Basic Principles of Learning Bayesian Logic Programs

Kristian Kersting and Luc De Raedt
Institute for Computer Science, Machine Learning Lab
University of Freiburg,
Georges-Koehler-Allee, Building 079,
D-79110 Freiburg/Brg., Germany
E-mail: {kersting,deraedt}@informatik.uni-freiburg.de

Bayesian logic programs tightly integrate definite logic programs with Bayesian networks in order to incorporate the notions of objects and relations into Bayesian networks. They establishing a one-to-one mapping between ground atoms and random variables, and between the *immediate consequence operator* and the *directly influences by* relation. In doing so, they nicely separate the qualitative (i.e. logical) component from the quantitative (i.e. the probabilistic) one providing a natural framework to describe general, probabilistic dependencies among sets of random variables. In this paper, we present results on combining *Inductive Logic Programming* with Bayesian networks to learn both the qualitative and the quantitative components of Bayesian logic programs from data. More precisely, we show how the qualitative components can be learned by combining the inductive logic programming setting *learning from interpretations* with score-based techniques for learning Bayesian networks. The estimation of the quantitative components is reduced to the corresponding problem of (dynamic) Bayesian networks.
Keywords: Bayesian networks, first order logic, learning from interpretations, parameter estimation, structural learning, gradient, EM

1. Introduction

In recent years, there has been an increasing interest in integrating probability theory with first order logic. One of the research streams [Poo93,NH97, Jae97,FGKP99,KD01] concentrates on first order extensions of Bayesian networks [Pea91]. The reason why this has attracted attention is, that even though Bayesian networks are one of the most important, efficient and elegant frameworks for representing and reasoning with probabilistic models, they suffer from an inherently propositional character. A single Bayesian network specifies a joint probability density over a finite set of random variables and consists of two components:

- a *qualitative* one that encodes the local influences among the random variables using a directed acyclic graph, and
- a *quantitative* one that encodes the probability densities over these local influences.

Imagine a Bayesian network modelling the localization of genes/proteins. Here, every gene would be a single random variable. There is no way of formulating general probabilistic regularities among the localizations of the genes such as *a gene G has localization L if G interacts with another gene G' that has localization L.*

Bayesian logic programs are a language that overcomes this propositional character by tightly integrating definite logic programs with Bayesian networks to incorporate the notions of objects and relations. In doing so, they can naturally be used to do first order classification, clustering, and regression. Their underlying idea is to establish a one-to-one mapping between ground atoms and random variables, and between the *immediate consequence operator* and the *directly influences by* relation. In doing so, they nicely separate the qualitative (i.e. logical) component from the quantitative (i.e. the probabilistic) one providing a natural framework to describe general, probabilistic dependencies among sets of random variables such as the rule stated above.

However, it is well-known that determining the structure of a Bayesian network, and therefore also of a Bayesian logic program, can be difficult and expensive. In 1997, Koller and Pfeffer [KP97] addressed the question "*where do the numbers come from?*" for similar frameworks. So far, this issue has not yet attracted much attention in the context of first order extensions of Bayesian networks (with the exception of [KP97,FGKP99]). In this context, we present for the first time how to calculate the *gradient* for a maximum likelihood estimation of the parameters of Bayesian logic programs. Together with the EM algorithm which we will present, this gives one a rich class of optimization techniques such as conjugate gradient and the possibility to speed up the EM algorithm, see e.g. [MK97],

Moreover, [KP97] rose the question whether techniques from inductive logic programming (ILP) could help to learn the logical component of first order probabilistic models. In [KDK00] we suggested that the ILP setting *learning from in-*

terpretations [DB93,DD97,BD98] is a good candidate for investigating this question. With this paper we would like to make our suggestions more concrete. We present a novel scheme to learn intensional clauses within Bayesian logic programs [KD00,KD01]. It combines techniques from inductive logic programming with techniques for learning Bayesian networks. More exactly, we will show that the *learning from interpretations* setting for inductive logic programming can be integrated with score-based Bayesian network learning techniques for learning Bayesian logic programs. Thus, we positively answer Koller and Pfeffer's question.

We proceed as follows. After briefly reviewing the framework of Bayesian logic programs in Section 2, we define the learning problem in Section 4. Based on that, we present a score-based greedy algorithm solving the problem in Sections 4 and 5. In Section 4, we introduce the scheme of the algorithm for structural learning of Bayesian logic programs, discuss it applied on a special class of propositional Bayesian logic programs, well-known under the name Bayesian networks, and applied on general Bayesian logic programs. In Section 5, we formulate the likelihood of the parameters of a Bayesian logic program given some data and, based on this, we present a gradient-based and an EM method to find that parameters which maximize the likelihood. Section 6 reports on first experiments. Before concluding the paper, we relate our approach to other work in Section 7.

We assume some familiarity with logic programming or Prolog (see e.g. [SS86,Llo89]) as well as with Bayesian networks (see e.g. [Pea91,CDLS99,Jen01]).

2. Bayesian Logic Programs

Throughout the paper we will use an example from genetics which is inspired by Friedman et al. [FGKP99]: "it is a genetic model of the inheritance of a single gene that determines a person's `X` blood type `bt(X)`. Each person `X` has two copies of the chromosome containing this gene, one, `mc(Y)`, inherited from her mother `m(Y,X)`, and one, `pc(Z)`, inherited from her father `f(Z,X)`." We will use $\mathbf{P}$ to denote a probability distribution, e.g. $\mathbf{P}(x)$, and the normal letter P to denote a probability value, e.g. $P(x = v)$, where v is a state of x.

2.1. Representation Language

The basic idea underlying our framework is that each Bayesian logic program specifies a (possibly infinite) Bayesian network, with one node for each (Bayesian) ground atom (see below). A Bayesian logic program B consist of two components:

- a *qualitative* or *logical* one, a set of Bayesian clauses (cf. below), and
- a *quantitative* one, a set of conditional probability distributions and combining rules (cf. below) corresponding to that logical structure.

Definition 1. (Bayesian Clause) A *Bayesian (definite) clause c is an expression of the form*

$$A \mid A_1, \ldots, A_n$$

where $n \geq 0$, the $A, A_1, \ldots, A_n$ are Bayesian atoms and all Bayesian atoms are (implicitly) universally quantified. We define $head(c) = A$ and $body(c) = \{A_1, \ldots, A_n\}$.

So, the differences between a *Bayesian clause* and a *logical* one are:

1. the atoms $p(t_1, ..., t_n)$ and predicates p arising are Bayesian, which means that they have an associated (finite) domain[1] $\mathrm{Dom}(p)$, and
2. we use " | " instead of ":-".

For instance, consider the Bayesian clause c

```
bt(X) | mc(X), pc(X).
```

where $\mathrm{Dom}(bt) = \{a, b, ab, 0\}$ and $\mathrm{Dom}(mc) = \mathrm{Dom}(pc) = \{a, b, 0\}$. It says that the blood type of a person X depends on the inherited genetical information of X. Note that the domain $\mathrm{Dom}(p)$ has nothing to do with the notion of a domain in the logical sense. The domain $\mathrm{Dom}(p)$ defines the states of random variables. Intuitively, a Bayesian predicate p generically represents a set of (finite) random variables. More precisely, each Bayesian ground atom g over p represents a (finite) random variable over the states $\mathrm{Dom}(g) := \mathrm{Dom}(p)$. E.g. $bt(ann)$ represents the blood type of a person named Ann as a random variable over the states $\{a, b, ab, 0\}$. Apart from that, most other *logical* notions carry over to Bayesian

[1] For the sake of simplicity we consider finite random variables, i.e. random variables having a finite set Dom of states. However, the ideas generalize to discrete and continuous random variables.

logic programs. So, we will speak of Bayesian predicates, terms, constants, substitutions, ground Bayesian clauses, Bayesian Herbrand interpretations etc. We will assume that all Bayesian clauses are range-restricted. A clause is *range-restricted* iff all variables occurring in the head also occur in the body. Range restriction is often imposed in the database literature; it allows one to avoid derivation of non-ground true facts.
In order to represent a probabilistic model we associate with each Bayesian clause c a conditional probability distribution $\mathit{cpd}(c)$ encoding $\mathbf{P}(\mathit{head}(c) \mid \mathit{body}(c))$. To keep the expositions simple, we will assume that $\mathit{cpd}(c)$ is represented as table, see Figure 1. More elaborate representations like decision trees or rules are also possible. The distribution $\mathit{cpd}(c)$ generically represents the conditional probability distributions of all ground instances $c\theta$ of the clause c. In general, one may have many clauses, e.g. clauses c_1 and the c_2

```
bt(X) | mc(X).
bt(X) | pc(X).
```

and corresponding substitutions θ_i that ground the clauses c_i such that $\mathit{head}(c_1\theta_1) = \mathit{head}(c_2\theta_2)$. They specify $\mathit{cpd}(c_1\theta_1)$ and $\mathit{cpd}(c_2\theta_2)$, but not the distribution required: $\mathbf{P}(\mathit{head}(c_1\theta_1) \mid \mathit{body}(c_1) \cup \mathit{body}(c_2))$. The standard solution to obtain the distribution required are so called *combining rules.*

Definition 2. (Combining Rule) A combining rule is a functions which maps finite sets of conditional probability distributions $\{\mathbf{P}(A \mid A_{i1}, \ldots, A_{in_i}) \mid i = 1, \ldots, m\}$ onto one (*combined*) conditional probability distribution $\mathbf{P}(A \mid B_1, \ldots, B_k)$ with $\{B_1, \ldots, B_k\} \subseteq \bigcup_{i=1}^{m}\{A_{i1}, \ldots, A_{in_i}\}$.

We assume that for each Bayesian predicate p there is a corresponding combining rule cr, such as *noisy or* (see e.g. [Jen01]) or *average*. The latter assumes $n_1 = \ldots = n_m$ and $\mathrm{Dom}(A_{ij}) = \mathrm{Dom}(A_{kj})$, and computes the average of the distributions over $\mathrm{Dom}(A)$ for each joint state over $\bigotimes_j \mathrm{Dom}(A_{ij})$.

To summarize, we could define Bayesian logic program in the following way:

Definition 3. (Bayesian Logic Program) A *Bayesian logic program B* consists of a (finite) set of Bayesian clauses. To each Bayesian clause c there is exactly one conditional probability distribution $\mathit{cpd}(c)$ associated, and for each Bayesian predicate p there is exactly one associated combining rule $cr(p)$.

```
m(ann,dorothy).
f(brian,dorothy).
pc(ann).
pc(brian).
mc(ann).
mc(brian).

mc(X)   | m(Y,X),mc(Y),pc(Y).
pc(X)   | f(Y,X),mc(Y),pc(Y).
bt(X)   | mc(X),pc(X).
```

(1)

$mc(X)$	$pc(X)$	$\mathbf{P}(bt(X))$
a	a	$(0.97, 0.01, 0.01, 0.01)$
b	a	$(0.01, 0.01, 0.97, 0.01)$
$\cdots$	$\cdots$	$\cdots$
0	0	$(0.01, 0.01, 0.01, 0.97)$

(2)

Figure 1. (1) The Bayesian logic program *bloodtype* encoding our genetic domain. To each Bayesian predicate, the identity is associated as combining rule. (2) A conditional probability distribution associated to the Bayesian clause `bt(X) | mc(X), pc(X)` represented as a table.

