263.85	55
Basin-hopping			
Complete CPT		Any combination of rules	
Log-Likelihood	# Parameters	Log-Likelihood	# Parameters
−1281.78	63	−1263.85	55

Table 5
Movie Genres experiments results.

Complete CPT		Any combination of rules	
Log-Likelihood	# Parameters	Log-Likelihood	# Parameters
−2032.38	62	−2031.49	42

For smaller datasets, PLPs produced by Algorithm 1 scored better than CPT-based Bayesian networks, and required fewer parameters. For larger datasets, both approaches tended to converge in terms of log-likelihood, but there was still a significant reduction on the number of parameters with Algorithm 1. As this is a nearly-deterministic PLP, it should be expected that logic rules would encode the local probability distributions much more compactly than complete CPT's.

Heart diagnosis This second test aimed at learning an AP-PLP(2) to reason over diagnosis made from observation of cardiac Single Proton Emission Computed Tomography (SPECT) images, using a standard dataset from the UCI repository [41]. The training and testing datasets contain 80 and 187 instances respectively. The 23 binary attributes are observed for each example, so there is no missing data. The learning algorithm was tested with both L-BFGS and Basin-Hopping optimization methods. Results obtained are listed in Table 4. Both CPT-based Bayesian networks and Algorithm 1 tended to produce equivalent results in terms of log-likelihood, but there was still a significant reduction on the number of parameters with Algorithm 1. In addition, we note that results obtained with both optimization methods, L-BFGS and Basin-Hopping, were identical.

Movie genres The third test aimed at learning an AP-PLP(2) to reason over movie genres; here the idea was to learn a relatively large program. Labels result from previous multi-classification of movies according to the genres they represent, with information extracted from the MovieLens dataset.[7] Training and testing datasets are obtained from a random 70/30 split of the original dataset, which contains 3449 movies. Algorithm 1 was tested with L-BFGS optimization method. Results are listed in Table 5; the resulting AP-PLP(2) and the corresponding dependency graph are shown in Figs. 6 and 7.

In this experiment we do not observe a significant improvement in terms of log-likelihood from CPT-based learning to Algorithm 1, but the number of parameters produced by the latter algorithm was considerably smaller than the number produced by the former. In this context the restriction of having at most two parents per variable may be too strong, but the dependency graph still gives us an idea of which labels are more correlated.

7. Conclusions

We have presented novel techniques to the problems of learning parameters and structure in acyclic probabilistic logic programs. We have put forward a new method to learn the parameters when dealing with probabilistic rules, showing how one can avoid the insertion of latent variables. We have also described techniques that can learn a whole PLP from a complete dataset by exact score maximization, by adapting techniques from CPT-based Bayesian networks.

In the complete data scenario, experiments indicate that our approach to parameter learning yields orders of magnitude gains when compared to LFI-ProbLog, a reasonable contender for the current state-of-art. Even when there is missing data, the smaller number of latent variables guarantees significant improvements in efficiency.

This paper offers initial results on learning whole PLPs via exact score maximization, presenting cases where closed-form solutions are viable. We have also implemented and tested our structure learning methods, and we have shown that learned PLPs contain less parameters than the corresponding CPT-based Bayesian networks, as intuitively expected. Whenever the model is nearly deterministic, the expressive power of rules leads to improved accuracy.

In future work we intend to extend our structure learning techniques to relational but still acyclic programs, and to cope with missing data in those cases. We also wish to explore parameter and structure learning for cyclic programs. For those

[7] Available at http://grouplens.org/datasets/movielens/.

0.04 :: war.
0.00 :: children.
0.14 :: comedy.
0.31 :: comedy :— thriller, war.
0.00 :: thriller :— war.
0.17 :: thriller :— war.
0.13 :: action :— comedy.
0.24 :: action :— thriller.
0.06 :: action :— comedy, thriller.
0.27 :: drama.
0.53 :: drama :— action, comedy.
0.01 :: western.
0.03 :: western :— comedy, drama.
0.03 :: horror.
0.23 :: horror :— comedy, drama.
0.00 :: documentary.
0.08 :: documentary :— comedy, drama.
0.03 :: sciFi :— drama.
0.27 :: sciFi :— action, drama.
0.07 :: sciFi :— action, drama.
0.27 :: adventure :— action.
0.10 :: adventure :— drama.
0.03 :: adventure :— action, drama.
0.02 :: romance.
0.16 :: romance :— documentary, horror.
0.09 :: animation :— adventure.
0.00 :: animation :— drama.
0.04 :: animation :— drama.
0.02 :: crime :— horror.
0.27 :: crime :— horror, thriller.
0.06 :: crime :— horror, thriller.
0.17 :: fantasy :— adventure.
0.23 :: fantasy :— animation.
0.03 :: fantasy :— adventure, animation.
0.00 :: musical :— thriller.
0.33 :: musical :— animation, thriller.
0.03 :: musical :— animation, thriller.
0.01 :: filmNoir.
0.13 :: filmNoir :— action, crime.
0.22 :: mystery :— filmNoir.
0.13 :: mystery :— thriller.
0.02 :: mystery :— filmNoir, thriller.

