Bayesian Logic Networks

Technical Report IAS-2009-03

Dominik Jain, Stefan Waldherr and Michael Beetz

Intelligent Autonomous Systems Group, Technische Universität München
Boltzmannstr. 3, 85748 Garching bei München, Germany
{jain, waldhers, beetz}@cs.tum.edu

This report introduces Bayesian logic networks (BLNs), a statistical relational knowledge representation formalism that is geared towards practical applicability. A BLN is a meta-model for the construction of a probability distribution from local probability distribution fragments (as in a Bayesian network) and global logical constraints formulated in first-order logic. An instance is thus a mixed network with probabilistic and deterministic constraints.
We provide the formal semantics of BLNs and explain their practical realization as implemented in the open-source software distribution called ProbCog, which supports learning and a wide range of inference algorithms.

Contents

1 Introduction

When modelling relational domains in the context of AI applications, where both expressiveness and tractability are key, we need to be able to cope with both a high degree of complexity as well as a high degree of uncertainty. Representation formalisms must therefore combine ways of abstractly specifying rules that generalize across domains of relevant objects with probabilistic semantics. In the field that has emerged as statistical relational learning, a number of such formalisms have recently been proposed [5].

Among the most expressive such formalisms are Markov logic networks, which elegantly extend first-order logic to a probabilistic setting by attaching weights to formulas. The weighted formulas collectively represent a template for the construction of undirected graphical models (i.e. Markov random fields). Unfortunately, parameter learning in MLNs is an ill-posed problem and approximate inference is typically expensive even for conceptually simple queries. Other approaches are loosely based on Bayesian networks [1, **?**], thus enabling exact and efficient learning methods that even scale to large amounts of training data – a much-needed quality in real-world applications. However, these formalisms typically sacrifice expressiveness for tractability, as they usually support only local probabilistic constraints, while even simple relational properties required on a global level, such as the transitivity or symmetry of a relation, cannot be modelled.

The representation formalism we propose in this work, Bayesian logic networks (BLNs), is a reasonable compromise in this regard. Its probabilistic components are based on conditional probability distribution templates (for the construction of a Bayesian network), which can straightforwardly be obtained from statistical data. Global constraints are supported by a second model component, a template for the construction of a constraint network, in which we represent deterministic constraints using first-order logic.

2 Formalism and Semantics

A *Bayesian logic network* is a template for the construction of a mixed network [7] with probabilistic and deterministic dependencies. Formally, a Bayesian logic network (BLN) $\mathcal{B}$ is a tuple $(\mathcal{D}, \mathcal{F}, \mathcal{L})$, where

- $\mathcal{D} = (\mathcal{T}, S, E, t)$ comprises the model's fundamental *declarations*. $\mathcal{T}$ is a taxonomy of types, which is represented as a directed forest (T, I), where T is the actual set of types and $I \subset T \times T$ is the *generalizes* relation (inverse is-a), i.e. $(T_i, T_j) \in I$ iff T_i is a generalization of T_j. S is a set of signatures of functions, and E is a set of (abstract) entities that are to exist in all instantiations, whose types are given by the function

  $$t : E \to 2^T \setminus \{\emptyset\}$$

  which maps every entity to the non-empty subset of types $T = \left\{T_1, \ldots, T_{|T|}\right\}$ it belongs to. The function t thus induces a cover of the set of entities with

sets $E_{T_i} = \{e \in E \mid T_i \in t(e)\}$. (We assume that t is consistent with $\mathcal{T}$, i.e. if $(T_i, T_j) \in I$, then $e \in E_{T_j}$ implies $e \in E_{T_i}$.)

The set S contains the signature of every function f, defining the domain and the range of the function in terms of types, i.e.

$$(f, (T_{i_1}, \ldots, T_{i_n}), T_r) \in S \;\Leftrightarrow\; f : E_{T_{i_1}} \times \cdots \times E_{T_{i_n}} \to E_{T_r}$$

Logical predicates are simply boolean functions, i.e. functions that map to $E_{T_r} = \mathbb{B}$, and we implicitly assume that the corresponding type symbol *Boolean* is always contained in T and that $\mathbb{B} = \{\mathit{True}, \mathit{False}\} \subseteq E$.

We regard a set E_{T_r} that corresponds to a type $T_r \in T$ which appears as the return type of a function as a (fixed) *domain*, i.e. as a fixed set of entities that must be fully contained in E, whereas the extensions of other types may vary from instantiation to instantiation (and the corresponding subsets of E may even be empty).

