Reasoning with Recursive Loops Under the PLP Framework

YI-DONG SHEN
Chinese Academy of Sciences

Recursive loops in a logic program present a challenging problem to the PLP (Probabilistic Logic Programming) framework. On the one hand, they loop forever so that the PLP backward-chaining inferences would never stop. On the other hand, they may 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 article, we propose a novel solution to this problem by making use of recursive loops to build a stationary dynamic Bayesian network. We introduce a new PLP formalism, called a *Bayesian knowledge base*. It allows recursive loops and contains logic clauses of the form $A \leftarrow A_1, \ldots, A_l, \mathit{true}, \mathit{Context}, \mathit{Types}$, which naturally formulate the knowledge that the A_is have direct influences on A in the context *Context* under the type constraints *Types*. We 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. This establishes a clear declarative semantics for a Bayesian knowledge base. We view a logic program with recursive loops as a special temporal model, where backward-chaining cycles of the form $A \leftarrow \cdots A \leftarrow \cdots$ are interpreted as feedbacks. This extends existing PLP approaches, which mainly aim at (nontemporal) relational models.

Categories and Subject Descriptors: D.1.6 [**Programming Techniques**]: Logic Programming; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and constraint programming*; G.3 [**Probability and Statistics**]—*Probabilistic algorithms (including Monte Carlo)*

General Terms: Languages, Theory

Additional Key Words and Phrases: Logic programming, Bayesian networks, recursive loops, cyclic influences, the well-founded model

ACM Reference Format:

Shen, Y.-D. 2008. Reasoning with recursive loops under the PLP framework. ACM Trans. Comput. Logic 9, 4, Article 27 (August 2008), 31 pages. DOI = 10.1145/1380572.1380576 http://doi.acm.org/10.1145/1380572.1380576

A preliminary version of this article appeared in *Proceedings of the 15th International Conference on Inductive Logic Programming* (Bonn, Germany).
This work is supported in part by the National Natural Science Foundation of China (grants 60721016, 60673103, 60421001, and 60373052).
Author's address: State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China; email: ydshen@ios.ac.cn.

1. INTRODUCTION

Since Pearl's [1988] pioneering work, Bayesian networks have become a standard probability model and have been widely used in many applications [CACM 1995]. However, it is well recognized that due to their attribute-based nature, Bayesian networks are limited in the power of expressiveness and applicability. Using first-order logic to extend the expressive power of Bayesian networks has received particular attention, especially since the notion of the *knowledge-based model construction* (KBMC) was introduced [Breese 1992; Wellman et al. 1992]. The basic idea of the KBMC approach is to encode general knowledge in an expressive language such as first-order logic, then dynamically construct a Bayesian network for each specific situation or problem instance. In particular, logic clauses are used to represent general causal knowledge or direct influences. In this case, a knowledge base is a logic program whose clauses are each associated with a probability parameter expressing the degree of uncertainty about the direct influences. Due to its enhancement of a logic program with probability, such a framework is also called *probabilistic logic programming* (PLP) [Haddawy 1994; Ngo and Haddawy 1997; Poole 1993].[1]

The core of the PLP framework is a backward-chaining procedure, which generates a Bayesian network graphic structure from a logic program in a way that when a ground atom A is placed in the network (as a node), each of the atoms in the body of a ground clause whose head is A is added to the network as a parent node of A. This is quite like the process of query evaluation in logic programming. Therefore, existing PLP methods use a slightly adapted *SLD-* or *SLDNF-resolution* [Lloyd 1987] as the backward-chaining procedure.

Recursive loops in a logic program are SLD-derivations of the form

$$A_1 \leftarrow \cdots \leftarrow A_2 \ \cdots \leftarrow A_3 \ \cdots, \tag{1}$$

where, for any $i \geq 1$, A_i is the same as A_{i+1} up to variable renaming, which are generated from a set of recursive clauses like

$$\begin{aligned}
&p_1(X_1) :- \ldots, p_2(X_2), \ldots\\
&p_2(X_2) :- \ldots, p_3(X_3), \ldots\\
&\quad\vdots\\
&p_n(X_n) :- \ldots, p_1(X_1), \ldots
\end{aligned}$$

Such loops present a challenging problem to the PLP framework. On the one hand, they loop forever so that the PLP backward-chaining inferences will never stop. On the other hand, they may generate cyclic influences, which are disallowed in Bayesian networks. Therefore, exploiting methods for reasoning with recursive loops is of particular importance in PLP research.

Two representative approaches have been proposed to avoid recursive loops. The first one is by Ngo and Haddawy [1997] and Kersting and De Raedt [2000], who restricted themselves to considering only acyclic logic programs [Apt and Bezem 1991]. One major issue with this restriction is that not only is the

[1]This term was first introduced by Ng and Subrahmanian [1992] for a purpose different from the KBMC approach. It was used for a probabilistic characterization of logic programming semantics.

expressive power limited, it is also quite difficult to write acyclic logic programs without sacrificing the completeness of problem description, especially for most unskilled users [Bol et al. 1991, Shen et al. 2001, 2003]. The second approach, proposed by Glesner and Koller [1995], 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 clause head is at least one time step later than the time arguments of the predicates in the clause body. In this way, each predicate $p(X)$ is changed to $p(X, T)$ and each clause $p(X) \leftarrow q(X)$ is rewritten into $p(X, T1) \leftarrow T2 = T1 - 1, q(X, T2)$, where T, $T1$, and $T2$ are time parameters. As we will show in Section 6.3, enforcing acyclicity of a logic program by introducing time parameters suffers from important drawbacks and thus is not an effective way to handle recursive loops.

In this article, we propose a novel solution to the problem of recursive loops under the PLP framework. Major characteristics and significance of our method are as follows. First, we do not avoid recursive loops by either restricting to acyclic logic programs or relying on explicit time parameters. Instead, we make use of recursive loops to build a stationary dynamic Bayesian network. Second, we introduce a new PLP formalism, called *a Bayesian knowledge base*. It allows recursive loops and contains logic clauses of the form $A \leftarrow A_1, \ldots, A_l, true, Context, Types$, which naturally formulates the knowledge that the A_is have direct influences on A in the context *Context* under the type constraints *Types*. Third, we introduce the well-founded semantics [Van Gelder et al. 1991] 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 [Chen and Warren 1996] (or SLTNF-resolution [Shen et al. 2004]) to make the backward-chaining inferences. This establishes a clear declarative semantics for a Bayesian knowledge base. Fourth, we observe that under the PLP framework cyclic influences caused by recursive loops define feedbacks, thus implying a time sequence. For instance, the clause $aids(X) \leftarrow aids(Y), contact(X, Y)$ introduces recursive loops

$$aids(X) \leftarrow aids(Y) \;\cdots \leftarrow aids(Y1) \;\cdots .$$

Together with some other clauses in a logic program, these recursive loops may generate cyclic influences of the form

$$aids(p1) \leftarrow \cdots \leftarrow aids(p1) \;\cdots \leftarrow aids(p1) \;\cdots .$$

Such cyclic influences represent feedback connections, that is, that $p1$ *is* infected with aids (in the current time slice t) depends on whether $p1$ *was* infected with aids earlier (in the last time slice $t - 1$). Therefore, recursive loops of form (1) potentially imply a time sequence of the form

$$A \underbrace{\leftarrow \cdots \leftarrow}_{t} A \underbrace{\cdots \leftarrow}_{t-1} A \underbrace{\cdots \leftarrow}_{t-2} A \;\cdots , \tag{2}$$

where A is a ground instance of A_1. 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. This extends existing PLP approaches, such as those in Goldman and Charniak [1993], Haddawy [1994], [Kersting and Raedt 2000; Ngo and Haddawy 1997; Poole 1993], which aim at (non-temporal) relational models.

The article is structured as follows. In Section 2, we review some concepts concerning Bayesian networks and logic programs. In Section 3, we introduce Bayesian knowledge bases. A Bayesian knowledge base consists mainly of a logic program that defines a direct influence relation over a space of random variables. In Section 4, we establish a declarative semantics for a Bayesian knowledge base based on a key notion of influence clauses. Influence clauses contain only ground atoms from the space of random variables and define the same direct influence relation as the original Bayesian knowledge base does. In Section 5, we present algorithms for building a Bayesian network from a Bayesian knowledge base. We describe related work in Section 6 and summarize our work in Section 7.

2. PRELIMINARIES AND NOTATION

We assume the reader is familiar with basic ideas of Bayesian networks [Pearl 1988] and logic programming [Lloyd 1987]. In particular, we assume the reader is familiar with the well-founded semantics [Van Gelder et al. 1991] as well as SLG-resolution [Chen et al. 1995]. A Bayesian network is a directed acyclic graph whose nodes represent random variables and whose edges express *direct influences*. A conditional probability table, $\mathbf{P}(A|B_1, \ldots, B_n)$, is attached to each node/random variable A, which describes the probabilistic relation between A and the set B_is of its parent nodes. When no confusion would occur, we will refer to nodes and random variables exchangeably. A node without parent nodes is referred to as an *input node*.

We now review some basic concepts concerning dynamic Bayesian networks (DBNs). DBNs are introduced to model the evolution of the state of the environment over time [Kanazawa et al. 1995]. Briefly, a DBN is a Bayesian network whose random variables are subscripted with time steps (basic units of time) or time slices (i.e., intervals). In this article, we use time slices. 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 intraprobabilistic relations between random variables in each individual time slice t ($t > 0$) and the interprobabilistic relations between the random variables of each two consecutive time slices $t-1$ and t.[2] If both the intra- and interprobabilistic relations (conditional probability tables) are the same for all time slices (in this case, the DBN is a repetition of a Bayesian network over time; see Figure 1 where all conditional probability tables are omitted), the DBN is called a *stationary* DBN [Russell and Norvig 1995]. As far as we know, most existing DBN systems reported in the literature are stationary DBNs.

[2]We consider first-order Morkov models by assuming that each state only depends on the immediately preceding state.

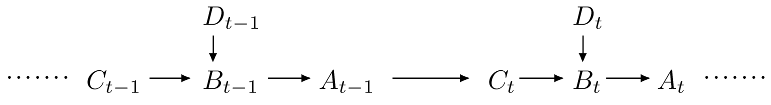

Fig. 1. A stationary DBN structure.

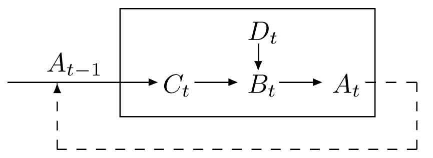

Fig. 2. A two-slice DBN structure (a feedback system).

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 (i.e., they produce cycles of direct influences). Such random variables are called *state random variables*. Note that D is not a state random variable. Each state random variable A in a stationary DBN is assumed to have an initial/prior probability distribution $\mathbf{P}(A_0)$ at time zero. A stationary DBN will make use of such prior probability distributions (together with the probability distributions of nonstate random variables, which are the same for all time slices) to compute a posterior probability distribution $\mathbf{P}(A_1|\cdots)$ for A in time slice $t = 1$.

