The SLGAD Procedure for Inference on Logic Programs with Annotated Disjunctions

Fabrizio Riguzzi

ENDIF, Università di Ferrara, Via Saragat, 1, 44100 Ferrara, Italy.
`fabrizio.riguzzi@unife.it`

Abstract. Logic Programs with Annotated Disjunctions (LPADs) allow to express probabilistic information in logic programming. The semantics of an LPAD is given in terms of well founded models of the normal logic programs obtained by selecting one disjunct from each ground LPAD clause. The paper presents SLGAD resolution that computes the (conditional) probability of a ground query from an LPAD and is based on SLG resolution for normal logic programs. The performances of SLGAD are evaluated on classical benchmarks for normal logic programs under the well founded semantics, namely the stalemate game and the ancestor relation. The results show that SLGAD has good scaling properties and is able to deal with cyclic programs.

Topics: Probabilistic Logic Programming, Well Founded Semantics, Logic Programs with Annotated Disjunctions, SLG resolution.

1 Introduction

The combination of logic and probability is a long standing problem in philosophy and artificial intelligence, dating back to [1]. Recently, the work on this topic has thrived leading to the proposal of novel languages that combine relational and statistical aspects, such as Independent Choice Logic [2], ProbLog [3], Stochastic Logic Programs [4], Bayesian Logic Programs [5], PRISM [6] and CLP($\mathcal{BN}$) [7]. Each of these languages has a different semantics that makes it suitable for different domains: the identification of the best setting for each language is currently under study [8, 9].

When we are reasoning about actions and effects and we have causal independence [10] among different causes for the same effect, Logic Programs with Annotated Disjunctions (LPADs) [11] seem particularly suitable. They extend logic programs by allowing program clauses to be disjunctive and by annotating each atom in the head with a probability. A clause can be causally interpreted in the following way: the truth of the body causes the truth of one of the atoms in the head non-deterministically chosen on the basis of the annotations. The semantics of LPADs is given in terms of well founded models [12] of the normal logic programs obtained by selecting one head for each disjunctive clause.

In order to compute the (conditional) probability of queries, various options are possible. [13] showed that ground acyclic LPADs can be converted to Bayesian networks. However, the conversion requires the complete grounding of the LPAD, thus making the technique impractical.

[13] also showed that acyclic LPADs can be converted to Independent Choice Logic (ICL) programs. Thus inference can be performed by using the Cilog2 system [14]. An algorithm for performing inference directly with LPADs was proposed in [15]. The algorithm, that will be called SLDNFAD in the following, is an extension of SLDNF derivation and uses Binary Decision Diagrams, similarly to what is presented in [3] for the ProbLog language. Both Cilog2 and SLDNFAD are complete and correct for programs for which the Clark's completion semantics [16] and the well founded semantics coincide, as for acyclic [17] and modularly acyclic programs [18].

In this paper we present the SLGAD top down procedure for performing inference with possibly (modularly) cyclic LPADs. SLGAD is based on the SLG procedure [19] for normal logic programs and extends it in a minimal way.

SLGAD is evaluated on classical benchmarks for well founded semantics inference algorithms, namely the stalemate game and the ancestor relation. In both cases, various extensional databases are considered, encoding linear, cyclic or tree-shaped relations. SLGAD is compared with Cilog2 and SLDNFAD on the modularly acyclic programs. The results show that SLGAD, while being more expensive than SLDNFAD on problems where SLDNFAD succeeds, is faster than Cilog2 when the query is true in an exponential number of instances.

The paper is organized as follows. In Section 2 we present the syntax and semantics of LPADs together with some properties of normal programs and of LPADs. Section 3 provides the definition of the SLGAD procedure. In Section 4 we prove the correctness of SLGAD. Section 5 presents the experiments and, finally, Section 6 concludes and presents directions for future work.

2 Preliminaries

A Logic Program with Annotated Disjunctions [11] T consists of a finite set of formulas of the form

$$(H_1 : \alpha_1) \vee (H_2 : \alpha_2) \vee \ldots \vee (H_n : \alpha_n) : -B_1, B_2, \ldots B_m$$

called *annotated disjunctive clauses*. In such a clause the H_i are logical atoms, the B_i are logical literals and the α_i are real numbers in the interval $[0, 1]$ such that $\sum_{i=1}^{n} \alpha_i \leqslant 1$. The head of LPAD clauses implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{i=1}^{n} \alpha_i$.