2.2. Declarative Semantics

The declarative semantics of Bayesian logic programs is given by the annotated *dependency graph*. The *dependency graph* $DG(B)$ is that directed graph whose nodes correspond to the ground atoms in the least Herbrand model $\mathrm{LH}(B)$ (cf. below). It encodes the *directly influenced by* relation over the random variables in $\mathrm{LH}(B)$:

> there is an edge from a node x to a node y if and only if there exists a clause $c \in B$ and a substitution θ, s.t. $y = head(c\theta)$, $x \in body(c\theta)$ and for all atoms z in $c\theta : z \in \mathrm{LH}(B)$.

The direct predecessors of a graph node x are called its parents, $\mathbf{Pa}(x)$. The Herbrand base $\mathrm{HB}(B)$ is the set of all random variables we can talk about. It is defined as if B were a logic program (cf. [Llo89]). The least Herbrand model $\mathrm{LH}(B) \subseteq \mathrm{HB}(B)$ consists of all *relevant* random variables, the random variables over which a probability distribution is well-defined by B, as we will see. It is the least fix point of the *immediate consequence operator* applied on the empty interpretation. Therefore, a ground atom which is true in the logical sense corresponds to a relevant random variables. Now, to each node x in $DG(B)$ we associate the combined conditional probability distribution which is the result of the combining rule $cr(p)$ of the corresponding Bayesian predicate p applied to the set of $cpd(c\theta)$'s where $head(c\theta) = x$ and $\{x\} \cup body(c\theta) \subseteq \mathrm{LH}(B)$. Thus, if $DG(B)$ is acyclic and not empty then it encodes a (possibly infinite) Bayesian network, because the least Herbrand model always exists and is unique. Therefore, the following independence assumption holds:

Independence Assumption 1. Each node x is independent of its non-descendants given a joint state of its parents $\mathbf{Pa}(x)$ in the dependency graph.

E.g. in the program in Figure 1, the random variable $bt(dorothy)$ is independent from $pc(brian)$ given a joint state of $pc(dorothy), mc(dorothy)$. Using this assumption the following proposition holds:

Proposition 2.1. *Let B be a Bayesian logic program. If*

1. $\mathrm{LH}(B) \neq \emptyset$,
2. *$DG(B)$ is acyclic, and*
3. *each node in $DG(B)$ is influenced by a finite set of random variables*

then B specifies a unique probability distribution $\mathbf{P}_B$ over $\mathrm{LH}(B)$.

Proof sketch [for a detailed proof see [KD01]]. The least Herbrand $\mathrm{LH}(B)$ always exists, is unique and countable. Thus, $DG(B)$ exists and is unique, and due to condition (3) the combined probability distribution for each node of $DG(B)$ is computable. Furthermore, because of condition (1) a total order π on $DG(B)$ exists, so that one can see B together with π as a stochastic process over $\mathrm{LH}(B)$. An induction "along" π together with condition 2 shows that the family of finite-dimensional distribution of the process is projective (cf. [Bau91]), i.e the joint probability density over each finite subset $s \subseteq \mathrm{LH}(B)$ is uniquely defined and $\int_y p(s, x = y)\, dy = p(s)$. Thus, the preconditions of *Kolmogorov's theorem* [Bau91, page 307] hold, and it follows that B given π specifies a probability density function p over $\mathrm{LH}(B)$. This proves the proposition because the total order π used for the induction is arbitrary.

A program B satisfying the conditions (1), (2) and (3) of proposition 2.1 is called *well-defined.* The program *bloodtype* in Figure 1 is an example of a well-defined Bayesian logic program. It encodes the regularities in our genetic example. Its grounded version, which is a Bayesian network, is shown in Figure 2. This illustrates that Bayesian networks [Pea91] are well-defined propositional Bayesian logic programs. Each node-parents pair uniquely specifies a propositional Bayesian clause; we associate the identity as combining rule to each predicate; the conditional probability distributions are those of the Bayesian network.

```
m(ann,dorothy).
f(brian,dorothy).
pc(ann).
pc(brian).
mc(ann).
mc(brian).
mc(dorothy) | m(ann, dorothy),mc(ann),pc(ann).
pc(dorothy) | f(brian, dorothy),mc(brian),pc(brian).
bt(ann)     | mc(ann), pc(ann).
bt(brian)   | mc(brian), pc(brian).
bt(dorothy) | mc(dorothy),pc(dorothy).
```

Figure 2. The grounded version of the Bayesian logic program of Figure 1. It (directly) encodes a Bayesian network.

Some interesting insights follow from the proof sketch. We interpreted a Bayesian logic program as a stochastic process. This places them in a wider context of what Cowell et. al. call *highly structured stochastic systems* (HSSS), cf. [CDLS99], because Bayesian logic programs represent discrete-time stochastic processes in a more flexible manner. Well-known probabilistic frameworks such as dynamic Bayesian networks, first order hidden Markov models or Kalman filters are special cases of Bayesian logic programs. Moreover, the proof in [KD01] indicates the important *support network* concept. Support networks are a graphical representation of the finite-dimensional distribution, cf. [Bau91], and are needed for the formulation of the likelihood function (see below) as well as for answering probabilistic queries in Bayesian logic programs.

Definition 4. (Support Network). The *support network* N of a variable $x \in \mathrm{LH}(B)$ is defined as the induced subnetwork of $S = \{x\} \cup \{y \mid y \in \mathrm{LH}(B)$ and y is influencing $x\}$. The support network of a finite set $\{x_1, \ldots, x_k\} \subseteq \mathrm{LH}(B)$ is the union of the networks of each single x_i.

Because we consider well-defined Bayesian logic programs, each $x \in \mathrm{LH}(B)$ is influenced by a finite subset of $\mathrm{LH}(B)$. So, the support network N of a finite set $\{x_1, \ldots, x_k\} \subseteq \mathrm{LH}(B)$ of random variables is always a finite Bayesian network and computable in finite time. The distribution factorizes in the usual way, i.e. $\mathbf{P}_N(x_1 \ldots, x_n) = \prod_{i=1}^{n} \mathbf{P}_N(x_i \mid \mathbf{Pa}\, x_i)$, where $\{x_1 \ldots, x_n\} = S$, and $\mathbf{P}(x_i \mid \mathbf{Pa}\, x_i)$ is the combined conditional probability distribution associated to x_i. Because N

models the finite-dimensional distribution specified by S, any interesting probability value over subsets of S is specified by N. For the proofs and an effective inference procedure (together with a Prolog implementation) we refer to [KD01].

3. The Learning Problem

So far, we have assumed that there is an expert who provides both the structure and the conditional probability distributions of the Bayesian logic program. This is not always easy. Often, there is no-one possessing necessary the expertise or knowledge. However, instead of an expert we may have access to data. In this section, we investigate and formally define the problem of learning Bayesian logic programs. While doing so, we exploit analogies with Bayesian network learning as well as with inductive logic programming.

3.1. Data Cases

In the last section, we have introduced Bayesian logic programs and argued that they contain two components, the quantitative (the combining rules and the conditional probability distributions) and the qualitative ones (the Bayesian clauses). Now, if we want to learn Bayesian logic programs, we need to employ data. Hence, we need to formally define the notions of a data case.

Let B be a Bayesian logic program consisting of the Bayesian clauses $c_1, \ldots, c_n$, and let $\mathbf{D} = \{D_1, \ldots, D_m\}$ be a set of data cases.

Definition 5. (Data Case) A data case $D_i \in \mathbf{D}$ for a Bayesian logic program B consists of a

logical part which is a Herbrand interpretation $Var(D_i)$ such that $Var(D_i) = \mathrm{LH}(B \cup Var(D_i))$, and a

probabilistic part which is a partially observed joint state of some variables, i.e. an assignment of values to some of the facts in $Var(D_i)$.

Examples of data cases are

$$\begin{aligned}
D_1 = \{& m(\mathit{cecily}, \mathit{fred}) = \mathit{true}, f(\mathit{henry}, \mathit{fred}) = ?, pc(\mathit{cecily}) = a, \\
& pc(\mathit{henry}) = b, pc(\mathit{fred}) = ?, mc(\mathit{cecily}) = b, mc(\mathit{henry}) = b, \\
& mc(\mathit{fred}) = ?, bt(\mathit{cecily}) = ab, bt(\mathit{henry}) = b, bt(\mathit{fred}) = ?\},
\end{aligned}$$

$$D_2 = \{m(ann, dorothy) = true, f(brian, dorothy) = true, pc(ann) = b, \\ mc(ann) =?, mc(brian) = a, mc(dorothy) = a, \\ pc(dorothy) = a, pc(brian) =?, bt(ann) = ab, bt(brian) =?, \\ bt(dorothy) = a\},$$

where '?' stands for an unobserved state. Notice that – for ease of writing – we merged the two components of a data case into one. Indeed, the *logical part* of a data case $D_i \in \mathbf{D}$, denoted as $Var(D_i)$, is a Herbrand interpretation, such as

$$\begin{aligned} Var(D_1) = \{&m(cecily, fred), f(henry, fred), pc(cecily), pc(henry), \\ &pc(fred), mc(cecily), mc(henry), mc(fred), bt(cecily), \\ &bt(henry), bt(fred)\}, \\ Var(D_2) = \{&m(ann, dorothy), f(brian, dorothy), pc(ann), \\ &mc(ann), mc(brian), mc(dorothy), pc(dorothy), \\ &pc(brian), bt(ann), bt(brian), bt(dorothy)\}, \end{aligned}$$

satisfy this logical property w.r.t. the target Bayesian logic program B

```
mc(X)  | m(Y,X),mc(Y),pc(Y).
pc(X)  | f(Y,X),mc(Y),pc(Y).
bt(X)  | mc(X),pc(X).
```

Indeed, $Var(B \cup Var(D_i)) = Var(D_i)$ for all $D_i \in \mathbf{D}$.

So, the logical components of the data cases should be seen as the least Herbrand models of the target Bayesian logic program. They specify different sets of *relevant* random variables, depending on the given "extensional context". If we accept that the genetic laws are the same for both families then a learning algorithm should find regularities among the Herbrand interpretations that can be to compress the interpretations. The key assumption underlying any inductive technique is that the rules that are valid in one interpretation are likely to hold for any interpretation. This is exactly what the *learning from interpretations* in inductive logic programming [DD97,BD98] is doing. Thus, we will adapt this setting for learning the structure of the Bayesian logic program, cf. Section 4.