A stationary DBN is often compactly represented as a two-slice DBN together with a prior probability distribution $\mathbf{P}(A_0)$ for each state random variable A at time zero.

Definition 2.1. A *two-slice* DBN for a stationary DBN consists of two consecutive time slices, $t-1$ and t, which describes (1) the intraprobabilistic relations between the random variables in slice t and (2) the interprobabilistic relations between the random variables in slice $t-1$ and the random variables in slice t.

A two-slice DBN models a feedback system, where a cycle of direct influences establishes a feedback connection. For convenience, we depict feedback connections using dashed edges. Moreover, we refer to nodes coming from slice $t-1$ as *state input nodes* (or *state input random variables*).

Example 2.1. The stationary DBN of Figure 1 can be represented by a two-slice DBN as shown in Figure 2, where A, C, and B form a cycle of direct influences and thus establish a feedback connection. This stationary DBN can also be represented by a two-slice DBN starting from a different state input node such as C_{t-1} or B_{t-1}. These two-slice DBN structures are equivalent in the sense that they model the same cycle of direct influences and can be unrolled into the same stationary DBN (Figure 1).

Observe that in a two-slice DBN, all random variables except state input nodes have the same subscript t. When no confusion would arise, the subscript

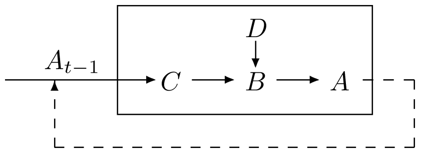

Fig. 3. A simplified two-slice DBN structure.

t is omitted for simplification of the structure. For instance, the two-slice DBN of Figure 2 is simplified to that of Figure 3.

In the rest of this section, we introduce some necessary notation for logic programs. Variables begin with a capital letter, and predicate, function and constant symbols with a lower-case letter. We use $p(.)$ to refer to any predicate/atom whose predicate symbol is p and use $p(\overrightarrow{X})$ to refer to $p(X_1, \ldots, X_n)$ where all X_is are variables. There is one special predicate, *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

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

where A, the B_is, and the C_js are atoms. We use $HU(P)$ and $HB(P)$ to denote the Herbrand universe and Herbrand base of P, respectively, and use $WF(P) = \langle I_t, I_f \rangle$ 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/atom 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 $A \leftarrow B_1, \ldots, B_k, \neg B_{k+1}, \ldots, \neg B_n$ of any clause in P, $map(A) > map(B_i)$ $(1 \leq i \leq n)$ [Apt and Bezem 1991]. P is said to have the *bounded-term-size property* with respect to a set of predicates $\{p_1(.), \ldots, p_t(.)\}$ 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 SLDNF- (or SLG-) derivations for G_0 have an argument whose term size exceeds $f(n)$ (this definition is adapted from Van Gelder [1989]). Obviously, all Datalog programs have the bounded-term-size property.

A *dependency graph* of a logic program P, denoted $DG(P)$, consists of all predicate symbols appearing in P (as its nodes) such that for any p and q, there is an edge $p \rightarrow q$ in the graph if P has a clause with p in the head and q in the body. p is a *recursive predicate symbol* if $DG(P)$ contains a cycle involving p.

3. DEFINITION OF A BAYESIAN KNOWLEDGE BASE

In this section, we introduce a new PLP formalism, called *Bayesian knowledge bases*. Bayesian knowledge bases accommodate recursive loops and define the direct influence relation in terms of the well-founded semantics.

Definition 3.1. A *Bayesian knowledge base* is a triple $\langle PB \cup CB, T_x, CR\rangle$, where

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

$$p(.) \leftarrow \underbrace{p_1(.), \ldots, p_l(.)}_{direct\ influences}, true, \underbrace{B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n}_{context},$$
$$\underbrace{member(X_1, DOM_1), \ldots, member(X_s, DOM_s)}_{type\ constraints}, \qquad (4)$$

where (i) the predicate symbols $p, p_1, \ldots, p_l$ 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 $\mathbf{P}(p(.)|p_1(.), \ldots, p_l(.))$, each being attached to a clause (4) in PB, together with a probability distribution $\mathbf{P}(p(\overrightarrow{X})_0)$ attached to each recursive predicate symbol p in PB;

—CR is a combination rule such as *noisy-or, min* or *max* [Kersting and Raedt 2000; Ngo and Haddawy 1997; Russell and Norvig 1995].

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 $p_1(.), \ldots, p_l(.)$ have direct influences on $p(.)$ in the context that $B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$, $member(X_1, DOM_1), \ldots, member(X_s, DOM_s)$ 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, that is, 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.[3] Clauses in CB do not describe direct influences.

Recursive clauses are allowed in PB and CB. In particular, when some $p_i(.)$ in clause (4) is the same as the head $p(.)$, a cyclic direct influence occurs. Such a cyclic influence models a feedback connection and is interpreted as $p(.)$ at present depending on itself in the past.

In this article, we focus on Datalog programs, although the proposed approach applies to logic programs with the bounded-term-size property (with respect to the set of predicates appearing in the heads of clauses in PB) as well. Datalog programs are widely used in database and knowledge base systems [Ullman 1988] and have a polynomial time data complexity in computing their well-founded models [Van Gelder et al. 1991]. 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, $\mathbf{P}(p(.)|p_1(.), \ldots, p_l(.))$, in T_x specifying the degree of the direct influences. Such a CPT is shared by all instances of clause (4). For each recursive predicate symbol p in PB, there is a unique $\mathbf{P}(p(\overrightarrow{X})_0)$ in T_x specifying a prior probability distribution (at time zero) for any instance of $p(\overrightarrow{X})$.

[3]The predicate *true* can be defined in CB using a unit clause.

A Bayesian knowledge base has the following important property.

THEOREM 3.1. (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.*

PROOF. (1) If the head of a clause in *PB* contains variables, there must be atoms of the form $member(X_i, DOM_i)$ in its body. This means that clauses whose head contains variables are not unit clauses. Therefore, all unit clauses in *PB* are ground.

(2) Let A be an answer of G_0 obtained by applying SLG-resolution to $PB \cup CB \cup \{G_0\}$. Then A must be produced by applying a clause in *PB* of form (4) with a most general unifier (mgu) θ such that $A = p(.)\theta$ and the body $(p_1(.), \ldots, p_l(.), true, B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n, member(X_1, DOM_1), \ldots, member(X_s, DOM_s))\theta$ is evaluated true in the well-founded model $WF(PB \cup CB)$. Note that the type constraints $(member(X_1, DOM_1), \ldots, member(X_s, DOM_s))\theta$ being evaluated true by SLG-resolution guarantees that all variables X_is in the head $p(.)$ are instantiated by θ into constants in their domains DOM_is. This means that A is ground. □

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

Example 3.1. We borrow the well-known AIDS program from Glesner and Koller [1995] (a simplified version) as a running example to illustrate our PLP approach. It is formulated by a Bayesian knowledge base BKB_1 with the following logic program[4]:

$$\begin{array}{ll} PB_1: & 1.\ aids(p1). \\ & 2.\ aids(p3). \\ & 3.\ aids(X) \leftarrow aids(X). \\ & 4.\ aids(X) \leftarrow aids(Y), contact(X, Y). \\ & 5.\ contact(p1, p2). \\ & 6.\ contact(p2, p1). \end{array}$$

Note that both the third and the fourth clauses produce recursive loops. The third clause also has a cyclic direct influence. Conceptually, the two clauses model the fact that the direct influences on $aids(X)$ come from whether X was infected with AIDS earlier (the feedback connection induced from the third clause) or whether X has contact with someone Y who is infected with AIDS (the fourth clause).

[4]This Bayesian knowledge base $BKB_1 = \langle PB_1 \cup CB_1, T_{x_1}, CR_1 \rangle$ may well contain contexts that describe a person's background information. The contexts together with CB_1, T_{x_1}, and CR_1 are omitted here for the sake of simplicity.

4. DECLARATIVE SEMANTICS

In this section, we formally describe the space of random variables and the direct influence relation defined by a Bayesian knowledge base *BKB*. We then define probability distributions induced by *BKB*.

4.1 Space of Random Variables and Influence Clauses

Recall that a node/random variable in a Bayesian network is either an input node, which has no parent node, or a node whose parent nodes are determined by a direct influence relation. A Bayesian knowledge base *BKB* defines a Bayesian network whose random variables are a subset of *HB*(*PB*), by taking atoms of all unit clauses in *PB* as input nodes and deducing the other nodes iteratively based on the direct influence relation defined by *PB*. Formally, we have

Definition 4.1. The *space of random variables* of *BKB*, denoted $\mathcal{S}(BKB)$, is recursively defined as follows:

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

Definition 4.2. For any random variables A, B in $\mathcal{S}(BKB)$, we say A is *influenced by* B if B has a direct influence on A, or for some C in $\mathcal{S}(BKB)$ A is influenced by C and C is influenced by B. A *cyclic influence* occurs if A is influenced by itself.

Example 4.1 (*Example* 3.1 *Continued*). The clauses 1, 2, 5, and 6 are unit clauses; thus their atoms are random variables. $aids(p2)$ is then derived by applying the fourth clause. Consequently,

$$\mathcal{S}(BKB_1) = \{aids(p1), aids(p2), aids(p3), contact(p1, p2), contact(p2, p1)\}.$$

$aids(p1)$ and $aids(p2)$ have a direct influence on each other. There are three cyclic influences: $aids(pi)$ is influenced by itself for each $i = 1, 2, 3$.

Let $WF(PB \cup CB) = \langle I_t, I_f \rangle$ 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 4.1. $\mathcal{S}(BKB) = I_{PB}$.

PROOF. First note that atoms of all unit clauses in *PB* are both in $\mathcal{S}(BKB)$ and in I_{PB}. We prove this theorem by induction on the maximum depth $d \geq 0$ of backward derivations of a random variable A.

($\Longrightarrow$) Let $A \in \mathcal{S}(BKB)$. When $d = 0$, A is a unit clause in PB, so $A \in I_{PB}$. For the induction step, assume $B \in I_{PB}$ for any $B \in \mathcal{S}(BKB)$ whose maximum depth d of backward derivations is below k. Let $d = k$ for A. There must be a ground instance $A \leftarrow A_1, \ldots, A_l, true, B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$ of a clause in PB such that the A_is are already in $\mathcal{S}(BKB)$ and $B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$ is true in the well-founded model $WF(PB \cup CB)$. Since the head A is derived from the A_is in the body, the maximum depth for each A_i must be below the depth k for the head A. By the induction hypothesis, the A_is are in I_{PB}. By definition of the well-founded model, A is true in $WF(PB \cup CB)$ and thus $A \in I_{PB}$.

