Parameter Learning for Relational Bayesian Networks

Manfred Jaeger JAEGER@CS.AAU.DK
Aalborg University, Fredrik Bajers Vej 7E, 9220 Aalborg, Denmark

Abstract

We present a method for parameter learning in relational Bayesian networks (RBNs). Our approach consists of compiling the RBN model into a computation graph for the likelihood function, and to use this likelihood graph to perform the necessary computations for a gradient ascent likelihood optimization procedure. The method can be applied to all RBN models that only contain differentiable combining rules. This includes models with non-decomposable combining rules, as well as models with weighted combinations or nested occurrences of combining rules. Experimental results on artificial random graph data explores the feasibility of the approach both for complete and incomplete data.

1. Introduction

We are concerned with probabilistic models for relational data that can be expressed in representation languages like Relational Bayesian Networks (Jaeger, 1997), Probabilistic Relational Models (Friedman et al., 1999), or Bayesian Logic Programs (Kersting & Raedt, 2001; Kersting, 2006). Closely related to these, yet semantically mostly a bit different are numerous other probabilistic modeling frameworks that combine logical with probabilistic elements, e.g. the Prism system (Sato, 1995), Stochastic Logic Programs (Muggleton, 1996), IBAL (Pfeffer, 2000), Blog (Milch et al., 2005), and Markov Logic (Richardson & Domingos, 2006).

1.1. Models, Languages and Combining Rules

A common ground shared by the aforementioned systems is that they define a probabilistic model for interdependent objects in a structured domain. These dependencies are represented by attributes and relations. In typical applications, objects could be persons in a pedigree, web-pages, or atoms in a molecule; the binary relation between the objects then could represent the parent/child relation, hyperlinks, or bonds between atoms, respectively.

Given such a structured input domain, a probabilistic model specifies a probability distribution over further uncertain attributes and relations, as, for example, the presence of a certain genetic traits, the relevance of a web page, or spatial adjacency of atoms in a molecule. We use S to denote the set of attributes and relations that describe the known structure of an input domain (also called the predefined relations (Jaeger, 2001), the skeleton structure (Friedman et al., 1999), or logical predicates (Kersting, 2006)). R denotes the uncertain relations on the objects of the domain. In the simplest case, all attributes and relations are boolean. RBNs only allow this boolean case. Most other frameworks also allow multi-valued attributes, and sometimes multi-valued relations. The restriction to boolean relations is no limitation in principle, but makes some modeling tasks more cumbersome.

A relational probabilistic model defines for input structures $\mathcal{D}$ described by S a probability distribution on the relations in R. This distribution can be seen as a joint distribution of boolean random variables given by the *ground atoms* $r(\boldsymbol{o})$ constructed from relations $r \in R$ and tuples $\boldsymbol{o}$ of domain objects. We denote with $\boldsymbol{A}(\mathcal{D}, R)$ (or simply $\boldsymbol{A}$) the set of all these ground atoms, and write $\boldsymbol{A} = \boldsymbol{a}$ to denote an instantiation of the atoms in $\boldsymbol{A}$ to truth values $\boldsymbol{a}$. For a particular ground atom $r(\boldsymbol{o})$ we denote with $\boldsymbol{a}[r(\boldsymbol{o})] \in \{true, false\}$ the value of $r(\boldsymbol{o})$ in the instantiation $\boldsymbol{A} = \boldsymbol{a}$.

A core component of relational probabilistic models are aggregation (Friedman et al., 1999) or combining rules (Ngo & Haddawy, 1995; Jaeger, 1997; Koller & Pfeffer, 1997; Kersting & Raedt, 2001; Natarajan et al., 2005) that describe how to integrate probabilistic dependencies on varying numbers of related objects. See (Natarajan et al., 2005) for a discussion of aggregating vs. combining probabilistic influences. In frameworks that are based on combining rules, it is often assumed that with each probabilistic relation there is associated exactly one combining rule. One

Appearing in *Proceedings of the 24th International Conference on Machine Learning*, Corvallis, OR, 2007.

of the motivating factors in the definition of the RBN language was the desire to allow for multiple, possibly nested, applications of combining rules in the definition of conditional probabilities. The consideration of weighted combinations of combining rules also is a main motivation in (Natarajan et al., 2005).

1.2. Relational Bayesian Networks

Relational Bayesian Networks define the probabilities for ground atoms of a given probabilistic relation r by a single functional expression $\texttt{F\_r(v)}$, called a *probability formula*. A formula $\texttt{F\_r(v)}$ evaluates for an object o from an input domain $\mathcal{D}$ to a probability value $\texttt{F\_r}(o) = P(r(o) = \mathit{true}) \in [0,1]$. The evaluation can depend on the input structure, and be conditional on the truth values of other probabilistic atoms $r'(o')$ (all relations can have any arity ≥ 0).