There is one further logical constraints to take into account while learning Bayesian logic programs. It is concerned with the acyclicity requirement (cf. property 2 in proposition 2.1) imposed on Bayesian logic programs. Thus, we require that for each $D_i \in \mathbf{D}$ the induced Bayesian network over $\mathrm{LH}(B \cup \mathit{Var}(D_i))$ has to be acyclic.

At this point, the reader should also observe that we require that the logical part of a data case is a *complete* model of the target Bayesian logic program and not a *partial* one[2]. This is motivated by 1) Bayesian network learning and 2) the problems with learning from partial models in inductive logic programming. First, data cases as they have been used in Bayesian network learning are the propositional equivalent of the data cases that we introduced above. Indeed, if we have a Bayesian network B over the propositional Bayesian predicates $\{p_1, ..., p_k\}$ then $\mathrm{LH}(B) = \{p_1, ..., p_k\}$ and a data case would assign values to some of the predicates in B. This also shows that the second component of a data case is pretty standard in the Bayesian network literature. Second, it is well-known that learning from partial models is harder than learning from complete models (cf. [De 97]). More specifically, learning from partial models is akin to multiple predicate learning, which is a very hard problem in general. These two points also clarify why the semantics of the set of relevant random variables coincided with the least Herbrand domain and at the same time why we do not restrict the domain of Bayesian predicates to $\{\mathit{true}, \mathit{false}\}$.

Before we are able to fully specify the problem of learning Bayesian logic programs, let us introduce the hypothesis space and scoring functions.

3.2. The Hypothesis Space

The *hypothesis space* $\mathcal{H}$ explored consists of Bayesian logic programs, i.e. finite set of Bayesian clauses to which conditional probability distributions are associated. More formally, let $\mathcal{L}$ be the language, which determines the set $\mathcal{C}$ of clauses that can be part of a hypothesis. It is common to impose syntactic restrictions on the space $\mathcal{H}$ of hypotheses.

> *Language Assumption*: In this paper, we assume that the alphabet of $\mathcal{L}$ only contains constant and predicate symbols that occur in one of the data cases, and we restrict $\mathcal{C}$ to range-restricted, constant-free clauses containing maximum $k = 3$ atoms in the body. Furthermore, we assume that the combining rules associated to the Bayesian predicates are given.

[2] Partial models specify the truth-value (false or true) of *some* of the elements in the Herbrand Base.

E.g. given the data cases D_1 and D_2, $\mathcal{C}$ looks like

```
mc(X)  | m(Y,X).
mc(X)  | mc(X).
mc(X)  | pc(X).
mc(X)  | m(Y,X),mc(Y).
...
pc(X)  | f(Y,X),mc(Y),pc(Y).
...
bt(X)  | mc(X),pc(X).
```

Not every element $H \in \mathcal{H}$ has to be a candidate. The logical parts of the data cases constraint the set of possible candidates. To be a candidate, H has to be

- (logically) valid on the data, and
- acyclic on the data i.e. the induced Bayesian network over $\mathrm{LH}(H \cup \mathit{Var}(D_i))$ has to be acyclic.

E.g. given the data cases D_1 and D_2, the Bayesian clause

```
mc(X) | mc(X)
```

is not included in any candidate, because the Bayesian network induced over the data cases would be cyclic.

3.3. Scoring Function

So far, we mainly exploit the logical part of the data cases. The probabilistic part of the data cases are partially observed joint states. They induce a joint distribution over the random variables of the logical parts of the data cases. A candidate $H \in \mathcal{H}$ should reflect this distribution. We assume that there is a scoring function $\mathit{score}_{\mathbf{D}} : \mathcal{H} \mapsto \mathbb{R}$ which expresses how well a given candidate H fits the data $\mathbf{D}$. Examples of scoring functions are the likelihood (see Section 5) or the *minimum description length* score (which bases on the likelihood).

Putting all together, we can define the basic learning problem as follows:

Definition 6. (Learning Problem) **Given** a set $\mathbf{D} = \{D_1, \ldots, D_m\}$ of data cases, a set $\mathcal{H}$ of sets of Bayesian clauses according to some language bias, and a scoring function $\mathit{score}_{\mathbf{D}} : \mathcal{H} \mapsto \mathbb{R}$, **find** a hypothesis $H^* \in \mathcal{H}$ such that for all $D_i \in \mathbf{D}$: $\mathrm{LH}(H^* \cup \mathit{Var}(D_i)) = \mathit{Var}(D_i)$, H^* is acyclic on the data, and H^* maximizes $\mathit{score}_{\mathbf{D}}$.

As usual, we assume the all data cases are idependently sampled from identical distributions. In the following section we will present an algorithm solving the learning problem.

4. An Algorithm for Learning Bayesian Logic Programs

An algorithm solving the learning problem is given in Table 1. First, we will illustrate the algorithm for a special class of Bayesian logic programs: Bayesian networks. For Bayesian networks, the algorithm coincides with score-based methods for learning within Bayesian networks which are proven to be useful by the UAI community (see e.g. [Hec95]). Therefore, an extension to the first order case seems reasonable. Then, we will show that the algorithm works for first order Bayesian logic programs, too.

For the sake of readability, we assume the existence of a method to compute the parameters maximizing the score given a candidate and data cases. Methods to do this will be discussed in Section 5. They assume that the combining rules are decomposable (see Section 5.2.3). Furthermore we will discuss the basic framework only. Extensions are possible.

4.1. A Propositional Case: Bayesian Networks

Here we will show that Algorithm 1 is a generalization of score-based techniques for structural learning of Bayesian networks (see e.g. [Hec95]).

Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be a fixed set of random variables. The set $\mathbf{x}$ corresponds to a least Herbrand model of an unknown propositional Bayesian logic program representing a Bayesian network. The probabilistic dependencies among the relevant random variables are not known, i.e. the propositional Bayesian clauses are unknown. Therefore, we have to select such a propositional Bayesian logic program as a *candidate* and estimate its parameters. Assume the data cases $\mathbf{D} = \{D_1, \ldots, D_m\}$ look like

$$\{m(ann, dorothy) = true, f(brian, dorothy) = true, pc(ann) = a,$$
$$mc(ann) = ?, mc(brian) = ?, mc(dorothy) = a, mc(dorothy) = a,$$
$$pc(brian) = b, bt(ann) = a, bt(brian) = ?, bt(dorothy) = a\}$$

```
function BASIC-GREEDY(H, D) returns a modified Bayesian logic program
   inputs:  H, a (valid) Bayesian logic program
            D, a finite set of data cases

S(H) := score_D(H)
repeat
   H' := H
   S(H') := S(H)
   for each H'' ∈ ρ_g(H') ∪ ρ_s(H')
      if H'' is (logically) valid on D
         if the Bayesian network induced by H'' on the data are acyclic
            if score_D(H'') > S(H)
                  H := H''
                  S(H) := S(H'')
until S(H') = S(H)
return H
```

Table 1

A greedy algorithm for searching the structure of Bayesian logic programs. Note that we have omitted the initialization of the conditional probability distributions associated to Bayesian clauses.

which is a data case for the Bayesian network in Figure 2. Note, that the atoms have to be interpreted as propositions. Each H in the hypothesis space $\mathcal{H}$ is a Bayesian logic program consisting of n propositional clauses: for each $x_i \in \mathbf{x}$ a single clause c with $head(c) = x_i$ and $body(c) \subseteq \mathbf{x} \setminus \{x_i\}$. To traverse $\mathcal{H}$ we specify two *refinement* operators $\rho_g : \mathcal{H} \mapsto 2^{\mathcal{H}}$ and $\rho_s : \mathcal{H} \mapsto 2^{\mathcal{H}}$, that take a hypothesis and modify it to produce a set of possible candidates. In the case of Bayesian networks the operator $\rho_g(H)$ deletes a Bayesian proposition from the body of a Bayesian clause $c_i \in H$, and the operator $\rho_s(H)$ adds a Bayesian proposition to the body of $c_i \in H$ (cf Figure 3). The search algorithm performs an greedy, informed search in $\mathcal{H}$ based on $score_{\mathbf{D}}$.

As a simple illustration we consider a greedy hill-climbing algorithm incorporating $score_{\mathbf{D}}(H) := LL(\mathbf{D}, H)$, the log-likelihood of the data $\mathbf{D}$ given a candidate structure H with the best parameters. We pick an initial candidate $S \in \mathcal{H}$ as starting point (e.g. the set of all propositions) and compute the likelihood $LL(\mathbf{D}, S)$ with the best parameters. Then, we use $\rho(S)$ to compute the legal "neighbours" (candidates being acyclic) of S in $\mathcal{H}$ and score them. All neighbours are valid (see below for a definition of validity). E.g. replacing `pc(dorothy)` with `pc(dorothy) | pc(brian)` gives such a "neighbour". We take that $S' \in \rho(S)$

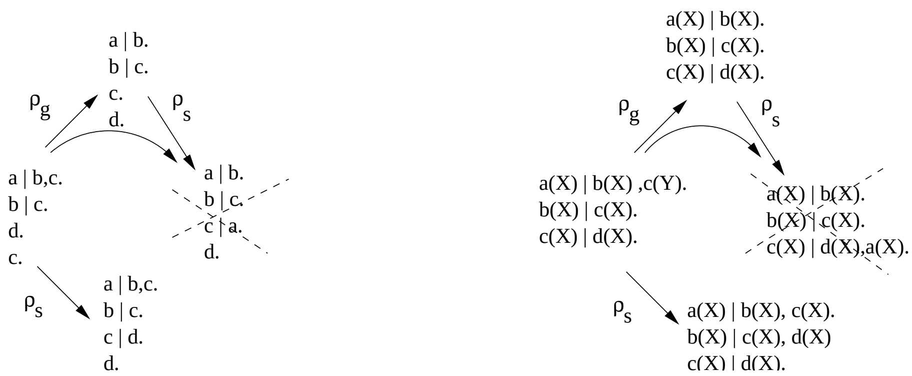

Figure 3. (1) The use of refinement operators during structural search for Bayesian networks. We can add (ρ_s) a proposition to the body of a clause or delete (ρ_g) it from the body. (2) The use of refinement operators during structural search within the framework of Bayesian logic programs. We can add (ρ_s) an atom to the body of a clause or delete (ρ_g) it from the body. Candidates crossed out in (1) and (2) are illegal because they are cyclic.

with the best improvements in the score. The process is continued until no improvements in score are obtained.

4.2. The First Order Case: Bayesian Logic Programs

Here, we will explain the ideas underlying our algorithm in the first order case. The key difference with the propositional case are

1. that some Bayesian logic programs will be logically invalid (see below for an example), and
2. that the traditional first order refinement operators must be used.