- $\mathcal{F}$ is a set of *fragments* of conditional probability distributions. Every fragment defines a dependency of an abstract random variable $f(p_1, \ldots, p_n)$ (the *fragment variable*) on a set of other abstract random variables (the *parents*), where f is one of the functions defined in S, and the parameters $p_1, \ldots, p_n$ are either variables typed according to f's signature or entities in E belonging to the respective type. The dependencies are simply encoded in a *conditional probability table* (CPT), which defines, for every setting of the parent variables (i.e. every element in the domain product of the parent variables, as specified by the ranges in the functions' signatures), a probability distribution over the elements in the domain of the fragment variable (i.e. the range of f).

  Additionally, a fragment may define preconditions for its applicability, which may involve arbitrary logical statements about the parameters $p_1, \ldots, p_n$ (or parameters that can be functionally determined by these).

  Note that for regular fragments, it is necessary to require a *domain restriction*, i.e. we require the set of variables appearing in the parents of $f(p_1, \ldots, p_n)$ to be limited to the variables in $p_1, \ldots, p_n$ (or variables whose values can be functionally determined from these). Otherwise, the conditional probability distribution of the fragment variable would be dependent on a variable number of parents (depending on the extension of the free variable's domain), which cannot be captured in a fixed-size CPT. We can nevertheless deal with such cases by associating with the fragment a *combination function* that specifies how the set of parents is to be aggregated to yield a single discrete value on which the fragment variable could depend.

- The set $\mathcal{L}$ consists of formulas in first-order logic (with equality) over the functions/predicates defined in S, which represent hard deterministic dependencies. Such formulas may help us to model global constraints that cannot concisely be representated by the conditional probability fragments in the set $\mathcal{F}$, which represent local dependencies by definition.

Instantiation. For any given set of entities E', whose types are given by a function $t' : E' \to 2^T \setminus \emptyset$, a Bayesian logic network $\mathcal{B}$ defines a ground mixed network $M_{\mathcal{B},E'} = (X, D, G, P, C)$ as follows:

- E and t in $\mathcal{B}$ are augmented to include E', t'. The instantiation is valid only if in the resulting set E, all type-specific subsets are non-empty, i.e. $\forall T_i \in T.\ E_{T_i} \neq \emptyset$.
- The set of random variables X contains, for each function $(f, (T_{i_1}, \ldots, T_{i_n}), T_r) \in S$ and each tuple of applicable entities $(e_1, \ldots, e_n) \in E_{T_{i_1}} \times \cdots \times E_{T_{i_n}}$, one element $X_i = f(e_1, \ldots, e_n)$. The corresponding domain $D_i \in D$ is simply E_{T_r}.
- The conditional probability table $P_i \in P$ applicable to a random variable $X_i = f(e_1, \ldots, e_n)$ is determined by $\mathcal{F}$, which must contain exactly one fragment for f whose preconditions are met given the actual parameters. The connectivity of the directed graph G is such that there is a directed edge from every parent to the fragment variable – as indicated by the applicable fragment.
- For every grounding of every formula in $\mathcal{L}$ (obtained by substituting quantified variables by the applicable elements of E accordingly), the set C contains one constraint $C_i = (S_i, R_i)$, where S_i, the scope of the constraint, is the product of the return type domains of the functions mentioned in the ground formula, and R_i is the relation indicating the combinations of values for which the formula is satisfied. (For universally quantified formulas, we generate a separate constraint for each binding of the quantified variables to keep the individual constraints small.)

In the special case where $\mathcal{L} = \emptyset$, the mixed network contains no constraints and is thus equivalent to the Bayesian network (X, D, G, P).

For purposes of inference, we may generally convert any mixed network $M_{\mathcal{B},E'}$ into an *auxiliary Bayesian network* $B_{\mathcal{B},E'}$, allowing us to leverage the large body of approximate inference algorithms available for Bayesian networks. $B_{\mathcal{B},E'}$ is constructed from $M_{\mathcal{B},E'}$ by adding to X for every constraint $C_i = (S_i, R_i) \in C$ a boolean auxiliary variable A_i that represents the corresponding constraint and has as its parents in G all the nodes that the constraint depends on (as indicated by its scope S_i). The probability table P_i associated with A_i contains (as its entry for the value *True*) 1 for every configuration of parents contained in R_i and 0 for all other configurations. Since all constraints are required to be satisfied, when we perform inference in the auxiliary Bayesian network, we condition on the auxiliary variables, requiring that they all take on a value of *True*. Therefore, if $|C| = k$, we have

$$P_{M_{\mathcal{B},E'}}(X = x) = P_{B_{\mathcal{B},E'}}(X = x \mid A_1 = \mathit{True}, \ldots, A_k = \mathit{True}) \tag{1}$$

Note, however, that most Bayesian network inference algorithms will perform poorly in the presence of unlikely evidence, and setting the auxiliary constraint variables to *True* typically constitutes such unlikely evidence.

