Deriving a Stationary Dynamic Bayesian Network from a Logic Program with Recursive Loops

Yi-Dong Shen[1] and Qiang Yang[2]

[1] Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing 100080, China
ydshen@cqu.edu.cn
[2] Department of Computing Science,
Hong Kong University of Science and Technology, Hong Kong, China
qyang@cs.ust.hk

Abstract. Recursive loops in a logic program present a challenging problem to the PLP framework. On the one hand, they loop forever so that the PLP backward-chaining inferences would never stop. On the other hand, they generate cyclic influences, which are disallowed in Bayesian networks. Therefore, in existing PLP approaches logic programs with recursive loops are considered to be problematic and thus are excluded. In this paper, we propose an approach that makes use of recursive loops to build a stationary dynamic Bayesian network. Our work stems from an observation that recursive loops in a logic program imply a time sequence and thus can be used to model a stationary dynamic Bayesian network without using explicit time parameters. We introduce a Bayesian knowledge base with logic clauses of the form $A \leftarrow A_1, ..., A_l, true, Context, Types$, which naturally represents the knowledge that the A_is have direct influences on A in the context $Context$ under the type constraints $Types$. We then use the well-founded model of a logic program to define the direct influence relation and apply SLG-resolution to compute the space of random variables together with their parental connections. We introduce a novel notion of influence clauses, based on which a declarative semantics for a Bayesian knowledge base is established and algorithms for building a two-slice dynamic Bayesian network from a logic program are developed.

Keywords: Probabilistic logic programming (PLP), the well-founded semantics, SLG-resolution, stationary dynamic Bayesian networks.

1 Introduction

Probabilistic logic programming (PLP) is a framework that extends the expressive power of Bayesian networks with first-order logic [18,21]. The core of the PLP framework is a backward-chaining procedure, which generates a Bayesian

S. Kramer and B. Pfahringer (Eds.): ILP 2005, LNAI 3625, pp. 330–347, 2005.

network graphic structure from a logic program in a way quite like query evaluation in logic programming. Therefore, existing PLP methods use a slightly adapted *SLD-* or *SLDNF-resolution* [16] as the backward-chaining procedure.

Recursive loops in a logic program is of the form

$$A_1 \leftarrow ...A_2 \leftarrow ...A_3 \leftarrow ...A_4 \leftarrow ... \tag{1}$$

where for any $i \geq 1$, A_i is the same as A_{i+1} up to variable renaming. Such loops present a challenging problem to the PLP framework. On the one hand, they loop forever so that the PLP backward-chaining inferences would never stop. On the other hand, they generate cyclic influences, which are disallowed in Bayesian networks. Two representative approaches have been proposed to avoid recursive loops. The first one is by Ngo and Haddawy [18] and Kersting and De Raedt [15], who restrict to considering only acyclic logic programs [1]. The second approach, proposed by Glesner and Koller [11], uses explicit time parameters to avoid occurrence of recursive loops. It enforces acyclicity using time parameters in the way that every predicate has a time argument such that the time argument in the rule head is at least one time step later than the time arguments of the predicates in the rule body. In this way, each predicate $p(X)$ is changed to $p(X,T)$ and each clause like $p(X) \leftarrow q(X)$ is rewritten into $p(X,T1) \leftarrow q(X,T2), T2 = T1 - 1$, where T, $T1$ and $T2$ are time parameters.

In this paper, we propose a solution to the problem of recursive loops under the PLP framework. Our method is not restricted to acyclic programs, nor does it rely on explicit time parameters. Instead, it makes use of recursive loops to derive a stationary dynamic Bayesian network. We will make two novel contributions. First, we introduce the *well-founded* semantics [29] of logic programs to the PLP framework; in particular, we use the well-founded model of a logic program to define the direct influence relation and apply *SLG-resolution* [4] (or *SLTNF-resolution* [25]) to make the backward-chaining inferences. As a result, termination of the PLP backward-chaining process is guaranteed. Second, we observe that under the PLP framework recursive loops (cyclic influences) define feedbacks, thus implying a time sequence. For instance, the following two clauses

$$\begin{aligned}&aids(X) \leftarrow aids(X),\\ &aids(X) \leftarrow aids(Y), contact(X,Y)\end{aligned}$$

model that the direct influences on $aids(X)$ (in the current time slice t) come from whether X was already infected with aids earlier (in the last time slice $t-1$) or whether X had contact with someone Y who was infected (in time slice $t-1$). As a result, recursive loops of form (1) imply a time sequence

$$A_1 \underbrace{\leftarrow ...}_{t} A_2 \underbrace{\leftarrow ...}_{t-1} A_3 \underbrace{\leftarrow ...}_{t-2} A_4 \leftarrow ... \tag{2}$$

It is this observation that leads us to viewing a logic program with recursive loops as a special temporal model. Such a temporal model corresponds to a stationary dynamic Bayesian network and thus can be compactly represented as a two-slice dynamic Bayesian network.

1.1 Preliminaries and Notation

We assume the reader is familiar with basic ideas of Bayesian networks [19] and logic programming [16]. In particular, we assume the reader is familiar with the well-founded semantics [29] as well as SLG-resolution [4]. Here we review some basic concepts concerning dynamic Bayesian networks (DBNs). DBNs are introduced to model the evolution of the state of the environment over time [14]. Briefly, a DBN is a Bayesian network whose random variables are subscripted with time slices (or intervals). For instance, $Weather_{t-1}$, $Weather_t$ and $Weather_{t+1}$ are random variables representing the weather situations in time slices $t-1$, t and $t+1$, respectively. We can then use a DBN to depict how $Weather_{t-1}$ influences $Weather_t$.

A DBN is represented by describing the intra-probabilistic relations between random variables in each individual time slice t and the inter-probabilistic relations between the random variables of each two consecutive time slices $t-1$ and t. If both the intra- and inter-probabilistic relations are the same for all time slices (in this case, the DBN is a repetition of a Bayesian network over time; see Figure 1), the DBN is called a *stationary* DBN [22]; otherwise it is called a *flexible* DBN [11]. As far as we know, most existing DBN systems reported in the literature are stationary DBNs.

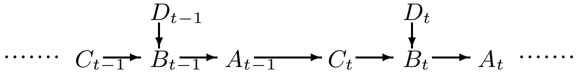

Fig. 1. A stationary DBN structure

In a stationary DBN as shown in Figure 1, the state evolution is determined by random variables like C, B and A, as they appear periodically and influence one another over time. Such variables are called *state variables.* Note that D is not a state variable. Due to the characteristic of stationarity, a stationary DBN is often compactly represented as a two-slice DBN.