Difference 1 is the most important one, because it determines the hypotheses that are candidate Bayesian logic programs. To account for this difference, two modifications of the traditional Bayesian network algorithm are needed.

The first modification concerns the initialisation phase where we have to choose a logically valid, acyclic Bayesian logic program. Such a program can be computed using a CLAUDIEN like procedure ([DB93,DD97,BD98]). CLAUDIEN is an ILP-program that computes a logically valid hypothesis H from a set of data cases. Furthermore, all clauses in H will be maximally general (w.r.t. θ-subsumption), and CLAUDIEN will compute all such clauses (within $\mathcal{L}$). This means non of the clauses in H can be generalized without violating the logical validity requirement (or leaving $\mathcal{L}$). Consider again the data cases

$$
\begin{aligned}
D_1 = \{& m(\mathit{cecily}, \mathit{fred}) = \mathit{true}, f(\mathit{henry}, \mathit{fred}) =?, pc(\mathit{cecily}) = a, \\
& pc(\mathit{henry}) = b, pc(\mathit{fred}) =?, mc(\mathit{cecily}) = b, mc(\mathit{henry}) = b, \\
& mc(\mathit{fred}) =?, bt(\mathit{cecily}) = ab, bt(\mathit{henry}) = b, bt(\mathit{fred}) =?\}, \\
D_2 = \{& m(\mathit{ann}, \mathit{dorothy}) = \mathit{true}, f(\mathit{brian}, \mathit{dorothy}) = \mathit{true}, pc(\mathit{ann}) = b, \\
& mc(\mathit{ann}) =?, mc(\mathit{brian}) = a, mc(\mathit{dorothy}) = a, \\
& pc(\mathit{dorothy}) = a, pc(\mathit{brian}) =?, bt(\mathit{ann}) = ab, bt(\mathit{brian}) =?, \\
& bt(\mathit{dorothy}) = a\},
\end{aligned}
$$

The clause `bt(X)` is not a member of $\mathcal{L}$. The clause `bt(X) | mc(X),pc(X)` is valid but not maximally general because the literal `pc(X)` can be deleted without violating the logical validity requirement. Any hypothesis including `m(X,Y) | mc(X),pc(Y)` would be logically invalid because *cecily* is not the mother of *henry*. Examples of maximally general clauses are

```
mc(X)  | m(Y, X).
pc(X)  | f(Y, X).
bt(X)  | mc(X).
bt(X)  | pc(X).
...
```

Roughly speaking, CLAUDIEN works as follows (for a detailed discussion we refer to [DD97]). It keeps track of a list of candidate clauses Q, which is initialized to the maximally general clause (in $\mathcal{L}$). It repeatedly deletes a clause c from Q, and tests whether c is valid on the data. If it is, c is added to the final hypothesis, otherwise, all maximally general specializations of c (in $\mathcal{L}$) are computed (using a so-called refinement operator ρ, see below) and added back to Q. This process continues until Q is empty and all relevant parts of the search space have been considered. The clauses generated by CLAUDIEN can be used as an initial hypothesis.

In the experiments, for each predicate, we selected one of the clause generated by CLAUDIEN for inclusion in the initial hypothesis such that the valid Bayesian logic program was also acyclic on the data cases (see below). An initial hypothesis is e.g.

```
m(ann,dorothy).   m(cecily,fred).
f(brian,dorothy). f(henry,fred).
pc(ann).  pc(brian).  pc(cecily). pc(henry).
mc(ann).  mc(brian).  mc(cecily). mc(henry).
mc(dorothy) | m(ann,dorothy).   mc(fred) | m(cecily,fred).
pc(dorothy) | f(brian,dorothy). pc(fred) | f(cecily,fred).
bt(ann)     | mc(ann).      bt(brian)   | mc(brian).
bt(dorothy) | mc(dorothy). bt(cecily)  | mc(cecily).
bt(henry)   | mc(henry).   bt(fred)    | mc(fred).
```

Figure 4. The support network induced by the initial hypothesis S (see text) over the the data cases D_1 and D_2.

```
mc(X)   | m(Y, X).
pc(X)   | f(Y, X).
bt(X)   | mc(X).
```

The second modification concerns filtering out those Bayesian logic programs that are logically invalid during search. This is realized by the first **if**-condition in the loop. The second **if**-condition tests whether cyclic dependencies are induced on the data cases. This can be done in time $O(s \cdot r^3)$ where r is the number of random variables of the largest data case in $\mathbf{D}$ and s is the number of clauses in H. To do so, we build the Bayesian networks induced by H over each $Var(D_i)$ by computing the ground instances for each clause $c \in H$ where the ground atoms are members of $Var(D_i)$. Thus, ground atoms, which are not appearing as a head atom of a valid ground instance, are apriori nodes, i.e. nodes with an empty parent set. This takes $O(s \cdot r_i^3)$. Then, we test in $O(r_i)$ for a topological order of the nodes in the induced Bayesian network. If it exists, then the Bayesian network is acyclic. Otherwise, it is cyclic. Figure 4 shows the support network induced by the initial hypothesis over D_1 and D_2.

For Difference 2, i.e. the refinements operators, we employ the traditional ILP refinement operators. In our approach we use the two refinement operators $\rho_s : 2^{\mathcal{H}} \mapsto \mathcal{H}$ and $\rho_g : 2^{\mathcal{H}} \mapsto \mathcal{H}$. The operator $\rho_s(H)$ adds constant-free atoms to the body of a single clause $c \in H$, and $\rho_g(H)$ deletes constant-free atoms from the body of a single clause $c \in H$. Figure 3 shows the different refinement operators for the first order case and the propositional case for learning Bayesian networks. Instead of adding (deleting) propositions to (from) the body of a clause, they add (delete) according to our language assumption constant-free atoms. Furthermore, Figure 3 shows that using the refinement operators each hypothesis can in principle be reached.

Finally, we need to mention that whereas the maximal general clauses are the most interesting ones from the logical point of view, this is not necessarily the case from the probabilistic point of view. E.g. having data cases D_1 and D_2 (see Section 3.1), the initial candidate S

```
mc(X)   | m(Y, X).
pc(X)   | f(Y, X).
bt(X)   | mc(X).
```

is likely not to score maximally on the data cases. E.g. the blood type does not depend on the fatherly genetical information.

As a simple instantiation of Algorithm 1 we consider a greedy hill-climbing algorithm incorporating $score_{\mathbf{D}}(H) := LL(\mathbf{D}, H)$ with $\mathbf{D} = \{D_1, D_2\}$. It takes $S \in \mathcal{H}$ (see above) as starting point and computes $LL(\mathbf{D}, S)$ with the best parameters. Then, we use $\rho_s(S)$ and $\rho_g(S)$ to compute the legal "neighbours" of S in $\mathcal{H}$ and score them. E.g. one such a "neighbour" is given by replacing `bt(X) | mc(X)` with `bt(X) | mc(X), pc(X)`. Let S' be that valid and acyclic neighbour which scores best. If $LL(\mathbf{D}, S) < LL(\mathbf{D}, S')$, then we take S' as new hypothesis. The process is continued until no improvements in score are obtained.

4.3. Discussion

The algorithm presented serves as a basic, unifying framework. Several extensions and modifications based on ideas developed in both fields, inductive logic programming and Bayesian networks are possible. These include: lookaheads, background knowledge, mode declarations and improved scoring functions. Let us briefly address some of these:

Lookahead: In some cases, an atom might never be chosen by our algorithm because it will not – in itself – result in a better score. However, such an atom, while not useful in itself, might introduce new variables that make a better score possible by adding another atom later on. Within inductive logic programming this is solved by allowing the algorithm to *look ahead* in the search space. Immediately after refining a clause by putting some atom A into the body, the algorithm checks whether any other atom involving some variable of A results in a better score [BD97]. The same problem is encountered when learning Bayesian networks [XWC96].

Background knowledge: Inductive logic programming emphasizes background knowledge, i.e. predefined, fixed regularities which are common to all examples. Background knowledge can be incorporated into our approach in the following way. It is expressed as a fixed Bayesian logic program BK. Now, we search for a candidate H^* which is together with BK acyclic on the data such that for all $D_i \in \mathbf{D}$: $\mathrm{LH}(BK \cup H^* \cup \mathit{Var}(D_i)) = \mathit{Var}(D_i)$, and $BK \cup H^*$ matches the data $\mathbf{D}$ best according to $score_{\mathbf{D}}$. Therefore, all the Bayesian facts that can be derived from the background knowledge and an example are part of the corresponding "extended" example. This is particularly interesting to specify deterministic knowledge as in inductive logic programming. In [KD01], we showed how pure Prolog programs can be represented as Bayesian logic programs w.r.t. the conditions 1,2 and 3 of Proposition 1.

Improved scoring function: Using the likelihood directly as scoring function, score-based algorithm to learn Bayesian networks prefer fully connected networks. To overcome the problem advanced scoring functions were developed. On of these is the *minimum description length* (MDL) score which trades off the fit to the data with complexity. In the context of learning Bayesian networks, the whole Bayesian network is encoded to measure the compression [LB94]. In the context of learning clause programs, other compression measure were investigated such as the average length of proofs [SMB94]. For Bayesian logic programs, a combination of both seems to be appropriate.

Finally, an extension for learning predicate definitions consisting of more than one clause is in principle possible. The refinement operators could be modified in such a way that for a clause $c \in H'$ with head predicate p another (valid) clause c' (e.g. computed by CLAUDIEN) with head predicate p is added or deleted.

5. Learning Probabilities in a Bayesian Logic Program

So far, we have assumed that there is a method estimating the parameters of an candidate program given data. In this section, we show how to learn the quantitative component of a Bayesian logic program, i.e. the conditional probability distributions. The learning problem can be stated as follows:

Definition 7. (Learning Problem) **Given** a set $\mathbf{D} = \{D_1, \ldots, D_m\}$ of data cases [3], a set H of Bayesian clauses according to some language bias, which is logically valid and acyclic on the data, and a scoring function $score_{\mathbf{D}} : \mathcal{H} \mapsto \mathbb{R}$, **find** the parameters of H maximizing $score_{\mathbf{D}}$.

We will concentrate on the classical *maximum likelihood estimation* (MLE) approach.