Example. Let's imagine a simple "parent–child" scenario, in which we have entities of the types $T = \{T_1 = \text{parentT}, T_2 = \text{childT}, T_3 = \text{propertyT}\}$. In particular, we want to determine whether the *isParentof* relation holds given the properties of a parent p and a child c. The properties that parents and children can have are defined by the type T_3, and the functions `propParent(p)` and `propChild(c)` are defined to map parents and children to their respective property represented by an element of E_{T_3}. Therefore,

$$\begin{aligned} S = \{&(\mathit{isParentOf}, (\mathit{parentT}, \mathit{childT}), \mathit{Boolean}),\\ &(\mathit{propParent}, \mathit{parentT}, \mathit{propertyT}),\\ &(\mathit{propChild}, \mathit{childT}, \mathit{propertyT})\} \end{aligned}$$

Since *propertyT* is a fixed domain, the set of entities E will contain the constants of this domain, i.e. $E = E_{\mathit{propertyT}} = \{A1, A2\}$, while the model contains no entities for the types *parentT* and *childT*. As shown in Figure 1, the set $\mathcal{F}$ contains three fragments,

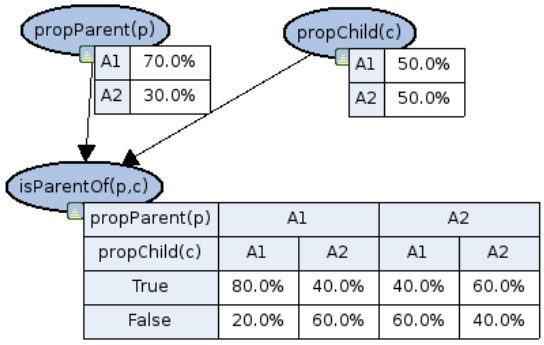

Figure 1: Graphical representation of the conditional probability fragments in the "parent-child" scenario (one for each of the functions declared in the model). The conditional distribution of *isParentOf* indicates that it is more likely for parents to have the same property as their children.

one that defines the conditional probability distribution of *isParentOf* given the attributes of the parent and the child, i.e. $P(\text{isParentof}(p, c) \mid \text{propParent}(c), \text{propChild}(c))$, and two for the marginal distributions of the properties. The set $\mathcal{L}$ contains a single logical formula that states that every child must have exactly two parents:

$$\begin{aligned} &\forall x.\ \exists p_1, p_2.\ \mathit{isParentOf}(p_1, x) \wedge \mathit{isParentOf}(p_2, x) \wedge \neg(p_1 = p_2) \wedge \\ &\quad \neg\exists p_3.\ \mathit{isParentOf}(p_3, x) \wedge \neg(p_1 = p_3) \wedge \neg(p_2 = p_3) \end{aligned}$$

We can instantiate this model for any (non-empty) set of parents and children. Figure 2 shows the auxiliary Bayesian network for an instantiation with two parents, $\{X, X2\}$, and two children, $\{Y, Y2\}$. The nodes `GF1` and `GF0` are the auxiliary constraint variables introduced for the logical constraint in $\mathcal{L}$, which results in two ground formulas (one for each child).

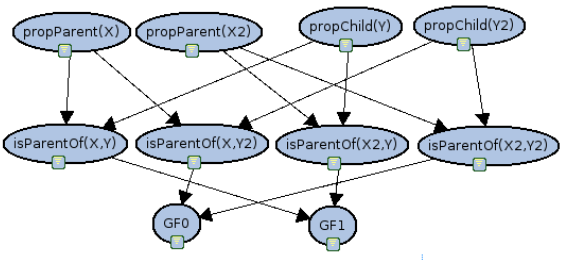

Figure 2: An auxiliary Bayesian network obtained for an instantiation of the parent-child model involving two parents and two children.