($\Longleftarrow$) Let $A \in I_{PB}$. When $d = 0$, A is a unit clause in PB, so $A \in \mathcal{S}(BKB)$. For the induction step, assume $B \in \mathcal{S}(BKB)$ for any $B \in I_{PB}$ whose maximum depth d of backward derivations is below k. Let $d = k$ for A. There must be a ground instance $A \leftarrow A_1, \ldots, A_l, true, \ldots$ of a clause in PB such that the body is true in $WF(PB \cup CB)$. Note that the predicate symbol of each A_i occurs in the head of a clause in PB. Since the head A is derived from the literals in the body, the maximum depth of backward derivations for each A_i in the body must be below the depth k for the head A. By the induction hypothesis, the A_is are in $\mathcal{S}(BKB)$. By Definition 4.1, $A \in \mathcal{S}(BKB)$. $\square$

Theorem 4.1 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 [Sagonas et al. 1998], in this article we apply it for the PLP backward-chaining inferences. SLG-resolution is a tabling mechanism for top-down computation of the well-founded model. For any atom A, during the process of evaluating a goal $\leftarrow A$, SLG-resolution stores all answers of A in a space called *table*, denoted $\mathcal{T}_A$.

Let $\{p_1, \ldots, p_t\}$ be the set of predicate symbols occurring in the heads of clauses in PB, and let $GS_0 = \{\leftarrow p_1(\overrightarrow{X_1}), \ldots, \leftarrow p_t(\overrightarrow{X_t})\}$.

Algorithm 1. Computing random variables

```
(1) $\mathcal{S}'(BKB) = \emptyset$.
(2) For each $\leftarrow p_i(\overrightarrow{X}_i)$ in $GS_0$
    (a) Compute the goal $\leftarrow p_i(\overrightarrow{X}_i)$ by applying SLG-resolution to $PB \cup CB \cup \{\leftarrow p_i(\overrightarrow{X}_i)\}$.
    (b) $\mathcal{S}'(BKB) = \mathcal{S}'(BKB) \cup \mathcal{T}_{p_i(\overrightarrow{X_i})}$.
(3) Return $\mathcal{S}'(BKB)$.
```

THEOREM 4.2. *Algorithm 1 terminates, yielding a finite set $\mathcal{S}'(BKB) = \mathcal{S}(BKB)$.*

PROOF. Since GS_0 is finite, Algorithm 1 terminates if SLG-resolution terminates for each $\leftarrow p_i(\overrightarrow{X_i})$ in GS_0. Let $WF(PB \cup CB) = \langle I_t, I_f \rangle$ be the well-founded model of $PB \cup CB$. Since SLG-resolution is sound and complete for the well-founded semantics and terminates for any logic programs with the bounded-term-size property [Chen and Warren 1996], Algorithm 1 will terminate with a finite output $\mathcal{S}'(BKB)$ that consists of all answers of $p_i(\overrightarrow{X_i})$ $(1 \leq i \leq t)$. By Theorem 3.1, all answers in $\mathcal{S}'(BKB)$ are ground. This means $\mathcal{S}'(BKB) = I_{PB}$. Hence, by Theorem 4.1 $\mathcal{S}'(BKB) = \mathcal{S}(BKB)$. □

We introduce the following principal concept.

Definition 4.3. Let $A \leftarrow A_1, \ldots, A_l, true, B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$ be a ground instance of the kth clause in PB such that its body is true in the well-founded model $WF(PB \cup CB)$. We call

$$k.\ A \leftarrow A_1, \ldots, A_l \tag{5}$$

an *influence clause*.[5] All influence clauses derived from all clauses in PB constitute the *set of influence clauses* of BKB, denoted $\mathcal{I}_{clause}(BKB)$.

The following result is immediate from Definition 4.1 and Theorem 4.1.

THEOREM 4.3. *For any influence clause* (5)*, A and all A_is are random variables in $\mathcal{S}(BKB)$.*

Influence clauses have the following principal property.

THEOREM 4.4. *For any A_i and A in HB(PB), A_i has a direct influence on A, which is derived from the kth clause in PB, if and only if there is an influence clause in $\mathcal{I}_{clause}(BKB)$ of the form $k.\ A \leftarrow A_1, \ldots, A_i, \ldots, A_l$.*

PROOF. ($\Longrightarrow$) Assume A_i has a direct influence on A, which is derived from the kth clause in PB. By Definition 4.1, the kth clause has a ground instance of the form $A \leftarrow A_1, \ldots, A_i, \ldots, A_l, true, B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$ such that $B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$ is true in $WF(PB \cup CB)$ and $\{A_1, \ldots, A_i, \ldots, A_l\} \subseteq \mathcal{S}(BKB)$. By Theorem 4.1, $A_1, \ldots, A_i, \ldots, A_l$ is true in $WF(PB \cup CB)$. Thus, $k.\ A \leftarrow A_1, \ldots, A_i, \ldots, A_l$ is an influence clause in $\mathcal{I}_{clause}(BKB)$.

($\Longleftarrow$) Assume that $\mathcal{I}_{clause}(BKB)$ contains an influence clause $k.\ A \leftarrow A_1, \ldots, A_i, \ldots, A_l$. Then the kth clause in PB has a ground instance of the form $A \leftarrow A_1, \ldots, A_i, \ldots, A_l, true, B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n$ such that its body is true in $WF(PB \cup CB)$ and (by Theorem 4.3) $\{A_1, \ldots, A_i, \ldots, A_l\} \subseteq \mathcal{S}(BKB)$. By Definition 4.1, $A \in \mathcal{S}(BKB)$ and A_i has a direct influence on A. □

The following result is immediate from Theorem 4.4.

COROLLARY 4.5. *For any atom A, A is in $\mathcal{S}(BKB)$ if and only if there is an influence clause in $\mathcal{I}_{clause}(BKB)$ whose head is A.*

Theorem 4.4 shows the significance of influence clauses: they define the same direct influence relation over the same space of random variables as the original Bayesian knowledge base BKB does. Therefore, a Bayesian network can be built directly from $\mathcal{I}_{clause}(BKB)$ provided the influence clauses are available.

[5]The prefix "$k.$" will be omitted sometimes for the sake of simplicity.

Observe that to compute the space of random variables (see Algorithm 1), SLG-resolution will construct a proof tree rooted at the goal $\leftarrow p_i(\overrightarrow{X_i})$ for each $1 \leq i \leq t$ [Chen et al. 1995]. For each answer A of $p_i(\overrightarrow{X_i})$ in $\mathcal{S}(BKB)$ there must be a success branch (i.e., a branch starting at the root node and ending at a node marked *success*) in the tree that generates the answer. Let $p_i(.) \leftarrow A_1, \ldots, A_l, true, \ldots$ be the kth clause in *PB* that is applied to expand the root goal $\leftarrow p_i(\overrightarrow{X_i})$ in the success branch and let θ be the composition of all mgus along the branch. Then $A = p_i(.)\theta$ and the clause body $A_1, \ldots, A_l, true, \ldots$ is evaluated true with the mgu θ in $WF(PB \cup CB)$ by SLG-resolution. This means that, for each $1 \leq j \leq l$, $A_j\theta$ is an answer of A_j that is derived by applying SLG-resolution to $PB \cup CB \cup \{\leftarrow A'_j\}$ where A'_j is A_j or some instance of A_j. By Theorem 3.1, all $A_j\theta$s are ground atoms. Therefore, $k.\ p_i(.)\theta \leftarrow A_1\theta, \ldots, A_l\theta$ is an influence clause. Hence we have the following result.

THEOREM 4.6. *Let B_r be a success branch in a proof tree of SLG-resolution, $p_i(.) \leftarrow A_1, \ldots, A_l,\ true, \ldots$ be the kth clause in PB that expands the root goal in B_r, and θ be the composition of all mgus along B_r. B_r produces an influence clause $k.\ p_i(.)\theta \leftarrow A_1\theta, \ldots, A_l\theta$.*

Every success branch in a proof tree for a goal in GS_0 produces an influence clause. The set of influence clauses can then be obtained by collecting all influence clauses from all such proof trees in SLG-resolution.

Algorithm 2. Computing influence clauses.

```
(1) $\mathcal{I}'_{clause}(BKB) = \emptyset$.
(2) For each goal $\leftarrow p_i(\overrightarrow{X_i})$ in $GS_0$, compute all answers of $p_i(\overrightarrow{X_i})$ by applying SLG-
    resolution to $PB \cup CB \cup \{\leftarrow p_i(\overrightarrow{X_i})\}$ while for each success branch starting at the
    root goal $\leftarrow p_i(\overrightarrow{X_i})$, collecting an influence clause from the branch and adding it into
    $\mathcal{I}'_{clause}(BKB)$.
(3) Return $\mathcal{I}'_{clause}(BKB)$.
```

THEOREM 4.7. *Algorithm 2 terminates, yielding a finite set $\mathcal{I}'_{clause}(BKB) = \mathcal{I}_{clause}(BKB)$.*

PROOF. That Algorithm 2 terminates is immediate from Theorem 4.2, as except for collecting influence clauses, Algorithm 2 makes the same derivations as Algorithm 1. The termination of Algorithm 2 then implies $\mathcal{I}'_{clause}(BKB)$ is finite.

By Theorem 4.6, any clause in $\mathcal{I}'_{clause}(BKB)$ is an influence clause in $\mathcal{I}_{clause}(BKB)$. We now prove the converse. Let $k.\ A \leftarrow A_1, \ldots, A_l$ be an influence clause in $\mathcal{I}_{clause}(BKB)$. Then the kth clause in *PB* $A' \leftarrow A'_1, \ldots, A'_l, true, \ldots.$ has a ground instance of the form $A \leftarrow A_1, \ldots, A_l, true, \ldots$ whose body is true in $WF(PB \cup CB)$. By the completeness of SLG-resolution, there must be a success branch in the proof tree rooted at a goal $\leftarrow p_i(\overrightarrow{X_i})$ in GS_0 where (1) the root goal is expanded by the kth clause, (2) the composition of all mgus along the branch is θ, and (3) $A \leftarrow A_1, \ldots, A_l, true, \ldots$ is an instance of $(A' \leftarrow A'_1, \ldots, A'_l, true, \ldots)\theta$. By Theorem 4.6, $k.\ A'\theta \leftarrow A'_1\theta, \ldots, A'_l\theta$ is an

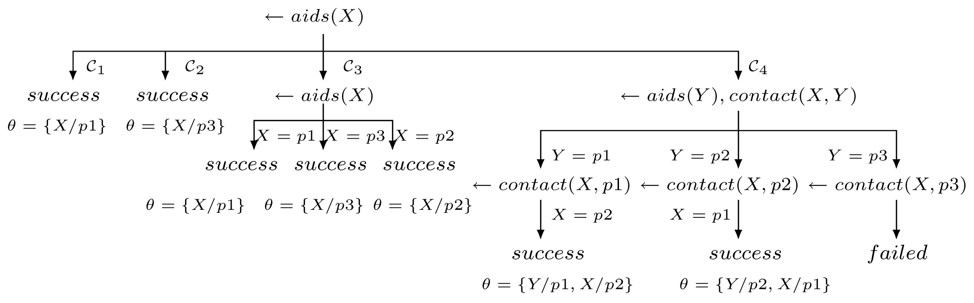

Fig. 4. The proof tree for ← *aids*(X).

