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

Yi-Dong Shen

Laboratory of Computer Science, Institute of Software

Chinese Academy of Sciences, Beijing 100080, China

Email: ydshen@ios.ac.cn

Qiang Yang

Department of Computing Science, Hong Kong University of Science and Technology

Hong Kong, China

Email: qyang@cs.ust.hk

Jia-Huai You and Li-Yan Yuan

Department of Computing Science, University of Alberta

Edmonton, Alberta, Canada T6G 2H1

Email: {you, yuan}@cs.ualberta.ca

Abstract

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

*A preliminary version appears in the 15th International Conference on Inductive Logic Programming.

SLG-resolution to compute the space of random variables together with their parental connections. We introduce a novel notion of influence clauses, based on which a declarative semantics for a Bayesian knowledge base is established and algorithms for building a two-slice dynamic Bayesian network from a logic program are developed.

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

1 Introduction

Probabilistic logic programming (PLP) is a framework that extends the expressive power of Bayesian networks with first-order logic [20, 23]. 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 quite like query evaluation in logic programming. Therefore, existing PLP methods use a slightly adapted *SLD-* or *SLDNF-resolution* [18] as the backward-chaining procedure.

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

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

where for any $i \geq 1$, A_i is the same as A_{i+1} up to variable renaming.[1] Such loops present a challenging problem to the PLP framework. On the one hand, they loop forever so that the PLP backward-chaining inferences would never stop. On the other hand, they may generate cyclic influences, which are disallowed in Bayesian networks.

Two representative approaches have been proposed to avoid recursive loops. The first one is by Ngo and Haddawy [20] and Kersting and De Raedt [17], who restrict to considering only acyclic logic programs [1]. The second approach, proposed by Glesner and Koller [13], 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.

In this paper, we propose a solution to the problem of recursive loops under the PLP framework. Our method is not restricted to acyclic logic programs, nor does it rely on explicit time parameters. Instead, it makes use of recursive loops to derive a stationary dynamic Bayesian network. We will make two novel contributions. First, we introduce the *well-founded* semantics [33] 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

[1] The *left-most* computation rule [18] is assumed in this paper.

SLG-resolution [6] (or *SLTNF-resolution* [29]) to make the backward-chaining inferences. As a result, termination of the PLP backward-chaining process is guaranteed. Second, we observe that under the PLP framework recursive loops (cyclic influences) define feedbacks, thus implying a time sequence. For instance, the clause $aids(X) \leftarrow aids(Y), contact(X, Y)$ introduces recursive loops

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

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

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

Such cyclic influences represent feedback connections, i.e., 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) imply a time sequence of the form

$$A \underbrace{\leftarrow ... \leftarrow}_{t} A \underbrace{... \leftarrow}_{t-1} A \underbrace{... \leftarrow}_{t-2} A \ ... \tag{2}$$

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 and thus can be compactly represented as a two-slice dynamic Bayesian network.

The paper is structured as follows. In Section 2, we review some concepts concerning Bayesian networks and logic programs. In Section 3, we introduce a new PLP formalism, called 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 two-slice dynamic 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 [21] and logic programming [18]. In particular, we assume the reader is familiar with the well-founded semantics [33] as well as SLG-resolution [5]. Here we review some basic concepts concerning dynamic Bayesian networks (DBNs). DBNs are introduced to model the evolution of the state of the

environment over time [16]. 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 paper, 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 intra-probabilistic relations between random variables in each individual time slice t ($t > 0$) and the inter-probabilistic relations between the random variables of each two consecutive time slices $t-1$ and t. If both the intra- and inter-probabilistic relations are the same for all time slices (in this case, the DBN is a repetition of a Bayesian network over time; see Figure 1), the DBN is called a *stationary* DBN [24]; otherwise it is called a *flexible* DBN [13]. As far as we know, most existing DBN systems reported in the literature are stationary DBNs.

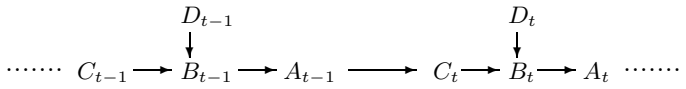


Figure 1: A stationary DBN structure.

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

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

A two-slice DBN models a feedback system, where a cycle of direct influences establishes a feedback connection. For convenience, we depict feedback connections with dashed edges. Moreover, we refer to nodes coming from slice $t-1$ as *state input nodes* (or *state input variables*).[2]

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).

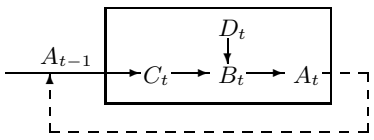


Figure 2: A two-slice DBN structure (a feedback system).

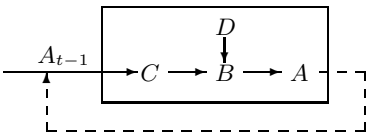


Figure 3: A simplified two-slice DBN structure.

Observe that in a two-slice DBN, all random variables except state input nodes have the same subscript t. In the sequel, the subscript t is omitted for simplification of the structure. For instance, the two-slice DBN of Figure 2 is simplified to 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 $p(X_1, ..., X_n)$ 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

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

where A, the B_is and C_js are atoms. We use $HU(P)$ and $HB(P)$ to denote the Herbrand universe and Herbrand base of P, respectively, and use $WF(P) = <I_t, I_f>$ to denote the well-founded model of P, where $I_t, I_f \subseteq HB(P)$, and every A in I_t is true and every A in I_f is false in $WF(P)$. By a *(Herbrand) ground instance* of a clause/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 $A \leftarrow B_1, ..., B_k, \neg B_{k+1}, ..., \neg B_n$ of any clause in P, $map(A) > map(B_i)$ $(1 \leq i \leq n)$ [1]. P is said to have the *bounded-term-size property* w.r.t. a set of predicates $\{p_1(.), ..., p_t(.)\}$ if there is a function $f(n)$ such that for any $1 \leq i \leq t$ whenever a top goal $G_0 = \leftarrow p_i(.)$ has no argument whose term size exceeds n, no atoms in any SLDNF- (or SLG-) derivations for G_0 have an argument whose term size exceeds $f(n)$ (this definition is adapted from [32]).