3 Representing Bayesian Logic Networks in Practice

In this section, we show how a Bayesian logic network $\mathcal{B} = (\mathcal{D}, \mathcal{F}, \mathcal{L})$ can be represented in practice. The declarations $\mathcal{D}$ are defined in a text file, in which we declare the types, the entities belonging to these types and the signatures of functions. For the declaration of fragments contained in $\mathcal{F}$, we make use of a graphical editor, while the set $\mathcal{L}$ of logical formulas is once again defined in a simple text file. The concrete syntax and semantics are discussed below.

3.1 Declarations

We define the set of types T, the set E of known entities (and, implicitly, the function t assigning types to these entities), as well as the set of signatures S in a text file as follows:

```
1 type Type1, Type2 isa Type1, Domain1;
2
3 guaranteed Domain1 dom11, dom12, dom13;
4
5 random Domain1 Function1(Type1);
6 random Boolean Predicate1(Type2);
```

Note that we use a notation that is largely equivalent to the one used for BLOG models [8].[1]

The keyword `type` is used to list the set of types T, and `isa` is used to build up the taxonomy. The keyword `guaranteed` is used to list constant symbols referring to entities that belong to a particular type and are guaranteed to be contained in all instantiations of the model (which is mandatory for fixed domains, i.e. types that appear as function ranges, such as `Domain1`). The declaration of guaranteed domain elements does not, however, necessarily prohibit further elements from being added

[1] The background is that, in special cases, BLNs may coincide with BLOG models, and using the same syntax thus allows the application of BLOG-specific software to BLN models.

to the type's extension during instantiation. The keyword `random` precedes a function signature declaration, which serves as the basis for the instantiation of random variables. In line 5, we define a function that maps from `Type1` to `Domain1`, and in line 6 a predicate applicable to entities of `Type2`.

In addition to these declarations, we may want to make statements about functional dependencies between the entities connected by a relation. The declaration of such dependencies makes it possible to perform a functional lookup (provided that the predicate itself is an evidence predicate, i.e. its full extension is always given as evidence). For example, imagine the model of a sequence of actions in which we provide the `after(a1,a2)` relation, which states that an action `a1` happens after `a2`. In this model, it is may be unacceptable for a given action to not have exactly one predecessor and one successor (ignoring, for the moment, the problem of defining the first and the last action in a closed sequence). We declare functional relations using *relation keys*: We may define any (sub-)tuple of the relation's arguments as a key for a functional lookup. A relation key definition thus takes a function f and declares some of its arguments to form such a key. The relation key then guarantees that for any tuple of key objects, there exists exactly one functionally determined object (or tuple of objects). The syntax is as follows:

```
1 type action;
2
3 random Boolean after(action, action);
4
5 relationKey after(a1,_)
6 relationKey after(_,a2)
```

In such declarations, an underscore indicates a functionally determined argument while the remaining arguments (which can be arbitrarily named in the declaration) make up the key. In the above example, we declare two relation keys for the `after` relation. In line 5, we state that for an action `a2`, there exists exactly one other action that precedes it, i.e. there is exactly one action to which $a1$ is related via the relation `after` such that $a1$ can be functionally mapped to the action to which it is related. Line 6 declares the inverse mapping. Functional dependencies are used to look up variables whose values could not otherwise be determined during the instantiation of a fragment (see below). Because we may have fragments that are not applicable at all times (see Section 3.2.1), the requirement that the function be total may be relaxed.

3.2 Conditional Probability Fragments

The second defining part of our BLN $\mathcal{B}$, the set $\mathcal{F}$, specifies generalized conditional probability fragments for the functions/predicates defined in $\mathcal{D}$, indicating the dependencies of generalized random variables (i.e. usually function terms where at least some of the arguments are variables). We use a graphical representation method to define $\mathcal{F}$ in a *fragment network*, which makes it easy to create generalized random variables and their dependencies: We model dependencies similar to pure Bayesian networks, where the arcs indicate dependencies. Usually, we will manually define only the connectivity

of the fragment network and learn distribution parameters and variable domains from data. The template structure is the foundation for the materialization of the probabilistic part of ground mixed network $M_{\mathcal{B},E'}$: For every function and every tuple of parameters, we instantiate a node and determine its parents in the ground network according to the sub-structure of the fragment network that is applicable.

By default, every node defines a fragment and thus represents a fragment variable. The parents of a node may be fragment variables themselves, interconnecting different fragments in the network. However, to keep the fragment network clearly arranged, a fragment may also be completely detached from other fragments. We then define the parents of the fragment variable using auxiliary nodes that do not represent fragments themselves. The # operator allows us to declare such auxiliary parent nodes without simultaneously declaring a fragment (see Figure 3). Apart from clarity, it may also be practical to do this if a fragment variable is to depend on two variables referring to the same function, or to allow context-specific naming of the meta-variables appearing as parameters of the fragment variable (which, for obvious reasons, must also appear in the parents).