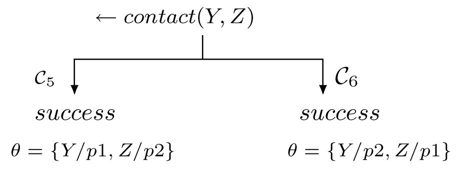

Fig. 5. The proof tree for ← *contact*(Y, Z).

influence clause. Since any influence clause is ground, $k.\ A'\theta \leftarrow A'_1\theta, \ldots, A'_l\theta$ is the same as $k.\ A \leftarrow A_1, \ldots, A_l$. This influence clause from the success branch will be collected into $\mathcal{I}'_{clause}(BKB)$ by Algorithm 2. Thus, any clause in $\mathcal{I}_{clause}(BKB)$ is in $\mathcal{I}'_{clause}(BKB)$. □

Example 4.2 (*Example* 4.1 *Continued*). There are only two predicate symbols, *aids* and *contact*, in the heads of clauses in PB_1. Let $GS_0 = \{\leftarrow aids(X), \leftarrow contact(Y, Z)\}$. Algorithm 2 will generate two proof trees rooted at $\leftarrow aids(X)$ and $\leftarrow contact(Y, Z)$, respectively, as shown in Figures 4 and 5. In the proof trees, a label $\mathcal{C}_i$ on an edge indicates that the ith clause in *PB* is applied, and the other labels like $X = p1$ on an edge show that an answer from a table is applied. Each success branch yields an influence clause. For instance, expanding the root goal $\leftarrow aids(X)$ by the third clause produces a child node $\leftarrow aids(X)$ (Figure 4). Then applying to this node the answers of $aids(X)$ in the table $\mathcal{T}_{aids(X)}$ leads to three success branches. Then, by applying the mgu θ on each success branch to instantiate the third clause, we obtain three influence clauses of the form

$$3.\ aids(pi) \leftarrow aids(pi) \quad (i = 1, 2, 3).$$

As a result, we obtain the following set of influence clauses:

$$\begin{aligned}
\mathcal{I}_{clause}(BKB_1):\ & 1.\ aids(p1).\\
& 2.\ aids(p3).\\
& 3.\ aids(p1) \leftarrow aids(p1).\\
& 3.\ aids(p2) \leftarrow aids(p2).\\
& 3.\ aids(p3) \leftarrow aids(p3).\\
& 4.\ aids(p2) \leftarrow aids(p1), contact(p2, p1).\\
& 4.\ aids(p1) \leftarrow aids(p2), contact(p1, p2).
\end{aligned}$$

5. *contact*$(p1, p2)$.
6. *contact*$(p2, p1)$.

For the computational complexity, we observe that the cost of Algorithm 2 is dominated by applying SLG-resolution to evaluate the goals in GS_0. It has been shown that for a Datalog program P, the time data complexity, as defined by Vardi [1982], of computing the well-founded model $WF(P)$ is polynomial in the number of ground unit clauses in P [Van Gelder et al. 1991]. This applies to SLG-resolution as well [Chen and Warren 1996].

$P = PB \cup CB$ is a Datalog program except for the $member(X_i, DOM_i)$ predicates used in PB (see Definition 3.1). So if the cost of handling all $member(X_i, DOM_i)$ predicates is polynomial, it is polynomial applying SLG-resolution to compute the well-founded model $WF(PB \cup CB)$.

Since each domain DOM_i is a finite list of constants, checking if X_i is in DOM_i takes time linear in the size of DOM_i. Let K_1 be the maximum number of $member(X_i, DOM_i)$ predicates used in a clause in P and K_2 be the maximum size of a domain DOM_i. Then the time of handling all $member(X_i, DOM_i)$ predicates in a clause is bounded by $K_1 * K_2$. Note that each clause in P is applied at most N times in SLG-resolution, where N is the number of atoms of predicates (except $member(X_i, DOM_i)$) in P that are not variants of each other. N is a polynomial in the number of ground unit clauses in P [Chen and Warren 1996]. So the time of handling all $member(X_i, DOM_i)$s in all clauses of P is bounded by $|P| * N * K_1 * K_2$, where $|P|$ is the number of clauses in P. Clearly, this is a polynomial. Therefore, we have the following result.

THEOREM 4.8. *The time data complexity of Algorithm* 2 *is polynomial in the number of ground unit clauses in* $PB \cup CB$.

4.2 Probability Distributions Induced by *BKB*

For any random variable A, 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 5.2). Furthermore, we make the following *independence assumption*.

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

We define probability distributions induced by *BKB* in terms of whether there are cyclic influences.

Definition 4.4. When no cyclic influence occurs, the probability distribution induced by *BKB* is $\mathbf{P}(\mathcal{S}(BKB))$.

THEOREM 4.9. $\mathbf{P}(\mathcal{S}(BKB)) = \prod_{A_i \in \mathcal{S}(BKB)} \mathbf{P}(A_i|pa(A_i))$ *under the independence assumption.*

PROOF. When no cyclic influence occurs, the random variables in $\mathcal{S}(BKB)$ can be arranged in a partial order such that if A_i is influenced by A_j then $j > i$.

By the independence assumption, we have $\mathbf{P}(\mathcal{S}(BKB)) = \mathbf{P}(\bigwedge_{A_i \in \mathcal{S}(BKB)} A_i) = \mathbf{P}(A_1 | \bigwedge_{i=2} A_i) * \mathbf{P}(\bigwedge_{i=2} A_i) = \mathbf{P}(A_1 | pa(A_1)) * \mathbf{P}(A_2 | \bigwedge_{i=3} A_i) * \mathbf{P}(\bigwedge_{i=3} A_i) = \ldots = \prod_{A_i \in \mathcal{S}(BKB)} \mathbf{P}(A_i | pa(A_i))$. □

When there are cyclic influences, we cannot have a partial order on $\mathcal{S}(BKB)$. Consider a cyclic influence: A_1 is influenced by itself. By Definition 4.2, this cyclic influence must come from a chain (cycle) of direct influences

$$A_1 \leftarrow A_2 \leftarrow A_3 \leftarrow \cdots \leftarrow A_n \leftarrow A_1. \tag{6}$$

By Theorem 4.4, this chain of direct influences must be generated from a set of influence clauses in $\mathcal{I}_{clause}(BKB)$ of the form

$$\begin{array}{lcl} A_1 & \leftarrow & \ldots, A_2, \ldots, \\ A_2 & \leftarrow & \ldots, A_3, \ldots, \\ & \ldots\ldots & \\ A_n & \leftarrow & \ldots, A_1, \ldots. \end{array} \tag{7}$$

Note that the cycle of direct influences (6) defines a feedback connection and thus the set of influence clauses (7) constitutes a feedback system. Since a feedback system can be modeled by a two-slice DBN (see Section 2), influence clauses (7) represent the same knowledge as the following ones do:

$$\begin{array}{lcl} A_1 & \leftarrow & \ldots, A_2, \ldots, \\ A_2 & \leftarrow & \ldots, A_3, \ldots, \\ & \ldots\ldots & \\ A_n & \leftarrow & \ldots, A_{1_{t-1}}, \ldots. \end{array} \tag{8}$$

Here the A_is are each a state random variable and $A_{1_{t-1}}$ is a state input random variable. As a result, A_1 being influenced by itself becomes A_1 being influenced by $A_{1_{t-1}}$. By applying this transformation (from influence clauses (7) to (8)) for every cycle of direct influences, we can get rid of all cyclic influences and obtain a *generalized set* $\mathcal{I}_{clause}(BKB)_g$ of influence clauses from $\mathcal{I}_{clause}(BKB)$. (The algorithm for detecting influence cycles will be discussed in detail in the next section.)

Example 4.3 (*Example* 4.2 *Continued*). $\mathcal{I}_{clause}(BKB_1)$ can be transformed to the following generalized set of influence clauses by introducing three state input random variables $aids(p1)_{t-1}$, $aids(p2)_{t-1}$ and $aids(p3)_{t-1}$.

$\mathcal{I}_{clause}(BKB_1)_g$:
1. $aids(p1)$.
2. $aids(p3)$.
3. $aids(p1) \leftarrow aids(p1)_{t-1}$.
3. $aids(p2) \leftarrow aids(p2)_{t-1}$.
3. $aids(p3) \leftarrow aids(p3)_{t-1}$.
4. $aids(p2) \leftarrow aids(p1)_{t-1}, contact(p2, p1)$. ♣
4. $aids(p1) \leftarrow aids(p2), contact(p1, p2)$.
5. $contact(p1, p2)$.
6. $contact(p2, p1)$.

For instance, the clause marked ♣ results from the following cycle of direct influences:

$$aids(p1) \leftarrow aids(p2) \leftarrow aids(p1),$$

which is generated from the following two influence clauses in $\mathcal{I}_{clause}(BKB_1)$:

4. $aids(p2) \leftarrow aids(p1), contact(p2, p1).$
4. $aids(p1) \leftarrow aids(p2), contact(p1, p2).$

When there is no cyclic influence, *BKB* is a nontemporal model, represented by $\mathcal{I}_{clause}(BKB)$. When cyclic influences occur, however, *BKB* becomes a temporal model, represented by $\mathcal{I}_{clause}(BKB)_g$. Let $\mathcal{S}(BKB)_g$ be $\mathcal{S}(BKB)$ plus all state input random variables introduced in $\mathcal{I}_{clause}(BKB)_g$.

Definition 4.5. When there are cyclic influences, the probability distribution induced by *BKB* is $\mathbf{P}(\mathcal{S}(BKB)_g)$.

By extending the independence assumption from $\mathcal{S}(BKB)$ to $\mathcal{S}(BKB)_g$, we obtain the following result.

THEOREM 4.10. $\mathbf{P}(\mathcal{S}(BKB)_g) = \prod_{A_i \in \mathcal{S}(BKB)_g} \mathbf{P}(A_i|pa(A_i))$ *under the independence assumption.*

PROOF. First recall that in a two-slice DBN, only the subscript $t-1$ for each state input random variable is explicitly written; the subscript t for the remaining random variables is omitted for the sake of simplicity (see Section 2). Therefore, in the proof of this theorem, each random variable A_j in $\mathcal{I}_{clause}(BKB)_g$ is assumed to have either a subscript t or $t-1$.