[2]When no confusion would occur, we will refer to nodes and random variables exchangeably.

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 $<PB \cup CB, T_x, CR>$, where

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

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

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

- T_x is a set of conditional probability tables (CPTs) of the form $\mathbf{P}(p(.)|p_1(.), ..., p_l(.))$, each being attached to a clause (4) in PB.

- CR is a combination rule such as *noisy-or, min* or *max* [17, 20, 24].

A Bayesian knowledge base contains a logic program that can be divided into two parts, PB and CB. PB defines a direct influence relation, each clause (4) saying that the atoms $p_1(.), ..., p_l(.)$ have direct influences on $p(.)$ in the context that $B_1, ..., B_m, \neg C_1, ..., \neg C_n$, $member(X_1, DOM_1), ..., member(X_s, DOM_s)$ is true in $PB \cup CB$ under the well-founded semantics. Note that the special literal *true* is used in clause (4) to mark the beginning of the context; it is always true in the well-founded model $WF(PB \cup CB)$. For each variable X_i in the head $p(.)$, $member(X_i, DOM_i)$ is used to enforce the type constraint on X_i, i.e. the value of X_i comes from its domain DOM_i. CB assists PB in defining the direct influence relation by introducing some auxiliary predicates (such as $member(.)$) to describe contexts.[3] Clauses in CB do not describe direct influences.

Recursive loops 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 paper, we focus on Datalog programs, although the proposed approach applies to logic programs with the bounded-term-size property (w.r.t. the set of predicates appearing in the heads of clauses in PB) as well. Datalog programs are widely used in database and knowledge base systems [31] and have a polynomial time complexity in computing their

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

well-founded models [33]. In the sequel, we assume that except for the predicate $member(.)$, $PB \cup CB$ is a Datalog program.

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

A Bayesian knowledge base has the following important property.

Theorem 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 $(p_1(.), ..., p_l(.), true, B_1, ..., B_m, \neg C_1, ..., \neg C_n, member(X_1, DOM_1), ..., member(\ X_s, DOM_s))\theta$ is evaluated true in the well-founded model $WF(PB \cup CB)$. Note that the type constraints $(member(X_1, DOM_1), ..., member(X_s, DOM_s))\theta$ 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 $B_1, ..., B_m, \neg C_1, ..., \neg C_n$ is true, we assume that the related type constraints are true as well.

Example 3.1 We borrow the well-known AIDS program from [13] (a simplified version) as a running example to illustrate our PLP approach. It is formulated by a Bayesian knowledge base KB_1 with the following logic program:[4]

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. $contact(p2, p1)$.

Note that both the 3rd and the 4-th clause produce recursive loops. The 3rd clause also has a cyclic direct influence. Conceptually, the two clauses model the fact that the direct

[4]This Bayesian knowledge base $KB_1 = <PB_1 \cup CB_1, T_{x_1}, CR_1>$ 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.

influences on $aids(X)$ come from whether X was infected with aids earlier (the feedback connection induced from the 3rd clause) or whether X has contact with someone Y who is infected with aids (the 4-th clause).

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 KB. We then define probability distributions induced by KB.

4.1 Space of Random Variables and Influence Clauses

A Bayesian knowledge base KB defines a direct influence relation over a subset of $HB(PB)$. Recall that any random variable in a Bayesian network is either an input node (with no parent nodes) or a node on which some other nodes (i.e. its parent nodes) in the network have direct influences. Since an input node can be viewed as a node whose direct influences come from an empty set of parent nodes, we can define a space of random variables from a Bayesian knowledge base KB by taking all unit clauses in PB as input nodes and deriving the other nodes iteratively based on the direct influence relation defined by PB. Formally, we have

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

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

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

Example 4.1 (Example 3.1 continued) The clauses 1, 2, 5 and 6 are unit clauses, thus random variables. $aids(p2)$ is then derived applying the 4-th clause. Consequently, $\mathcal{S}(KB_1) = \{aids(p1), aids(p2), aids(p3), contact(p1, p2), contact(p2, p1)\}$. $aids(p1)$ 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 $WF(PB \cup CB) = <I_t, I_f>$ be the well-founded model of $PB \cup CB$ and let $I_{PB} = \{p(.) \in I_t | p$ occurs in the head of some clause in $PB\}$. The following result shows that the space of random variables is uniquely determined by the well-founded model.

Theorem 4.1 $\mathcal{S}(KB) = I_{PB}$.

Proof: First note that all unit clauses in PB are both in $\mathcal{S}(KB)$ 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 $A \in \mathcal{S}(KB)$. 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 $B \in \mathcal{S}(KB)$ whose maximum depth d of backward derivations is below k. Let $d = k$ for A. There must be a ground instance $A \leftarrow A_1, ..., A_l, true, B_1, ..., B_m, \neg C_1, ..., \neg C_n$ of a clause in PB such that the A_is are already in $\mathcal{S}(KB)$ and $B_1, ..., B_m, \neg C_1, ..., \neg C_n$ 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 $A \in \mathcal{S}(KB)$. For the induction step, assume $B \in \mathcal{S}(KB)$ 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 $A \leftarrow A_1, ..., A_l, true, ...$ 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 $\mathcal{S}(KB)$. By Definition 4.1, $A \in \mathcal{S}(KB)$. $\square$

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

Algorithm 1: Computing random variables.