Fig. 6. Probabilistic logic program learned in the Movie Genres experiment. Some probabilities are zero due to rounding.

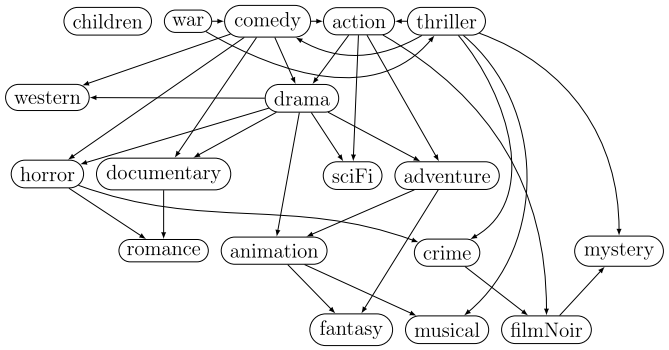


Fig. 7. Dependency graph of the AP-PLP(2) learned in the Movie Genres experiment.

cases non-trivial extensions will have to be developed as the direct relationship with Bayesian network learning is lost, and it is not obvious how to express the likelihood function.

Acknowledgements

The first author was supported by a scholarship from Toshiba Corporation. The second author was supported by a scholarship from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), and was also supported by a scholarship from the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grant 2017/19007-6. The third author was supported by FAPESP grant 2016/25928-4. The fourth author is partially supported by CNPq grants 303920/2016-5 and 420669/2016-7. The fifth author is partially supported by CNPq grant 308433/2014-9. Support was also provided in part by FAPESP grants 2016/01055-1, 2015/21880-4, and 2016/18841-0, and in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) — finance code 001.

Appendix A. Closed-form estimation of a NoisyOr gate

Here we focus on the structure given by Expression (6):

$$\theta_1 :: A_1., \quad \theta_2 :: A_2., \quad \theta_3 :: A_3 :- A_1., \quad \theta_4 :: A_3 :- A_2.,$$

to be learned from a complete dataset with N complete observations; we use the notation $N_{a_1a_2a_3}$ to denote the number of times we observe $\{A_1 = a_1, A_2 = a_2, A_3 = a_3\}$. The likelihood is

$$\theta_1^{N'_1}(1-\theta_1)^{N'_0}\theta_2^{N''_1}(1-\theta_2)^{N''_0}\theta_3^{N_{101}}(1-\theta_3)^{N_{100}}\theta_4^{N_{011}}(1-\theta_4)^{N_{010}}\theta_{3;4}^{N_{111}}(1-\theta_{3;4})^{N_{110}},$$

where $N'_1 = N_{100} + N_{101} + N_{110} + N_{111}$, $N'_0 = N - N'_1$, $N''_1 = N_{010} + N_{011} + N_{110} + N_{111}$, $N''_0 = N - N''_1$. By operating with terms in θ_1 and θ_2, we easily obtain $\hat{\theta}_1 = N'_1/N$ and $\hat{\theta}_2 = N''_1/N$. Expressions for $\hat{\theta}_3$ and $\hat{\theta}_4$ are much more complicated; for instance we have $\hat{\theta}_3$ to be either $(-K_1 + \sqrt{K_2})/K_3$ or $(-K_1 - \sqrt{K_2})/K_3$, where

$$
\begin{aligned}
K_1 &= N_{011}N_{100} - N_{010}N_{101} + N_{011}N_{101} + 2N_{100}N_{101} + 2N_{101}^2 + \\
&\quad N_{011}N_{110} + N_{101}N_{110} - N_{010}N_{111} + N_{100}N_{111} + 2N_{101}N_{111} \\
K_2 &= 4(N_{010} - N_{100} - N_{101})N_{101}(N_{011} + N_{101} + N_{111})(N_{100} + N_{101} + N_{110} + N_{111}) + \\
&\quad (2N_{100}N_{101} + 2N_{101}^2 + N_{101}N_{110} + N_{011}(N_{100} + N_{101} + N_{110}) + \\
&\quad N_{100}N_{111} + 2N_{101}N_{111} - N_{010}(N_{101} + N_{111}))^2, \\
K_3 &= 2(N_{010} - N_{100} - N_{101})(N_{100} + N_{101} + N_{110} + N_{111}).
\end{aligned}
$$