Definition 1. A *two-slice* DBN for a stationary DBN consists of two consecutive time slices, $t-1$ and t, which describes (1) the intra-probabilistic relations between the random variables in slice t and (2) the inter-probabilistic relations between the random variables in slice $t-1$ and the random variables in slice t.

A two-slice DBN models a feedback system. For convenience, we depict feedback connections with dashed edges. Moreover, we refer to nodes coming from slice $t-1$ as *state input nodes.*

Example 1. The stationary DBN of Figure 1 can be represented by a two-slice DBN as shown in Figure 2. It can also be represented by a two-slice DBN starting from a different state input node (C_{t-1} or B_{t-1}). These two-slice DBN structures are equivalent in the sense that they can be unrolled into the same stationary DBN (Figure 1).

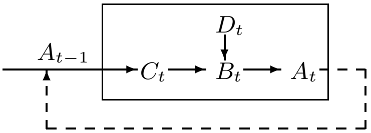

Fig. 2. A two-slice DBN structure (a feedback system)

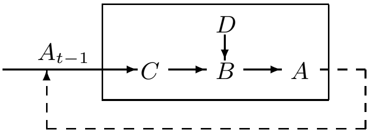

Fig. 3. A simplified two-slice DBN structure

Observe that in a two-slice DBN, all random variables except state input nodes have the same subscript t. In the sequel, the subscript t is omitted for simplification of the structure. For instance, the two-slice DBN of Figure 2 is simplified to Figure 3.

In the rest of this section, we introduce some necessary notation for logic programs. We use $p(.)$ to refer to any atom/predicate whose predicate symbol is p and use $p(\overrightarrow{X})$ to refer to $p(X_1, ..., X_n)$ where all X_is are variables. There is one special atom, *true*, which is always logically true. A predicate $p(\overrightarrow{X})$ is *typed* if its arguments $\overrightarrow{X}$ are typed so that each argument takes on values in a well-defined finite domain. A (general) logic program P is a finite set of clauses of the form

$$A \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \tag{3}$$

where A, the B_is and C_js are atoms. We use $HU(P)$ and $HB(P)$ to denote the Herbrand universe and Herbrand base of P, respectively, and use $WF(P) = <I_t, I_f>$ to denote the well-founded model of P, where $I_t, I_f \subseteq HB(P)$, and every A in I_t is true and every A in I_f is false in $WF(P)$. By a *(Herbrand) ground instance* of a clause C we refer to a ground instance of C that is obtained by replacing all variables in C with some terms in $HU(P)$.

A logic program P is a *positive logic program* if no negative literal occurs in the body of any clause. P is a *Datalog program* if no clause in P contains function symbols. P is an *acyclic logic program* if there is a mapping *map* from the set of ground instances of atoms in P into the set of natural numbers such that for any ground instance $A \leftarrow B_1, ..., B_k, \neg B_{k+1}, ..., \neg B_n$ of any clause in P, $map(A) > map(B_i)$ $(1 \leq i \leq n)$ [1]. P is said to have the *bounded-term-size property* w.r.t. a set of predicates $\{p_1(.), ..., p_t(.)\}$ if there is a function $f(n)$ such that for any $1 \leq i \leq t$ whenever a top goal $G_0 = \leftarrow p_i(.)$ has no argument whose term size exceeds n, no atoms in any backward derivations for G_0 have an argument whose term size exceeds $f(n)$ [28].

2 Bayesian Knowledge Bases

Definition 2. A *Bayesian knowledge base* is a triple $<PB \cup CB, T_x, CR>$, where

– $PB \cup CB$ is a logic program, each clause in PB being of the form

$$p(.) \leftarrow \underbrace{p_1(.), ..., p_l(.)}_{direct\ influences}, true, \underbrace{B_1, ..., B_m, \neg C_1, ..., \neg C_n}_{context},$$

$$\underbrace{member(X_1, DOM_1), ..., member(X_s, DOM_s)}_{type\ constraints} \qquad (4)$$

where (i) the predicate symbols $p, p_1, ..., p_l$ only occur in PB and (ii) $p(.)$ is typed so that for each variable X_i in it with a finite domain DOM_i (a list of constants) there is an atom $member(X_i, DOM_i)$ in the clause body.

- T_x is a set of conditional probability tables (CPTs) of the form $\mathbf{P}(p(.)|p_1(.),$ ..., $p_l(.))$, each being attached to a clause (4) in PB.
- CR is a combination rule such as *noisy-or, min* or *max* [15,18,22].

A Bayesian knowledge base contains a logic program that can be divided into two parts, PB and CB. PB defines a direct influence relation, each clause (4) saying that the atoms $p_1(.), ..., p_l(.)$ have direct influences on $p(.)$ in the context that $B_1, ..., B_m, \neg C_1, ..., \neg C_n$, $member(X_1, DOM_1), ...,$ $member(X_s, DOM_s)$ is true in $PB \cup CB$ under the well-founded semantics. Note that the special literal *true* is used in clause (4) to mark the beginning of the context; it is always true in the well-founded model $WF(PB \cup CB)$. For each variable X_i in the head $p(.)$, $member(X_i, DOM_i)$ is used to enforce the type constraint on X_i, i.e. the value of X_i comes from its domain DOM_i. CB assists PB in defining the direct influence relation by introducing some auxiliary predicates (such as $member(.)$) to describe contexts. Clauses in CB do not describe direct influences. Note that recursive loops are allowed in PB and CB.

In this paper, we focus on Datalog programs, although the proposed approach applies to logic programs with the bounded-term-size property (w.r.t. the set of predicates appearing in the heads of PB) as well. Datalog programs are widely used in database and knowledge base systems [27] and have a polynomial time complexity in computing their well-founded models [29]. In the sequel, we assume that except for the predicate $member(.)$, $PB \cup CB$ is a Datalog program.

For each clause (4) in PB, there is a unique CPT, $\mathbf{P}(p(.)|p_1(.), ..., p_l(.))$, in T_x specifying the degree of the direct influences. Such a CPT is shared by all instances of clause (4).

A Bayesian knowledge base has the following important property.

Theorem 1. *(1) All unit clauses in PB are ground. (2) Let* $G_0 = \leftarrow p(.)$ *be a goal with p being a predicate symbol occurring in the head of a clause in PB. Then all answers of* G_0 *derived from* $PB \cup CB \cup \{G_0\}$ *by applying SLG-resolution are ground.*

For simplicity of presentation, in the sequel for each clause (4) in PB, we omit all of its type constraints $member(X_i, DOM_i)$ $(1 \leq i \leq s)$. Therefore, when we say that the context $B_1, ..., B_m$, $\neg C_1, ..., \neg C_n$ is true, we assume that the related type constraints are true as well.