1. $\mathcal{S}'(KB) = \emptyset$.
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) $\mathcal{S}'(KB) = \mathcal{S}'(KB) \cup \mathcal{T}_{p_i(\overrightarrow{X_i})}$.

3. Return $\mathcal{S}'(KB)$.

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

Proof: Let $WF(PB \cup CB) = <I_t, I_f>$ be the well-founded model of $PB \cup CB$. By the soundness and completeness of SLG-resolution, Algorithm 1 will terminate with a finite output $\mathcal{S}'(KB)$ that consists of all answers of $p_i(\overrightarrow{X_i})$ $(1 \leq i \leq t)$. By Theorem 3.1, all answers in $\mathcal{S}'(KB)$ are ground. This means $\mathcal{S}'(KB) = I_{PB}$. Hence, by Theorem 4.1 $\mathcal{S}'(KB) = \mathcal{S}(KB)$. □

We introduce the following principal concept.

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

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

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

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_i*s are random variables in* $\mathcal{S}(KB)$.

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* k*-th clause in* PB*, if and only if there is an influence clause in* $\mathcal{I}_{clause}(KB)$ *of the form* $k.\ A \leftarrow A_1, ..., A_i, ..., A_l$.

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

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

The following result is immediate from Theorem 4.4.

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

Corollary 4.5 *For any atom A, A is in $\mathcal{S}(KB)$ if and only if there is an influence clause in $\mathcal{I}_{clause}(KB)$ 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 does. Therefore, a Bayesian network can be built directly from $\mathcal{I}_{clause}(KB)$ provided the influence clauses are available.

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 $1 \leq i \leq t$ [5]. For each answer A of $p_i(\overrightarrow{X_i})$ in $\mathcal{S}(KB)$ there must be a success branch (i.e. a branch starting at the root node and ending at a node marked with *success*) in the tree that generates the answer. Let $p_i(.) \leftarrow A_1, ..., A_l, true, ...$ be the k-th 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 body $A_1, ..., A_l, true, ...$ 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, $k.\ p_i(.)\theta \leftarrow A_1\theta, ..., A_l\theta$ 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, $p_i(.) \leftarrow A_1, ..., A_l, true, ...$ be the k-th 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 $k.\ p_i(.)\theta \leftarrow A_1\theta, ..., A_l\theta$.*

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. 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 into $\mathcal{I}'_{clause}(KB)$.
2. Return $\mathcal{I}'_{clause}(KB)$.

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

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 $\mathcal{I}'_{clause}(KB)$ is finite.

By Theorem 4.6, any clause in $\mathcal{I}'_{clause}(KB)$ is an influence clause in $\mathcal{I}_{clause}(KB)$. We now prove the converse. Let $k.\ A \leftarrow A_1, ..., A_l$ be an influence clause in $\mathcal{I}_{clause}(KB)$. Then

the k-th clause in PB $A' \leftarrow A'_1, ..., A'_l, true, ...$ has a ground instance of the form $A \leftarrow A_1, ..., A_l, true, ...$ 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 k-th clause, (2) the composition of all mgus along the branch is θ, and (3) $A \leftarrow A_1, ..., A_l, true, ...$ is an instance of $(A' \leftarrow A'_1, ..., A'_l, true, ...)\theta$. By Theorem 4.6, $k.\ A'\theta \leftarrow A'_1\theta, ..., A'_l\theta$ is an influence clause. Since any influence clause is ground, $k.\ A'\theta \leftarrow A'_1\theta, ..., A'_l\theta$ is the same as $k.\ A \leftarrow A_1, ..., A_l$. This influence clause from the success branch will be collected into $\mathcal{I}'_{clause}(KB)$ by Algorithm 2. Thus, any clause in $\mathcal{I}_{clause}(KB)$ is in $\mathcal{I}'_{clause}(KB)$. □

Example 4.2 (Example 4.1 continued) There are 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 i-th 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 3rd clause produces a child node $\leftarrow aids(X)$ (Figure 4). Then applying the answers of $aids(X)$ from the table $\mathcal{T}_{aids(X)}$ to the goal of this node leads to three success branches. Applying the mgu θ on each success branch to the 3rd clause yields three influence clauses of the form 3. $aids(pi) \leftarrow aids(pi)$ $(i = 1, 2, 3)$. As a result, we obtain the following set of influence clauses:

$\mathcal{I}_{clause}(KB_1)$: 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. $aids(p1) \leftarrow aids(p2), contact(p1, p2)$.
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 complexity of computing the well-founded model $WF(P)$ is polynomial [33, 34]. More precisely, the time complexity of SLG-resolution is $O(|P| * N^{\Pi_P+1} * logN)$, where $|P|$ is the number of clauses in P, Π_P is the maximum number of literals in the body of a clause, and N, the number of atoms of predicates in P that are not variants of each other, is a polynomial in the number of ground unit clauses in P [6].

$PB \cup CB$ is a Datalog program except for the $member(X_i, DOM_i)$ predicates (see Definition 3.1). Since each domain DOM_i is a finite list of constants, checking if X_i is

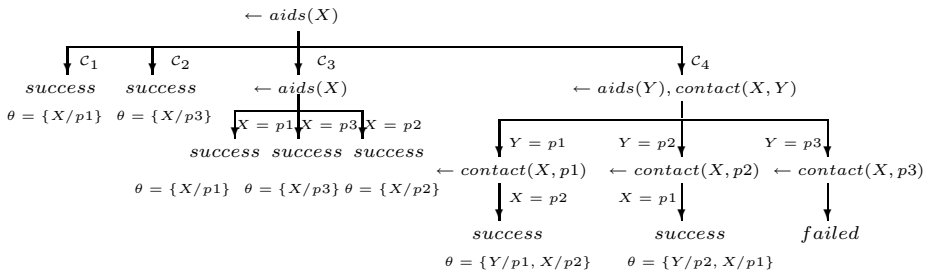


Figure 4: The proof tree for $\leftarrow aids(X)$.