Appendix B. Distinct rule sets with two parents

After pruning redundant sets (see Table 1), we are left with 27 sets of rules for a head (A_1) with two parents (A_2, A_3):

$$\{\theta_1 :: A_1 :- A_2, A_3\bullet\}, \{\theta_1 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet\}, \{\theta_1 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- A_2\bullet, \theta_2 :: A_1 :- A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- A_2\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2\bullet, \theta_2 :: A_1 :- A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet, \theta_2 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- A_2, A_3\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- A_2, A_3\bullet, \theta_2 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- A_2, A_3\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\bullet\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2\bullet, \theta_2 :: A_1 :- A_2, A_3\bullet, \theta_3 :: A_1 :- A_2, \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- A_2\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_3\bullet, \theta_2 :: A_1 :- A_2, A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, A_3\}$$
$$\{\theta_1 :: A_1 :- A_3\bullet, \theta_2 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- A_2, A_3\bullet, \theta_2 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, A_3\}$$
$$\{\theta_1 :: A_1 :- A_2, A_3\bullet, \theta_2 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- A_2, A_3\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- A_2\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2\bullet, \theta_3 :: A_1 :- A_3\}$$
$$\{\theta_1 :: A_1 :- A_2\bullet, \theta_2 :: A_1 :- \mathbf{not}\ A_2\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- A_2\bullet, \theta_2 :: A_1 :- A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2\bullet, \theta_2 :: A_1 :- A_3\bullet, \theta_3 :: A_1 :- \mathbf{not}\ A_3\}$$
$$\{\theta_1 :: A_1 :- \mathbf{not}\ A_2, A_3\bullet, \theta_2 :: A_1 :- A_2, \mathbf{not}\ A_3\bullet,$$
$$\theta_3 :: A_1 :- A_2, A_3\bullet, \theta_4 :: A_1 :- \mathbf{not}\ A_2, \mathbf{not}\ A_3\bullet\}$$