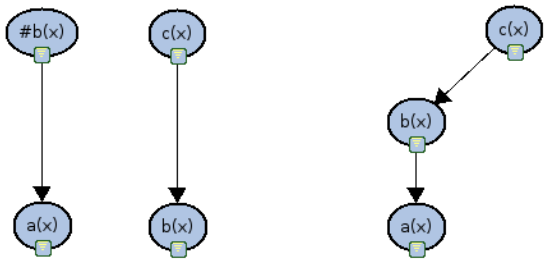

Figure 3: Demonstration of equivalent fragments using the # operator. To the left, the three fragments for a, b and c are defined in two detached parts, using an auxiliary node in the specification of the fragment for a, whereas to the right, the three fragments are defined in a single connected network without auxiliary parent nodes.

3.2.1 Preconditions

The fragment network may contain more than one fragment for any function declared in the set of signatures S. To decide which fragment is applicable, we need to define appropriate preconditions. This is, for example, useful when we want to instantiate different structures of the ground network depending on the evaluation of given evidence variables.

Preconditions are formulated as special parent nodes of the fragment variable. One very simple way of defining a precondition is to prefix a boolean parent (i.e. a parent referring to a boolan function) with the + operator: By using the + operator on a parent node, we state that the fragment in question is to apply only if (for the variable

assignment in question) the parent (an evidence variable) evaluates to true. A usage example is shown in Figure 4.

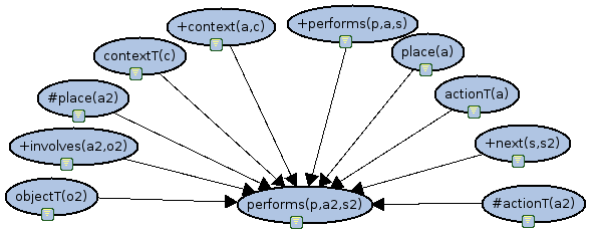

Figure 4: Preconditions using the + operator. In the fragment for $performs(p, a2, s2)$ (person p performs action $a2$ in situation $s2$), we for example require that $next(s, s2)$ holds by prefixing the parent with the + operator, since $performs(p, a2, s2)$ also depends on properties relating to the successor situation s of $s2$. By declaring this requirement, we also enable the functional lookup from $s2$ to s (*next* being a functional predicate).

A more elaborate way of specifying preconditions is through logical *precondition nodes*. Precondition nodes allow us to create completely different substructures in the ground network depending on the evaluation of a formula given in first-order logic (for formula syntax, see Section 3.3). When the formula is evaluated during instantiation, the meta-variables that appear in the fragment variable are always bound to their respective values and can be used in the formula as expected. Any other variables appearing in the formula must be (either universally or existentially) quantified.

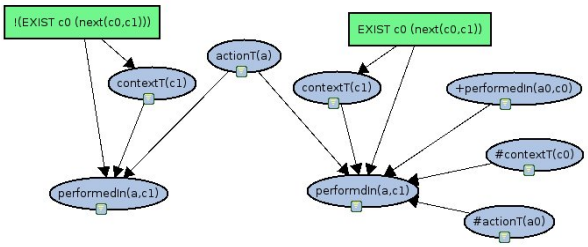

Figure 5: Precondition Nodes

As an example, consider the problem we encounter when modelling a sequence of time steps, where the first time step needs to be treated differently, because it has no predecessor to influence its properties. Figure 5 illustrates how we can differentiate the respective cases: We only want to instantiate the template to the right whenever the situation has a predecessor, while we want to instantiate the left hand template otherwise. This example thus illustrates how one could represent time series models such as hidden Markov models or, more generally, dynamic Bayesian networks as BLNs.

It should be noted that all the predicates appearing in preconditions (be it in the case of the + operator or using the more elaborate precondition nodes) are required to be evidence predicates. Their extensions must therefore be fully specified in any evidence database, which is why we generally apply the *closed-world assumption* by default to such predicates (i.e. any value that is not explicitly given in the evidence database is assumed to be *False*).