The formal syntax and semantics of probability formulas is defined in (Jaeger, 2001). We here review the main elements by an example.

Example 1.1 *Let input structures be given by arbitrary directed acyclic graphs consisting of a (finite) domain, a binary* edge *relation, and a unary* root *attribute identifying nodes without incoming edges. On such input structures we define a boolean probabilistic attribute* on *as follows: for any root node o,* $\mathrm{on}(o)$ *is true with probability θ_1. For any non-root node o, the probability for* $\mathrm{on}(o)$ *depends on the truth value of* $\mathrm{on}(o')$ *for all parents o' of o. If o has a single parent o', then* $P(\mathrm{on}(o) = \mathrm{true}) = \theta_2$ *if* $\mathrm{on}(o') = \mathrm{true}$*, and* $P(\mathrm{on}(o) = \mathrm{true}) = \theta_3$ *if* $\mathrm{on}(o') = \mathrm{false}$*. For multiple parents these probabilities are combined using noisy-or.*

A RBN representation of this model is shown in Table 1. It consists basically of a single formula $\texttt{F\_on(v)}$ *defining the probabilities for* on *atoms. For better readability the formula is broken up into several subformulas that are denoted by identifiers starting with* '@'.

The main formula $\texttt{F\_on(v)}$ *is a* convex combination*, which has the general syntactic form* $\texttt{(F1:F2,F3)}$ *with sub-formulas* $\texttt{F1}, \texttt{F2}, \texttt{F3}$*. A convex combination evaluates for an object o to* $\texttt{F1}(o)\texttt{F2}(o) + (1 - \texttt{F1}(o))\texttt{F3}(o)$*. In the case where* F1 *is a formula evaluating to either 0 or 1, this amounts to an if-then-else condition. In our example: if o is a root, then* $\texttt{F\_on}(o)$ *is evaluated as* $\texttt{theta1}$*, else as* $\texttt{@nonroot}(o)$*. The subformula* $\texttt{@nonroot}(v)$ *combines by noisy-or the values returned by the sub-formula* $\texttt{@norarg}(o')$ *for all o' with* $\mathrm{edge}(o', v)$*. The* @norarg *sub-formula again uses the convex combination construct to make an if-then-else choice.*

Table 1. A simple RBN

```
@norarg(w) = (on(w):  theta2,theta3);
@nonroot(v) = n-or{@norarg(w)|w:edge(w,v)};
F_on(v) = (root(v):theta1,@nonroot(v));
```

The preceding example illustrated all syntactic elements of the RBN language: convex combinations, *combination functions*, and the two basic constructs of numerical constants and atoms from S and R. Combination functions are the RBN version of combining rules: they are functions that map multisets I of probability values to a single probability value. Examples of combination functions are

$$\begin{aligned} \mathit{noisy\text{-}or}(I) &= 1 - \textstyle\prod_{p \in I}(1-p) \\ \mathit{mean}(I) &= 1/\,|I|\,\textstyle\sum_{p \in I} p \\ \mathit{esum}(I) &= e^{-\sum_{p \in I} p} \end{aligned}$$

(esum stands for exponential sum). A combination function is differentiable if it is differentiable in all elements of $p \in I$. Clearly, the three combination functions above are differentiable.

In Example 1.1 the convex combination construct was only used to express if-then-else conditions. However, by setting the first component F1 to some numerical constant $\lambda \in [0,1]$, one obtains the weighted mean $\lambda\texttt{F2} + (1-\lambda)\texttt{F3}$ of two probability formulas. By nested application of the the convex combination construct, this can be extended to weighted means $\lambda_1\texttt{F1} + \ldots + \lambda_m\texttt{Fm}$ with arbitrarily many components. Since the Fi here could also be constructed using combination functions, one can form weighted means of combining rules, as suggested in (Natarajan et al., 2005).

The RBN in Table 1 contains 3 parameters θ_i that have to be set to constants in $[0,1]$ before a distribution is actually defined. Other than being numbers in $[0,1]$ the parameters in a RBN are unconstrained, i.e. the parameter space for a RBN with k parameters is $[0,1]^k$. Given a particular $\boldsymbol{\theta} \in [0,1]^k$, and input domain $\mathcal{D}$, the RBN defines a distribution $P^{\mathcal{D}}_{\boldsymbol{\theta}}$ on $\boldsymbol{A}(\mathcal{D}, R)$ as

$$P^{\mathcal{D}}_{\boldsymbol{\theta}}(\boldsymbol{A} = \boldsymbol{a}) = \prod_{r \in R} \prod_{\substack{\boldsymbol{o}: \\ \boldsymbol{a}[r(\boldsymbol{o})] = \mathit{true}}} \texttt{F\_r}(\boldsymbol{o}) \prod_{\substack{\boldsymbol{o}: \\ \boldsymbol{a}[r(\boldsymbol{o})] = \mathit{false}}} (1 - \texttt{F\_r}(\boldsymbol{o})) \quad (1)$$