Due to the introduction of state input random variables, $\mathcal{I}_{clause}(BKB)_g$ produces no cycles; thus the random variables in $\mathcal{S}(BKB)_g$ can be arranged in a partial order such that if A_i is influenced by A_j then $j > i$. The proof can then proceed in the same way as that of Theorem 4.9 provided that, for any time slice $t > 0$, a probability distribution $\mathbf{P}(A_{j_{t-1}})$ is available for each state input random variable $A_{j_{t-1}}$ in $\mathcal{S}(BKB)_g$. The latter can be proved by a simple induction on t. For the induction base, when $t = 1$, $\mathbf{P}(A_{j_0})$ is available as an initial/prior probability distribution for each state random variable A_j at time zero in a DBN. As the induction hypothesis, assume that $\mathbf{P}(A_{j_{t-1}})$ is available for any $0 < t \leq k$. We now prove that $\mathbf{P}(A_{j_k})$ is available. When $t = k$, each state input random variable in $\mathcal{I}_{clause}(BKB)_g$ is subscripted with $k-1$ and all the remaining random variables subscripted with k. Then, by the independence assumption we have $\mathbf{P}(\mathcal{S}(BKB)_g) = \mathbf{P}(\bigwedge_{A_i \in \mathcal{S}(BKB)_g} A_i) = \mathbf{P}(A_1 | \bigwedge_{i=2} A_i) * \mathbf{P}(\bigwedge_{i=2} A_i)$ $= \mathbf{P}(A_1|pa(A_1)) * \mathbf{P}(A_2 | \bigwedge_{i=3} A_i) * \mathbf{P}(\bigwedge_{i=3} A_i) = \ldots = \prod_{A_i \in \mathcal{S}(BKB)_g} \mathbf{P}(A_i|pa(A_i))$, where by the induction hypothesis each state input random variable $A_{j_{k-1}}$ has a probability distribution $\mathbf{P}(A_{j_{k-1}})$ availabe. As A_{j_k} is in $\mathcal{S}(BKB)_g$, $\mathbf{P}(A_{j_k})$ is derivable from $\mathbf{P}(\mathcal{S}(BKB)_g)$. This concludes the proof. □

5. BUILDING A BAYESIAN NETWORK FROM A BAYESIAN KNOWLEDGE BASE

5.1 Building a Two-Slice DBN Structure

From a Bayesian knowledge base BKB, we can derive a set of influence clauses $\mathcal{I}_{clause}(BKB)$, which defines the same direct influence relation over the same space $\mathcal{S}(BKB)$ of random variables as $PB \cup CB$ does (see Theorem 4.4). Therefore, given a probabilistic query together with some evidences, we can depict a network structure from $\mathcal{I}_{clause}(BKB)$, 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 come from $\mathcal{S}(BKB)$ (i.e., each random variable in Q and E is a head of some influence clause in $\mathcal{I}_{clause}(BKB)$). Let

$$TOP = \{A | A \text{ is a random variable in } Q \text{ or } E\}.$$

An *influence network* of Q and E,[6] denoted $\mathcal{I}_{net}(BKB)_{Q,E}$, is constructed from $\mathcal{I}_{clause}(BKB)$ using the following algorithm.

Algorithm 3. Building an influence network

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

Example 5.1 (*Example 4.2 Continued*). To build an influence network from $\mathcal{I}_{clause}(BKB_1)$ that covers $aids(p1)$, $aids(p2)$, and $aids(p3)$, we apply Algorithm 3 with $TOP = \{aids(p1), aids(p2), aids(p3)\}$. The resulting influence network $\mathcal{I}_{net}(BKB_1)_{Q,E}$ is depicted in Figure 6. As an illustration, consider the case that the first random variable $aids(p1)$ is removed from TOP. There are three influence clauses for $aids(p1)$ in $\mathcal{I}_{clause}(BKB_1)$:

1. $aids(p1)$.
3. $aids(p1) \leftarrow aids(p1)$.
4. $aids(p1) \leftarrow aids(p2), contact(p1, p2)$.

[6]Note the differences between influence networks and *influence diagrams*. Influence diagrams (also known as *decision networks*) are a formalism introduced in decision theory that extends Bayesian networks by incorporating actions and utilities [Russell and Norvig 1995].

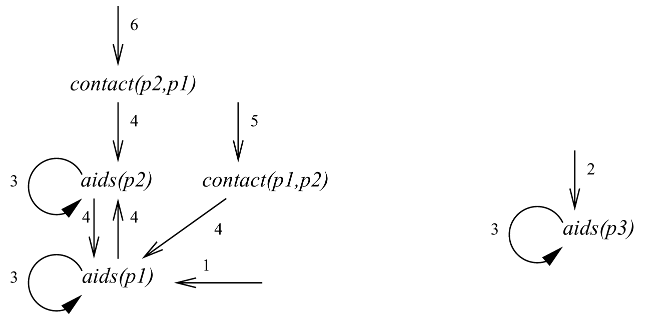

Fig. 6. An influence network built from the AIDS program BKB_1.

The first clause has no body, so an edge $aids(p1) \overset{1}{\leftarrow}$ is added to the network. Applying the second clause gives an edge $aids(p1) \overset{3}{\leftarrow} aids(p1)$, which forms a cycle. The third clause has two atoms in its body, so two edges, $aids(p1) \overset{4}{\leftarrow} aids(p2)$ and $aids(p1) \overset{4}{\leftarrow} contact(p1, p2)$, are added respectively.

An influence network is a graphical representation for influence clauses. This claim is supported by the following properties of influence networks.

THEOREM 5.1. *For any A_i, A_j in $\mathcal{I}_{net}(BKB)_{Q,E}$, A_j is a parent node of A_i, connected via an edge $A_i \overset{k}{\leftarrow} A_j$, if and only if there is an influence clause of the form $k.\ \ A_i \leftarrow A_1, \ldots, A_j, \ldots, A_l$ in $\mathcal{I}_{clause}(BKB)$.*

PROOF. First note that termination of Algorithm 3 is guaranteed by the fact that any random variable in $\mathcal{S}(BKB)$ will be added to *TOP* no more than one time (line 2a). Let A_i, A_j be nodes in $\mathcal{I}_{net}(BKB)_{Q,E}$. If A_j is a parent node of A_i, connected via an edge $A_i \overset{k}{\leftarrow} A_j$, this edge must be added at line 2b, due to applying an influence clause in $\mathcal{I}_{clause}(BKB)$ of the form $k.\ \ A_i \leftarrow A_1, \ldots, A_j, \ldots, A_l$ (line 2). Conversely, if $\mathcal{I}_{clause}(BKB)$ contains such an influence clause, it must be applied at line 2, with edges of the form $A_i \overset{k}{\leftarrow} A_j$ added to the network at line 2b. □

THEOREM 5.2. *For any A_i, A_j in $\mathcal{I}_{net}(BKB)_{Q,E}$, A_i is a descendant node of A_j if and only if A_i is influenced by A_j.*

PROOF. Assume A_i is a descendant node of A_j, with a path

$$A_i \overset{k}{\leftarrow} B_1 \overset{k_1}{\leftarrow} \ldots B_m \overset{k_m}{\leftarrow} A_j. \tag{9}$$

By Theorem 5.1, $\mathcal{I}_{clause}(BKB)$ must contain the following influence clauses:

$$\begin{array}{rl} k. & A_i \leftarrow \ldots, B_1, \ldots, \\ k_1. & B_1 \leftarrow \ldots, B_2, \ldots, \\ & \ldots\ldots \\ k_m. & B_m \leftarrow \ldots, A_j, \ldots. \end{array} \tag{10}$$

By Theorem 4.4 and Definition 4.2, A_i is influenced by A_j. Conversely, if A_i is influenced by A_j, there must be a chain of influence clauses of the form as above. Since A_i, A_j are in $\mathcal{I}_{net}(BKB)_{Q,E}$, by Theorem 5.1 there must be a path of form (9) in the network. □

THEOREM 5.3. *Let V be the set of nodes in $\mathcal{I}_{net}(BKB)_{Q,E}$ and let $W = \{A_j \in \mathcal{S}(BKB) \mid$ for some $A_i \in TOP$, A_i is influenced by $A_j\}$. $V = TOP \cup W$.*[7]

PROOF. That $\mathcal{I}_{net}(BKB)_{Q,E}$ covers all random variables in TOP follows from line 1 of Algorithm 3. We first prove that if $A_j \in W$ then $A_j \in V$. Assume $A_j \in W$. There must be a chain of influence clauses of form (10) with $A_i \in TOP$. In this case, $B_1, B_2, \ldots, B_m, A_j$ will be recursively added to the network (line 2). Thus $A_j \in V$. We then prove that if $A_j \in V$ and $A_j \notin TOP$ then $A_j \in W$. Assume $A_j \in V$ and $A_j \notin TOP$. A_j must not be added to V at line 1. Instead, it is added to V at line 2a. This means that for some $A_i \in TOP$, A_i is a descendant of A_j. By Theorem 5.2, A_i is influenced by A_j. Hence $A_j \in W$. □

Remark 5.1. Theorem 5.3 shows that to build an influence network $\mathcal{I}_{net}(BKB)_{Q,E}$, only those influence clauses relevant to Q and E will be used. An influence clause with head A is *relevant to* Q and E if A is in TOP or for some $B \in TOP$, B is influenced by A. One way of computing the relevant set of influnce clauses for Q and E is to apply SLG-resolution to evaluate all atoms in TOP, while collecting all influence clauses from all proof trees rooted at a goal $\leftarrow p(.)$ where p is a predicate symbol in the head of a clause in PB. Another way is to directly apply Algorithm 2 by letting GS_0 contain only goals $\leftarrow p(\overrightarrow{X})$ where (i) p is a predicate symbol in the head of a clause in PB, and (ii) p occurs in TOP or for some q in TOP there is a path $q \to \cdots \to p$ in the depndency graph $DG(PB)$.

Theorem 4.9 shows that the probability distribution induced by BKB can be computed over $\mathcal{I}_{clause}(BKB)$. Let $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$ denote an influence network that covers all random variables in $\mathcal{S}(BKB)$. We show that the same distribution can be computed over $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$. For any node A_i in $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$, let $parents(A_i)$ denote the set of parent nodes of A_i in the network. Observe the following facts: first, by Theorem 5.1, $parents(A_i) = pa(A_i)$. Second, by Theorem 5.2, A_i is a descendant node of A_j in $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$ if and only if A_i is influenced by A_j in $\mathcal{I}_{clause}(BKB)$. This means that the independence assumption (Assumption 1) applies to $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$ as well, and that $\mathcal{I}_{clause}(BKB)$ produces a cycle of direct influences if and only if $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$ contains the same (direct) loop. Combining these facts leads to the following immediate result.