5.1. Maximum Likelihood Estimation

Let B be a Bayesian logic program consisting of the Bayesian clauses $c_1, \ldots, c_n$, and let $\mathbf{D} = \{D_1, \ldots, D_m\}$ be a set of data cases. The parameters

$$cpd(c_i)_{jk} = P(= u_j \mid \mathbf{u}_k),$$

where $u_j \in \mathrm{Dom}(head(c_i))$ and $\mathbf{u}_j \in \mathrm{Dom}(body(c_i))$, affecting the associated conditional probability distributions $cpd(c_i)$ constitute the set $\boldsymbol{\lambda} = \bigcup_{i=1}^{n} cpd(c_i)$. The version of B where the parameters are set to $\boldsymbol{\lambda}$ is denoted by $B(\boldsymbol{\lambda})$, and as long as no ambiguities occur we will not distinguish between the parameters $\boldsymbol{\lambda}$ themselves and a particular instance of them. Now, the likelihood $L(\mathbf{D}, \boldsymbol{\lambda})$ is the probability of the data $\mathbf{D}$ as a function of the unknown parameters $\boldsymbol{\lambda}$:

$$L(\mathbf{D}, \boldsymbol{\lambda}) := P_B(\mathbf{D} \mid \boldsymbol{\lambda}) = P_{B(\boldsymbol{\lambda})}(\mathbf{D}). \tag{5.1}$$

Thus, the search space $\mathcal{H}$ is spanned by the product space over the possible values of $\lambda(c_i)$ and we seek to find the parameter values $\boldsymbol{\lambda}^*$ that maximize the likelihood, i.e.

$$\boldsymbol{\lambda}^* = \max_{\boldsymbol{\lambda} \in \mathcal{H}} P_{B(\boldsymbol{\lambda})}(\mathbf{D}).$$

Usually, B specifies a distribution over a (countably) infinite set of random variables namely $\mathrm{LH}(B)$ and hence we cannot compute $P_{B(\boldsymbol{\lambda})}(\mathbf{D})$ by considering the whole dependency graph. But as we have argued at the end of the preceding section it is sufficient to consider the support network $N(\boldsymbol{\lambda})$ of the random variables occurring in $\mathbf{D}$ to compute $P_{B(\boldsymbol{\lambda})}(\mathbf{D})$. Thus, remembering that the logarithm is monotone

$$\boldsymbol{\lambda}^* = \max_{\boldsymbol{\lambda} \in \mathcal{H}} \log P_{N(\boldsymbol{\lambda})}(\mathbf{D}), \tag{5.2}$$

[3] Given a well-defined Bayesian network B, we can weak definition 5 of data cases if we only estimate the parameters and do not learn the structure. The random variables $Var(D_i)$ are a subset of $\mathrm{LH}(B)$.

so $score_{\mathbf{D}}(B) = \log P_{N(\boldsymbol{\lambda})}(\mathbf{D}) =: LL(\mathbf{D}, B)$.

To summarize, we have expressed the original problem in terms of the maximum likelihood parameter estimation problem of Bayesian networks. However, we need to to take into account that

1. Some of the nodes in $N(\boldsymbol{\lambda})$ are hidden, that is, their values are not observed in $\mathbf{D}$.
2. We are not interested in the conditional probability distributions associated to ground instances of Bayesian clauses, but in those associated to the Bayesian clauses themselves.
3. Not only $L(\mathbf{D}, \boldsymbol{\lambda})$ but also $N(\boldsymbol{\lambda})$ itself depends on the data, i.e. the data cases determine the subnetwork of $DG(B)$ which is sufficient to calculate the likelihood.

Due to the first point, some random variables are not observed. In the presence of missing data, the maximum likelihood estimate for Bayesian networks typically cannot be written in closed form. It is a numerical optimization problem, and all known algorithms involve nonlinear optimization and multiple calls to a Bayesian network inference procedure as a subroutine. The most prominent techniques within Bayesian networks are the EM (Expectation and Maximization) algorithm [DLR77,Lau95] and gradient-based approaches [BKRK97,Jen99,Jen01]. The second point indicates that both types of techniques can not directly be applied. In the next two subsections we will show how to adapt gradient-based approaches and EM in the context of Bayesian logic programs. The third point may affect the generalization performance of the learned program, but it is a similar situation as for "unrolling" dynamic Bayesian networks [DK88] or recurrent neural networks [WZ95].

5.2. Gradient-based approach

Gradient ascent, also known as *hill climbing*, is a classical method for finding a maximum of an evaluation function. Here, one computes the *gradient* vector $\nabla_{\boldsymbol{\lambda}}$ of partial derivatives with respect to the parameters of the conditional probability distributions at a given point $\boldsymbol{\lambda} \in \mathcal{H}$. Then, one takes a small step in the direction of the gradient to the point $\boldsymbol{\lambda} + \alpha \nabla_{\boldsymbol{\lambda}}$ where α is the step-size parameter. The algorithm will converge to a local maximum for small enough α. Thus, to apply gradient ascent to Bayesian logic programs, we have to compute the partial derivatives of $P_{N(\boldsymbol{\lambda})}(\mathbf{D})$ with respect to some particular parameter $cpd(c_i)_{jk}$.

5.2.1. Computing the Gradient

We will now adapt [BKRK97]'s solution for dynamic Bayesian networks based on the chain rule of differentiation. For simplicity, we fix the current instantiation of the parameters $\boldsymbol{\lambda}$ and, hence, we write B and $N(\mathbf{D})$. Applying the chain rule to (5.2) yields

$$\frac{\partial \log P_N(\mathbf{D})}{\partial cpd(c_i)_{jk}} = \sum_{\substack{\text{subst. } \theta \text{ s.t.} \\ support(c_i\theta)}} \frac{\partial \log P_N(\mathbf{D})}{\partial cpd(c_i\theta)_{jk}}$$

where θ refers to grounding substitutions and $support(c_i\theta)$ is true iff $\{head(c_i\theta)\} \cup body(c_i\theta) \subset N$. Assuming that the data cases $D_l \in \mathbf{D}$ are independently sampled from the same distribution we can separate the contribution of the different data cases to the partial derivative of a single ground instance $c\theta$:

$$\begin{aligned} \frac{\partial \log P_N(\mathbf{D})}{\partial cpd(c_i\theta)_{jk}} &= \frac{\partial \log \prod_{l=1}^{m} P_N(D_l)}{\partial cpd(c_i\theta)_{jk}} \\ &= \sum_{l=1}^{m} \frac{\partial \log P_N(D_l)}{\partial cpd(c_i\theta)_{jk}} \\ &= \sum_{l=1}^{m} \frac{\partial \log P_N(D_l) / \partial cpd(c_i\theta)_{jk}}{P_N(D_l)} \end{aligned} \tag{5.3}$$

In order to get computations local to the parameter $cpd(c_i\theta)_{jk}$ we introduce the variables $head(c_i\theta)$ and $body(c_i\theta)$ into (5.3) and average over their possible values:

$$\begin{aligned} & \frac{\partial \log P_N(D_l)}{\partial cpd(c_i\theta)_{jk}} \\ &= \frac{\partial}{\partial cpd(c_i\theta)_{jk}} \Big(\sum_{j',k'} P_N(D_l \mid head(c_i\theta) = u_{j'}, body(c_i\theta) = \mathbf{u}_{k'}) \cdot \\ & \qquad\qquad P_N(head(c_i\theta) = u_{j'}, body(c_i\theta) = \mathbf{u}_{k'}) \Big) \\ &= \frac{\partial}{\partial cpd(c_i\theta)_{jk}} \Big(\sum_{j',k'} P_N(D_l \mid head(c_i\theta) = u_{j'}, body(c_i\theta) = \mathbf{u}_{k'}) \cdot \\ & \qquad\qquad P_N(head(c_i\theta) = u_{j'} \mid body(c_i\theta) = \mathbf{u}_{k'}) \cdot P_N(body(c_i\theta) = \mathbf{u}_{k'}) \Big) \end{aligned}$$

where $u_j \in \mathrm{Dom}(head(c_i))$, $\mathbf{u}_k \in \mathrm{Dom}(body(c_i))$ and j, k refer to the corresponding entries in $cpd(c_i)$, respectively $cpd(c_i\theta)$. Now, $cpd(c_i\theta)_{jk}$ appears only in linear form. Moreover, it appears only when $j' = j$, and $k' = k$. Hence

$$
\begin{aligned}
&\frac{\partial \log P_N(D_l)/\partial cpd(c_i\theta)_{jk}}{P_N(D_l)} \\
&= \frac{P_N(D_l \mid head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k) \cdot P_N(body(c_i\theta) = \mathbf{u}_k)}{P_N(D_l)} \\
&= \frac{P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l) \cdot P_N(D_l) \cdot P_N(body(c_i\theta) = \mathbf{u}_k)}{P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k) \cdot P_N(D_l)} \\
&= \frac{P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)}{P_N(head(c_i\theta) = u_j \mid body(c_i\theta) = \mathbf{u}_k)} \\
&= \frac{P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)}{cpd(c_i\theta)_{jk}}
\end{aligned}
$$

Combining all these, we obtain

$$
\begin{aligned}
&\frac{\partial \log P_N(\mathbf{D})}{\partial cpd(c_i)_{jk}} \\
&= \sum_{\substack{\text{subst. } \theta \text{ with} \\ support(c_i\theta)}} \sum_{l=1}^{m} \frac{P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)}{cpd(c_i\theta)_{jk}}
\end{aligned}
\tag{5.4}
$$

where

$$
\sum_{l=1}^{m} P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l) \tag{5.5}
$$

specifies the *expected counts* of the joint state $head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k$ given the data. Equation (5.4) shows that $P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)$ is all what is needed. This can essentially be computed using any standard Bayesian network inference engine. The last equation differs from the one for Bayesian networks given in [BKRK97] in that we sum over all ground instances of a Bayesian clause holding in the data. A simplified skeleton of a gradient-based algorithm is shown in Table 2. In general, it is subject to local maxima, plateaux and ridges. Well-known techniques such as random restarts or random perturbations are possibilities to avoid them.

Before showing how to adapt the EM algorithm, we have to explain two points which we have left out so far for the sake of simplicity: Constraint satisfaction and decomposable combining rules.

Table 2

A simplified skeleton of the algorithm for *adaptive Bayesian logic programs.*

```
function BASIC-ABLP(B, D) returns a modified Bayesian logic program
   inputs:   B, a Bayesian logic program; associated cpds are parameterized by λ
             D, a finite set of data cases

λ ←INITIALPARAMETERS
N ←SUPPORTNETWORK(B, D)
repeat until Δλ ≈ 0
     Δλ ← 0
     set associated conditional probability distribution of N according to λ
     for each D_l ∈ D
        set the evidence in N from D_l
        for each clause c ∈ B
              for each ground instance cθ s.t. {head(cθ)} ∪ body(cθ) ⊂ N
                     for each single parameter cpd(cθ)_jk
                            Δcpd(c)_jk ← Δcpd(c)_jk + (∂ log P_N(D_l)/∂cpd(cθ)_jk)
     Δλ ←PROJECTIONONTOCONSTRAINTSURFACE(Δλ)
     λ ← λ + α · Δλ
return B
```

5.2.2. Constraint Satisfaction

In the problem at hand, the gradient ascent has to be modified to take into account the constraint that the parameter vector $\boldsymbol{\lambda}$ consists of probability values, i.e. $cpd(c_i)_{jk} \in [0,1]$ and $\sum_j cpd(c_i)_{jk} = 1$. There are two [BKRK97] ways to enforce this:

1. Projecting the gradient onto the constraint surface (as used to formulate the algorithm in Table 2), and
2. reparameterizing the problem.