For a clause C of the form above, we define *head*(C) as $\{(H_i : \alpha_i)|1 \leqslant i \leqslant n\} \cup \{(null : 1 - \sum_{i=1}^{n} \alpha_i)\}$, *body*$(C)$ as $\{B_i|1 \leqslant i \leqslant m\}$, $H_i(C)$ as H_i and $\alpha_i(C)$ as α_i. Let $H_B(T)$ be the Herbrand base of T and let $\mathcal{I}_T$ be the set of all the possible Herbrand interpretations of P (i.e., subsets of $H_B(T)$). If T contains function symbols, then $H_B(T)$ is infinite, otherwise it is finite.

In order to define the semantics of a non-ground T, we must generate the grounding T' of T. Each ground annotated disjunctive clause represents a probabilistic choice between the ground non-disjunctive clauses obtained by selecting only one head. The intuitive interpretation of a ground clause is that the body represents an event that, when it happens (i.e. it becomes true), causes an atom in the head (an effect) to happen (i.e. to become true). If the atom selected is *null*, this is equivalent to having no effect.

The semantics of an LPAD, given in [11], requires the grounding to be finite, so the program must not contain function symbols if it contains variables.

By choosing a head atom for each ground clause of an LPAD we get a normal logic program called an *instance* of the LPAD. A probability distribution is defined over the space of instances by assuming independence between the choices made for each clause.

A *choice* κ is a set of triples (C, θ, i) where $C \in T$, θ is a substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for ground clause $C\theta$, the head $H_i : \alpha_i$ was chosen. A choice κ is *consistent* if $(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, i.e. only one head is selected for a ground clause.

A consistent choice is a *selection* σ if for each clause $C\theta$ in the grounding of T there is a triple (C, θ, i) in σ. We denote the set of all selections σ of a program T by $\mathcal{S}_T$.

A consistent choice κ identifies a normal logic program $T_\kappa = \{(H_i(C) : -body(C))\theta | (C, \theta, i) \in \kappa\}$ that is called a *sub-instance* of T. If σ is a selection, T_σ is called an *instance*. For a consistent choice κ, let $U(\kappa)$ be the set of instances that are supersets of T_κ, i.e., the set of instances T_σ with σ a selection such that $\sigma \supseteq \kappa$.

We now assign a probability to a consistent choice κ. The *probability of a consistent choice* κ is the product of the probabilities of the individual choices made, i.e. $P_\kappa = \prod_{(C,\theta,i)\in\kappa} \alpha_i(C)$. The *probability of instance* T_σ is P_σ.

The semantics of the instances of an LPAD is given by the well founded semantics (WFS) [12]. Given a normal program T, we call $WFM(T)$ its *well founded partial model*. For each instance T_σ, we require that $WFM(T_\sigma)$ is two-valued, since we want to model uncertainty solely by means of disjunctions. An LPAD T is called *sound* iff, for each selection σ in $\mathcal{S}_T$, $WFM(T_\sigma)$ is two-valued. In the following we consider only sound programs.

The probability of an interpretation $I \in \mathcal{I}_T$ according to T is given by the sum of the probabilities of the instances that have I as the well founded model, i.e.

$$P_T(I) = \sum_{\sigma\in\mathcal{S}_T, WFM(T_\sigma)=I} P_\sigma.$$

The probability of a formula χ according to T is given by the sum of the probabilities of interpretations in which the formula is true, i.e.

$$P_T(\chi) = \sum_{I\in\mathcal{I}_T, I\models\chi} P(I).$$

Equivalently, the probability of a formula χ is given by the sum of the probabilities of the instances in which the formula is true according to the WFS:

$$P_T(\chi) = \sum_{T_\sigma \models_{WFS} \chi} P_\sigma$$

LPADs show patterns of *causal independence* [10]: each ground clause with atom a in the head is a potential cause of a that is activated when the body becomes true. Each cause is independent of the other so they combine with the noisy or law [20]. Such a law states that, if there are n causes (represented by binary variables $c_1, \ldots, c_n$) for an effect e (a binary variable) and the probabilities of the causes of happening (i.e. of assuming the value 1) are $p_1, \ldots, p_n$, the probability of happening of the effect (i.e. of assuming the value 1) is given by