3.2.2 Domain Nodes (Per-Constant Dependencies)

In previous examples, a fragment's conditional distribution was dependent on values of functions declared in S. In some cases, it can be useful to have conditional distributions dependent on elements of a (fixed) domain, i.e. in effect, to have a separate conditional distribution per constant belonging to the domain. To support this, the parent of a fragment variable may simply bear the name of one of the meta-variables appearing in the fragment variable. The parent then then stands for the values/constants in the domain that corresponds to the type the meta-variable belongs to.

As an example, consider Figure 6, where we want to model the habits of people participating in meals. People are characerized by their type, as are the meals (e.g. breakfast, lunch, dinner). We allow several types of goods and utensils that may be used and consumed respectively. We use domain nodes for utensils and goods in this model (u and g), because here, we do not want to distinguish between the actual objects being used or consumed but are interested only in the class of the object being used or consumed (but we do want to distinguish between different people and different meals of the same type because of additional attributes).

3.2.3 Combination Functions

We can deal with the case that a fragment variable can be dependent on a variable number of parents by associating with the fragment a combination function. This function specifies how the set of parents is to be aggregated to yield a single discrete value on which the fragment variable could depend. At this time Bayesian logic networks support one such combination function, the *logical disjunction* which is a useful tool to declare one variable as a disjunction of other variables (as one would obtain it using existential quantification). The function represents a disjunction over the objects of a stated type. We use the `=OR:` operator for the conjunction function and it is represented as a node with the syntax =OR:pred1$(v_{i_1}, \ldots, v_{i_n})|v_{j_1}, \ldots, v_{j_m}$ whose parent is pred2$(v_1, \ldots, v_l)$ where $\{v_{i_1}, \ldots, v_{i_n}\} \uplus \{v_{j_1}, \ldots, v_{j_m}\} = \{v_1, \ldots, v_l\}$. During instantiation, where the typed variables v_i are substituted by entities E_i, it has the following semantics,

$$\mathit{pred1}(E_{i_1}, \ldots, E_{i_n}) = \begin{cases} 1 & \text{if } \exists (E_{j_1}, \ldots, E_{j_m}) \in \mathrm{dom}(v_{j_1}) \times \ldots \times \mathrm{dom}(v_{j_1}) \\ & \text{s.t. } \mathit{pred2}(E_1, \ldots, E_l) = \mathit{True} \\ 0 & \text{otherwise} \end{cases}$$

In other words, the `=OR:` operator sets a given prediate *pred1* to true iff there is an assignment of the free variables $v_{j_1}, \ldots, v_{j_m}$ such that *pred2* evaluates to true.

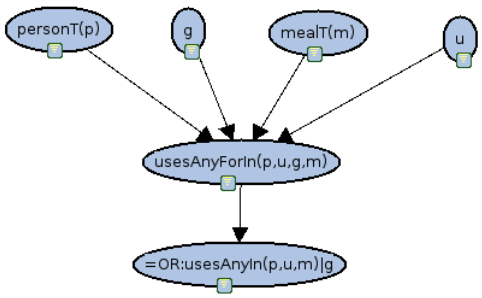

Figure 6: Domain nodes and the OR combination function

While we could achieve the same using a logical formula in $\mathcal{L}$ that uses existential quantification, the `=OR:` operator can be the more efficient solution (depending on the inference algorithm).

We currently support only the disjunction combination function. One can, however, think of many other combination functions that might be useful and that might be included in the future, as modelling demands increase, e.g. combination functions that represent the maximum, mean or mode operations.

To clarify the use of the `=OR:` operator, consider again Figure 6. The predicate `usesAnyForIn(p,u,g,m)` evaluates to true whenever a person `p` uses a good a utensil `u` to consume some good `g` in meal `m`, while the predicate `usesAnyIn(p,u,m)` should evaluate to true whenever the person uses the utensil for anything in a meal. This is done with the help of the `=OR:` operator: In the fragment for *usesAnyIn*, g is a free variable, causing a variable size set of parents depending on the extension of its domain, which is mapped to a single value by the combination function.

3.3 Logical Formulas

The set $\mathcal{L}$ in a Bayesian logic network may contain a set of logical formulas constraining the set of possible worlds. Formulas are declared in a text file, which contains one formula per line, each terminated by a period. We use the following plain text syntax for the logical operators:

`^`	conjunction
`v`	disjunction
`=>`	implication
`<=>`	biimplication

Quantifiers are expressed as follows:

`EXIST vars (formula)`	existential quantification
`FORALL vars (formula)`	universal quantification

where `vars` is a comma-separated list of variables (words beginning with lowercase letters) and `formula` is any complex formula (in which these variables appear)

expressed using the logical operators above. Universal quantification may be implicit, i.e. if a formula contains free variables, these variables are assumed to be universally quantified.