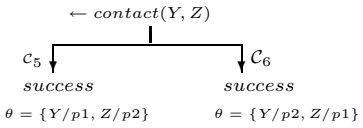


Figure 5: The proof tree for $\leftarrow contact(Y, Z)$.

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 $K_1 * K_2$. Since each clause in P is applied at most N times in SLG-resolution, the time of handling all $member(X_i, DOM_i)$s in all clauses in P is bounded by $|P| * N * K_1 * K_2$. This is also a polynomial, hence SLG-resolution computes the well-founded model $WF(PB \cup CB)$ in polynomial time. Therefore, we have the following result.

Theorem 4.8 *The time complexity of Algorithm 2 is polynomial.*

4.2 Probability Distributions Induced by KB

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 $\mathcal{S}(KB)$ that are not influenced by A.

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

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

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

Proof: When no cyclic influence occurs, the random variables in $\mathcal{S}(KB)$ 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 $\mathbf{P}(\mathcal{S}(KB)) = \mathbf{P}(\bigwedge_{A_i \in \mathcal{S}(KB)} A_i) = \mathbf{P}(A_1 | \bigwedge_{i=2} A_i) * \mathbf{P}(\bigwedge_{i=2} A_i) = \mathbf{P}(A_1|pa(A_1)) * \mathbf{P}(A_2 | \bigwedge_{i=3} A_i) * \mathbf{P}(\bigwedge_{i=3} A_i) = ... = \prod_{A_i \in \mathcal{S}(KB)} \mathbf{P}(A_i|pa(A_i))$ □

When there are cyclic influences, we cannot have a partial order on $\mathcal{S}(KB)$. By Definition 4.2 and Theorem 4.4, any cyclic influence, say "A_1 is influenced by itself," must be resulted from a set of influence clauses in $\mathcal{I}_{clause}(KB)$ of the form

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

These influence clauses generate a chain (cycle) of direct influences

$$A_1 \leftarrow A_2 \leftarrow A_3 \leftarrow ... \leftarrow A_n \leftarrow A_1 \tag{7}$$

which defines a feedback connection. Since a feedback system can be modeled by a two-slice DBN (see Section 2), the above influence clauses represent the same knowledge as the following ones do:

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

Here the A_is are state variables and $A_{1_{t-1}}$ is a state input 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 (6) to (8)), we can get rid of all cyclic influences and obtain a *generalized set* $\mathcal{I}_{clause}(KB)_g$ of influence clauses from $\mathcal{I}_{clause}(KB)$.[6]

Example 4.3 (Example 4.2 continued) $\mathcal{I}_{clause}(KB_1)$ can be transformed to the following generalized set of influence clauses by introducing three state input variables $aids(p1)_{t-1}$, $aids(p2)_{t-1}$ and $aids(p3)_{t-1}$.

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

$\mathcal{I}_{clause}(KB_1)_g$: 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. $contact(p2, p1)$.

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

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

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

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

Proof: Since $\mathcal{I}_{clause}(KB)_g$ produces no cyclic influences, the random variables in $\mathcal{S}(KB)_g$ can be arranged in a partial order such that if A_i is influenced by A_j then $j > i$. The proof then proceeds in the same way as that of Theorem 4.9. □

5 Building a Bayesian Network from a Bayesian Knowledge Base

5.1 Building a Two-Slice DBN Structure

From a Bayesian knowledge base KB, we can derive a set of influence clauses $\mathcal{I}_{clause}(KB)$, which defines the same direct influence relation over the same space $\mathcal{S}(KB)$ of random variables as $PB \cup CB$ does (see Theorem 4.4). Therefore, given a probabilistic query together with some evidences, we can depict a network structure from $\mathcal{I}_{clause}(KB)$, which covers the random variables in the query and evidences, by backward-chaining the related random variables via the direct influence relation.

Let Q be a probabilistic query and E a set of evidences, where all random variables come from $\mathcal{S}(KB)$ (i.e., they are heads of some influence clauses in $\mathcal{I}_{clause}(KB)$). Let TOP

consist of these random variables. An *influence network* of Q and E,[7] denoted $\mathcal{I}_{net}(KB)_{Q,E}$, is constructed from $\mathcal{I}_{clause}(KB)$ using the following algorithm.

Algorithm 3: Building an influence network.

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

Example 5.1 (Example 4.2 continued) To build an influence network from KB_1 that covers $aids(p1)$, $aids(p2)$ and $aids(p3)$, we apply Algorithm 3 to $\mathcal{I}_{clause}(\ KB_1)$ while letting $TOP = \{aids(p1), aids(p2), aids(p3)\}$. It generates an influence network $\mathcal{I}_{net}(KB_1)_{Q,E}$ as shown in Figure 6.

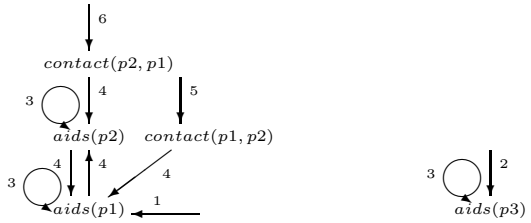


Figure 6: An influence network built from the AIDS program KB_1.