$$1 - \prod_{i=1}^{n}(1 - p_i)$$

Example 1. Consider the dependency of a person's itching from him having allergy or measles:

$$\begin{aligned}
C_1 &= \mathit{itching}(X, \mathit{strong}) : 0.3 \vee \mathit{itching}(X, \mathit{moderate}) : 0.5 :-\ \mathit{measles}(X).\\
C_2 &= \mathit{itching}(X, \mathit{strong}) : 0.2 \vee \mathit{itching}(X, \mathit{moderate}) : 0.6 :-\ \mathit{allergy}(X).\\
C_3 &= \mathit{allergy}(\mathit{david}).\\
C_4 &= \mathit{measles}(\mathit{david}).
\end{aligned}$$

This program models the fact that itching can be caused by allergy or measles. Measles causes strong itching with probability 0.3, moderate itching with probability 0.5 and no itching with probability $1 - 0.3 - 0.5 = 0.2$; allergy causes strong itching with probability 0.2, moderate itching with probability 0.6 and no itching with probability $1 - 0.2 - 0.6 = 0.2$.

If only one cause happens, the probability of the effect is given by the parameter in the head. If more than one cause happens, the probability of the effect is given by the noisy or relation. For example, $\mathit{itching}(\mathit{david}, \mathit{strong})$ is true in 5 of the 9 instances of the program and its probability is

$$0.3 \cdot 0.2 + 0.3 \cdot 0.6 + 0.3 \cdot 0.2 + 0.5 \cdot 0.2 + 0.2 \cdot 0.2 = 0.44$$

If we compute the probability by noisy or we get $1 - (1 - 0.3) \cdot (1 - 0.2) = 0.44$.

Example 2. Consider a probabilistic game in which a position is winning with a certain probability if there is a move to another position that is losing for the opponent. This game can be represented by

$$\begin{aligned}
\mathit{win}(X, \mathit{white}) : 0.8 &:-\ \mathit{move}(X, Y), \neg \mathit{win}(Y, \mathit{black}).\\
\mathit{win}(X, \mathit{balck}) : 0.8 &:-\ \mathit{move}(X, Y), \neg \mathit{win}(Y, \mathit{white}).
\end{aligned}$$

plus facts for the *move* predicate.

If *move* is acyclic, then the program is sound. Otherwise, there are instances that do not have a total well founded model, namely those where the *win* head is selected for all the ground clauses whose instance of the $move(X, Y)$ atom is in the cycle.

Let us now see other properties of LPADs. For a consistent choice κ, P_κ is the sum of the probability of the instances that are supersets of T_κ, i.e.

$$P_\kappa = \sum_{\sigma \in \mathcal{S}_T, \sigma \supseteq \kappa} P_\sigma$$

Two consistent choices κ_1 and κ_2 are incompatible if there exists a couple (C, θ) such that $(C, \theta, i) \in \kappa_1$, $(C, \theta, j) \in \kappa_2$ and $i \neq j$. In this case $U(\kappa_1)$ and $U(\kappa_2)$ are disjoint, so

$$\sum_{T_\sigma \in U(\kappa_1) \cup U(\kappa_2)} P_\sigma = P_{\kappa_1} + P_{\kappa_2}$$

Two important properties of normal logic program are acyclicity and modular acyclicity, we refer to [17] and [18] respectively for the definitions. For acyclic and modularly acyclic programs the least Herbrand model of the Clark's completion and the well founded partial model coincide, so queries can be answered in the WFS by means of SLDNF.

An LPAD is (modularly) acyclic if all of its instances are (modularly) acyclic.

An LPAD is *range restricted* if all the variables appearing in the head of clauses also appear in the body.

The notion of relevance is adapted from the one given in [21] for extended logic programs: a clause C is *directly relevant* to a ground goal Q if there exists an atom A that appears in a literal of Q and is in the head of a grounding of C. A clause C is *relevant* to a ground goal Q if it is directly relevant to Q or if there exists a clause C' that has A in the head of a grounding $C'\theta$ of C' and C is relevant to $body(C'\theta)$.

[21] showed that the WFSX semantics for extended logic programs (and thus also the WFS for normal logic programs) has the property of *relevance*: given a normal program T, the truth of ground atom A in $WFM(T)$ does not change if we add to T clauses that are not relevant to A.

3 SLGAD Resolution Algorithm

In this section we present *Linear resolution with Selection function for General logic programs with Annotated Disjunctions* (SLGAD) that extends SLG resolution [22, 19] for dealing with LPADs.