In the logical coupling, non-boolean functions in the BLN are converted to predicates with the value of the function as an additional parameter, i.e. the value expression $f(A, B) = C$ of function $(f, (T_1, T_2), T_3) \in S$ would become $f(A, B, C)$.

As an extension to pure first-order logic, we also support (the concise formulation of) cardinality restrictions, as many real-world relations are subject to such restrictions. A cardinality restriction takes the form

$$\text{count}(\text{rel}(x_1, \ldots, x_n) \mid x_{i_1}, \ldots, x_{i_m}) \in C \tag{2}$$

where $m \leq n$ and $S \subset \mathbb{N}$, the semantics being that if the parameters that are given by the index set $\{i_1, \ldots, i_m\}$ are bound to some fixed vector of constants, the number of bindings for the remaining parameters for which the relation holds true is required to be in the set C.

For example, in a model of parent-child relationships in which we do not want to distinguish between males and females we want to state that every child should have exactly two parents. This can easily be expressed by using the count constraint:

```
count(parent(x,y)|x) = 2
```

Although this formula could also be expressed in pure first-order logic, we can think of relations with a higher count constraint, where the specification would become rather lengthy.

4 Inference

There are several different approaches to inference in Bayesian logic networks. A straightforward approach would be to instantiate a ground network, i.e. either the ground auxiliary Bayesian network or the ground mixed network, and then apply one of many standard inference algorithms. More elaborate methods will seek to abstract away from the ground instances, exploiting the repeated structures that result from the application of the template model. BLNs may also be translated into MLNs, which effectively enables MLN inference algorithms to be applied to BLNs.

4.1 Inference in the Ground Auxiliary Bayesian Network

By converting a ground mixed to a standard Bayesian network, we can leverage a large body of approximate inference algorithms devised in the past, including

- likelihood weighting [2]
- backward sampling [3]
- importance sampling based on evidence prepropagation (EPIS-BN) [12]

- Gibbs sampling [4]

The first three are based on importance sampling, while Gibbs sampling is a Markov Chain Monte Carlo (MCMC) method.

Moreover, exact methods, such as Pearl's algorithm or algorithms based on variable elimination, may be applicable to smaller instances.

It should be noted, however, that Bayesian network algorithms typically cannot handle high degrees of determinism and fail to produce (usable) results if evidence with low probability is provided. As all logical constraints in a BLN may result in such low-probability evidence, especially large ground networks often call for methods that specifically take determinism into account.

4.2 Inference in the Ground Mixed Network

In [7], Mateescu and Dechter propose two exact inference algorithms for mixed networks, one based on bucket elimination, the other on AND/OR search. An approximate inference algorithm that adapts importance sampling to the specific requirements of mixed networks, SampleSearch, is introduced in [6]. MC-SAT [9], another approximate technique, which was originally proposed for Markov logic networks, is an algorithm that is well-suited to inference in mixed networks, as it essentially reduces, within a slice sampling framework, probabilistic inference to repeated SAT sampling. One of its drawbacks is, however, that inference problems are typically not significantly simplified if the number of hard logical constraints is reduced.

4.3 Inference Via Translation into Markov Logic Networks

A BLN $B = (\mathcal{D}, \mathcal{F}, \mathcal{L})$ can be straightforwardly translated into a Markov logic network (MLN) $L = \{(F_i, w_i)\}$ [10], i.e. a set of weighted formulas in first-order logic that collectively represent a template for the construction of a Markov random field. The translation simply involves transforming every fragment in $\mathcal{F}$ into a set of weighted logical formulas as follows: For every entry of the fragment's CPT, we include in L the conjunction of the fragment variable and all its parent variables (and, if the fragment has preconditions, add the conjunction of these preconditions as a further conjunct) with the logarithm of the probability value as the weight of the formula (if the probability value is 0, we choose a sufficiently large negative value). We model non-boolean functions in $\mathcal{D}$ as functional predicates, adding the return value as an additional parameter. The hard logical constraints in $\mathcal{L}$ are added directly to L, using a sufficiently large positive weight.

The translation procedure outlined above allows us to apply MLN inference algorithms in order to perform inference in BLNs. As inference algorithms, especially new developments such as lifted first-order belief propagation [11] could be of interest.