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 $\mathcal{I}_{net}(KB)_{Q,E}$, 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 $k.\ A_i \leftarrow A_1, ..., A_j, ..., A_l$ in $\mathcal{I}_{clause}(KB)$.*

[7]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 [24].

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

Theorem 5.2 *For any A_i, A_j in $\mathcal{I}_{net}(KB)_{Q,E}$, 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

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

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

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

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 $\mathcal{I}_{net}(KB)_{Q,E}$, 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 $\mathcal{I}_{net}(KB)_{Q,E}$ and let $W = \{A_j \in \mathcal{S}(KB) |$ for some $A_i \in TOP$, A_i is influenced by $A_j\}$. $V = TOP \cup W$.*[8]

Proof: That $\mathcal{I}_{net}(KB)_{Q,E}$ 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, $B_1, B_2, ..., B_m, A_j$ 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 $A_j \in W$. □

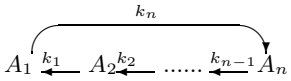
Theorem 4.9 shows that the probability distribution induced by KB can be computed over $\mathcal{I}_{clause}(KB)$. Let $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ denote an influence network that covers all random variables in $\mathcal{S}(KB)$. We show that the same distribution can be computed over

[8]This result suggests that an influence network is similar to a *supporting network* introduced in [20].

$\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$. For any node A_i in $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$, 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 $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ if and only if A_i is influenced by A_j in $\mathcal{I}_{clause}(KB)$. This means that the independence assumption (Assumption 1) applies to $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ as well, and that $\mathcal{I}_{clause}(KB)$ produces a cycle of direct influences if and only if $\mathcal{I}_{net}(KB)_{\mathcal{S}(KB)}$ contains the same (direct) loop. Combining these facts leads to the following immediate result.

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

Theorem 5.4 implies that an influence network without loops is a Bayesian network structure. Let us consider influence networks with loops. By Theorem 5.2, loops in an influence network are generated from recursive influence clauses of form (6) and thus they depict feedback connections of form (7). This means that an influence network with loops can be converted into a two-slice DBN, simply by converting each loop of the form



into a two-slice DBN path

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

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 variables backward to the same set of state variables (i.e., state input nodes). This process corresponds to generating an influence network $\mathcal{I}_{net}(KB)_{Q,E}$ from $\mathcal{I}_{clause}(KB)$ incrementally (adding nodes and edges one 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, $2\mathcal{S}_{net}(KB)_{Q,E}$, directly from $\mathcal{I}_{clause}(KB)$, where Q, E and TOP are the same as defined in Algorithm 3.

Algorithm 4: Building a two-slice DBN structure.

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

(b) If adding $A \overset{k}{\leftarrow} A_i$ to $2\mathcal{S}_{net}(KB)_{Q,E}$ produces a loop, then add to $2\mathcal{S}_{net}(KB)_{Q,E}$ a node $A_{i_{t-1}}$ and an edge $A \overset{k}{\leftarrow} A_{i_{t-1}}$, else add an edge $A \overset{k}{\leftarrow} A_i$ to $2\mathcal{S}_{net}(KB)_{Q,E}$.

3. Repeat step 2 until TOP becomes empty.

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

Example 5.2 (Example 5.1 continued) To build a two-slice DBN structure from KB_1 that covers $aids(p1)$, $aids(p2)$ and $aids(p3)$, we apply Algorithm 4 to $\mathcal{I}_{clause}(\ KB_1)$ while letting $TOP = \{aids(p1), aids(p2), aids(p3)\}$. It generates $2\mathcal{S}_{net}(KB_1)_{Q,E}$ as 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 as shown in Figure 8.

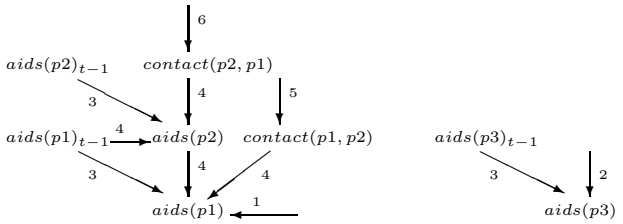


Figure 7: A two-slice DBN structure built from the AIDS program KB_1.

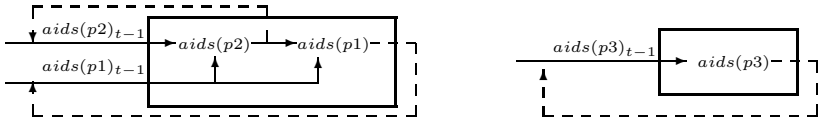


Figure 8: The feedback connections created by the AIDS program KB_1.

Algorithm 4 is Algorithm 3 enhanced with a mechanism for cutting loops (item 2b), i.e. when adding the current edge $A \overset{k}{\leftarrow} A_i$ to the network forms a loop, we replace it with an edge $A \overset{k}{\leftarrow} A_{i_{t-1}}$, where $A_{i_{t-1}}$ is a state input node. This is a process of transforming influence clauses (6) to (8). Therefore, $2\mathcal{S}_{net}(KB)_{Q,E}$ can be viewed as an influence network built from a generalized set $\mathcal{I}_{clause}(KB)_g$ of influence clauses. Let $\mathcal{S}(KB)_g$ be the set of random variables in $\mathcal{I}_{clause}(KB)_g$, as defined in Theorem 4.10. Let $2\mathcal{S}_{net}(KB)_{\mathcal{S}(KB)}$ denote a two-slice DBN structure (produced by applying Algorithm 4) that covers all random variables in $\mathcal{S}(KB)_g$. We then have the following immediate result from Theorem 5.4.