In the following we give a brief sketch of the implementation of SLG resolution as presented in [19]. SLG builds a search forest for a subgoal (i.e. a (partially instantiated) atom) by performing depth first search. Besides a stack $\mathcal{S}$ of subgoals, SLG keeps a table $\mathcal{T}$ in which it stores, for each subgoal A considered in the computation, the set of answers (i.e. instantiations of the subgoal) already computed $Anss$, the set of resolvents that wait for new answers for A, separated into a set $Poss$ that has A selected and a set $Negs$ that has $\neg A$ selected, and a boolean flag $Comp$ that indicates whether A has been completely evaluated. After every resolution step for a subgoal A, SLG tests whether all possible answers for A have been computed: if so, it sets $Comp$ to true. When it encounters a case of a possible loop through negation, it "delays" the selected literal $\neg A$ by inserting it into a dedicated data structure of the resolvent. Delayed literals are then removed if they turn out to be true.

SLG uses X-clauses to represent resolvents with delayed literals.

Definition 1 (X-Clause). *An* X-clause *X is a clause of the form $A : -D|B$ where A is an atom, D is a sequence of ground negative literals and (possibly unground) atoms and B is a sequence of literals. Literals in D are called* delayed literals*. If B is empty, an X-clause is called an* X-answer *clause.*

An ordinary program clause is seen as a X-clause with an empty set of delayed literals.

SLG is based on the operation of SLG resolution and SLG factoring on X-clauses. In particular, SLG resolution is performed between an X-clause $A : -|A$ and a program clause or between an X-clause and an X-answer.

In SLGAD, X-clauses are replaced by XD-clauses.

Definition 2 (XD-Clause). *An* XD-clause *G is a quadruple (X, C, θ, i) where X is an X-clause, C is a clause of T, θ is a substitution for the variables of C and $i \in \{1, \ldots, |head(C)|\}$. Let X be $A : -D|B$: if B is empty, the XD-clause is called an* XD-answer *clause.*

In SLGAD, SLG resolution between an X-clause $A : -|A$ and a program clause is replaced by SLGAD goal resolution and SLG resolution between an X-clause and an X-answer is replaced by SLGAD answer resolution.

Definition 3 (SLGAD Goal Resolution). *Let A be a subgoal and let C be a clause of T such that A is unifiable with an atom H_i in the head of C. Let C' be a variant of C with variables renamed so that A and C' have no variables in common. We say that A is* SLGAD goal resolvable *with C and the XD-clause*

$$((A : -|body(C'))\theta, C', \theta, i)$$

is the SLGAD goal resolvent *of A with C on head H_i, where θ is the most general unifier of A and H'_i. C' is kept in the resolvent because we must be able to recover the ground program clause to which the XD-clause refers, namely $C'\theta$.*

Definition 4 (SLGAD Answer Resolution). *Let G be a XD clause $(A : -D|L_1, \ldots, L_n, C, \theta, i)$ where $n > 0$ and L_j be the selected atom. Let F be an XD-answer clause with an empty set of delayed literals, and let F', of the form $(A' : -|, E', \theta', i')$, be a variant of F with variables renamed so that G and F'*

have no variables in common. If L_j and A' are unifiable then we say that G is SLGAD answer resolvable *with F and the XD-clause*

$$((A : -D|L_1, \dots, L_{j-1}, L_{j+1}, \dots, L_n)\delta, C, \theta\delta, i)$$

is the SLGAD answer resolvent *of G with F, where δ is the most general unifier of A' and L_j.*

SLG factoring is replaced by SLGAD factoring.

Definition 5 (SLGAD Factoring). *Let G be a XD-clause $(A : -D|L_1, \dots, L_n, C, \theta, i)$ where $n > 0$ and L_j be the selected atom. Let F be an XD-answer clause, and let F', of the form $(A' : -D'|, E', \theta', i')$, be a variant of F with variables renamed so that G and F' have no variable in common. If D' is not empty and L_j and A' are unifiable then the* SLGAD factor *of G with F is*

$$((A : -D, L_j|L_1, \dots, L_{j-1}, L_{j+1}, \dots, L_n)\delta, C, \theta\delta, i)$$

where δ is the most general unifier of A' and L_j.