3 Declarative Semantics

In this section, we formally describe the space of random variables and the direct influence relation defined by a Bayesian knowledge base KB. We then derive formulas for computing probability distributions induced by KB.

3.1 Space of Random Variables and Influence Clauses

A Bayesian knowledge base KB defines a direct influence relation over a subset of $HB(PB)$. Recall that any random variable in a Bayesian network is either an input node or a node on which some other nodes (i.e. its parent nodes) in the network have direct influences. Since an input node can be viewed as a node whose direct influences come from an empty set of parent nodes, we can define a space of random variables from a Bayesian knowledge base KB by taking all unit clauses in PB as input nodes and deriving the other nodes iteratively based on the direct influence relation defined by PB. Formally, we have

Definition 3. The *space of random variables* of KB, denoted $\mathcal{S}(KB)$, is recursively defined as follows:

1. All unit clauses in PB are random variables in $\mathcal{S}(KB)$.
2. Let $A \leftarrow A_1, ..., A_l, true, B_1, ..., B_m, \neg C_1, ..., \neg C_n$ be a ground instance of a clause in PB. If the context $B_1, ..., B_m, \neg C_1, ..., \neg C_n$ is true in the well-founded model $WF(PB \cup CB)$ and $\{A_1, ..., A_l\} \subseteq \mathcal{S}(KB)$, then A is a random variable in $\mathcal{S}(KB)$. In this case, each A_i is said to have a *direct influence* on A.
3. $\mathcal{S}(KB)$ contains only those ground atoms satisfying the above two conditions.

Definition 4. For any random variables A, B in $\mathcal{S}(KB)$, we say A is *influenced by* B if B has a direct influence on A, or for some C in $\mathcal{S}(KB)$ A is influenced by C and C is influenced by B. A *cyclic influence* occurs if A is influenced by itself.

Let $WF(PB \cup CB) = <I_t, I_f>$ be the well-founded model of $PB \cup CB$ and let $I_{PB} = \{p(.) \in I_t | p$ occurs in the head of some clause in $PB\}$. The following result shows that the space of random variables is uniquely determined by the well-founded model.

Theorem 2. $\mathcal{S}(KB) = I_{PB}$.

Theorem 2 suggests that the space of random variables can be computed by applying an existing procedure for the well-founded model such as SLG-resolution or SLTNF-resolution. Since SLG-resolution has been implemented as the well-known XSB system [23], in this paper we apply it for the PLP backward-chaining inferences. Let $\{p_1, ..., p_t\}$ be the set of predicate symbols occurring in the heads of clauses in PB, and let $G_0 = \leftarrow p_1(\overrightarrow{X_1}), ..., p_t(\overrightarrow{X_t})$ be a top goal where $\overrightarrow{X_i}$ and $\overrightarrow{X_j}$ are disjoint for any $i \neq j$. During the process of evaluating G_0, SLG-resolution stores answers of each $p_i(\overrightarrow{X_i})$ in a space called *table*, denoted $\mathcal{T}_{p_i(\overrightarrow{X_i})}$.

Algorithm 1: Computing random variables.

1. Compute all answers of G_0 by applying SLG-resolution to $PB \cup CB \cup \{G_0\}$.
2. Return $\mathcal{S}'(KB) = \bigcup_{i=1}^{t} \mathcal{T}_{p_i(\overrightarrow{X_i})}$.

Theorem 3. *Algorithm 1 terminates, yielding a finite set $\mathcal{S}'(KB) = \mathcal{S}(KB)$.*

When evaluating G_0, SLG-resolution will construct a proof tree, rooted at $\leftarrow p_i(\overrightarrow{X_i})$, for each subgoal $p_i(\overrightarrow{X_i})$ $(1 \leq i \leq t)$ [4]. For each answer A' of $p_i(\overrightarrow{X_i})$ in $\mathcal{S}'(KB)$ there must be a success branch (i.e. a branch starting at the root node and ending at a node marked with *success*) in the tree that generates the answer. Let

$$p_i(.) \leftarrow A_1, ..., A_l, true, B_1, ..., B_m, \neg C_1, ..., \neg C_n$$

be the k-th clause in PB that was applied to expand the root goal $\leftarrow p_i(\overrightarrow{X_i})$ in the success branch and let θ be the composition of all the mgus (most general unifiers) along the branch. Then $A' = p_i(.)\theta$ and $(A_1, ..., A_l, true, B_1, ..., B_m, \neg C_1, ..., \neg C_n)\theta$ is true in $WF(PB \cup CB)$. In this case, we refer to

$$k.\;\; p_i(.)\theta \leftarrow A_1\theta, ..., A_l\theta \tag{5}$$

as an *influence clause* (the prefix "k." would be omitted sometimes for simplicity of presentation). Obviously, every success branch in the proof tree for $\leftarrow p_i(\overrightarrow{X_i})$ produces an influence clause. All influence clauses from the proof trees for $\leftarrow p_i(\overrightarrow{X_i})$ $(1 \leq i \leq t)$ constitute the *set of influence clauses* of KB, denoted $\mathcal{I}_{clause}(KB)$.

Let $G_0 = \leftarrow p_1(\overrightarrow{X_1}), ..., p_t(\overrightarrow{X_t})$ be a top goal as in Algorithm 1. The above process of generating influence clauses can be described more formally as follows.

Algorithm 2: Computing influence clauses.

1. Compute all answers of G_0 by applying SLG-resolution to $PB \cup CB \cup \{G_0\}$, while for each success branch starting at a root goal $\leftarrow p_i(\overrightarrow{X_i})$ $(1 \leq i \leq t)$, we collect an influence clause from the branch into $\mathcal{I}_{clause}(KB)$.
2. Return $\mathcal{I}_{clause}(KB)$.

Influence clauses have two principal properties.