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

Remark 5.1 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 $\mathcal{I}_{clause}(KB)$. When no query and evidences are provided, we may apply Algorithm 4 to build a *complete* two-slice DBN structure, $2\mathcal{S}_{net}(KB)_{\mathcal{S}(KB)}$, which covers the space $\mathcal{S}(KB)$ of random variables, by letting TOP consist of all heads of influence clauses in $\mathcal{I}_{clause}(KB)$. This is a very useful feature, as in many situations the user may not be able to present the right queries unless a Bayesian network structure is shown.

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 $2\mathcal{S}_{net}(KB)_{Q,E}$ has been constructed from a Bayesian knowledge base KB, we associate each (non-state-input) node A in the network with a CPT. There are three cases. (1) If A (as a head) only has unit clauses in $\mathcal{I}_{clause}(KB)$, we build from the unit clauses a *prior* CPT for A as its prior probability distribution. (2) If A only has non-unit clauses in $\mathcal{I}_{clause}(KB)$, we build from the clauses a *posterior* CPT for A as its posterior probability distribution. (3) Otherwise, we prepare for A both a prior CPT (from the unit clauses) and a posterior CPT (from the non-unit clauses). In this case, A is attached with the posterior CPT; the prior CPT for A would be used, if A is a state variable, as the probability distribution of A in time slice 0 (only in the case that a two-slice DBN is unrolled into a stationary DBN starting with time slice 0).

Assume that the parent nodes of A are derived from n ($n \geq 1$) different influence clauses in $\mathcal{I}_{clause}(KB)$. Suppose these clauses share the following CPTs in T_x: $\mathbf{P}(A_1|B_1^1, ..., B_{m_1}^1)$, ..., and $\mathbf{P}(A_n|B_1^n, ..., B_{m_n}^n)$. (Recall that an influence clause prefixed with a number k shares the CPT attached to the k-th clause in PB.) Then the CPT for A is computed by combining the n CPTs in terms of the combination rule CR specified in Definition 3.1.

Example 5.3 (Example 5.2 continued) Let CPT_i denote the CPT attached to the i-th clause in PB_1. Consider the random variables in $2\mathcal{S}_{net}(KB_1)_{Q,E}$. Since $aids(p1)$ has three parent nodes, derived from the 3rd and 4-th clause in PB_1 respectively, the posterior CPT for $aids(p1)$ is computed by combining CPT_3 and CPT_4. $aids(p1)$ has also a prior CPT, CPT_1, derived from the 1st clause in PB_1. For the same reason, the posterior CPT for $aids(p2)$ is computed by combining CPT_3 and CPT_4. The posterior CPT for $aids(p3)$ is CPT_3 and its prior CPT is CPT_2. $contact(p1, p2)$ and $contact(p2, p1)$ have only prior CPTs, namely CPT_5 and 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. If the resulting stationary DBN starts with time slice 0, the prior CPTs, $\text{CPT}_{aids(p1)_0}$ and $\text{CPT}_{aids(p3)_0}$, for $aids(p1)$ and $aids(p3)$ are used as the probability distributions of $aids(p1)_0$ and $aids(p3)_0$.

Note that $aids(p2)$ is a state variable, but there is no unit influence clause available to build a prior CPT for it. We have two ways to derive a prior CPT, $\text{CPT}_{aids(p2)_0}$, for $aids(p2)$ from some existing CPTs. (1) $\text{CPT}_{aids(p2)_0}$ comes from averaging $\text{CPT}_{aids(p1)_0}$ and $\text{CPT}_{aids(p3)_0}$. For instance, let the probability of $aids(p1) = yes$ be 0.7 in $\text{CPT}_{aids(p1)_0}$ and the probability of $aids(p3) = yes$ be 0.74 in $\text{CPT}_{aids(p3)_0}$. Then the probability of $aids(p2) = yes$ is $(0.7 + 0.74)/2 = 0.72$ in $\text{CPT}_{aids(p2)_0}$. (2) $\text{CPT}_{aids(p2)_0}$ comes from averaging the posterior probability distributions of $aids(p2)$. For instance, let $\{0.9, 0.7, 0.4, 0.8\}$ be the posterior probabilities of $aids(p2) = yes$ in the posterior CPT for $aids(p2)$. Then the probability of $aids(p2) = yes$ is $(0.9 + 0.7 + 0.4 + 0.8)/4 = 0.7$ in $\text{CPT}_{aids(p2)_0}$.

6 Related Work

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 De Raedt and Kersting [8]. Typical non-logic-based approaches include probabilistic relational models (PRM), which are based on the entity-relationship (or object-oriented) model [12, 15, 22], and relational Markov networks, which combine Markov networks and SQL-like queries [30]. Representative logic-based approaches include frameworks based on the KBMC (Knowledge-Based Model Construction) idea [3, 4, 10, 13, 14, 17, 20, 23], stochastic logic programs (SLP) based on stochastic context-free grammars [7, 19], parameterized logic programs based on distribution semantics (PRISM) [26], and more. Most recently, a unifying framework, called *Markov logic*, has been proposed by Domingos and Richardson [9]. Markov logic subsumes first-order logic and Markov networks. Since our work follows the KBMC idea focusing on how to build a Bayesian network directly from a logic program, it is closely related to three representative existing PLP approaches: the context-sensitive PLP developed by Haddawy and Ngo [20], Bayesian logic programming proposed by Kersting and Raedt [17], and the time parameter-based approach presented by Glesner and Koller [13]. In this section, we make a detailed comparison of our work with the three closely related approaches.