SLGAD goal resolution, SLGAD answer resolution and SLGAD factoring are equivalent to their SLG counterparts on the underlying X-clauses.

The SLGAD algorithm is defined with a procedural pseudo code that contains a non-deterministic choice point. The main function of the algorithm is shown in Figure 1. It takes as input a ground atom A and a program T and it keeps four global variables. The first three are shared with SLG: the table $\mathcal{T}$, the stack of subgoals $\mathcal{S}$ and the counter Count. The fourth variable is specific of SLGAD and is used to record all the clauses used in the SLGAD derivation together with the head selected: it is a choice κ, i.e., a set of triples (C, θ, i) where C is a clause of T, θ is a substitution that grounds C and i is the index of an atom in the head of C. We assume that the global variables are copied in different branches of the

Fig. 1. Procedure SLGAD

```
function SLGAD($A$,$T$)
begin
    Initialize Count, $\mathcal{T}$, $\mathcal{S}$, DFN, PosMin and NegMin as in SLG;
    $\kappa := \emptyset$;
    let $\psi$ be the set of all the values for $\kappa$ after an execution of
    SLG_SUBGOAL(A,PosMin,NegMin) such that $\mathcal{T}$ contains $A$ as an answer;
    return $\sum_{\kappa \in \psi} P_\kappa$
end
```

search tree generated by the choice points, so that a modification in a branch does not influence the other branches. The search tree is explored depth first.

The SLGAD algorithm modifies in a minimal way SLG: it is composed of the same procedures as SLG [19], plus procedure ADD_CLAUSE. We refer to

[19] for a detailed description of the individual SLG procedures, here we report only the differences, that are indicated in italics in the figures. Procedure SLG_SUBGOAL (see Figure 2) differs from that of SLG because in line 3 each SLGAD goal resolvent is considered rather than each SLG resolvent. Procedure SLG_NEWCLAUSE (see Figure 2) performs resolution on the selected positive or negative literal in the body of the clause or adds an answer if the body is empty. SLG_NEWCLAUSE is the same as in SLG with X-clauses replaced by XD-clauses. The main difference is in procedure SLG_ANSWER (see Figure 3) where a call to ADD_CLAUSE is added in line 4.

Fig. 2. Procedures SLG_SUBGOAL and SLG_NEWCLAUSE

```
1  procedure SLG_SUBGOAL($A$,PosMin,NegMin)
2  begin
3      for each  SLGAD goal resolvent $G$ of $A$ with some clause $C \in T$
4          on the head $H_i$ do begin
5          SLG_NEWCLAUSE($A, G$,PosMin,NegMin);
6      end;
7      SLG_COMPLETE($A$,PosMin,NegMin,$\kappa$);
8  end;
9
10 procedure SLG_NEWCLAUSE($A, G$,PosMin,NegMin)
11 begin
12     if $G$ has no body literals on the right of $|$ then
13         SLG_ANSWER($A, G$,PosMin,NegMin)
14     else if $G$ has a selected atom $B$
15         SLG_POSITIVE($A, G, B$,PosMin,NegMin)
16     else if $G$ has a selected ground negative literal $\neg B$
17         SLG_NEGATIVE($A, G, B$,PosMin,NegMin)
18     else begin /* $G$ has a selected non-ground negative literal */
19         halt with an error message
20     end
```

If the answer G is not subsumed by an answer already present in the table, ADD_CLAUSE is called (see Figure 4) that modifies κ and returns a value to SLG_ANSWER in the variable Derivable. If $G = (X, C, \theta, i)$[1], ADD_CLAUSE adds a new triple (C, θ, j) to the current κ set . If the program is range restricted, $C\theta$ is ground, see Lemma 3. ADD_CLAUSE first checks whether the clause $C\theta$ already appears in the current choice with a head index different from i: if so, it fails the derivation. Otherwise, it non-deterministically selects a head

[1] C is the clause of the program from which the XD-clause G was obtained by SLGAD goal resolution, i is the index of the head used in the goal resolution and θ is the composition of the substitutions of all the derivations and factorings performed on G