Theorem 4. *Let $A \leftarrow A_1, ..., A_l$ be an influence clause. Then A and all the A_is are ground atoms.*

Theorem 5. *For any $A, A_i \in HB(PB)$, A_i has a direct influence on A, which is derived from the k-th clause in PB, if and only if there is an influence clause in $\mathcal{I}_{clause}(KB)$ of the form $k.\ A \leftarrow A_1, ..., A_i, ..., A_l$.*

Corollary 1. *For any $A \in HB(PB)$, A is in $\mathcal{S}(KB)$ if and only if there is an influence clause in $\mathcal{I}_{clause}(KB)$ whose head is A.*

Example 2. Let us consider the AIDS program, adapted from [11]. Let KB_1 be a Bayesian knowledge base with $CB_1 = \emptyset$ and PB_1 containing the following eleven clauses:

1-3. $aids(pi).\quad (i = 1, 2, 3)$
4. $aids(X) \leftarrow aids(X).$
5. $aids(X) \leftarrow aids(Y), contact(X, Y).$
6-11. $contact(pi, pj).\quad (i, j = 1, 2, 3 \text{ and } i \neq j)$

Let $G_0 = \leftarrow aids(X), contact(Y, Z)$. Algorithm 2 will generate two proof trees rooted at $\leftarrow aids(X)$ and $\leftarrow contact(Y, Z)$, respectively, and produce the set $\mathcal{I}_{clause}(KB_1)$ with the following eighteen influence clauses:

1-3. $aids(pi).\quad (i = 1, 2, 3)$
4. $aids(pi) \leftarrow aids(pi).\quad (i = 1, 2, 3)$
5. $aids(pi) \leftarrow aids(pj), contact(pi, pj).\quad (i, j = 1, 2, 3 \text{ and } i \neq j)$
6-11. $contact(pi, pj).\quad (i, j = 1, 2, 3 \text{ and } i \neq j)$

For example, the third line above represents six influence clauses that are derived by applying the 5-th clause in PB_1 to the root goal $\leftarrow aids(X)$.

3.2 Probability Distributions Induced by *KB*

For any random variable A in $\mathcal{S}(KB)$, we use $pa(A)$ to denote the set of random variables that have direct influences on A; namely $pa(A)$ consists of random variables in the body of all influence clauses whose head is A. Assume that the probability distribution $\mathbf{P}(A|pa(A))$ is available (see Section 4.2). Furthermore, we make the following independence assumption.

Assumption 1. For any random variable A in $\mathcal{S}(KB)$, we assume that given $pa(A)$, A is probabilistically independent of all random variables in $\mathcal{S}(KB)$ that are not influenced by A.

Theorem 6. *When no cyclic influence occurs, the probability distribution induced by KB is $\mathbf{P}(\mathcal{S}(KB)) = \prod_{A_i \in \mathcal{S}(KB)} \mathbf{P}(A_i|pa(A_i))$ under the independence assumption.*

When there are cyclic influences, we cannot have a partial order on $\mathcal{S}(KB)$. By Definition 4 and Theorem 5, any cyclic influence, say "A_1 is influenced by itself," must be resulted from a set of influence clauses in $\mathcal{I}_{clause}(KB)$ of the form

$$\begin{array}{rcl} A_1 & \leftarrow & ..., A_2, ... \\ A_2 & \leftarrow & ..., A_3, ... \\ & ...... & \\ A_n & \leftarrow & ..., A_1, ... \end{array} \tag{6}$$

Observe that these clauses generate a chain of direct influences

$$A_1 \leftarrow A_2 \leftarrow A_3 \leftarrow ... \leftarrow A_n \leftarrow A_1$$

which defines a feedback connection. Since a feedback system can be modeled by a two-slice DBN (see Section 1.1), the above clauses represent the same knowledge as the following ones

$$\begin{array}{rcl} A_1 & \leftarrow & ..., A_2, ... \\ A_2 & \leftarrow & ..., A_3, ... \\ & ...... & \\ A_n & \leftarrow & ..., A_{1_{t-1}}, ... \end{array} \qquad (7)$$

Here the A_is are state variables and $A_{1_{t-1}}$ is a state input variable. That is, A_1 being influenced by itself becomes A_1 being influenced by $A_{1_{t-1}}$. By applying this transformation (from clauses (6) to (7)), we can get rid of all cyclic influences and obtain a *generalized set* $\mathcal{I}_{clause}(KB)_g$ of influence clauses from $\mathcal{I}_{clause}(KB)$.[1]

Let $\mathcal{V}_{input}(KB)$ be the set of state input variables introduced in $\mathcal{I}_{clause}(KB)_g$ and let $\mathcal{S}(KB)_g = \mathcal{S}(KB) \cup \mathcal{V}_{input}(KB)$. By extending the independence assumption from $\mathcal{S}(KB)$ to $\mathcal{S}(KB)_g$ and defining $pa(A_i)$ over $\mathcal{I}_{clause}(KB)_g$, we obtain the following result.

Theorem 7. *When $\mathcal{I}_{clause}(KB)$ produces cyclic influences, the probability distribution induced by KB is $\mathbf{P}(\mathcal{S}(KB)_g) = \prod_{A_i \in \mathcal{S}(KB)_g} \mathbf{P}(A_i|pa(A_i))$ under the independence assumption.*

4 Building a Bayesian Network from a Bayesian Knowledge Base

4.1 Building a Two-Slice DBN Structure

From a Bayesian knowledge base KB, we can derive a set of influence clauses $\mathcal{I}_{clause}(KB)$, which defines the same direct influence relation over the same space $\mathcal{S}(KB)$ of random variables as $PB \cup CB$ does (see Theorem 5). For any influence clause $A \leftarrow A_1, ..., A_l$, its head A and the body atoms A_js are all ground and true in the well-founded model. Therefore, given a probabilistic query together with some evidences, we can depict a network structure from $\mathcal{I}_{clause}(KB)$, which covers the random variables in the query and evidences, by backward-chaining the related random variables via the direct influence relation.

Let Q be a probabilistic query and E a set of evidences, where all random variables (with time subscripts removed, if any) come from $\mathcal{S}(KB)$ (i.e., they

[1] Depending on starting from which influence clause to generate an influence cycle, a different generalized set containing different state input variables would be obtained. All of them are equivalent in the sense that they define the same feedback connections and can be unrolled into the same stationary DBN.

are heads of some influence clauses in $\mathcal{I}_{clause}(KB)$). Let TOP consist of these random variables. An *influence network* of Q and E, denoted $\mathcal{I}_{net}(KB)_{Q,E}$, is constructed from $\mathcal{I}_{clause}(KB)$ using the following algorithm.

Algorithm 3: Building an influence network.

1. Initially, $\mathcal{I}_{net}(KB)_{Q,E}$ has all random variables in TOP as nodes.
2. Remove the first random variable A from TOP. For each influence clause in $\mathcal{I}_{clause}(KB)$ of the form $k.\ B \leftarrow ..., A, ...$, if B is not in $\mathcal{I}_{net}(KB)_{Q,E}$ then add B to $\mathcal{I}_{net}(KB)_{Q,E}$ as a new node and to the end of TOP. For each influence clause in $\mathcal{I}_{clause}(KB)$ of the form $k.\ A \leftarrow A_1, ..., A_l$, if $l = 0$ then add to $\mathcal{I}_{net}(KB)_{Q,E}$ an edge $A \overset{k}{\leftarrow}$. Otherwise, for each A_i in the body:
   (a) If A_i is not in $\mathcal{I}_{net}(KB)_{Q,E}$ then add A_i to $\mathcal{I}_{net}(KB)_{Q,E}$ as a new node and to the end of TOP.
   (b) Add to $\mathcal{I}_{net}(KB)_{Q,E}$ an edge $A \overset{k}{\leftarrow} A_i$.
3. Repeat step 2 until TOP becomes empty.