THEOREM 5.4. *When no cyclic influence occurs, the probability distribution induced by BKB can be computed over $\mathcal{I}_{net}(BKB)_{\mathcal{S}(BKB)}$. That is, $\mathbf{P}(\mathcal{S}(BKB)) = \prod_{A_i \in \mathcal{S}(BKB)} \mathbf{P}(A_i \mid pa(A_i)) = \prod_{A_i \in \mathcal{S}(BKB)} \mathbf{P}(A_i \mid parents(A_i))$ under the independence assumption.*

Theorem 5.4 implies that an influence network without loops is a Bayesian network structure. Let us consider influence networks with loops. By

[7]This result suggests that an influence network is similar to a *supporting network* introduced in Ngo and Haddawy [1997].

Theorem 5.2, loops in an influence network are generated from recursive influence clauses of form (7) and thus they depict feedback connections of form (6). 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

$$A_1 \overset{k_1}{\longleftarrow} A_2 \overset{k_2}{\longleftarrow} \cdots \overset{k_{n-1}}{\longleftarrow} A_n \overset{k_n}{\longleftarrow} A_{1_{t-1}}$$

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

As illustrated in Section 2, 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 random variables backward to the same set of state random variables (i.e., state input nodes). This process corresponds to generating an influence network $\mathcal{I}_{net}(BKB)_{Q,E}$ from $\mathcal{I}_{clause}(BKB)$ incrementally (adding nodes and edges one at a time) while wrapping up loop nodes with state input nodes. This leads to the following algorithm for building a two-slice DBN structure, $2\mathcal{S}_{net}(BKB)_{Q,E}$, directly from $\mathcal{I}_{clause}(BKB)$ (or from a subset of $\mathcal{I}_{clause}(BKB)$ that are relevant to Q and E; see Remark 5.1), where Q, E, and *TOP* are the same as defined in Algorithm 3.

Algorithm 4. Building a two-slice DBN structure

(1) Initially, $2\mathcal{S}_{net}(BKB)_{Q,E}$ has all random variables in *TOP* as nodes.

(2) Remove the first random variable A from *TOP*. For each influence clause in $\mathcal{I}_{clause}(BKB)$ of the form $k.\ \ A \leftarrow A_1, \ldots, A_l$, if $l = 0$ then add to $2\mathcal{S}_{net}(BKB)_{Q,E}$ an edge $A \overset{k}{\leftarrow}$; else for each A_i in the body
   - (a) If A_i is not in $2\mathcal{S}_{net}(BKB)_{Q,E}$ then add A_i to $2\mathcal{S}_{net}(BKB)_{Q,E}$ as a new node and add it to the end of *TOP*.
   - (b) If adding $A \overset{k}{\leftarrow} A_i$ to $2\mathcal{S}_{net}(BKB)_{Q,E}$ would produce a loop, then add to $2\mathcal{S}_{net}(BKB)_{Q,E}$ 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 $2\mathcal{S}_{net}(BKB)_{Q,E}$.

(3) Repeat step 2 until *TOP* becomes empty.

(4) Return $2\mathcal{S}_{net}(BKB)_{Q,E}$.

Example 5.2 (*Example* 5.1 *Continued*). To build a two-slice DBN structure from BKB_1 that covers $aids(p1)$, $aids(p2)$ and $aids(p3)$, we apply Algorithm 4 to $\mathcal{I}_{clause}(\ BKB_1)$ with $TOP = \{aids(p1), aids(p2), aids(p3)\}$. $2\mathcal{S}_{net}(BKB_1)_{Q,E}$ is shown in Figure 7. Note that loops are cut by introducing three state input nodes $aids(p1)_{t-1}$, $aids(p2)_{t-1}$, and $aids(p3)_{t-1}$. The two-slice DBN structure concisely depicts a feedback system where the feedback connections are shown in Figure 8.

Algorithm 4 is Algorithm 3 enhanced with a mechanism for cutting loops (item 2b); that is, when adding the current edge $A \overset{k}{\leftarrow} A_i$ to the network

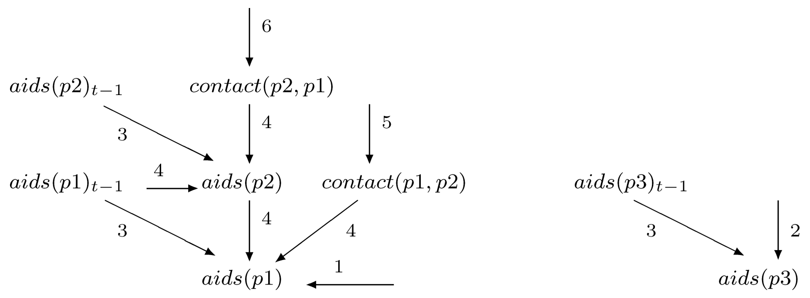

Fig. 7. A two-slice DBN structure built from the AIDS program BKB_1.

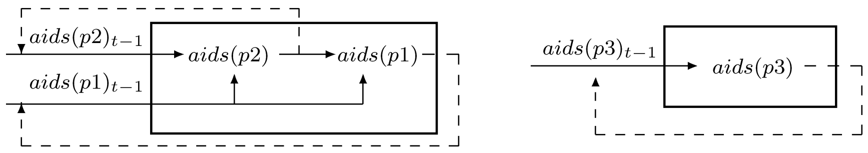

Fig. 8. The feedback connections created by the AIDS program BKB_1.

forms a loop, we replace the edge with $A \overset{k}{\leftarrow} A_{i_{t-1}}$, where $A_{i_{t-1}}$ is a state input node. This is a process of transforming influence clauses (7) to (8). Therefore, $2\mathcal{S}_{net}(BKB)_{Q,E}$ can be viewed as an influence network built from a generalized set $\mathcal{I}_{clause}(BKB)_g$ of influence clauses. Let $\mathcal{S}(BKB)_g$ be the set of random variables in $\mathcal{I}_{clause}(BKB)_g$, as defined in Theorem 4.10. Let $2\mathcal{S}_{net}(BKB)_{\mathcal{S}(BKB)}$ denote a two-slice DBN structure (produced by applying Algorithm 4) that covers all random variables in $\mathcal{S}(BKB)_g$. We then have the following immediate result from Theorem 5.4.

THEOREM 5.5. *When $\mathcal{I}_{clause}(BKB)$ produces cyclic influences, the probability distribution induced by BKB can be computed over $2\mathcal{S}_{net}(BKB)_{\mathcal{S}(BKB)}$. That is, $\mathbf{P}(\mathcal{S}(BKB)_g) = \prod_{A_i \in \mathcal{S}(BKB)_g} \mathbf{P}(A_i \mid pa(A_i)) = \prod_{A_i \in \mathcal{S}(BKB)_g} \mathbf{P}(A_i \mid parents\,(A_i))$ under the independence assumption.*

Remark 5.2. Note that Algorithm 4 produces a DBN structure without using any explicit 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 $\mathcal{I}_{clause}(BKB)$.

Also note that, when there is no cyclic influence, Algorithm 4 becomes Algorithm 3 and thus it builds a regular Bayesian network structure.

5.2 Building CPTs

After a Bayesian network structure $2\mathcal{S}_{net}(BKB)_{Q,E}$ has been constructed from a Bayesian knowledge base *BKB*, we associate each non-state-input node A in the network with a CPT. Recall that each clause in *PB* is attached with a CPT in T_x and that each edge in $2\mathcal{S}_{net}(BKB)_{Q,E}$ is labeled with a number referring to a clause in PB that produces this edge. We distinguish between two cases:

(1) A has no parent node in $2\mathcal{S}_{net}(BKB)_{Q,E}$. This means that A (as a head) only has unit clauses in $\mathcal{I}_{clause}(BKB)$. The CPT for A, denoted $\mathbf{P}(A)$, is then built from the CPTs attached to the unit clauses. (Recall that an influence clause prefixed with a number k shares the CPT attached to the kth clause in PB.) $\mathbf{P}(A)$ represents the prior probability distribution of A, thus being referred to as a *prior* CPT.

(2) Otherwise, the CPT for A, denoted $\mathbf{P}(A|B_1, \ldots, B_m)$, is built from the CPTs attached to all of its nonunit clauses (with A as a head) in $\mathcal{I}_{clause}(BKB)$, where the B_is are the parent nodes of A. $\mathbf{P}(A|B_1, \ldots, B_m)$ represents a probability distribution of A conditioned on its parent nodes, thus being referred to as a *posterior* CPT.

The construction of a CPT from a set of CPTs is done by combining these CPTs in terms of the combination rule CR specified in BKB (see Definition 3.1).

Example 5.3 (*Example* 5.2 *Continued*). Let CPT$_i$ denote the CPT attached to the ith clause in PB_1. We build a CPT for each non-state-input node in $2\mathcal{S}_{net}(BKB_1)_{Q,E}$ (Figure 7).

—$aids(p1)$ has three parent nodes, derived from the third and fourth clause in PB_1, respectively, so a posterior CPT for $aids(p1)$, $\mathbf{P}(aids(p1)|aids(p1)_{t-1}$, $aids(p2)$, $contact(p1, p2))$, is built by combining CPT$_3$ and CPT$_4$. For the same reason, a posterior CPT for $aids(p2)$, $\mathbf{P}(aids(p2)|aids(p1)_{t-1}, aids$ $(p2)_{t-1}, contact(p2, p1))$, is computed by combining CPT$_3$ and CPT$_4$.

—$aids(p3)$ has only one parent node, derived from the third clause in PB_1, so the posterior CPT for $aids(p3)$, $\mathbf{P}(aids(p3)|aids(p3)_{t-1})$, is CPT$_3$.

—$contact(p1, p2)$ has no parent node and has only one edge labeled 5, so it has a prior CPT, $\mathbf{P}(contact(p1, p2))$, which is CPT$_5$. For the same reason, $contact(p2, p1)$ has a prior CPT, $\mathbf{P}(contact(p2, p1))$, which is CPT$_6$.

Note that state input nodes, $aids(p1)_{t-1}$, $aids(p2)_{t-1}$, and $aids(p3)_{t-1}$, do not need to have a CPT; they will be expanded, during the process of unrolling the two-slice DBN into a stationary DBN, to cover the time slices involved in the given query and evidence nodes.

We have presented ways to build a two-slice DBN (its network structure as well as associated CPTs) from a Bayesian knowledge base $BKB = \langle PB \cup CB, T_x, CR\rangle$. Recall that a stationary DBN consists of a two-slice DBN together with a prior probability distribution $\mathbf{P}(A_0)$ at time zero for each state random variable A. Let p be the predicate symbol of A. It is easy to prove that p is a recursive predicate symbol in PB. Therefore, $\mathbf{P}(A_0)$ is defined by $\mathbf{P}(p(\overrightarrow{X})_0)$ in T_x. As a result, BKB defines a stationary DBN.