Fig. 3. Procdure SLG_ANSWER

```
procedure SLG_ANSWER($A, G$,PosMin,NegMin)
begin
    if $G$ is not subsumed by any answer in $Anss(A)$ in $\mathcal{T}$ then begin
        ADD_CLAUSE($G$,Derivable);
        if Derivable then begin
            insert $G$ into $Anss(A)$;
            if $G$ has no delayed literals then begin;
                reset $Negs(A)$ to empty;
                let $L$ be the list of all pairs $(B, H')$, where $(B, H) \in Poss(A)$ and
                    $H$ is the SLGAD answer resolvent of $H$ with $G$;
                for each $(B, H')$ in $L$ do begin
                    SLG_NEWCLAUSE(B,H',PosMin,NegMin);
                end;
            end else begin /* $G$ has a non empty delay */
                if no other answer in $Anss(A)$ has the same head as $G$ does then
                begin
                    let $L$ be the list of all pairs $(B, H')$, where $(B, H) \in Poss(A)$
                        and $H$ is the SLGAD factor of $H$ with $G$;
                    for each $(B, H')$ in $L$ do begin
                        SLG_NEWCLAUSE($B, H'$,PosMin,NegMin);
                    end;
                end;
            end;
        end;
    end;
end;
```

index j from $\{1, \ldots, |head(C)|\}$: if $j = i$ this means that the subgoal in the head is derivable in the sub-instance represented by κ, so it sets Derivable to true. If $j \neq i$, then Derivable is set to false. In backtracking, all elements of $\{1, \ldots, |head(C)|\}$ are selected.

Fig. 4. Procedure ADD_CLAUSE

```
procedure ADD_CLAUSE($G$,Derivable)
begin
    let $G$ be $(X, C, \theta, i)$;
    if $\exists k : (C, \theta, k) \in \kappa, k \neq i$ then begin
        fail;
    end else begin
        choose an index $j$ from $\{1, \ldots, |head(C)|\}$ (choice point);
        if $i = j$ then begin
            Derivable:= true;
        end else begin
            Derivable:= false;
        end
        $\kappa := \kappa \cup \{(C, \theta, j)\}$;
    end
end
```

Since every clause relevant to a subgoal is eventually reduced to an XD-answer, it is sufficient to update κ only in SLG_ANSWER by means of ADD_CLAUSE. The cases where $j \neq i$ are necessary because we must consider also the possibility that the subgoal A is derived not using head i of clause $C\theta$. It may be that A could be derived in a possibility $j \neq i$ using other clauses and/or that the possibility j is used to derive a subgoal necessary for this second derivation branch: if we do not consider these possibilities we could miss some explanations for A.

SLG_ANSWER then behaves differently depending on the value of Derivable: if it is true, a new answer has been found so the rest of the code of the SLG_ANSWER procedure of SLG is performed with X-clauses replaced by XD-clauses, otherwise it exits without modifying the global variables.

Procedure SLG_POSITIVE, that performs resolution on a positive literal, (see Figure 5) modifies the one of SLG by replacing SLG resolution with SLGAD answer resolution and SLG factoring with SLGAD factoring (see the instructions in italics in the figure). The other SLG procedure are modified simply by replacing X-clauses with XD-clauses.

Example 3. Let us consider the behaviour of the procedure for the query $A_1 = itching(david, strong)$ from the program of example 1. SLGAD is called with A_1 as the subgoal. SLGAD initializes the table by adding the entry

$$t_1 = (itching(david, strong), \{\}, [], [], false)$$

Fig. 5. Procdure SLG_POSITIVB

```
procedure SLG_POSITIVE($A, G, B$,PosMin,NegMin)
begin
    if $B$ is not in table $\mathcal{T}$ then begin
        insert $(B, \{\}, [(A, G)], [], \mathit{false})$ into $\mathcal{T}$;
        DFN:= Count; PosLink:=Count; NegLink:=maxint;
        push ($B$,DFN,PosLink,NegLink) onto stack $\mathcal{S}$
        Count:=Count+1;
        BPosMin:=DFN; BNegMin:=maxint;
        SLG_SUBGOAL($B$,BPosMin,BNegMin);
        UPDATE_SOLUTION($A, B$,pos,PosMin,NegMin,BPosMin,BNegMin);
    end else begin
        if $Comp(B)$ is not true then begin
            insert $(A, G)$ into $Poss(B)$;
            UPDATE_LOOKUP($A, B$,pos,PosMin,NegMin);
        end;
        let $L$ be the empty list;
        for each atom $B$ in the head of some answer in $Anss(B)$ do begin
            if $B' : -| \in Anss(B)$ then begin
                lef $G'$ be the $SLGAD$ $answer$ $resolvent$ of $G$ with $B' : -|$;
                insert $(A, G')$ into $L$;
            end else begin
                let $H \in Anss(B)$ with head atom $B'$;
                let $G'$ be the $SLGAD$ $factor$ of $G$ with $H$;
                insert $(A, G')$ into $L$;
            end;
        end;
        for each $(A, G')$ in $L$ do begin
            SLG_NEWCLAUSE($A, G'$,PosMin,NegMin);
        end;
    end;
end
```