In the experiments, we chose the latter approach using

$$cpd(c_i)_{jk} = \frac{\beta_{ijk}^2}{\sum_l \beta_{ilk}^2}$$

because the reparameterized problem is fully unconstrained, and projection is unnecessary. Applying the chain rule of derivatives yields

$$\frac{\partial log P_N(\mathbf{D})}{\partial \beta_{ijk}} = \sum_{i'j'k'} \frac{\partial log P_N(\mathbf{D})}{\partial cpd(c_{i'})_{j'k'}} \cdot \frac{\partial cpd(c_{i'})_{j'k'}}{\partial \beta_{ijk}} \tag{5.6}$$

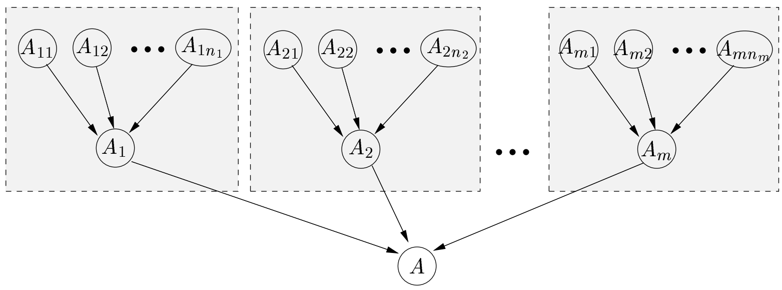

Figure 5. The scheme of decomposable combining rules. Each rectangle corresponds to a ground instance of a Bayesian clause (cf. definition of a combining rule). The node A is a deterministic node.

Since $\partial cpd(c_{i'})_{j'k'}/\partial\beta_{ijk} = 0$ unless $i = i'$ and $j = j'$, equation (5.6) simplifies to

$$\begin{aligned}\frac{\partial logP_N(\mathbf{D})}{\partial\beta_{ijk}} &= \sum_{j'} \frac{\partial logP_N(\mathbf{D})}{\partial cpd(c_i)_{j'k}} \cdot \frac{\partial cpd(c_i)_{j'k}}{\partial\beta_{ijk}} \\ &= \sum_{j'} \frac{\partial logP_N(\mathbf{D})}{\partial cpd(c_i)_{j'k}} \cdot \frac{\frac{\partial\beta^2_{ij'k}}{\partial\beta_{ijk}} \sum_l \beta^2_{ilk} - \frac{\partial\sum_l \beta^2_{ilk}}{\partial\beta_{ijk}}\beta^2_{ij'k}}{(\sum_l \beta^2_{ilk})^2} \\ &= \frac{2\beta_{ijk}\frac{\partial logP_N(\mathbf{D})}{\partial cpd(c_i)_{jk}} \sum_l \beta^2_{ilk} - \sum_l \frac{\partial logP_N(\mathbf{D})}{\partial cpd(c_i)_{lk}}\beta^2_{ilk}}{(\sum_l \beta^2_{ilk})^2}.\end{aligned}$$

5.2.3. Decomposable Combining Rules

We assumed *decomposable* combining rules. Such rules can be expressed using a set of separate, deterministic nodes in the support network, as shown in Figure 5. Most combining rules commonly employed in Bayesian networks such as *noisy or* or linear regression are decomposable (cp. [HB94]). Decomposable combining rules imply that for each node $x \in N$ there exist at most one clause c and a substitution θ s.t. $body(c\theta) \subset \mathrm{LH}(B)$ and $head(c\theta) = x$. Thus, while the same clause c can induce more than one node in N, all of these nodes have identical local structure: the associated conditional probability distributions (and so the parameters) have to be identical, i.e. $\forall$ *subst.* $\theta : cpd(c\theta) = cpd(c)$. As an example consider the clause defining `bt(X)` in the program *bloodtype* and the nodes `mc(ann)`, `pc(ann)` and `mc(brain)`, `pc(brian)`. This is the same situation as for dynamic Bayesian networks where the parameters that encode the stochastic model of state evolution appear many times in the network. However, gradient methods might be applied to non-decomposable combining function, too. In the general case, the partial derivatives of an inner function has to be computed. E.g. [BKRK97] derive the gradient for *noisy or* when it is not expressed in the structure. This seems to be more difficult in the case of the EM algorithm which we will now dicuss.

5.3. Expectation-Maximization

The Expectation-Maximization algorithm [DLR77] is another classical approach to do maximum likelihood estimation in the presence of missing values.

The basic idea of the Expectation-Maximization algorithm is that if we know the values for all random variables, learning would be easy. Assuming that no value is missing, Lauritzen [Lau95] showed that maximum likelihood estimation of Bayesian network parameters simply corresponds to frequency counting in the following way. Let $n(\mathbf{a} \mid \mathbf{D})$ denote the *counts* for a particular joint state $\mathbf{a}$ of variables $\mathbf{A}$ in the data, i.e. the number of cases in which the variables in $\mathbf{A}$ are assigned the evidence $\mathbf{a}$, then

$$cpd(c_i)^*_{jk} = \frac{n(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)}{n(body(c_i\theta) = \mathbf{u}_k \mid D_l)} \tag{5.7}$$

However, in the presence of missing values, the maximum likelihood estimate typically cannot be written in closed form. Therefore, the Expectation-Maximization algorithm iteratively performs the following two steps: First, based on the current parameters $\boldsymbol{\lambda}$ and the observed data $\mathcal{D}$ the algorithm computes a distribution over all possible completions of each partially observed data case. Each completion is then treated as a fully-observed data case weighted by its probability. A new set of parameters is then computed based on equation (5.7). Lauritzen [Lau95] showed that this idea leads to a modified equation (5.7) where the *expected counts*

$$\sum_{l=1}^{m} P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)$$

are used instead of *counts* as already encountered to compute the gradient (cf. equation (5.5)). Again, essentially any Bayesian network engine can be used to compute $P(\mathbf{a} \mid D_l)$.

To adapt the EM algorithm to our problem, we assume decomposable combining rules. Thus, each node in the support network was “produced” by exactly one Bayesian clause c, and each node derived from c can be seen as a separate “experiment” for the conditional probability distribution $cpd(c)$. Formally, due to the reduction of our problem at hand to parameter estimation within the support network N, the update rule becomes

$$cpd(c_i)_{jk} \leftarrow \sum_{\substack{\text{subst. } \theta \text{ with} \\ support(c_i\theta)}} \frac{\sum_{l=1}^{m} P_N(head(c_i\theta) = u_j, body(c_i\theta) = \mathbf{u}_k \mid D_l)}{\sum_{l=1}^{m} P_N(body(c_i\theta) = \mathbf{u}_k \mid D_l)}$$

The fact that we get the maximum likelihood estimate for our parametes in the fully observable case implies that the algorithm converges to a stationary point [MK97].

5.4. Relationship between Gradient and EM

The comparison between EM and (advanced) gradient techniques such as conjugate gradient is yet not well understood. Both methods perform a greedy local search which is guaranteed to converge to stationary points. They both exploit expected counts as their primary computation step However, there are important differences. EM is easier to implement because one e.g. need not the constraint that the parameters are probability distributions. It converges much faster than simple gradient, and is somewhat less sensitive to starting points. (Conjugate) gradients estimate the step size (see below) with a line search involving several additional Bayesian network inferences compared to EM. But, gradients are more flexible than EM, as they allow e.g. to learn non-multinomial parameterizations using the chain rule for derivatives [BKRK97] or to choose other scoring functions than the likelihood [Jen99]. The EM algorithm may slowly converge, but can be speed up near the maximum using gradient-based approaches [Thi95,OK99]. Other acceleration approaches execute only a partial maximization step of the EM algorithm based on gradient techniques leading to generalized EM algorithms [Thi95,MK97,BKS97,OK99]. Finally, [KL02] gave experimental evidence that advanced gradient methods namely *scaled conjugate gradients* are comparable to the EM algorithm in the context of Bayesian networks with respect to normalized-loss evaluation and to the number of iteration needed to converge. A similar comparison in the context of Bayesian logic programs is out of the scope of the present paper. But given that we estimate the parameters of a Bayesian network, where additional equality constraints among parameters are employed, similar results are to be expected.

6. Experiments

The presented learning algorithm for Bayesian logic programs is mainly meant as an overall and general framework. Indeed, it leaves several aspects open such as scoring functions. Nevertheless in this section, we report on experiments that show that the algorithm and its underlying principles work.

We implemented the score-based algorithm in Sicstus Prolog 3.9.0 on a Pentium-III 700 MHz Linux machine. The implementation has an interface to the Netica API (`http://www.norsys.com`) for Bayesian network inference and maximum likelihood estimation. To do the maximum likelihood estimation, we adapted the scaled conjugate gradient (SCG) as implemented in Bishop and Nabney's Netlab library (`http://www.ncrg.aston.ac.uk/netlab/`, see also [Bis95]) with an upper bound on the scale parameter of $2 \cdot 10^6$. Parameters were initialized randomly. To avoid zero entries in the conditional probability tables, m-estimates were used.

6.1. Genetic Domain

The goal was to learn a global, descriptive model for our genetic domain, i.e. to learn the program *bloodtype*. We considered two totally independent families using the predicates given by *bloodtype* having 12 respectively 15 family members. For each least Herbrand model 1000 data cases from the induced Bayesian network were sampled with a fraction of 0.4 of missing at random values of the observed nodes making in total 2000 data cases.

Therefore, we first had a look at the (logical) hypotheses space. The space could be seen as the first order equivalent of the space for learning the structure of Bayesian networks (see Figure 3). The generating hypothesis is a member of it. In a further experiment, we fixed the definitions for `m/2` and `f/2`. The hypothesis scored best included `bt(X) | mc(X), pc(X)`, i.e. the algorithm rediscovered the intensional definition which was originally used to build the data cases. However, the definitions of `mc/1` and `pc/1` considered genetic information of the grandparents to be important. It failed to re-discover the original definitions for reasons explained above. The predicates `m/2` and `f/2` were not part of the learned model rendering them to be extensionally defined. Nevertheless, the founded global model had a slightly better likelihood than the original one.

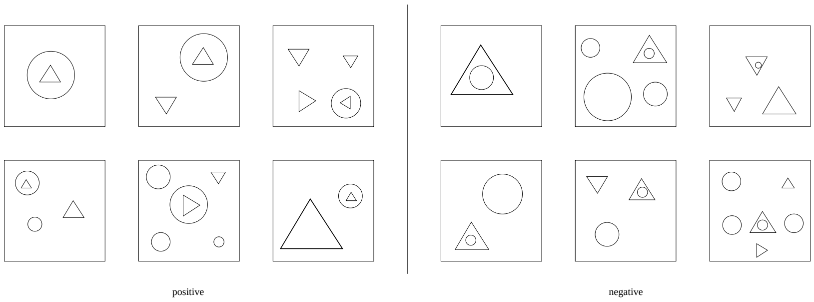

Figure 6. A Bongard problem consisting of 12 scenes, six positive ones and six negative ones. The goal is to discriminate between the two classes.