5 Parameter Learning

Parameter learning in Bayesian logic networks consists of learning the conditional probabilities that parameterize the fragment network. Given a database containing observations that fully specify a possible world for a given set of entities, the probabilities are learnt based on maximum likelihood, which, as in Bayesian networks, reduces to counting relative frequencies of parent-child occurrences in the data, as these already constitute a sufficient statistic.

6 Software Tools

BLNs are supported by a number of software tools, implemented in the ProbCog software suite for statistical relational learning, including

- a graphical editor for the creation of fragment networks
- command-line tools for learning and inference along with graphical wrapper applications (see Figure 7)
- tools for the scripting of relational stochastic processes in order to generate relational data
- a model server for easy integration of probabilistic reasoning into other projects

The software suite is freely available for download.[2]

7 Conclusion and Future Work

Bayesian logic networks are a simple yet powerful representation formalism that unifies probabilistic and logical representations. As meta-models for the generation of mixed networks or (auxiliary) Bayesian networks, BLNs can be straightforwardly trained even on large data sets and support a wide range of exact and approximate inference techniques.

As directions for future work, it is conceivable to add support for soft logical constraints (as in Markov logic networks), which could increase expressiveness (at the expense of learning performance, however). Furthermore, there is still a great need for more efficient inference mechanisms if statistical relational models are to be used as reasoning components in cognitive technical systems, where near-real-time performance is highly desirable. Especially approaches that seek to exploit repeated sub-structures in ground networks (i.e. lifted or semi-lifted approaches) would be worthwhile exploring.

[2] http://ias.cs.tum.edu/research/probcog

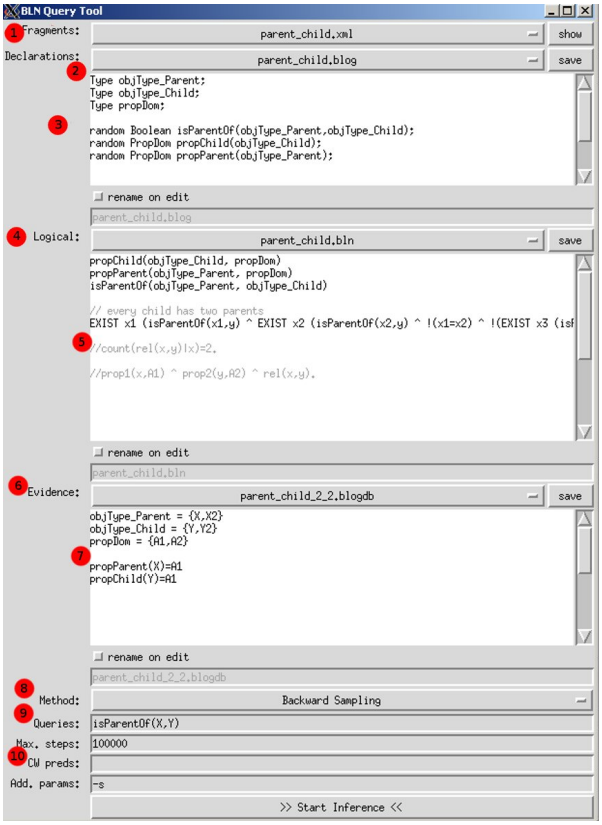

Figure 7: The inference tool

References

[1] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.

[2] R. M. Fung and K.-C. Chang. Weighing and integrating evidence for stochastic simulation in bayesian networks. In *UAI '89: Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 209–220, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[3] R. M. Fung and B. D. Favero. Backward simulation in bayesian networks. In R. L. de Mántaras and D. Poole, editors, *UAI*, pages 227–234. Morgan Kaufmann, 1994.

[4] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[5] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[6] V. Gogate and R. Dechter. SampleSearch: A Scheme that Searches for Consistent Samples. In *AISTATS*, 2007.

[7] R. Mateescu and R. Dechter. Mixed deterministic and probabilistic networks. *Annals of Mathematics and Artificial Intelligence*, 2008.

[8] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI*, pages 1352–1359, 2005.

[9] H. Poon and P. Domingos. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *AAAI*. AAAI Press, 2006.

[10] M. Richardson and P. Domingos. Markov Logic Networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[11] P. Singla and P. Domingos. Lifted first-order belief propagation. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 1094–1099. AAAI Press, 2008.

[12] C. Yuan and M. J. Druzdzel. An importance sampling algorithm based on evidence pre-propagation. In *In Proceedings of the Nineteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 624–631. Morgan Kaufmann Publishers, 2003.