1.3. Data

Data for learning relational models consists of pairs $(\mathcal{D}_i, \boldsymbol{a}_i)$, where $\mathcal{D}_i$ is an input structure, and $\boldsymbol{a}_i$ is an instantiation of the ground probabilistic

atoms (this is the "learning from interpretations" paradigm (De Raedt & Kersting, 2003)). Some truth values of probabilistic atoms may be unobserved. We use $\boldsymbol{a}_{obs}$ to denote a partial instantiation of the ground atoms, and $\boldsymbol{a}_{mis}$ for the missing values. Thus, $\boldsymbol{a} = (\boldsymbol{a}_{obs}, \boldsymbol{a}_{mis})$ is a complete instantiation.

The likelihood function induced by data $(\mathcal{D}_1, \boldsymbol{a}_{obs,1}), \ldots, (\mathcal{D}_N, \boldsymbol{a}_{obs,N})$ is

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{N} P_{\boldsymbol{\theta}}^{\mathcal{D}_i}(\boldsymbol{a}_{obs,i}). \tag{2}$$

2. Parameter Learning

The most frequently proposed method for learning parameters of a relational model specification is by reduction to parameter learning in Bayesian networks. Figure 1 shows a Bayesian network representing the probability distribution induced by the model of Table 1, and an input domain $\mathcal{D}$ with objects $A, \ldots, E$ and a binary *edge* relation with $A \to D, B \to D, C \to D, C \to E$. The parameters of the original model specification here appear inside arithmetic expressions defining the entries in the conditional probability tables. Standard Bayesian network learning methods would learn each cpt entry as an independent parameter, rather than the underlying model parameters $\theta_1, \ldots, \theta_3$. The easiest way around this problem is to *decompose* the network via insertion of auxiliary or hidden nodes in such a way, that the marginal distribution on the observable ground atoms is unchanged, and in the resulting network all cpt entries are either one of the original model parameters, or constants not depending on the model parameters (Figure 2). Such a decomposition is possible when the relational model only uses a restricted class of combining rules. The definitions of what constitutes a *decomposable* combining rule vary to some extent (Koller & Pfeffer, 1997; Natarajan et al., 2005; Kersting, 2006). For RBNs the decomposability of a combination function has been identified with the arithmetic property of multi-linearity (Jaeger, 2001). When a decomposed Bayesian network has been constructed, then parameters can be learned using a standard EM procedure. Since the generated Bayesian networks are usually too complex for exact inference, one has to implement the E step using a suitable approximation method.

The possibility of applying gradient-ascent optimization directly on non-decomposed parameterizations of the cpt entries has been mentioned e.g. in (Binder et al., 1997). Such an approach seems attractive for at least two reasons: combining rules only need to be differentiable, not decomposable, and, for complete data,

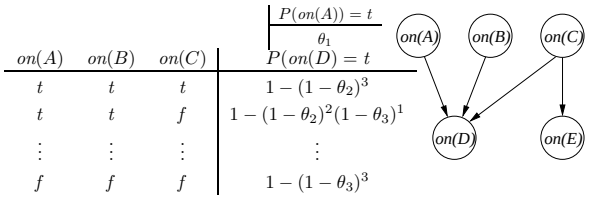

Figure 1. Bayesian network without decomposition

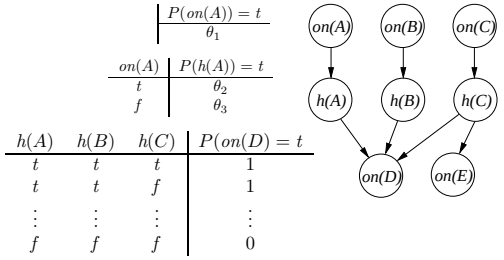

Figure 2. Bayesian network with decomposition

no probabilistic inference is required. In spite of these attractive features, to the best of our knowledge, this approach has not previously been developed for any general, expressive relational modeling language. One possible difficulty is the need to compile the model into a suitable representation of the likelihood function, such that partial derivatives can be computed. Whereas most languages are designed to support the compilation into a Bayesian network, they are not very easily transformed into an arithmetic expression for the likelihood function. RBNs, in contrast, are very closely linked to the likelihood function, since the basic representation elements of RBNs, the probability formulas, are just the factors of the likelihood function (1).