$\mathcal{I}_{net}(KB)_{Q,E}$ covers all random variables in TOP. Moreover, for any node A in $\mathcal{I}_{net}(KB)_{Q,E}$, its parent nodes come from the body atoms of all influence clauses of the form $k.\ A \leftarrow A_1, ..., A_l$. Each parent node A_i is connected to A via an edge $A \overset{k}{\leftarrow} A_i$, indicating that the parental relationship comes from applying an influence clause that is derived from the k-th clause in PB. We see that an influence network is a Bayesian network structure unless it contains loops (cyclic influences).

Let $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ denote an influence network that covers all random variables in $\mathcal{S}(KB)$. It is easy to show the following. First, for any node A_i in $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$, the set $parents(A_i)$ of its parent nodes is $pa(A_i)$, as defined in Theorem 6. Second, A_i is a descendant node of A_j if and only if A_i is influenced by A_j. This means that the independence assumption (Assumption 1) applies to $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ as well, and that $\mathcal{I}_{clause}(KB)$ produces a cycle of direct influences if and only if $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ contains the same loop. Combining these facts leads to the following immediate result.

Theorem 8. *When no cyclic influence occurs, the probability distribution induced by KB can be computed over $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$. That is, $\mathbf{P}(\mathcal{S}(KB)) = \prod_{A_i \in \mathcal{S}(KB)} \mathbf{P}(A_i|pa(A_i)) = \prod_{A_i \in \mathcal{S}(KB)} \mathbf{P}(A_i|parents(A_i))$ under the independence assumption.*

Let us consider influence networks with loops. Loops in an influence network are generated from recursive influence clauses of form (6). They establish feedback connections like that in Figure 3, which can be unrolled into a stationary DBN as in Figure 1. This means that an influence network with loops can be converted into a two-slice DBN, simply by converting each loop of the form

$$\begin{tikzcd}
A_1 \arrow[rrr, bend left, "k_n"] & A_2 \arrow[l, "k_1"'] & \cdots\cdots \arrow[l, "k_2"'] & A_n \arrow[l, "k_{n-1}"']
\end{tikzcd}$$

into a two-slice DBN path

$$A_1 \overset{k_1}{\leftarrow} A_2 \overset{k_2}{\leftarrow} ... \overset{k_{n-1}}{\leftarrow} A_n \overset{k_n}{\leftarrow} \mathbf{A_{1_{t-1}}}$$

by introducing a state input node $A_{1_{t-1}}$.

As illustrated in Section 1.1, a two-slice DBN is a snapshot of a stationary DBN across any two time slices, which can be obtained by traversing the stationary DBN from a set of state variables backward to the same set of state variables (i.e., state input nodes). This process corresponds to generating an influence network $\mathcal{I}_{net}(KB)_{Q,E}$ from $\mathcal{I}_{clause}(KB)$ incrementally (adding nodes and edges one by one) while wrapping up loop nodes with state input nodes (like A_{t-1}). This leads to the following algorithm for building a two-slice DBN structure, $2\mathcal{S}_{net}(KB)_{Q,E}$, directly from $\mathcal{I}_{clause}(KB)$, where Q, E and TOP are as defined in Algorithm 3.

Algorithm 4: Building a two-slice DBN structure.

1. Initially, $2\mathcal{S}_{net}(KB)_{Q,E}$ has all random variables in TOP as nodes.
2. Remove the first random variable A from TOP. For each influence clause in $\mathcal{I}_{clause}(KB)$ of the form $k.\ B \leftarrow ..., A, ...$, if B is not in $\mathcal{I}_{net}(KB)_{Q,E}$ then add B to $\mathcal{I}_{net}(KB)_{Q,E}$ as a new node and to the end of TOP. For each influence clause in $\mathcal{I}_{clause}(KB)$ of the form $k.\ \ A \leftarrow A_1, ..., A_l$, if $l = 0$ then add to $2\mathcal{S}_{net}(KB)_{Q,E}$ an edge $A \overset{k}{\leftarrow}$. Otherwise, for each A_i in the body:
   (a) If A_i is not in $2\mathcal{S}_{net}(KB)_{Q,E}$ then add A_i to $2\mathcal{S}_{net}(KB)_{Q,E}$ as a new node and to the end of TOP.
   (b) If adding $A \overset{k}{\leftarrow} A_i$ to $2\mathcal{S}_{net}(KB)_{Q,E}$ produces a loop, then add to $2\mathcal{S}_{net}(KB)_{Q,E}$ a node $A_{i_{t-1}}$ and an edge $A \overset{k}{\leftarrow} A_{i_{t-1}}$, else add an edge $A \overset{k}{\leftarrow} A_i$ to $2\mathcal{S}_{net}(KB)_{Q,E}$.
3. Repeat step 2 until TOP becomes empty.

Algorithm 4 is Algorithm 3 enhanced with a mechanism for cutting loops (item 2b), i.e. when adding the current edge $A \overset{k}{\leftarrow} A_i$ to the network forms a loop, we replace it with an edge $A \overset{k}{\leftarrow} A_{i_{t-1}}$, where $A_{i_{t-1}}$ is a state input node. This is a process of transforming influence clauses (6) to (7). Therefore, $2\mathcal{S}_{net}(KB)_{Q,E}$ is essentially built from a generalized set $\mathcal{I}_{clause}(KB)_g$ of influence clauses.

Let $\mathcal{S}(KB)_g$ be the set of random variables in $\mathcal{I}_{clause}(KB)_g$, as defined in Theorem 7. Let $2\mathcal{S}_{net}(KB)_{\mathcal{S}(KB)}$ denote a two-slice DBN structure (produced by applying Algorithm 4) that covers all random variables in $\mathcal{S}(KB)_g$. We have the following result.

Theorem 9. *When $\mathcal{I}_{clause}(KB)$ produces cyclic influences, the probability distribution induced by KB can be computed over $2\mathcal{S}_{net}(KB)_{\mathcal{S}(KB)}$. That is, $\mathbf{P}(\mathcal{S}(KB)_g) = \prod_{A_i \in \mathcal{S}(KB)_g} \mathbf{P}(A_i|pa(A_i)) = \prod_{A_i \in \mathcal{S}(KB)_g} \mathbf{P}(A_i|parents(A_i))$ under the independence assumption.*

Remark 1. Note that Algorithm 4 does not use any time parameters. It only requires the user to specify, via the query and evidences, what random variables are necessarily included in the network. Algorithm 4 builds a two-slice DBN structure for any given query and evidences whose random variables are heads of some influence clauses in $\mathcal{I}_{clause}(KB)$. When no query and evidences are provided, we may apply Algorithm 4 to build a *complete* two-slice DBN structure, $2\mathcal{S}_{net}(KB)_{\mathcal{S}(KB)}$, which covers the space $\mathcal{S}(KB)$ of random variables, by letting TOP consist of all heads of influence clauses in $\mathcal{I}_{clause}(KB)$. This is a very useful feature, as in many situations the user may not be able to present the right queries unless a Bayesian network structure is shown.