References

[1] M. Jaeger, Relational Bayesian networks, in: Conference on Uncertainty in Artificial Intelligence, 1997, pp. 266–273.
[2] L. Getoor, B. Taskar, Introduction to Statistical Relational Learning, MIT Press, 2007.
[3] M. Richardson, P. Domingos, Markov logic networks, Mach. Learn. 62 (1) (2006) 107–136.
[4] L. De Raedt, Logical and Relational Learning, Springer Science & Business Media, 2008.
[5] A.D. Gordon, T.A. Henzinger, A.V. Nori, S.K. Rajmani, Probabilistic programming, in: Future of Software Engineering, ACM, 2014, pp. 167–181.
[6] J.Y. Halpern, An analysis of first-order logics of probability, Artif. Intell. 46 (3) (1990) 311–350.
[7] Z. Ognjanovic, M. Raškovic, Some first-order probability logics, Theor. Comput. Sci. 247 (1) (2000) 191–212.
[8] N. Fuhr, Probabilistic Datalog — a logic for powerful retrieval methods, in: ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 1995, pp. 282–290.
[9] T. Lukasiewicz, Probabilistic logic programming, in: European Conference on Artificial Intelligence, 1998, pp. 388–392.
[10] R. Ng, V.S. Subrahmanian, Probabilistic logic programming, Inf. Comput. 101 (2) (1992) 150–201.
[11] D. Poole, Probabilistic Horn abduction and Bayesian networks, Artif. Intell. 64 (1) (1993) 81–129.
[12] T. Sato, A statistical learning method for logic programs with distribution semantics, in: International Conference on Logical Programming, 1995.
[13] L. De Raedt, P. Frasconi, K. Kersting, S.H. Muggleton, Probabilistic Inductive Logic Programming, Springer, 2008.
[14] F. Riguzzi, E. Bellodi, R. Zese, A history of probabilistic inductive logic programming, Front. Robot. AI 1 (2014) 6.
[15] T. Sato, Y. Kameya, Parameter learning of logic programs for symbolic-statistical modeling, J. Artif. Intell. Res. 15 (2001) 391–454.
[16] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted Boolean formulas, Theory Pract. Log. Program. 15 (3) (2015) 358–401.
[17] L. De Raedt, A. Dries, I. Thon, G. Van den Broeck, M. Verbeke, Inducing probabilistic relational rules from probabilistic examples, in: International Joint Conference on Artificial Intelligence, 2015, pp. 1835–1842.
[18] F.H.O.V. de Faria, A.C. Gusmão, G. De Bona, D.D. Mauá, F.G. Cozman, Parameter learning in ProbLog with probabilistic rules, in: Symposium on Knowledge Discovery, Mining and Learning, 2017, pp. 27–34.
[19] F.H.O.V. de Faria, F.G. Cozman, D.D. Mauá, Closed-form solutions in learning probabilistic logic programs by exact score maximization, in: S. Moral, O. Pivert, D. Sánchez, N. Marín (Eds.), Conference on Scalable Uncertainty Management, 2017, pp. 119–133.
[20] F.H.O.V. de Faria, A.C. Gusmão, F.G. Cozman, D.D. Mauá, Speeding-up ProbLog's parameter learning, in: International Workshop on Statistical Relational AI, 2017.
[21] L. De Raedt, A. Kimmig, H. Toivonen, ProbLog: a probabilistic Prolog and its application in link discovery, in: International Joint Conference on Artificial Intelligence, 2007, pp. 2468–2473.
[22] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (3) (2001) 374–425.
[23] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Elsevier, 1988.
[24] D. Poole, The independent choice logic and beyond, in: Probabilistic Inductive Logic Programming, Springer, 2008, pp. 222–243.
[25] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, J. R. Stat. Soc. B (1977) 1–38.
[26] D. Koller, N. Friedman, Probabilistic Graphical Models: Principles and Techniques, MIT Press, 2009.
[27] O. Pourret, P. Naim, B. Marcot, Bayesian Networks — A Practical Guide to Applications, Wiley, 2008.
[28] E. Bellodi, F. Riguzzi, Structure learning of probabilistic logic programs by searching the clause space, Theory Pract. Log. Program. 15 (2) (2015) 169–212.
[29] L. De Raedt, I. Thon, Probabilistic rule learning, in: International Conference on Inductive Logic Programming, Springer, 2010, pp. 47–58.
[30] F. Yang, Z. Yang, W.W. Cohen, Differentiable learning of logical rules for knowledge base reasoning, in: Advances in Neural Information Processing Systems, 2017, pp. 2316–2325.
[31] F. Riguzzi, Learning logic programs with annotated disjunctions, in: Proceedings of the 13th International Conference on Inductive Logic Programming, 2004, pp. 270–287.
[32] S. Kok, P. Domingos, Learning the structure of Markov logic networks, in: Proceedings of the 22nd International Conference on Machine Learning, 2005, pp. 441–448.
[33] H. Blockeel, W. Meert, Towards learning non-recursive LPADs by transforming them into Bayesian networks, in: Proceedings of the 15th International Conference on Inductive Logic Programming, 2006, pp. 94–108.
[34] W. Meert, J. Struyf, H. Blockeel, Learning ground CP-logic theories by leveraging Bayesian network learning techniques, Fundam. Inform. (2008) 1–30.
[35] J. Cussens, Bayesian network learning with cutting planes, in: Conference on Uncertainty in Artificial Intelligence, AUAI Press, 2011, pp. 153–160.
[36] P. Van Beek, H.-F. Hoffmann, Machine learning of Bayesian networks using constraint programming, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2015, pp. 429–445.
[37] C.P. De Campos, Q. Ji, Efficient structure learning of Bayesian networks using constraints, J. Mach. Learn. Res. 12 (2011) 663–689.
[38] C. Yuan, B. Malone, Learning optimal Bayesian networks: a shortest path perspective 48 (2013) 23–65.
[39] D.J. Wales, J.P. Doye, Global optimization by basin-hopping and the lowest energy structures of Lennard–Jones clusters containing up to 110 atoms, J. Phys. Chem. A, Mol. Spectrosc. Kinet. Environ. Gen. Theory 101 (28) (1997) 5111–5116.
[40] D. Wales, Energy Landscapes: Applications to Clusters, Biomolecules and Glasses, Cambridge University Press, 2003.
[41] K. Bache, M. Lichman, UCI Machine Learning Repository (2013).