2.1. Likelihood Graph: Complete Data

We compile a RBN model together with an input domain and observed data into a data structure that supports the required gradient computations. An example of this data structure, called the *likelihood graph*, is shown in Figure 3. A likelihood graph is a directed acyclic graph with a single root node, called the likelihood node. The likelihood node has one child for each ground atom in the data; the edge leading to such an *upper ground atom node* is labeled with the truth value of the atom in the data. An upper ground atom node is labeled with the ground probability formula that defines the probability for the atom in the given input structure. Each ground atom node has one child node for each of the *relevant* sub-formulas of the probability formula it is labeled with. A sub-formula is relevant, if the evaluation of the sub-formula depends on one of the unknown parameters, and the evaluation of the

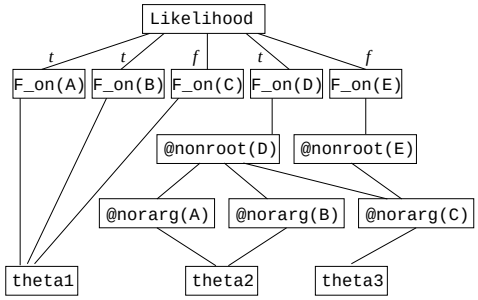

Figure 3. Likelihood graph for data *on(A),on(B),on(D)=true*; *on(C),on(E)=false*

parent formula depends on the sub-formula. For example, the ground atom node for `F_on(A)` in Figure 3 is labeled with the ground probability formula

```
F_on(A) = (root(A):theta1,@nonroot(A))
```

This formula has three sub-formulas `root(A)`, `theta1` and `@nonroot(A)`. The first one, `root(A)` evaluates to 1, independent of the parameters, and so is not relevant. The sub-formula `theta1` is relevant. The sub-formula `@nonroot(A)` depends on the parameters θ_2 and θ_3. However, since `root(A)`=1, the value of this sub-formula has no impact on the value of `on(A)`, and therefore also is not relevant. The decomposition of probability formulas into sub-formulas and the addition to the graph of nodes for relevant sub-formulas continues recursively until the base case of parameter formulas is encountered (note that here we are talking about a purely syntactic decomposition that is always possible. Even when a node is labeled with a probability formula constructed from a non-decomposable combination function, the node can be decomposed!). Nodes in the likelihood graph with identical sub-formulas will share the children corresponding to these sub-formulas. In our example the sub-formula `@norarg(C)` appears both in `@nonroot(D)` and `@nonroot(E)`.

Apart from their probability formula label and pointers to their children, nodes in the likelihood graph contain the numerical values for those sub-formulas that were irrelevant because they evaluated to a constant independent of any parameter, but whose constant value still has an impact on the value of the probability formula at the node. In our example, the node `F_on(A)` stores the numerical value 1 for the sub-formula `root(A)`.

Given the likelihood graph, and a setting of the parameters to specific values, one can compute by a simple bottom-up propagation the value of the probability formulas at all nodes. At the root node one obtains the likelihood value according to (1). Furthermore, partial derivatives of a probability formula are functions of the values and partial derivatives of its sub-formulas. For example, a (partial) derivative $(\mathtt{F1}:\mathtt{F2},\mathtt{F3})'$ of a convex combination is given by $\mathtt{F1}'(\mathtt{F2}-\mathtt{F3})+\mathtt{F1F2}'+(1-\mathtt{F1})\mathtt{F3}'$. Thus, partial derivatives, too, can be efficiently computed bottom-up (Rote, 1990). More precisely, both likelihood values and partial derivatives can be computed in time linear in the size of the graph, where the size is given by the number of edges in the graph, plus the number of constant values stored at the nodes.

2.2. Likelihood Graph: Incomplete Data

The definition and construction of likelihood graphs for incomplete data is a simple extension of the complete data case. One starts as before with a root likelihood node, and children for all ground atoms that are instantiated in the data. However, now the decomposition may end in a ground atom that is not observed in the data. In that case, a node for this ground atom is added to the graph (we call such a node a *lower ground atom node*), and the probability formula defining its probability is added as a new upper ground atom node. The decomposition of the new upper ground atom node may, in turn, lead to new unobserved atoms, so that the process has to be iterated (this is completely analogous to backward-chaining from observed nodes in a Bayesian network). Links from the likelihood node to upper ground atom nodes for unobserved atoms are labeled with a link to the corresponding lower ground atom node, instead of a fixed instantiation value. Figure 4 shows our previous example in the case where *on(A)* and *on(E)* are not observed. The decomposition of the formula `F_on(D)` leads to the atom `on(A)`, so upper and lower ground atom nodes for this atom have to be added.

Given a likelihood graph for incomplete data, one now can compute likelihood values and partial derivatives for any particular setting of truth values for the unobserved atoms in the graph: setting e.g. `on(A)` to *true* in the graph of Figure 4 turns the node `on(A)` into the numerical constant 1, and the edge-label leading to `F_on(A)` into *t*. The graph can now be evaluated as before in the complete data case.