Example 3 (Example 2 continued). Suppose that we want to build a Bayesian network from KB_1 that covers $aids(p1)$, $aids(p2)$ and $aids(p3)$. We may present a query $?-\mathbf{P}(aids(p1))$ along with the evidences $aids(p2) = yes$ and $aids(p3) = no$. Thus $TOP = \{aids(p1), aids(p2), aids(p3)\}$. Algorithm 4 builds from $\mathcal{I}_{clause}$ (KB_1) a two-slice DBN structure $2\mathcal{S}_{net}(KB_1)_{Q,E}$ as shown in Figure 4 where for simplicity, edges of the form $A \overset{k}{\leftarrow}$ are omitted. Note that loops are cut by introducing three state input nodes $aids(p1)_{t-1}$, $aids(p2)_{t-1}$ and $aids(p3)_{t-1}$. We see that the two-slice DBN structure $2\mathcal{S}_{net}(KB_1)_{Q,E}$ concisely depicts a feedback system where the feedback connections are as shown in Figure 5.

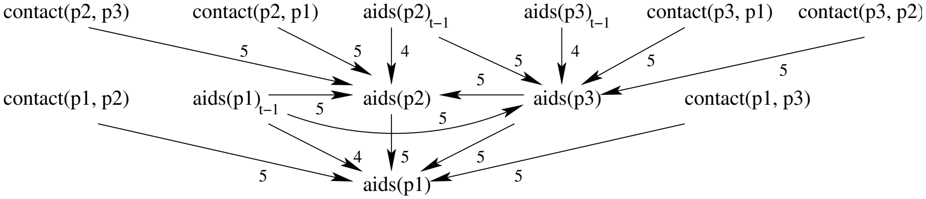

Fig. 4. A two-slice DBN structure built from the AIDS program KB_1

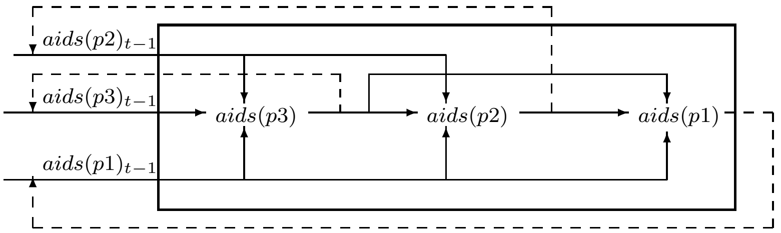

Fig. 5. The feedback connections created by the AIDS program KB_1

4.2 Building CPTs

After a Bayesian network structure $2\mathcal{S}_{net}(KB)_{Q,E}$ has been constructed from a Bayesian knowledge base KB, we associate each (non-state-input) node A in the network with a CPT. There are three cases. (1) If A only has unit clauses in

$\mathcal{I}_{clause}(KB)$, we build from the unit clauses a *prior* CPT for A as its prior probability distribution. (2) If A only has non-unit clauses in $\mathcal{I}_{clause}(KB)$, we build from the clauses a *posterior* CPT for A as its posterior probability distribution. (3) Otherwise, we prepare for A both a prior CPT (from the unit clauses) and a posterior CPT (from the non-unit clauses). In this case, A is attached with the posterior CPT; the prior CPT for A would be used, if A is a state variable, as the probability distribution of A in time slice 0 (only in the case that a two-slice DBN is unrolled into a stationary DBN starting with time slice 0; see Section 1.1).

Assume that the parent nodes of A are derived from n ($n \geq 1$) different influence clauses in $\mathcal{I}_{clause}(KB)$. Suppose these clauses share the following CPTs in T_x: $\mathbf{P}(A_1|B_1^1, ..., B_{m_1}^1)$, ..., and $\mathbf{P}(A_n|B_1^n, ..., B_{m_n}^n)$. (Recall that an influence clause prefixed with a number k shares the CPT attached to the k-th clause in PB.) Then the CPT of A is computed by combining the n CPTs in terms of the combination rule CR.

5 Related Work

A recent overview of existing representational frameworks that combine probabilistic reasoning with logic (i.e. logic-based approaches) or relational representations (i.e. non-logic-based approaches) is given by De Raedt and Kersting [6]. Typical non-logic-based approaches include probabilistic relational models (PRM), which are based on the entity-relationship (or object-oriented) model [10,13,20], and relational Markov networks, which combine Markov networks and SQL-like queries [26]. Representative logic-based approaches include frameworks based on the KBMC (Knowledge-Based Model Construction)idea [2,3,8,11,12,15,18,21], stochastic logic programs (SLP) based on stochastic context-free grammers [5,17], parameterized logic programs based on distribution semantics (PRISM) [24], and more. Most recently, a unifying framework, called *Markov logic*, has been proposed by Domingos and Richardson [7]. Markov logic subsumes first-order logic and Markov networks. Since our work follows the KBMC idea focusing on how to build a Bayesian network directly from a logic program, it is closely related to three representative existing PLP approaches: the context-sensitive PLP developed by Haddawy and Ngo [18], Bayesian logic programming proposed by Kersting and Raedt [15], and the time parameter-based approach presented by Glesner and Koller [11].

5.1 Comparison with the Context-Sensitive PLP Approach

The core of the context-sensitive PLP is a probabilistic knowledge base (PKB). In order to see the main differences from our Bayesian knowledge base (BKB), we reformulate its definition here.

Definition 5. A *probabilistic knowledge base* is a four tuple $<PD, PB, CB, CR>$, where

- PD defines a set of probabilistic predicates (*p-predicates*) of the form $p(T_1, ..., T_m, V)$ where all arguments T_is are typed with a finite domain and the last argument V takes on values from a probabilistic domain DOM_p.
- PB consists of *probabilistic rules* of the form

$$P(A_0|A_1, ..., A_l) = \alpha \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \tag{8}$$

  where $0 \leq \alpha \leq 1$, the A_is are p-predicates, and the B_js and C_ks are context predicates (*c-predicates*) defined in CB.
- CB is a logic program, and both PB and CB are acyclic.
- CR is a combination rule.