6.2. Bongard Domain

The Bongard problems (due to the Russian scientist M. Bongard) are well-known problems within inductive logic programming. Consider Figure 6. Each example or scene consists of

- a variable number of geometrical objects such as triangles, rectangles and circles etc (predicate `obj/2` with $\mathrm{Dom}(obj) = \{triangle, circle\}$)., each having a number of different properties such as color, orientation, size etc., and
- a variable number of relations between objects such as in (predicate `in/3` having states *true*, *false*), leftof, above etc.

The task is to find a set of rules which *discriminates* positive from negative examples (represented by `class/1` over the states *pos*, *neg*) by looking at the kind of objects they consists of. Though the Bongard problems are toy problems, they are very similar to real-world problems in e.g. the field of molecular biology where essentially the same representational problems arise. Data consists of molecules, each of which is composed of several atoms with specific properties such as charge. There exists a number of relations between atoms like e.g. bonds, structure etc.

In most real-world applications, the data is *noisy*. Class labels or object properties might be wrong or missing in the data cases. One extreme case concerns clustering where no class labels are given. Furthermore, we might be uncertain about relations among objects. Some positive examples might state that a triangle is not in a circle due to noise. In such cases, Bayesian logic programs are a natural method of choice. We conducted the follwing experiments.

First, we generated 20 positive and 20 negative examples of the concept *"there is a triangle in a circle.“* The number of objects varied from 2 to 8. We conducted three different experiments. We assumed the *in* relation to be deterministic and given as background knowledge `in(Example,Obj1,Obj2)`, i.e. we assumed that there is no uncertainty about it. Due to that, no conditional probability distribution has to take `in/3` into account. Because each scene is independent of the other, we represented the whole training data as one data case

$$\begin{aligned}&\{class(e1) = pos, obj(e1, o1) = triangle, obj(e1, o2) = circle,\\ &class(e2) = neg, obj(e2, o1) = triangle, size(e2, o1) = large,\\ &obj(e2, o2) = '?', \ldots\}\end{aligned}$$

with the background knowledge $in(e1, o1, o2), \ldots$ where $e1, e2, \ldots$ are identifiers for examples and $o1, o2, \ldots$ for objects. A fraction of 0.2 of the random variables were not observed. Our algorithm scored the hypothesis

```
class(Ex) | obj(Ex,O1),in(Ex,O1,O2),obj(Ex,O2).
```

best after specifying `obj(Ex,O2)` as a lookahead for `in(Ex,O1,O2)`. The conditional probability distribution assigned *pos* a probability higher than 0.6 only if object `O1` was a triangle and `O2` a circle. Without the lookahead, adding `in(Ex,O1,O2)` yield no improvement in score, and the correct hypothesis was not considered. The hypothesis is not a well-defined Bayesian networks, but it says that ground atoms over `obj/2` extensional defined. Therefore, we estimated the maximum likelihood parameters of

```
obj(Ex,O) | dom(Ex,O).
class(Ex) | obj(Ex,O1),in(Ex,O1,O2),obj(Ex,O2).
```

where `dom/2` ensured range-restriction and was part of the deterministic background knowledge. Using 0.6 as threshold, the learned Bayesian logic program had accuracy 1.0 on the training set and on an independently generated validation set consisting of 10 positive and 10 negative examples.

In a second experiments, we fixed the structure of the program learned in the first experiment, and estimated its parameters on a data set consisting of 20 positive and 20 negative examples of the disjunctive concept *"there is a (triangle or a circle) in a circle.“* The estimated conditional probability distribution gave almost equal probability for the object `O1` to be a triangle or circle.

In third experiment, we assumed uncertainty about the *in* relation. We enriched the data case used for the first experiment in the following way

$$\{class(e1) = pos, obj(e1, o1) = triangle, obj(e1, o2) = circle,$$
$$in(e1, o1, o2) = true, class(e2) = neg, obj(e2, o1) = triangle,$$
$$size(e2, o1) = large, obj(e2, o2) ='?', in(e2, o1, o2) = false, \ldots\},$$

i.e. for each pair of objects that could be related by *in* a ground atom over *in* was included. Note that the state need not to be observed. Here, the algorithm did not re-discovered the correct rule but

```
class(X) | obj(X,Y)
obj(X,Y) | in(X,Y,Z), obj(Z).
```

This is interesting, because when these rules are used to classify examples, only the first rule is needed. The class is independent of any information about `in/3` given full knowledge about the objects. The likelihood of the founded solution was close to the one of `class(Ex) | obj(Ex,O1),in(Ex,O1,O2),obj(Ex,O2)` on the data (absolute difference less than 0.001). However, the accuracy decreased (about 0.6 on an independently generated training set (10 pos / 10 neg)) for the reasons explained above: We learned a global model not focusing on the classification error. [FGG97] showed for Bayesian network classifier that maximizing the conditional likelihood of the class variable comes close to minimizing the classification error. In all experiments we assumed *noisy or* as combining rule.

Finally, we conducted a simple clustering experiments. We generated 20 positive and 20 negative examples of the disjunctive concept *"there is a triangle."* There were triangles, circles and squares. The number of objects varied from 2 to 8. All class labels were said to be observed, and 20% of the remaining stated were missing at random. The learned hypothesis was `class(X) | obj(X,Y)` totally separating the two classes.

6.3. KDD Cup 2001

We also started to conduct experiments on *large scale* data sets namely the KDD Cup 2001 [4] data sets, cf. [CHK$^+$02]. Task 3 is to predict the localizations `local(G)` of proteins encoded by the genes `G`. This is a multi-class problem because there are 16 possible localizations. The training set consisted of 862 genes,

[4] For details see `http://www.cs.wisc.edu/~dpage/kddcup2001/`.

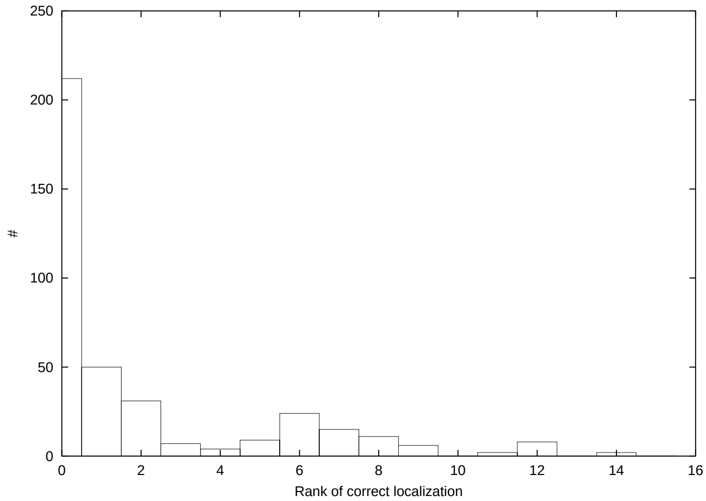

Figure 7. An histogram of the ranks of the correct localization computed by an Bayesian logic program on the KDD Cup 2001 test set.

the test set of 381 genes. The information we used included whether organisms with an mutation in this gene can survive `ess(G)` (3 states), the class `class(G)` of a gene/protein `G` (24 states), the complex `compl(G)` (56 states), and the other proteins `G` with which each protein `G` is known to interact encoded by `inter(G1,G2)`. To avoid a large interaction table, we considered only those interactions with a correlation higher than 0.85. Furthermore, we introduced a hidden predicate `hidden/1` with domain $0, 1, 2$ to compress the representation size because e.g. the conditional probability table of `local(G1) | inter(G1,G2),local(G2)` would consists of 225 entries (instead of 45 using `hidden(G1)`). The ground atoms over `hidden/1` were never observed in the data. Nevertheless, the naive Prolog representation of the support networks induced by some hypothesis (more than 4.400 random variables with more than 60.000 parameters) in our current implementation broke the memory capabilities of Sicstus Prolog. Due to that, we can only report on preliminary results. We only considered maximum likelihood parameter estimation on the training set. The (logical) structure is based on naive Bayes taking relational information into account:

```
local(G1)  | gene(G1).
hidden(G1) | local(G1).
hidden(G1) | inter(G1,G2),local(G2).
class(G1)  | hidden(G1).
```

```
compl(G1)  | local(G1).
ess(G1)    | local(G1).
```

As combining rule, we used for all predicates *average*. The given ground atoms over `inter/2` were used as pure logical background knowledge. Therefore, the conditional probability distribution associated to `hidden(G1) | inter(G1,G2),local(G2)` had not to take it into account. The parameters were randomly initialized. Again, the training set was represented as one data case, so that no artificial independencies among genes were postulated. Estimate the parameters took 12 iteration (about 30 min). The learned Bayesian logic program achieved an accuracy of 0.57 (top 50% level of submitted models was 0.61, best predictive accuracy was 0.72). A learner predicting always the majority class would achieve an predictive accuracy of 0.44. Furthermore, when we rank for each test gene its possible localizations according to the probability computed by the program, then the correct localization was among the three highest ranked localizations in 293 out of 381 cases (77%) (cf. Figure 7). Not that it took 40 iterations to learn the corresponding grounded Bayesian logic program.

7. Related Work

The learning of Bayesian networks has been thoroughly investigated in the Uncertainty in AI community, see e.g. [Hec95,Bun96]. [BKRK97], whose approach we have adapted, present results for a gradient-based method. But so far – to the best of our knowledge – there has not been much work on learning within first order extensions of Bayesian networks. [KP97] adapt the EM algorithm for probabilistic logic programs [NH97], a framework which in contrast to Bayesian logic programs sees ground atoms as states of random variables. Although the framework seems to theoretically allow for continuous random variables there exists no (practical) query-answering procedure for this case; to the best of our knowledge [NH97] give only a procedure for variables having finite domains. Furthermore, Koller and Pfeffer's approach utilizes support networks, too, but requires the intersection of the support networks of the data cases to be empty. This could be in our opinion in some cases too restrictive, e.g. in the case of dynamic Bayesian networks. [FGKP99,GKTF00] adapted the Structural-EM to learn the structure of probabilistic relational models. It applies the idea of the standard EM algorithm for maximum likelihood parameter estimation to the problem of learning the structure. If we know the values for all random variables, then the maximum likelihood estimate can be written in closed from. Based on the current hypothesis a distribution over all possible completions of each partially observed data case is computed. Then, new hypotheses are computed using a score-based method. However, the algorithm does not consider logical constraints on the space of hypotheses. Indeed, the considered clauses need not be logically valid on the data. Therefore, combining our approach with the structural EM seems to be reasonable and straightforward. Finally, there is work on learning object-oriented Bayesian networks [LB01,BLN01].