2.3. Gradient Ascent

We use a quite simple gradient ascent strategy: given a current parameter vector $\boldsymbol{\theta}$, we first compute the gradient of the likelihood function, $(\delta_1,\ldots,\delta_n) := gradL(\boldsymbol{\theta})$. Given the gradient, we determine a search direction $\boldsymbol{d}$

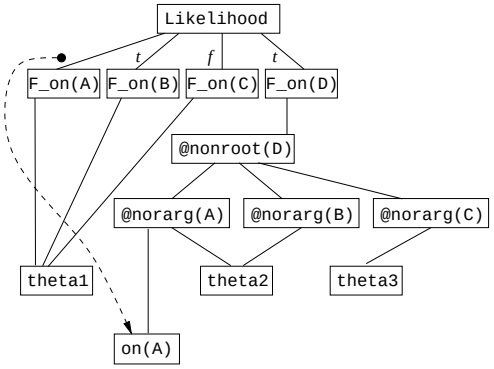

Figure 4. Likelihood graph for data *on(B),on(D)=true*; *on(C)=false*

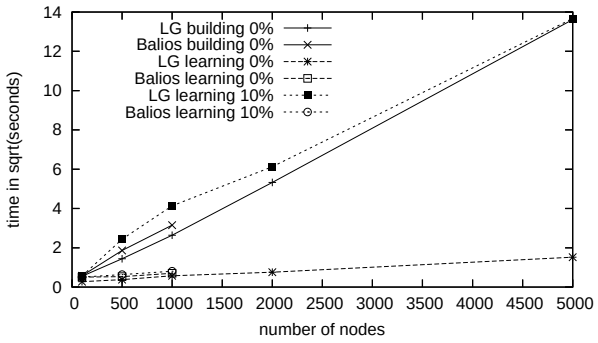

Figure 5. Comparison with EM: time

defined by

$$d_i = \begin{cases} \delta_i \cdot (1 - \theta_i) & \text{if } \delta_i \geq 0 \\ \delta_i \cdot \theta_i & \text{if } \delta_i < 0 \end{cases}$$

Thus, the search direction is essentially in the direction of the gradient, but the components of the gradient are weighted by the distance to the boundary of the parameter space when following that gradient component. When $\boldsymbol{\theta}$ is on the boundary, then components of the gradient leading out of the parameter space have 0 weight.

Given the direction $\boldsymbol{d}$ we optimize the likelihood on the line segment $[\boldsymbol{\theta}, \boldsymbol{\theta}^*]$, where $\boldsymbol{\theta}^*$ is the point where the line $\boldsymbol{\theta} + \lambda \boldsymbol{d}$ ($\lambda \geq 0$) intersects the boundary of $[0,1]^n$. This linesearch is currently implemented as an un-sophisticated binary search that is guaranteed to find a local maximum (but can be quite inefficient).

At the point $\boldsymbol{\theta}'$ returned by the linesearch the gradient is again evaluated, and the process continues until the distance between two successive parameter vectors $\boldsymbol{\theta}, \boldsymbol{\theta}'$ falls below a user defined threshold.

When data is complete, then the required evaluations of likelihood values and gradients are easily performed on the likelihood graph. When data is incomplete, then we have to use an approximate evaluation based on random samples for the unobserved atoms in the likelihood graph. Any sampling method could be used for this purpose, including Gibbs sampling or MCMC methods. We use the following sampling scheme: to estimate

$$L(\boldsymbol{\theta} \mid \boldsymbol{a}_{obs}) = \sum_{\boldsymbol{a}_{mis}} P_{\boldsymbol{\theta}}^{\mathcal{D}}(\boldsymbol{a}_{obs}, \boldsymbol{a}_{mis}) \qquad (3)$$

we generate a sample $\boldsymbol{a}_{mis,1}, \ldots, \boldsymbol{a}_{mis,k}$, and estimate (3) as

$$\frac{1}{k} \sum_{i=1}^{k} P_{\boldsymbol{\theta}}^{\mathcal{D}}(\boldsymbol{a}_{obs}, \boldsymbol{a}_{mis,i}).$$

An initial sample is drawn from a uniform distribution. After each termination of a linesearch we resample components of each $\boldsymbol{a}_{mis,i}$ according to their conditional distribution given $\boldsymbol{a}_{obs}$ and the other components of $\boldsymbol{a}_{mis,i}$ (according to the current parameters). This is similar to performing one step in Gibbs sampling, only that the values of several variables are changed in one step. Also, estimates here are not obtained by a sequence of samples from one chain, but by one sample each from k independent chains. In the experiments reported in Section 3 we used samplesize $k = 30$.

Partial derivatives are computed analogously, based on the same samples as used for the likelihood estimate.