In a probabilistic rule (8), each p-predicate A_i is of the form $q(t_1, ..., t_m, v)$, which simulates an equation $q(t_1, ..., t_m) = v$ with v being a value from the probabilistic domain of $q(t_1, ..., t_m)$. For instance, let $DOM_{nbrhd} = \{average, good, bad\}$ be the probabilistic domain of $nbrhd(X)$, then the p-predicate $nbrhd(X, good)$ simulates $nbrhd(X) = good$, meaning that the neighborhood of X is *good*. The left-hand side $P(A_0|A_1, ..., A_l) = \alpha$ expresses that the probability of A_0 conditioned on $A_1, ..., A_l$ is α. The right-hand side $B_1, ..., B_m, \neg C_1, ..., \neg C_n$ is the *context* of the rule where the B_js and C_ks are c-predicates. Note that the sets of p-predicate and c-predicate symbols are disjoint. A separate logic program CB is used to evaluate the context of a probabilistic rule. As a whole, the above probabilistic rule states that for each of its (Herbrand) ground instances

$$P(A'_0|A'_1, ..., A'_l) = \alpha \leftarrow B'_1, ..., B'_m, \neg C'_1, ..., \neg C'_n$$

if the context $B'_1, ..., B'_m, \neg C'_1, ..., \neg C'_n$ is true in CB under the program completion semantics, the probability of A'_0 conditioned on $A'_1, ..., A'_l$ is α.

PKB and BKB have the following important differences.

First, probabilistic rules of form (8) in PKB contain both logic representation (right-hand side) and probabilistic representation (left-hand side) and thus are not logic clauses. The logic part and the probabilistic part of a rule are separately computed against CB and PB, respectively. In contrast, our BKB uses logic clauses of form (4), which naturally integrate the direct influence information, the context and the type constraints. These logic clauses are evaluated against a single logic program $PB \cup CB$, while the probabilistic information is collected separately in T_x.

Second, logic reasoning in PKB relies on the program completion semantics and is carried out by applying SLDNF-resolution. But in BKB, logic inferences are based on the well-founded semantics and are performed by applying SLG-resolution. The well-founded semantics resolves the problem of inconsistency with the program completion semantics, while SLG-resolution eliminates the problem of infinite loops with SLDNF-resolution. Note that the key significance of BKB using the well-founded semantics lies in the fact that a unique set of influence clauses can be derived, which lays a basis on which both the declarative and procedural semantics for BKB are developed.

Third, most importantly PKB has no mechanism for handling cyclic influences. In PKB, cyclic influences are defined to be *inconsistent* (see Definition

9 of the paper [18]) and thus are excluded (PKB excludes cyclic influences by requiring its programs be acyclic). In BKB, however, cyclic influences are interpreted as feedbacks, thus implying a time sequence. This allows us to derive a stationary DBN from a logic program with recursive loops.

Recently, Fierens, Blockeel, Ramon and Bruynooghe [9] introduced *logical Bayesian networks* (LBN). LBN is similar to PKB except that it separates logical and probabilistic information. That is, LBN converts rules of form (8) into the form

$$A_0|A_1, ..., A_l \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n$$

where the A_is are p-predicates with the last argument V removed, and the B_js and C_ks are c-predicates defined in CB. This is not a standard clause of form (3) as defined in logic programming [16]. Like PKB, LBN differs from BKB in the following: (1) it has no mechanism for handling cyclic influences (see Section 3.2 of the paper [9]), and (2) although the well-founded semantics is also used for the logic contexts, neither declarative nor procedural semantics for LBN has been formally developed.

5.2 Comparison with Bayesian Logic Programming

Building on Ngo and Haddawy's work, Kersting and De Raedt [15] introduce the framework of Bayesian logic programs. A *Bayesian logic program* (BLP) is a triple $<P, T_x, CR>$ where P is a well-defined logic program, T_x consists of CPTs associated with each clause in P, and CR is a combination rule. According to [15], we understand that a *well-defined* logic program is an acyclic positive logic program satisfying the range restriction.[2] For instance, a logic program containing clauses like $r(X) \leftarrow r(X)$ (cyclic) or $r(X) \leftarrow s(Y)$ (not range-restricted) is not well-defined. BLP relies on the least Herbrand model semantics and applies SLD-resolution to make backward-chaining inferences.

BLP has two important differences from our BKB framework. First, it applies only to positive logic programs. Due to this, it cannot handle contexts with negated atoms. (In fact, no contexts are considered in BLP.) Second, it does not allow cyclic influences. BKB can be viewed as an extension of BLP with mechanisms for handling contexts and cyclic influences in terms of the well-founded semantics. Such extension is clearly non-trivial.

5.3 Comparison with the Time Parameter-Based Approach

The time parameter-based framework proposed by Glesner and Koller [11] is also a triple $<P, T_x, CR>$, where CR is a combination rule, T_x is a set of CPTs that are represented as decision trees, and P is a logic program with the property that each predicate contains a time parameter and that in each clause the time argument in the head is at least one time step (unit) later than the time arguments in

[2] A logic program is said to be *range-restricted* if all variables appearing in the head of a clause appear in the body of the clause.

the body. This framework is implemented in Prolog, i.e. clauses are represented as Prolog rules and goals are evaluated applying SLDNF-resolution. Glesner and Koller [11] state: "... In principle, this free variable Y can be instantiated with every domain element. (This is the approach taken in our implementation.)" By this we understand that they consider typed logic programs with finite domains.

An obvious difference is that our BKB framework is devoted to modeling stationary DBNs, whereas the time parameter-based framework targets flexible DBNs. One may say that stationary DBNs can also be modeled with the time parameter-based framework, as they are special cases of flexible DBNs. This appears not the case. We observe two major limitations of the time parameter-based framework. First, it uses time steps as time slices, thus for any A and B such that A is influenced by B, A and B will not be allowed to occur in the same time slice. Due to this, we are unable to use the time parameter-based framework to model intra-probabilistic relations between the random variables within a time slice t (like those in Figure 1). Second, introducing time parameters to enforce acyclicity may lose answers to some queries. Let P be a logic program and P_t be P with additional time parameters. It is easy to prove that P_t is acyclic. Let $p(.)$ be a query and $p(., N)$ be $p(.)$ with a ground time argument N added. Then evaluating $p(., N)$ over P_t (applying SLDNF-resolution) achieves the same effect as evaluating $p(.)$ over P with some depth-bound M (i.e. derivations are cut at depth M). Since the loop problem in logic programming is undecidable in general, it is impossible to determine an appropriate depth-bound (rep. a ground time argument) for an arbitrary query without losing answers.