There exist also methods for learning within first order probabilistic frameworks which do not build on Bayesian networks. [SK01] introduce an EM method for parameter estimation of PRISM programs. Cussens [Cus01] investigates EM like methods for estimating the parameters of stochastic logic programs. Within the same framework, Muggleton [Mug00] uses ILP techniques to learn the logical structure. The used ILP setting is different to *learning from interpretations*, it is not based on learning Bayesian networks, and so far considers only for single predicates definitions.

For a more detailed discussion of the relations of Bayesian logic programs to other first order extensions of Bayesian networks such as probabilistic logic programs [NH97], relational Bayesian networks [Jae97] and probabilistic relational models [Kol99] we refer to [KD00,KD01]. There, we show e.g. that probabilistic relational models could be expressed as Bayesian logic programs. This is particularly interesting because [TSK01] used probabilistic relational models for relational probabilistic clustering. Another approach to do relational probabilistic clustering is 1BC [FL99] which is an upgraded naive Bayes classifier. In contrast to Bayesian logic programs, it does not aim at representing arbitrary distribution. To some extent, Bayesian logic programs are related to the BUGS language [GTS94] which aims at carrying out Bayesian inference using Gibbs sampling. It uses concepts of imperative programming languages such as for-loops to model regularities in probabilistic models. Therefore, the relation between Bayesian logic programs and BUGS is akin to the general relation between logical and imperative languages. This holds in particular for relational domains such as we used in this paper: family relationships. Without the notion of objects and relations among objects family trees are hard to represent. Furthermore, a single BUGS program specifies a probability density over a finite set of random variables, whereas a Bayesian logic program can represent a distribution over an infinite set of random variables.

To summarize, the related work mainly differs in three points from ours:

- The underlying (logical) frameworks lack important knowledge representational features which Bayesian logic programs have.
- They adapt the EM algorithm to do parameter estimation which is particularly easy to implement. However, there are problematic issues both regarding speed of convergence as well as convergence towards a local (sub-optimal) maximum of the likelihood function. Different accelerations based on the gradient are discussed in [MK97]. Also, the EM algorithm is difficult to apply in the case of general probability density functions because it relies on computing the sufficient statistics (cf. [Hec95]).
- No probabilistic extension of the *learning from interpretations* is established.

8. Conclusions

A new link between ILP and learning of Bayesian networks was established. We have proposed a scheme for learning both the probabilities and the structure of Bayesian logic programs. We addressed the question "where do the numbers come from?" by showing how to compute the gradient of the likelihood based on ideas known for (dynamic) Bayesian networks. The intensional representation of Bayesian logic programs, i.e. their compact representation should speed up learning and provide good generalization. The general learning setting built on the ILP setting *learning from interpretations*. We have argued that by adapting this setting score-based methods for structural learning of Bayesian networks could be updated to the first order case. The ILP setting is used to define and traverse the space of (logical) hypotheses.

The experiments proved the basic principle of the algorithm. Their results highlight that future work on improved scoring functions is needed. We plan to conduct experiments on real-world scale problems. The use of refinement operators adding or deleting non constant-free atoms should be explored. Furthermore, it would be interesting to weaken the assumption that a data case corresponds to a complete interpretation. Not assuming all relevant random variables are known would be interesting for learning intensional rules like `nat(s(X)) | nat(X)`. Ideas for handling this within inductive logic programming might be adapted [DD97,BD98]. Furthermore, instead of traditional score-based greedy algorithm other UAI methods such as Structural-EM may be adapted taking advantage of the logical constraints implied by the data cases. In this sense, we believe that the proposed approach is a good point of departure for further research. But the link established between ILP and Bayesian networks seems to be bi-directional. Can ideas developed in the UAI community be carried over to ILP?

9. Acknowledgements

The authors would like to thank Manfred Jaeger, Stefan Kramer and David Page for helpful discussions on the ideas of the paper. Furthermore, the authors would like to thank Jan Ramon and Hendrik Blockeel for making available their Bongard problems generators. This research was partly supported by the European Union IST programme under contract number IST-2001-33053 (Application of Probabilistic Inductive Logic Programming – APRIL)

References

[Bau91] Heinz Bauer. *Wahrtscheinlichkeitstheorie*. Walter de Gruyter, Berlin, New York, 4. edition, 1991.

[BD97] H. Blockeel and L. De Raedt. Lookahead and discretization in ilp. In N. Lavrac and S. Dzeroski, editors, *Proceedingd of the Seventh International Workshop on Inductive Logic Programming (ILP 97)*, volume 1297 of *LNCS*, pages 77–85. Springer, 1997.

[BD98] H. Blockeel and L. De Raedt. ISIDD: An Interactive System for Inductive Databse Design. *Applied Artificial Intelligence*, 12(5):385, 421 1998.

[Bis95] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[BKRK97] J. Binder, D. Koller, S. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, pages 213–244, 1997.

[BKS97] E. Bauer, D. Koller, and Y. Singer. Update Rules for Parameter Estimation in Bayesian Networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, 1997.

[BLN01] O. Bangsø, H. Langseth, and T. D. Nielsen. Structural learning in object oriented domains. In *Proceedings of FLAIRS-2001*, 2001.

[Bun96] W. Buntine. A guide to the literature on learning probabilistic networks from data. *IEEE Transaction on Knowledge and Data Engineering*, 8:195 – 210, 1996.

[CDLS99] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic networks and expert systems*. Springer-Verlag New York, Inc., 1999.

[CHK+02] J. Cheng, C. Hatzis, M.–a. Krogel, S. Morishita, D. Page, and J. Sese. KDD Cup 2002 Report. *SIGKDD Explorations*, 3(2):47 – 64, 2002.

[Cus01] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 2001. to appear.

[DB93] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-1993)*, pages 1058–1063, 1993.

[DD97] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, (2-3):99–146, 1997.

[De 97] L. De Raedt. Logical settings for concept-learning. *Artificial Intelligence*, 95(1):197–201, 1997.

[DK88] T. Dean and K. Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-1988)*, 1988.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum liekelihood from incomplete data via the EM algorithm. *J. Royal Stat. Soc.*, B 39:1–39, 1977.

[FGG97] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[FGKP99] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the Sixteenth International Joint Conferences on Artificial Intelligence (IJCAI-1999)*, 1999.

[FL99] P. A. Flach and N. Lachiche. 1BC: A first-order Bayesian classifier. In *Proceedings of the 9th International Workshop on Inductive Logic Programming (ILP'99*, LNAI 1634, pages 92–103, 1999.

[GKTF00] L. Getoor, D. Koller, B. Taskar, and N. Friedman. Learning probabilistic relational models with structural uncertainty. In L. Getoor and D. Jensen, editors, *Proceedings of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data*, 2000.

[GTS94] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, 43, 1994.

[HB94] D. Heckerman and J. Breese. Causal Independence for Probability Assessment and Inference Using Bayesian Networks. Technical Report MSR-TR-94-08, Microsoft Research, 1994.

[Hec95] D. Heckerman. A Tutorial on Learning with Bayesian Networks. Technical Report MSR-TR-95-06, Microsoft Research,, 1995.

[Jae97] M. Jaeger. Relational Bayesian networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1997)*, 1997.

[Jen99] F. V. Jensen. Gradient descent training of bayesian networks. In *Proceedings of the Fifth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-99)*, 1999.

[Jen01] F. V. Jensen. *Bayesian networks and decision graphs.* Springer-Verlag New York, 2001.

[KD00] K. Kersting and L. De Raedt. Bayesian logic programs. In *Work-in-*

Progress Reports of the Tenth International Conference on Inductive Logic Programming (ILP -2000), 2000. `http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-35/`.

[KD01] K. Kersting and L. De Raedt. Bayesian logic programs. Technical Report 151, University of Freiburg, Institute for Computer Science, April 2001.

[KDK00] K. Kersting, L. De Raedt, and S. Kramer. Interpreting Bayesian Logic Programs. In L. Getoor and D. Jensen, editors, *Working Notes of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data (SRL)*, 2000.

[KL02] K. Kersting and N. Landwehr. Scaled Conjugate Gradients for Maximum likelihood: An Empirical Comparison with the EM Algorithm. submitted, 2002.

[Kol99] D. Koller. Probabilistic relational models. In *Proceedings of Ninth International Workshop on Inductive Logic Programming (ILP-1999)*, 1999.

[KP97] D. Koller and A. Pfeffer. Learning probabilities for noisy first-order rules. In *Proceedings of the Fifteenth Joint Conference on Artificial Intelligence (IJCAI-1997)*, 1997.

[Lau95] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, 19:191–201, 1995.

[LB94] W. Lam and F. Bacchus. Learning Bayesian belief networks: An approach based on the mdl principle. *Computational Intelligence*, 10(4), 1994.

[LB01] Helge Langseth and Olav Bangsø. Parameter learning in object oriented Bayesian networks. *Annals of Mathematics and Artificial Intelligence*, 2001.

[Llo89] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 2. edition, 1989.

[MK97] G. J. McKachlan and T. Krishnan. *The EM Algorithm and Extensions*. John Eiley & Sons, Inc., 1997.

[Mug00] S. H. Muggleton. Learning stochastic logic programs. In L. Getoor and D. Jensen, editors, *Proceedings of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data*, 2000.

[NH97] L. Ngo and P. Haddawy. Answering queries form context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171:147–177, 1997.

[OK99] L. E. Ortiz and L. P. Kaelbling. Accelerating EM: An Empirical Study. In *Proceedings of the Fifteenth Annual Conference on Uncertainty in Articial Intelligence (UAI-99)*, pages 512–521, 1999.

[Pea91] J. Pearl. *Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2. edition, 1991.

[Poo93] D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

[SK01] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[SMB94] A. Srinivasan, S. Muggleton, and M. Bain. The justification of logical theories based on data compression. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence 13*. Oxford University Press, 1994.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, 1986.

[Thi95] B. Thiesson. Accelerated quantification of Bayesian networks with incomplete data. In *Proceedings of First International Conference on Knowledge Discovery and Data Mining*, pages 306–311, 1995.

[TSK01] B. Taskar, E. Segal, and D. Koller. Probabilistic clustering in relational data. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 870–87, 2001.

[WZ95] R. J. Williams and D. Zipser. Gradient-Based Learning Algorithms for Recurrent Networks and Their Computatinal Complexity. In *Back-propagation:Theory, Architectures and Applications.* Hillsdale, NJ: Erlbaum, 1995.

[XWC96] Yang Xiang, S. K. M. Wong, and N. Cercone. Critical remarks on single link search in learning belief networks. In *Proceddings of the Twelfeth Conference on Uncertainty in Artificial Intelligenc (UAI-1996)*, pages 564–571, San Francisco, CA, 1996. Morgan Kaufmann Publishers.