3. Experiments

In our first experiment we compare the performance of the likelihood graph method against the implementation of EM on decomposed networks that is provided by the *Balios* system for Bayesian Logic Programs (BLPs) (http://www.informatik.uni-freiburg.de/∼kersting/profile). A very simple toy model for Mendelian inheritance was encoded both in the BLP and RBN language. The model contains a single unary random attribute *trait*. Random graphs with a pedigree-like structure and 100-5000 nodes were generated, and for each random pedigree instantiations of the *trait* atoms with varying percentages of missing values were sampled. A single pedigree with one sampled instantiation represents one dataset we learn from, i.e. we have $N = 1$ in (2), and the size of the dataset corresponds to the size of the input structure (this is the case in all experiments in this section).

Figure 5 shows the time used by EM learning in Balios and gradient ascent in the likelihood graph (LG) for different pedigree sizes, and for 0 and 10% missing values. Each value in the plot represents the average of

three repetitions of the experiment. The Balios system was unable to cope with the larger pedigrees, due to memory overflow.

Based on the thus limited data, we can still make the following observations: both Balios and LG require quadratic time for building the Bayesian network, respectively the likelihood graph (note that the time scale is in $\sqrt{s}$). The construction time is shown only for the case of 0% missing data, because it is very similar in the 10% case. The quadratic complexity is somewhat surprising at first, as the sizes of the constructed structures in both cases are linear in the input graph. For the likelihood graph, the quadratic complexity is explained by the fact that in our current implementation we store the input graph in a Java class that requires linear time for returning for a given node o all parents o' in the graph. Since the graph has to be queried for such parent sets at least during the decomposition of each upper ground atom node, we obtain the quadratic complexity. This complexity could be reduced by storing the input structures in a database to permit more efficient querying.

In the case of complete data, both systems required about the same time to perform the actual parameter learning on the constructed structure, and the construction time dominates the overall complexity. The only marked difference we observe in the two systems is the learning time in the presence of missing values. Here our implementation exhibits a quadratic behavior, whereas learning in Balios remains linear. The reason for this lies in our rather expensive sampling scheme, where we compute a conditional distribution for each unobserved atom $r(\boldsymbol{o})$. This computation is currently implemented using the global evaluation of the likelihood graph. Thus, each of these computations linear in the size of the graph, giving an overall quadratic complexity for all conditional distributions. Significant speedups for our method could be achieved by reducing the computation of the conditional distributions to local re-evaluations of those nodes in the likelihood graph that are actually affected by the instantiation of the single atom $r(\boldsymbol{o})$. Balios, on the other hand, already uses a Gibbs sampling scheme based on local computations.

Figure 6 shows the accuracy of the parameters learned by Balios and the likelihood graph. For both systems, we performed 5 random restarts of parameter learning, and selected the highest-scoring parameter values. As before, we show average accuracy values over 3 repetitions of the experiment. Accuracy of an estimated parameter vector $\boldsymbol{\theta}$ here is measured as $acc(\boldsymbol{\theta}) = 1/n \sum_{i=1}^{n} \mid \theta_i - \theta_{gen,i} \mid$, where $\boldsymbol{\theta}_{gen}$ are the

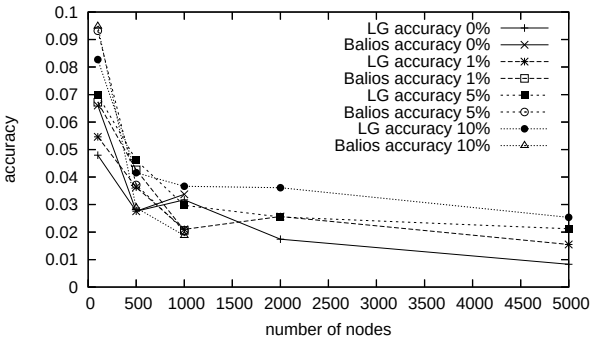

Figure 6. Comparison with EM: accuracy

Table 2. RBN with nested combination function

```
@base(u)=(blue(u):0.8,0.1);
@innercomb(w)=mean{@base(u)|u:edge(u,w)};
F_on(v)= esum{@innercomb(w)|w:edge(w,v)};
```

true parameters in the generating model (both encodings of the model contain exactly the same parameters). The results show that both methods obtain very similar accuracies, both with complete and incomplete data.