6. RELATED WORK

As far as we can determine, the Bayesian knowledge base proposed in this article is the first PLP formalism that allows logic programs with recursive loops, interprets cyclic influences caused by recursive loops as feedback connections, and makes use of recursive loops to build a stationary dynamic Bayesian network. Another distinct feature is that its declarative semantics is built on the well-founded semantics of a logic program, so that the space of random variables and the direct influence relation is uniquely determined by the well-founded model.

A recent overview of existing representational frameworks that combine probabilistic reasoning with logic (i.e., logic-based approaches) or with relational representations (i.e., non-logic-based approaches) is given by Raedt and Kersting [2003] (also see the additional bibliography in Getoor and Grant [2006] and Richardson and Domingos [2006]). Typical non-logic-based approaches include probabilistic relational models (PRM), which are based on the entity-relationship (or object-oriented) model [Getoor 2001; Jaeger 1997; Koller and Pfeffer 1998; Pfeffer and Koller 2000], and relational Markov networks, which combine Markov networks and SQL-like queries [Taskar et al. 2002]. Representative logic-based approaches include frameworks based on the KBMC idea [Bacchus 1994; Breese 1992; Fabian and Lambert 1998; Glesner and Koller 1995; Goldman and Charniak 1993; Kersting and Raedt 2000; Ngo and Haddawy 1997; Poole 1993], stochastic logic programs (SLP) based on stochastic context-free grammars [Cussens 2000; Muggleton 1996], parameterized logic programs based on distribution semantics (PRISM) [Sato and Kameya 2005], CLP(BN) based on constraint logic programming [Costa et al. 2003], and more. Most recently, a unifying framework, called *Markov logic networks*, has been proposed by Richardson and Domingos [2006]. Markov logic networks subsume first-order logic and Markov networks. Since our work follows the KBMC idea focusing on how to build a Bayesian network directly from a general logic program, it is closely related to three representative existing PLP approaches: the context-sensitive PLP developed by Haddawy and Ngo [1997], Bayesian logic programming proposed by Kersting and Raedt [2000], and the time parameter-based approach presented by Glesner and Koller [1995]. In this section, we make a detailed comparison of our work with the three closely related approaches. We also briefly discuss another recent related work, relational dynamic Bayesian networks introduced by Sanghai, Domingos and Weld [Sanghai et al. 2005].

6.1 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 6.1. A *probabilistic knowledge base* is a four tuple $\langle PD, PB, CB, CR\rangle$, where

—*PD* defines a set of probabilistic predicates (*p-predicates*) of the form $p(T_1, \ldots, T_m, V)$ 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

$$P(A_0|A_1,\ldots,A_l)=\alpha \leftarrow B_1,\ldots,B_m,\neg C_1,\ldots,\neg C_n, \qquad (11)$$

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 (11), each p-predicate A_i is of the form $q(t_1,\ldots,t_m,v)$, which simulates an equation $q(t_1,\ldots,t_m)=v$ with v being a value from the probabilistic domain of $q(t_1,\ldots,t_m)$. For instance, let $D_{color}=\{red, green, blue\}$ be the probabilistic domain of $color(X)$; then the p-predicate $color(X, red)$ simulates $color(X)=red$, meaning that the color of X is red. The left-hand side $P(A_0|A_1,\ldots,A_l)=\alpha$ expresses that the probability of A_0 conditioned on $A_1,\ldots,A_l$ is α. The right-hand side $B_1,\ldots,B_m,\neg C_1,\ldots,\neg C_n$ 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

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

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

PKB and BKB have the following important differences.

First, probabilistic rules of form (11) 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, *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 [Clark 1978] 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 Ngo and Haddawy [1997]) 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 et al. [2004, 2005] introduced *logical Bayesian networks* (LBN). LBN is similar to PKB except that it uses a special predicate $random(.)$ to define random variables and separates logical and probabilistic information by converting rules of form (11) into the form

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

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. LBN differs from BKB in the following: First, it defines (declares) random variables using special clauses of the form

$$random(A) \leftarrow B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n,$$

where A is a p-predicate and the B_js and C_ks are c-predicates. Note that such clauses contain no information about direct influences. In other words, random variables in LBN are defined independently of the direct influence relation. This does not quite seem to be appropriate; in many situations, the user may like to define random variables recursively in terms of the direct influence relation. Take the AIDS program (see Example 3.1) as an example. We should allow the user to deduce the random variable $aids(p2)$ from $aids(p1)$ by making use of the direct influence of $aids(p1)$ on $aids(p2)$, as was done in Example 4.1. Second, LBN has no mechanism for handling cyclic influences. In fact, it assumes acyclic logic programs (see Section 2.3 of the article [Fierens et al. 2005]). Finally, although the well-founded model is also used for the logic contexts, only a declarative semantics for LBN is briefly described and no procedural semantics for LBN has been developed.

Most recently, Getoor and Grant [2006] introduced a probabilistic relational language (PRL). PRL is a recasting of the major ideas introduced in PRM [Koller and Pfeffer 1998] into a logic programming framework. It is very similar in spirit to LBN except that a general notion of aggregates is introduced based on PRM.

6.2 Comparison with Bayesian Logic Programming

Building on Ngo and Haddawy's [1997] work, Kersting and De Raedt [2000] introduced the framework of Bayesian logic programs. A *Bayesian logic program* (BLP) is a triple $\langle P, T_x, CR\rangle$ 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. A distinct feature of BLP over PKB is its separation of probabilistic information (T_x) from logic clauses (P). According to Kersting and Raedt [2000], we understand that a *well-defined* logic program is a positive logic program satisfying the range restriction together with other restrictions that guarantee acyclicity.[8] 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. The AIDS program PB_1 (see Example 3.1) is not well-defined, even if the third clause $aids(X) \leftarrow aids(X)$ is removed. BLP relies on the least Herbrand

[8]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.

model semantics and applies SLD-resolution to make backward-chaining inferences.

BLP has two important differences from BKB. 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 has no mechanism for handling cyclic influences (BLP excludes cyclic influences by requiring its programs be well-defined). Due to choosing SLD-resolution for backward-chaining inferences, it does not allow recursive loops (otherwise, the procedure would not terminate). BKB can be viewed as an extension of BLP with mechanisms for handling contexts, recursive loops, and cyclic influences. Such an extension is substantial and clearly nontrivial.

6.3 Comparison with the Time Parameter-Based Approach

The time parameter-based framework (TPF) proposed by Glesner and Koller [1995] is also a triple $\langle P, T_x, CR\rangle$, 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 later than the time arguments in the body.[9] This framework is implemented in Prolog, that is, clauses are represented as Prolog rules and goals are evaluated applying SLDNF-resolution. Glesner and Koller [1995, p. 221] stated: “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.

We observe the following significant differences between TPF and BKB. First, TPF is a temporal model and its logic programs contain a time argument for every predicate. It always builds a DBN from a logic program even if there is no cyclic influence. In contrast, logic programs in BKB contain no time parameters. When there is no cyclic influence, BKB builds a regular Bayesian network from a logic program (in this case, BKB serves as a nontemporal model); when cyclic influences occur, it builds a stationary DBN, represented by a two-slice DBN (in this case, BKB serves as a special temporal model). Second, TPF uses time steps to describe direct influences (in the way that for any A and B such that B has a direct influence on A, the time argument T_2 in B is at least one time step earlier than T_1 in A), while BKB uses time slices (implied by recursive loops of form (1)) to model cycles of direct influences (feedbacks). TPF fully relies on the user to detect the temporal nature of the domain and to encode concrete time steps (T_1 in the head and T_2 in the body of each clause), whereas BKB detects time slices (feedbacks) automatically. Third, most importantly TPF avoids recursive loops by introducing time parameters to enforce acyclicity of a logic program. A serious problem with this method is that it may lose and/or produce wrong answers to some queries. To explain this, let P be a logic program and P_t be P with additional time arguments added to each

[9]The idea of including a time parameter in each predicate is also mentioned in BLP [Kersting and Raedt 2000] and CLP(BN) [Costa et al. 2003].

predicate (as in TPF). If the transformation from P to P_t is correct, it must hold that for any query $p(.)$ over P, an appropriate time argument $N = 0, 1, 2, \ldots$ can be determined such that the query $p(., N)$ over P_t has the same set of answers as $p(.)$ over P when the time arguments in the answers are ignored. It turns out, however, that this condition does not hold in general cases. Note that finding an appropriate N for a query $p(.)$ such that evaluating $p(., N)$ over P_t (applying SLDNF-resolution) yields the same set of answers as evaluating $p(.)$ over P corresponds to finding an appropriate depth-bound M such that cutting all SLDNF-derivations for the query $p(.)$ at depth M does not lose any answers to $p(.)$. The latter is the well-known loop problem in logic programming [Bol et al. 1991]. Since the loop problem is undecidable in general, it is very hard, if not impossible, to automatically determine such a depth-bound M (respectively a time argument N) for an arbitrary query $p(.)$ [Bol et al. 1991; Shen et al. 2001, 2003]. We further illustrate this claim using a concrete example.

Example 6.1. The following logic program defines a *path* relation; that is, there is a path from X to Y if either there is an edge from X to Y or, for some Z, there is a path from X to Z and an edge from Z to Y.

P : 1. $e(s, b1)$.
2. $e(b1, b2)$.
......
99. $e(b98, b99)$.
100. $e(b99, g)$.
101. $path(X, Y) \leftarrow e(X, Y)$.
102. $path(X, Y) \leftarrow path(X, Z), e(Z, Y)$.

To avoid recursive loops, TPF may transform P into the following program.

P_t : 1. $e(s, b1, 0)$.
2. $e(b1, b2, 0)$.
......
99. $e(b98, b99, 0)$.
100. $e(b99, g, 0)$.
101. $e(X, Y, T1) \leftarrow T2 = T1 - 1, e(X, Y, T2)$.
102. $path(X, Y, T1) \leftarrow T2 = T1 - 1, e(X, Y, T2)$.
103. $path(X, Y, T1) \leftarrow T2 = T1 - 1, path(X, Z, T2), e(Z, Y, T2)$.

P_t looks more complicated than P. In addition to having time arguments and time formulas, it has a new clause, the 101st clause, formulating that $e(X, Y)$ being true at present implies it is true in the future.

Let us see how to check if there is a path from s to g. In the original program P, we simply pose a query $? - path(s, g)$. In the transformed program P_t, however, we have to determine a specific time parameter N and then pose a query $? - path(s, g, N)$, such that evaluating $path(s, g)$ over P yields the same answer as evaluating $path(s, g, N)$ over P_t. Interested readers can practice this query evaluation using different values for N. The answer to $path(s, g)$ over P

is yes. However, we would get an answer *no* to the query $path(s, g, N)$ over P_t if we choose any $N < 100$.