6 Conclusions and Discussion

We have developed an approach to deriving a stationary DBN from a logic program with recursive loops. We observed that recursive loops in a logic program imply a time sequence and thus can be used to model a stationary DBN without using explicit time parameters. We introduced a Bayesian knowledge base with logic clauses of form (4). These logic clauses naturally integrate the direct influence information, the context and the type constraints, and are evaluated against a single logic program $PB \cup CB$ under the well-founded semantics. We established a declarative semantics for a Bayesian knowledge base and developed algorithms that build a two-slice DBN from a Bayesian knowledge base.

We emphasize the following two points. First, recursive loops (cyclic influences) and recursion through negation are unavoidable in modeling real-world domains, thus the well-founded semantics together with its top-down inference procedures is well suitable for the PLP application. Second, recursive loops define feedbacks, thus implying a time sequence. This allows us to derive a two-slice DBN from a logic program containing no time parameters. We point out, however, that the user is never required to provide any time parameters during the process of constructing such a two-slice DBN. A Bayesian knowledge base defines a unique space of random variables and a unique set of influence clauses, whether it contains recursive loops or not. From the viewpoint of logic, these random variables are ground atoms in the Herbrand base; their truth values are determined by the well-founded model and will never change over time.[3] Therefore, a Bayesian network is built over these random variables, independently of any time factors (if any). Once a two-slice DBN has been built, the time intervals over it would become clearly specified, thus the user can present queries and evidences over the DBN using time parameters at his/her convenience.

Acknowledgements

We thank the anonymous referees for their constructive comments, which have greatly improved the presentation. Yi-Dong Shen is supported in part by Chinese National Natural Science Foundation (grant # 60373052 and 60421001) and Trans-Century Training Program Foundation for the Talents by the Chinese Ministry of Education. Qiang Yang is supported in part by HKUST/6187/04E.

References

1. K. R. Apt and M. Bezem, Acyclic programs, *New Generation Computing* 29(3):335-363 (1991).
2. F. Bacchus, Using first-order probability logic for the construction of Bayesian networks, in: *Proc. of the Ninth Conference on Uncertainty in Artificial Intelligence*, 1994, pp. 219-226.
3. J. S. Breese, Construction of belief and decision networks, *Computational Intelligence* 8(4):624-647 (1992).
4. W. D. Chen, T. Swift and D. S. Warren, Efficient top-down computation of queries under the well-founded semantics, *Journal of Logic Programming* 24(3):161-199 (1995).
5. J. Cussens, Stochastic logic programs: sampling, inference and applications, in: *Proc. of The Sixteenth Annual Conference on Uncertainty in Artificail Intelligence*, 2000, pp. 115-122.
6. L. De Raedt and K. Kersting, Probabilistic logic learning, *SIGKDD Explorations* 5(1):31-48 (2003).
7. P. Domingos and M. Richardson, Markov logic: a unifying framework for statistical relational learning, in: *Proc. of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields* , Banff, Canada, 2004, pp. 49-54.
8. I. Fabian and D. A. Lambert, First-order Bayesian reasoning. In: *Proc. of the 11th Australian Joint Conference on Articial Intelligence*, number 1502 in LNAI. Springer, 1998, pp. 131-142.
9. D. Fierens, H. Blockeel, J. Ramon and M. Bruynooghe, Logical Bayesian networks, in: *3rd Workshop on Multi-Relational Data Mining*, Seattle, USA, 2005
10. L. Getoor, *Learning Statistical Models from Relational Data*, Ph.D. thesis, Stanford University, 2001.
11. S. Glesner and D. Koller, Constructing flexible dynamic belief networks from first-order probabilistic knowledge bases, in: C. Froidevaux and J. Kohlas, eds., *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning under Uncertainty*, Fribourg, Switzerland, July 1995, pages 217-226.

[3] However, from the viewpoint of Bayesian networks the probabilistic values of these random variables (i.e. values from their probabilistic domains) may change over time.

12. R. Goldman and E. Charniak, A language for construction of belief networks, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15(3):196-208 (1993).
13. M. Jaeger, Relational Bayesian networks, in: *Proc. of The Thirteenth Annual Conference on Uncertainty in Artificail Intelligence*, 1997, pp. 266-273.
14. K. Kanazawa, D. Koller and S. Russell, Stochastic simulation algorithms for dynamic probabilistic networks, in: *Proc. of the Eleventh Annual Conference on Uncertainty in Artificail Intelligence*, 1995.
15. K. Kersting and L. De Raedt, Bayesian logic programs, in: J. Cussens and A. Frisch, eds, *Work-in-Progress Reports of the Tenth International Conference on Inductive Logic Programming*, London,U.K., 2000. (A full version: Technical Report 151, University of Freiburg Institute for Computer Science.)
16. J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, Berlin, 1987.
17. S. Muggleton, Stochastic logic programs, in: *Advances in Inductive Logic Programming*, IOS Press, 1996.
18. L. Ngo and P. Haddawy, Answering queries from context-sensitive probabilistic knowledge bases, *Theoretical Computer Science*, 171:147-177 (1997).
19. J. Pearl, *Probabilistic Resoning in Intelligent Systems: Networks of Plausible inference*, Morgan Kaufmann, 1988.
20. A. Pfeffer and D. Koller, Semantics and inference for recursive probability models, in: *Proc. of the Seventeenth National Conference on Artificial Intelligence*, AAAI Press, 2000, pp.538-544.
21. D. Poole, Probabilistic Horn abduction and Bayesian networks, *Artificial Intelligence* 64(1):81-129 (1993).
22. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.
23. K. Sagonas, T. Swift, D.S. Warren, J. Freire and P. Rao, *The XSB Programmer's Manual (Version 1.8).* Department of Computer Science, SUNY at Stony Brook. Available from http://www.cs.sunysb.edu/ sbprolog/xsb-page.html.
24. T. Sato and Y. Kameya, Parameter learning of logic programs for symbolic-statistical modeling, *Journal of Artificial Intelligence Research* 15:391-454 (2001).
25. Y. D. Shen, J. H. You and L. Y. Yuan, Enhancing global SLS-resolution with loop cutting and tabling mechanisms, *Theoretical Computer Science* 328(3):271-287(2004).
26. B. Taskar, P. Abeel and D. Koller, Discriminative probabilistic models for relational data, in: *Proc. of the Eighteenth Conf. on Uncertainty in Artificial Intelligence*, Edmonton, Canada, 2002, pp.485-492.
27. J. D. Ullman, *Database and Knowledge-Base Systems*, vols. I and II, Computer Science Press, 1988.
28. A. Van Gelder, Negation as failure using tight derivations for general logic programs, *Journal of Logic Programming* 6(1&2):109-133 (1989).
29. A. Van Gelder, K. Ross, J. Schlipf, The well-founded semantics for general logic programs, *J. ACM* 38(3):620-650 (1991).