Our following experiments explore models with non-decomposable and nested combination functions, which are outside the scope of previous methods. First, we investigate the RBN given in Table 2. This is a model with a nested combination function, including a non-decomposable one. Input structures for this model consist of graphs with an *edge* relation, and an attribute *blue* on the nodes. The model represents a probabilistic attribute *on*. According to the model, the probability of $on(o)$ depends on parents o' and grandparents o'' of o. Note that $P(on(o))$ does not only depend on the number of grandparents o'' of o for which $blue(o'')$ is true/false. For example, the two objects o_1, o_2 in Figure 8 both have two blue and two

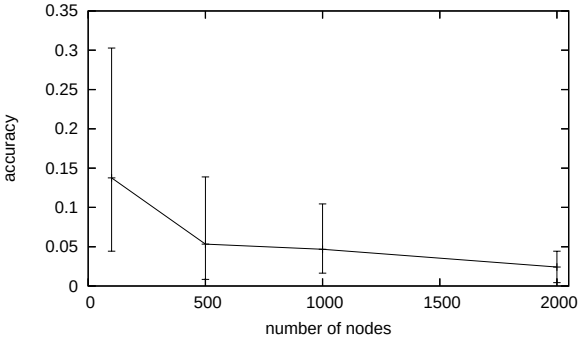

Figure 7. Accuracy result for RBN of Table 2

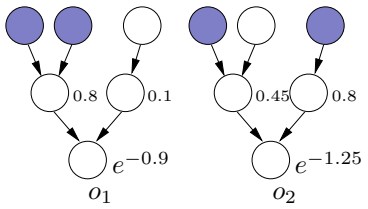

Figure 8. Nodes with different probability values

Table 3. Mixture of combination functions RBN

```
active(w) = theta_active
@basenor(w) = (active(w):theta_nor1,theta_nor2);
@basemean(w) = (active(w):theta_mean1,theta_mean2);
@baseesum(w) = (active(w):theta_esum1,theta_esum2);
@norform(v) = n-or{@basenor(w)|w:edge(v,w)};
@meanform(v) = mean{@basemean(w)|w:edge(v,w)};
@esumform(v) = esum{@baseesum(w)|w:edge(v,w)};
@redcomb(v)=theta_red_nor @norform(v)+
            theta_red_mean @meanform(v)+
            theta_red_esum @esumform(v);
F_on(v) = ( red(v):@redcomb(v),
           blue(v):@bluecomb(v),
           green(v):@greencomb(v));
```

non-blue grandparents. In Figure 8 the parent nodes o' are labeled with the values of `@innercomb`(o'), and o_1, o_2 with the resulting value of `F_on`. Intuitively, in this model, blueness of grandparents inhibits the attribute *on*. Parents pass this inhibition factor on to their children only depending on the relative, not the total, number of their blue parents.

Figure 7 shows the accuracy results for learning the two parameters of the the RBN in Table 2. For this, complete data was sampled from (random) graphs of different sizes (note that for this model incomplete data will just reduce the effective datasize, because an observed *on* atom will not depend on any unobserved atoms). On each dataset, parameters were learned with 10 random restarts, and the parameters obtaining the highest likelihood score were selected. For each input structure, this experiment was repeated 6 times. The graph in Figure 7 shows the average accuracy in the 6 repetitions, as well as maximal and minimal values. The results show that we can learn the parameters inside a two-level nested combination function.

In our third experiment we explore the possibility of identifying combination functions by parameter learning. We use random graphs with a node coloring *red, blue, green* as input structures, and consider two unary random attributes *active* and *on*. The *on* attribute depends via a combination function on the *active* attribute of parents in very much the same way as *on* depended recursively on *on* in the RBN of Table 1. The formulas `@meanform(v)`, `@norform(v)`, and `@esumform(v)` describe this dependency for three different combination functions. The formula `@redcomb(v)` is a weighted combination of these three combination functions with weight parameters $\boldsymbol{\theta}_{red} = \theta_{red,nor}, \theta_{red,mean}, \theta_{red,esum}$ (we are here taking some liberties with the RBN syntax, which in reality expresses this combination as a nested convex combination). According to the formula `F_on(v)`, `@redcomb(v)` is the formula that determines the probability of *on* for red nodes. The formulas `@bluecomb` and `@greencomb` (not shown in Table 3) are identical to `@redcomb(v)`, only with parameter vectors $\boldsymbol{\theta}_{blue}, \boldsymbol{\theta}_{green}$. We use a generating model in which $\theta_{red,nor} = \theta_{blue,mean} = \theta_{green,esum} = 1$, and all other weight parameters are 0, i.e. the *on* attribute is determined by a noisy-or for red nodes, by mean for blue nodes, and by esum for green nodes. We learn the parameters of the model from complete datasets sampled from an input structure with 2000 nodes (as usual, repeating the experiment 6 times with different datasets). Figure 9 illustrates the learned weight parameters. Here we have projected the three dimensional vectors $\boldsymbol{\theta}_{red}, \boldsymbol{\theta}_{blue}, \boldsymbol{\theta}_{green}$ into 2-dimensional space, such that $(1,0,0)$ is mapped to $(-1,0)$, $(0,1,0)$ is mapped to $(1,0)$, and $(0,0,1)$ is mapped to $(0,1)$. The result shows that the mixture coefficients for the red and green nodes (given by (1,0,0), respectively (0,0,1) in the generating model) were very accurately identified. The mixture coefficients for the blue nodes were not as accurately identified, i.e. the combination function *mean* was harder to identify.