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 $<PD, PB, CB, CR>$, where

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

- PB consists of *probabilistic rules* of the form

$$P(A_0|A_1, ..., A_l) = \alpha \leftarrow B_1, ..., B_m, \neg C_1, ..., \neg C_n \tag{11}$$

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

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

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

PKB and BKB have the following important differences.

First, probabilistic rules of form (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 and is carried out by applying SLDNF-resolution. But in BKB, logic inferences are based on the well-founded semantics and are performed by applying SLG-resolution. The well-founded semantics resolves the problem of inconsistency with the program completion semantics,

while SLG-resolution eliminates the problem of infinite loops with SLDNF-resolution. Note that the key significance of BKB using the well-founded semantics lies in the fact that a unique set of influence clauses can be derived, which lays a basis on which both the declarative and procedural semantics for BKB are developed.

Third, most importantly PKB has no mechanism for handling cyclic influences. In PKB, cyclic influences are defined to be *inconsistent* (see Definition 9 of the paper [20]) and thus are excluded (PKB excludes cyclic influences by requiring its programs be acyclic). In BKB, however, cyclic influences are interpreted as feedbacks, thus implying a time sequence. This allows us to derive a stationary DBN from a logic program with recursive loops.

Recently, Fierens, Blockeel, Ramon and Bruynooghe [11] introduced *logical Bayesian networks* (LBN). LBN is similar to PKB except that it separates logical and probabilistic information. That is, LBN converts rules of form (11) into the form

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

where the A_is are p-predicates with the last argument V removed, and the B_js and C_ks are c-predicates defined in CB. This is not a standard clause of form (3) as defined in logic programming [18]. Like PKB, LBN differs from BKB in the following: (1) it has no mechanism for handling cyclic influences (see Section 3.2 of the paper [11]), and (2) although the well-founded semantics is also used for the logic contexts, neither declarative nor procedural semantics for LBN has been formally developed.

6.2 Comparison with Bayesian Logic Programming

Building on Ngo and Haddawy's work, Kersting and De Raedt [17] introduce the framework of Bayesian logic programs. A *Bayesian logic program* (BLP) is a triple $<P, T_x, CR>$ where P is a well-defined logic program, T_x consists of CPTs associated with each clause in P, and CR is a combination rule. A distinct feature of BLP over PKB is its separation of probabilistic information (T_x) from logic clauses (P). According to [17], we understand that a *well-defined* logic program is an acyclic positive logic program satisfying the range restriction.[9] For instance, a logic program containing clauses like $r(X) \leftarrow r(X)$ (cyclic) or $r(X) \leftarrow s(Y)$ (not range-restricted) is not well-defined. BLP relies on the least Herbrand model semantics and applies SLD-resolution to make backward-chaining inferences.

BLP has two important differences from 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 does not allow cyclic influences. BKB can be viewed as an extension of BLP with mechanisms for handling contexts and cyclic influences in terms of the well-founded semantics. Such an extension is clearly nontrivial.

[9] 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.

6.3 Comparison with the Time Parameter-Based Approach

The time parameter-based framework (TPF) proposed by Glesner and Koller [13] is also a triple $<P, T_x, CR>$, where CR is a combination rule, T_x is a set of CPTs that are represented as decision trees, and P is a logic program with the property that each predicate contains a time parameter and that in each clause the time argument in the head is at least one time step later than the time arguments in the body. This framework is implemented in Prolog, i.e. clauses are represented as Prolog rules and goals are evaluated applying SLDNF-resolution. Glesner and Koller [13] state: "... In principle, this free variable Y can be instantiated with every domain element. (This is the approach taken in our implementation.)" By this we understand that they consider typed logic programs with finite domains.

We observe the following major 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 non-temporal 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 in B is at least one time step earlier than that in A), while BKB uses time slices (implied by recursive loops of form (1)) to model cycles of direct influences (feedbacks). Time-steps based frameworks like TPF are suitable to model flexible DBNs, whereas time-slices based approaches like BKB apply to stationary DBNs. 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 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 $N = 0, 1, 2, ...$ 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 [2]. Since the loop problem is undecidable in general, there is no algorithm for automatically determining such a depth-bound M (rep. a time argument N) for an arbitrary query $p(.)$ [2, 27, 28]. We further illustrate this claim using the following example.

Example 6.1 The following logic program defines a *path* relation; i.e. 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 $N < 100$.

7 Conclusions and Discussion

We have developed a novel theoretical framework for deriving a stationary DBN from a logic program with recursive loops. We observed that recursive loops in a logic program imply a time sequence and thus can be used to model a stationary DBN without using explicit time parameters. We introduced a Bayesian knowledge base with logic clauses of form (4). These logic clauses naturally integrate the direct influence information, the context and the

type constraints, and are evaluated under the well-founded semantics. We established a declarative semantics for a Bayesian knowledge base and developed algorithms that build a two-slice DBN from a Bayesian knowledge base.

We emphasize the following three points.

1. Recursive loops (cyclic influences) and recursion through negation are unavoidable in modeling real-world domains, thus the well-founded semantics together with its top-down inference procedures is well suitable for the PLP application.

2. 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, 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. Firstly, such a method transforms the original non-temporal 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 variable and the original program defines a regular Bayesian network). Secondly, it relies on time steps to define (individual) direct influences, but recursive loops need time slices (intervals) to model cycles of direct influences (feedbacks). Finally, to pose a query over the transformed program, an appropriate time parameter must be specified. As illustrated in Example 6.1, there is no algorithm for automatically determining such a time parameter for an arbitrary query.

Promising future work includes (1) developing algorithms for learning BKB clauses together with their CPTs from data and (2) applying BKB to model large real-world problems. We intend to build a large Bayesian knowledge base for traditional Chinese medicine, where we already have both a large volume of collected diagnostic rules and a massive repository of diagnostic cases.