and the stack by pushing the entry

$$s_1 = (itching(david, strong), 1, 1, maxint).$$

In SLG_SUBGOAL (call Q_1) two SLGAD goal resolvents of A_1 with the clauses of the program are found, namely

$$G_1 = (itching(david, strong) : -|measles(david), C_1, \{X/david\}, 1)$$

$$G_2 = (itching(david, strong) : -|allergy(david), C_2, \{X/david\}, 1)$$

SLG_NEWCLAUSE (call Q_2) is called first with A_1 as the subgoal and G_1 as the XD-clause. Since G_1 has a non empty body, SLG_POSITIVE (call Q_3) is invoked. The selected literal $measles(david)$ is not in the table so

$$t_2 = (measles(david), \{\}, [(A_1, G_1)], [], false)$$

is inserted into $\mathcal{T}$. Then

$$s_1 = (measles(david), 2, 2, \text{maxint})$$

is pushed onto the statck. SLG_SUBGOAL (call Q_4) is called with subgoal $A_2 = measles(david)$. One SLGAD goal resolvent of A_2 with clauses of the program is found, namely

$$G_3 = (measles(david) : -|, C_3, \{\}, 1)$$

SLG_NEWCLAUSE is invoked and, since the body of the XD-clause is empty (G_3 is an sntswer), SLG_ANSWER (call Q_5) is called. Since no answer for A_2 is in the table, ADD_CLAUSE is invoked but it does not generate choice points because C_3 is deterministic so it returns setting Derviable to true.

The answer $measles(david) : -|$ is added to the set of answers of A_2 in table entry t_2 that becomes

$$t_2 = (measles(david), \{\}, [(A_1, G_1)], [measles(david) : -|], false).$$

G_1 has no delayed literals, so SLGAD answer resolution is performed between G_1 and G_3 obtaining

$$G_4 = (itching(david, strong) : -|, C_1, \{X/david\}, 1)$$

SLG_NEWCLAUSE (call Q_6) is then called with $A = A_1$ and $G = G_4$.

Since G_4 is an answer, SLG_ANSWER (call Q_7) is invoked. No answer for A_1 is present in the table so ADD_CLAUSE is called. This procedure generates three derivation branches B_1, B_2 and B_3:

- in B_1, $\kappa = \{(C_1, \{X/david\}, 1)\}$ and Derivable=true,
- in B_2, $\kappa = \{(C_1, \{X/david\}, 2)\}$ and Derivable=false and
- in B_3, $\kappa = \{(C_1, \{X/david\}, 3)\}$ and Derivable=false.

Let us now consider derivation branch B_1: $itching(david, strong) : -|$ is added to the set of answers for A_1. Since no subgoal depends positively or negatively on A_1, SLG_ANSWER (call Q_7) returns, SLG_NEWCLAUSE (call Q_6) returns and SLG_COMPLETE is called that determines that $itching(david, strong)$ cannot be marked as completed yet.

The second SLGAD goal resolvent G_2 of A_1 is considered in call Q_1 of SLG_SUBGOAL. SLG_NEWCLAUSE (call Q_8) is invoked with $A = A_1$ and $G = G_2$. Since G_2 has a non- empty body, SLG_POSITIVE (call Q_9) is called that inserts

$$t_3 = (allergy(david), \{\}, [(A_1, G_2)], [], false)$$

into the table and invokes SLG_SUBGOAL (call Q_{10}) with subgoal $A_3 =$*allergy (david)*. As for $measles(david)$, this results in the addition to the table of the answer

$$allergy(david) : -|$$

and in SLG_ANSWER SLGAD answer resolution is performed obtaining

$$G_5 = (itching(david, strong) : -|, C_2, \{X/david\}, 1)$$

This is an answer, SLG_NEWCLAUSE and SLG_ANSWER are called but, since $itching(david, strong) : -|$ is already present in the table, they return without modifying the table. So call Q_1 returns and we have one successful derivation with

$$\kappa_1 = \{(C_1, \{X/david\}, 1)\}.$$

For derivation branch B_2 Derivable is false, so SLG_ANSWER (call Q_7) exits without modifying the table, SLG_NEWCLAUSE (call Q_6) returns and SLG_COMPLETE determines that $itching(david, strong)$ cannot be marked as completed yet.