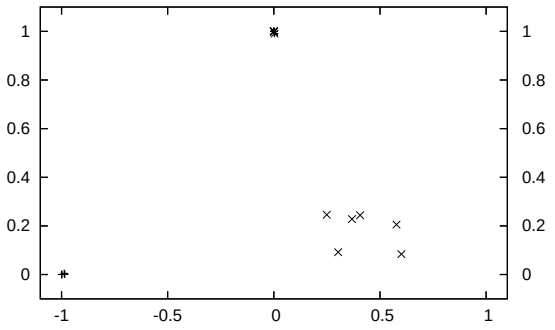

Figure 9. Learned mixture parameters

4. Conclusion

We have introduced a new method for parameter learning for relational models based on compiling a relational Bayesian network representation into a computational structure for the likelihood function and its partial derivatives. Apart from the limitation to dif-

ferentiable combination functions, the method is applicable (and implemented) for arbitrary models in the very expressive RBN language, including models with weighted combinations of combining rules, nested combining rules, or both. The addition of a new combination function to the modeling language only requires the specification for its value and partial derivative computations (the three combination functions implemented so far require about 20 lines of specific code each).

Our current prototype implementation has already provided very encouraging results. They show that our method can compete with existing alternative methods on models where both apply, and, more importantly, that the method can be applied to models which are outside the scope of previous approaches.

There is room for substantial improvement in the current implementation. Notably improved data management and better sampling techniques can be expected to lead to significant gains in efficiency. Our implementation is being integrated into the *Primula* system (http://www.cs.aau.dk/∼jaeger/Primula), and will become publicly available with the next version of *Primula*.

Another line of future work is a further reduction of the likelihood graph. Currently, nodes are only shared when they represent subformulas that are identified by the name of their probability formula as equal. However, many nodes in the likelihood graph can still represent the same function, without being merged into a single node (this is the case, for instance, for the two nodes `@norarg(A)`,`@norarg(B)` in Figure 3, which both just represent the value θ_2). Another intriguing possibility is to explicitly sum out unobserved atoms using methods borrowed from binary decision diagrams.

References

Binder, J., Koller, D., Russell, S., & Kanazawa, K. (1997). Adaptive probabilistic networks with hidden variables. *Machine Learning*, *29*, 213–244.

De Raedt, L., & Kersting, K. (2003). Probabilistic logic learning. *ACM-SIGKDD Explorations*, *5*, 31–48.

Friedman, N., Getoor, L., Koller, D., & Pfeffer, A. (1999). Learning probabilistic relational models. *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*.

Jaeger, M. (1997). Relational bayesian networks. *Proceedings of the 13th Conference of Uncertainty in Artificial Intelligence (UAI-13)* (pp. 266–273). Providence, USA: Morgan Kaufmann.

Jaeger, M. (2001). Complex probabilistic modeling with recursive relational Bayesian networks. *Annals of Mathematics and Artificial Intelligence*, *32*, 179–220.

Kersting, K. (2006). *An inductive logic programming approach to statistical relational learning*. IOS Press.

Kersting, K., & Raedt, L. D. (2001). Towards combining inductive logic programming with bayesian networks. *Proceedings of the 11th International Conference on Inductive Logic Programming (ILP-01)* (pp. 118–131).

Koller, D., & Pfeffer, A. (1997). Learning probabilities for noisy first-oreder rules. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*.

Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., & Kolobov, A. (2005). Blog: Probabilistic logic with unknown objects. *Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 1352–1359).

Muggleton, S. (1996). Stochastic logic programs. In de L. Raedt (Ed.), *Advances in inductive logic programming*, 254–264. IOS Press.

Natarajan, S., Tadepalli, P., Altendorf, E., Dietterich, T., Fern, A., & Restificar, A. (2005). Learning first-order probabilistic models with combining rules. *Proc. of the 22nd International Conference on Machine Learning (ICML-05)* (pp. 609–616).

Ngo, L., & Haddawy, P. (1995). Probabilistic logic programming and bayesian networks. *Algorithms, Concurrency and Knowledge (Proceedings ACSC95)* (pp. 286–300).

Pfeffer, A. (2000). *Probabilistic reasoning for complex systems*. Doctoral dissertation, Stanford University.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*, 107 – 136.

Rote, G. (1990). Path problems in graphs. In *Computational graph theory*, no. 7 in Computing Supplementum, 155–189. Springer.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)* (pp. 715–729).