[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.

Acknowledgements

We are grateful to several anonymous referees for their constructive comments, which greatly helped us improve the presentation.

References

[1] K. R. Apt and M. Bezem, Acyclic programs, *New Generation Computing* 29(3):335-363 (1991).

[2] R. N. Bol, K. R. Apt and J. W. Klop, An analysis of loop checking mechanisms for logic programs, *Theoretical Computer Science* 86(1):35-79 (1991).

[3] F. Bacchus, Using first-order probability logic for the construction of Bayesian networks, in: *Proc. of the Ninth Conference on Uncertainty in Artificial Intelligence*, 1994, pp. 219-226.

[4] J. S. Breese, Construction of belief and decision networks, *Computational Intelligence* 8(4):624-647 (1992).

[5] W. D. Chen, T. Swift and D. S. Warren, Efficient top-down computation of queries under the well-founded semantics, *Journal of Logic Programming* 24(3):161-199 (1995).

[6] W. D. Chen and D. S. Warren, Tabled evaluation with delaying for general logic programs, *J. ACM* 43(1):20-74 (1996).

[7] J. Cussens, Stochastic logic programs: sampling, inference and applications, in: *Proc. of The Sixteenth Annual Conference on Uncertainty in Artificial Intelligence*, 2000, pp. 115-122.

[8] L. De Raedt and K. Kersting, Probabilistic logic learning, *SIGKDD Explorations* 5(1):31-48 (2003).

[9] P. Domingos and M. Richardson, Markov logic: a unifying framework for statistical relational learning, in: *Proc. of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields* , Banff, Canada, 2004, pp. 49-54.

[10] I. Fabian and D. A. Lambert, First-order Bayesian reasoning. In: *Proc. of the 11th Australian Joint Conference on Artificial Intelligence*, number 1502 in LNAI. Springer, 1998, pp. 131-142.

[11] D. Fierens, H. Blockeel, J. Ramon and M. Bruynooghe, Logical Bayesian networks, in: *3rd Workshop on Multi-Relational Data Mining*, Seattle, USA, 2005

[12] L. Getoor, *Learning Statistical Models from Relational Data*, Ph.D. thesis, Stanford University, 2001.

[13] S. Glesner and D. Koller, Constructing flexible dynamic belief networks from first-order probabilistic knowledge bases, in: C. Froidevaux and J. Kohlas, eds., *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning under Uncertainty*, Fribourg, Switzerland, July 1995, pages 217-226.

[14] R. Goldman and E. Charniak, A language for construction of belief networks, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15(3):196-208 (1993).

[15] M. Jaeger, Relational Bayesian networks, in: *Proc. of The Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, 1997, pp. 266-273.

[16] K. Kanazawa, D. Koller and S. Russell, Stochastic simulation algorithms for dynamic probabilistic networks, in: *Proc. of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, 1995.

[17] K. Kersting and L. De Raedt, Bayesian logic programs, in: J. Cussens and A. Frisch, eds, *Work-in-Progress Reports of the Tenth International Conference on Inductive Logic Programming*, London,U.K., 2000. (A full version: Technical Report 151, University of Freiburg Institute for Computer Science.)

[18] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, Berlin, 1987.

[19] S. Muggleton, Stochastic logic programs, in: *Advances in Inductive Logic Programming*, IOS Press, 1996.

[20] L. Ngo and P. Haddawy, Answering queries from context-sensitive probabilistic knowledge bases, *Theoretical Computer Science*, 171:147-177 (1997).

[21] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible inference*, Morgan Kaufmann, 1988.

[22] A. Pfeffer and D. Koller, Semantics and inference for recursive probability models, in: *Proc. of the Seventeenth National Conference on Artificial Intelligence*, AAAI Press, 2000, pp.538-544.

[23] D. Poole, Probabilistic Horn abduction and Bayesian networks, *Artificial Intelligence* 64(1):81-129 (1993).

[24] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.

[25] K. Sagonas, T. Swift, D.S. Warren, J. Freire and P. Rao, *The XSB Programmer's Manual (Version 1.8)*. Department of Computer Science, SUNY at Stony Brook. Available from http://www.cs.sunysb.edu∼sbprolog/xsb-page.html.

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

[27] Y. D. Shen, L. Y. Yuan and J. H. You, Loop checks for logic programs with functions, *Theoretical Computer Science* 266(1-2):441-461 (2001).

[28] Y. D. Shen, J. H. You, L. Y. Yuan, S. P. Shen and Q. Yang, A dynamic approach to characterizing termination of general logic programs, *ACM Transactions on Computational Logic* 4(4):417-430 (2003).

[29] Y. D. Shen, J. H. You and L. Y. Yuan, Enhancing global SLS-resolution with loop cutting and tabling mechanisms, *Theoretical Computer Science* 328(3):271-287(2004).

[30] B. Taskar, P. Abeel and D. Koller, Discriminative probabilistic models for relational data, in: *Proc. of the Eighteenth Conf. on Uncertainty in Artificial Intelligence*, Edmonton, Canada, 2002, pp.485-492.

[31] J. D. Ullman, *Database and Knowledge-Base Systems*, vols. I and II, Computer Science Press, 1988.

[32] A. Van Gelder, Negation as failure using tight derivations for general logic programs, *Journal of Logic Programming* 6(1&2):109-133 (1989).

[33] A. Van Gelder, K. Ross, J. Schlipf, The well-founded semantics for general logic programs, *J. ACM* 38(3):620-650 (1991).

[34] M. Vardi, The complexity of relational query languages, in: *ACM Symposium on Theory of Computing*, 1982, pp. 137-146.