Finally, note that BKB focuses only on stationary DBNs and thus it is restricted to first-order Morkov models. However, TPF has no such a restriction.

Recently, Sanghai et al. [2003, 2005] introduced a relational dynamic Bayesian network (RDBN). RDBN is a pair of networks $(M_0, M_{\rightarrow})$, where M_0 is a relational Bayesian network representing the probability distribution over the state of the relational domain at time zero and $M_{\rightarrow}$ is a two-time-slice relational dynamic Bayesian network representing the transition probability distribution. RDBN is an extension of a dynamic Bayesian network to first-order logic, where nodes of the network are logic predicates (atoms). It models a stationary DBN (first-order Markov plus stationary transition).

RDBN is essentially different from BKB. First, it is not a PLP approach. In RDBN, the direct influences on each node R are defined by directly specifying a set of parent nodes $Pa(R)$ for R. In BKB, however, direct influences are defined using logic program clauses of form (4). Second, no contexts are considered in RDBN. Third, cyclic influences are excluded in RDBN by assuming a complete ordering $\prec$ on the predicate symbols and the constants in the relational domain. Finally, as a temporal model RDBN fully relies on the user to detect the temporal nature of the domain and explicitly specifies the inter-probabilistic relations, $M_{\rightarrow}$, between the random variables of time slices $t-1$ and t. In contrast, BKB can serve either as a nontemporal model or a temporal model, where the interprobabilistic relations (feedbacks) are detected automatically.

7. CONCLUSIONS AND DISCUSSION

We have developed a novel method for reasoning with recursive loops under the PLP framework. We observed that recursive loops in a logic program potentially imply a time sequence and thus can be used to model a stationary DBN without using explicit time parameters. We introduced a new PLP formalism—a Bayesian knowledge base. It allows recursive loops and contains logic clauses of form (4). These logic clauses naturally integrate the direct influence information, the context, and the type constraints, and are evaluated under the well-founded semantics. We established a declarative semantics for a Bayesian knowledge base and developed algorithms for building a two-slice DBN from a Bayesian knowledge base.

Our work provides a solution to the problem of recursive loops. To further highlight its significance, we summarize the following important observations.

(1) Recursive loops and recursion through negation are unavoidable in modeling real-world domains, so the well-founded semantics together with its top-down inference procedures is well suitable for the PLP backward-chaining inferences.
(2) Cyclic influences caused by 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.[10] 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, and thus the user can present queries and evidences over the DBN using time parameters at his/her convenience.

(3) Enforcing acyclicity of a logic program by introducing time parameters is not an effective way to handle recursive loops. First, such a method transforms the original nontemporal logic program into a more complicated temporal program and builds a dynamic Bayesian network from the transformed program even if there exist no cyclic influences (in this case, there is no state random variable and the original program defines a regular Bayesian network). Second, it relies on time steps to define (individual) direct influences, but time slices (intervals) are needed to model cycles of direct influences (feedbacks) caused by recursive loops. Finally, to pose a query over the transformed program, an appropriate time parameter must be specified. As illustrated in Example 6.1, it is hard to automatically determine such a time parameter for an arbitrary query.

A Bayesian knowledge base has its limitations. It is restricted to logic programs with the bounded-term-size property and only induces a first-order Morkov and stationary DBN.

Promising future work includes (1) developing algorithms for learning Bayesian knowledge bases from data and (2) using Bayesian knowledge bases to model large real-world problems. We are considering building a large Bayesian knowledge base for traditional Chinese medicine, where we have already collected a large volume of diagnostic rules.

ACKNOWLEDGMENTS

We are grateful to all anonymous referees for their constructive comments and valuable suggestions, which greatly helped improve the article. We would also like to thank Jia-Huai You, Li-Yan Yuan and Qiang Yang for their interesting discussions on the early version of this article.

REFERENCES

Apt, K. R. and Bezem, M. 1991. Acyclic programs. *New Gen. Comput. 29*, 3, 335–363.

Bacchus, F. 1994. Using first-order probability logic for the construction of bayesian networks. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*. 219–226.

Bol, R. N., Apt, K. R., and Klop, J. W. 1991. An analysis of loop checking mechanisms for logic programs. *Theoret. Comput. Sci. 86*, 1, 35–79.

[10]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.

BREESE, J. S. 1992. Construction of belief and decision networks. *Computat. Intell. 8*, 4, 624–647.

CACM. 1995. Real-world applications of Bayesian networks. *Commun. ACM 38*, 3, 24–57.

CHEN, W. D., SWIFT, T., AND WARREN, D. S. 1995. Efficient top-down computation of queries under the well-founded semantics. *J. Log. Program. 24*, 3, 161–199.

CHEN, W. D. AND WARREN, D. S. 1996. Tabled evaluation with delaying for general logic programs. *J. ACM 43*, 1, 20–74.

CLARK, K. L. 1978. Negation as failure. In *Logic and Databases*. Plenum, New York, NY, 293–322.

COSTA, V. S., PAGE, D., QAZI, M., AND CUSSENS, J. 2003. CLP(BN): Constraint logic programming for probabilistic knowledge. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*. 517–524.

CUSSENS, J. 2000. Stochastic logic programs: sampling, inference and applications. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence*. 115–122.

FABIAN, I. AND LAMBERT, D. A. 1998. First-order Bayesian reasoning. In *Proceedings of the 11th Australian Joint Conference on Artificial Intelligence*. Springer, Berlin, Germany, 131–142.

FIERENS, D., BLOCKEEL, H., BRUYNOOGHE, M., AND RAMON, J. 2005. Logical Bayesian networks and their relation to other probabilistic logical models. In *Proceedings of the Fifteenth International Conference on Inductive Logic Programming*. Springer, Bonn, Germany.

FIERENS, D., BLOCKEEL, H., RAMON, J., AND BRUYNOOGHE, M. 2004. Logical bayesian networks. In *Proceedings of the 3rd Workshop on Multi-Relational Data Mining* (Seattle, WA).

GETOOR, L. 2001. *Learning Statistical Models from Relational Data*. Stanford University, Ph.D. dissertation. Stanford, CA.

GETOOR, L. AND GRANT, J. 2006. PRL: A probabilistic relational language. *Mach. Learn. 62*, 1/2, 7–31.

GLESNER, S. AND KOLLER, D. 1995. Constructing flexible dynamic belief networks from first-order probabilistic knowledge bases. In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning under Uncertainty* (Fribourg, Switzerland). 217–226.

GOLDMAN, R. AND CHARNIAK, E. 1993. A language for construction of belief networks. *IEEE Trans. Patt. Anal. Mach. Intell. 15*, 3, 196–208.

HADDAWY, P. 1994. Generating Bayesian networks from probabilistic logic knowledge bases. In *Proceedings of the Tenth Annual Conference on Uncertainty in Artificail Intelligence*.

JAEGER, M. 1997. Relational Bayesian networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*. 266–273.

KANAZAWA, K., KOLLER, D., AND RUSSELL, S. 1995. Stochastic simulation algorithms for dynamic probabilistic networks. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*.

KERSTING, K. AND RAEDT, L. D. 2000. Bayesian logic programs. In *Work-in-Progress Reports of the Tenth International Conference on Inductive Logic Programming*. (A full version: Tech. Rep. 151, Institute for Computer Science, University of Freiburg, Freiburg, Germany, April 2001).

KOLLER, D. AND PFEFFER, A. 1998. Probabilistic frame-based systems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA, 580–587.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Germany.

MUGGLETON, S. 1996. Stochastic logic programs. In *Advances in Inductive Logic Programming*. IOS Press, Amsterdam, The Netherlands.

NG, R. AND SUBRAHMANIAN, V. 1992. Probabilistic logic programming. *Inform. Computat. 101*, 150–201.

NGO, L. AND HADDAWY, P. 1997. Answering queries from context-sensitive probabilistic knowledge bases. *Theoret. Comput. Sci. 171*, 147–177.

PEARL, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA.

PFEFFER, A. AND KOLLER, D. 2000. Semantics and inference for recursive probability models. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA, 538–544.

POOLE, D. 1993. Probabilistic Horn abduction and Bayesian networks. *Artific. Intell. 64*, 1, 81–129.

RAEDT, L. D. AND KERSTING, K. 2003. Probabilistic logic learning. *SIGKDD Explor. 5*, 1, 31–48.

RICHARDSON, M. AND DOMINGOS, P. 2006. Markov logic networks. *Mach. Learn. 62*, 1/2, 107–136.

RUSSELL, S. AND NORVIG, P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ.

SAGONAS, K., SWIFT, T., AND WARREN, D. 1998. *The XSB Programmer's Manual (Version 1.8)*. Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY. Available online at http://www.cs.sunysb.edu/sbprolog/xsb-page.html.

SANGHAI, S., DOMINGOS, P., AND WELD, D. 2003. Dynamic probabilistic relational models. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (Acapulco, Mexico).

SANGHAI, S., DOMINGOS, P., AND WELD, D. 2005. Relational dynamic bayesian networks. *J. Artific. Intell. Res. 24*, 759–797.

SATO, T. AND KAMEYA, Y. 2005. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artific. Intell. Res. 15*, 391–454.

SHEN, Y. D., YOU, J. H., AND YUAN, L. Y. 2004. Enhancing global SLS-resolution with loop cutting and tabling mechanisms. *Theoret. Comput. Sci. 328*, 3, 271–287.

SHEN, Y. D., YOU, J. H., YUAN, L. Y., SHEN, S. P., AND YANG, Q. 2003. A dynamic approach to characterizing termination of general logic programs. *ACM Trans. Computat. Log. 4*, 4, 417–430.

SHEN, Y. D., YUAN, L. Y., AND YOU, J. H. 2001. Loop checks for logic programs with functions. *Theoret. Comput. Sci. 266*, 1/2, 441–461.

TASKAR, B., ABEEL, P., AND KOLLER, D. 2002. Discriminative probabilistic models for relational data. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence* (Edmonton, Canada). 485–492.

ULLMAN, J. D. 1988. *Database and Knowledge-Base Systems*. Computer Science Press, New York.

VAN GELDER, A. 1989. Negation as failure using tight derivations for general logic programs. *J. Log. Program. 6*, 1/2, 109–133.

VAN GELDER, A., ROSS, K., AND SCHLIPF, J. 1991. The well-founded semantics for general logic programs. *J. ACM 38*, 3, 620–650.

VARDI, M. 1982. The complexity of relational query languages. In *Proceedings of the ACM Symposium on Theory of Computing*. 137–146.

WELLMAN, M. P., BREESE, J. S., AND GOLDMAN, R. P. 1992. From knowledge bases to decision models. *Knowl. Eng. Rev. 7*, 1, 35–53.

Received June 2005; revised September 2006; accepted March 2007