Now the second SLGAD goal resolvent G_2 of A_1 is considered in call Q_1 of SLG_SUBGOAL. SLG_NEWCLAUSE (call Q_{11}) and SLG_POSITIVE (call Q_{12}) are invoked: the latter inserts

$$t_3 = (allergy(david), \{\}, [(A_1, G_2)], [], false)$$

into the table and calls SLG_SUBGOAL (call Q_{13}) with subgoal

$$A_3 = allergy(david).$$

As before the answer $allergy(david) : -|$ is added to the table and in SLG_ANSW-ER (call Q_{14}) SLGAD answer resolution is performed obtaining

$$G_6 = (itching(david, strong) : -|, C_2, \{X/david\}, 1)$$

SLG_NEWCLAUSE is invoked with $A = A_1$ and $G = G_6$ (call Q_{15}). This is an answer not present in the table, SLG_ANSWER (call Q_{16}) and, in turn, ADD_CLAUSE are called. The latter procedure generates three computation branches $B_{2,1}, B_{2,2}$ and $B_{2,3}$:

- $B_{2,1}$ with $\kappa = \{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 1)\}$ and Derivable=true,
- $B_{2,2}$ with $\kappa = \{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 2)\}$ and Derivable=false and
- $B_{2,3}$ with $\kappa = \{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 3)\}$ and Derivable=false.

In derivation branch $B_{2,1}$: $itching(david, strong) : -|$ is added to the set of answers for A_1. Since no subgoal depends positively or negatively on A_1, SLG_ANSWER (call Q_{16}) returns, SLG_NEWCLAUSE returns (call Q_{15}) and SLG_COMPLETE is called that establish that $itching(david, strong)$ can be completed.

So SLG_SUBGOAL (call Q_1) exits and we have one successful derivation with

$$\kappa_2 = \{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 1)\}.$$

Similarly, for derivation branches B_3 we get the branhces

- $B_{3,1}$ with $\kappa = \{(C_1, \{X/david\}, 3), (C_2, \{X/david\}, 1)\}$ and Derivable=true,
- $B_{3,2}$ with $\kappa = \{(C_1, \{X/david\}, 3), (C_2, \{X/david\}, 2)\}$ and Derivable=false and
- $B_{3,3}$ with $\kappa = \{(C_1, \{X/david\}, 3), (C_2, \{X/david\}, 3)\}$ and Derivable=false.

From these, only $B_{3,1}$ leads to a success with

$$\kappa_3 = \{(C_1, \{X/david\}, 3), (C_2, \{X/david\}, 1)\}.$$

So, overall the probability of $itching(david, strong)$ is

$$0.3 + 0.5 \cdot 0.2 + 0.4 \cdot 0.3 = 0.44$$

.

If the conditional probability of a ground atom A given another ground atom E must be computed, rather then computing $P_T(A \wedge E)$ and $P_T(E)$ separately, an optimization can be done: we first identify the choices for all successful derivations for E and then we look for the choices for the successful derivations of A starting from a choice of E, as shown in Figure 6.

4 Proof of Correctness

The proof of the soundness and completeness of SLDAG with respect to the LPAD semantics is based on the theorem of partial correctness of SLG [22, 23]: SLG is sound and complete given an arbitrary but fixed computation rule when it does not flounders. The truth of computed answers with respect to the well founded partial model can be expressed by the following corollary of Theorem 5.5 of [23]:

Corollary 1. *Let T be a normal logic programs, R an arbitrary but fixed computation rule, A a ground subgoal, $\mathcal{T}$ the table that is built by procedure SLG (Figure 14 in [19]) for query atom A and program T and suppose that no floundering occurs, then:*

Fig. 6. Procedure SLGAD_COND

```
procedure SLGAD_COND($A$, $E$,$T$)
begin
    Initialize Count, $\mathcal{T}$, $\mathcal{S}$, DFN, PosMin and NegMin as in SLG;
    $\kappa := \emptyset$;
    let $\psi_E$ be the set of all the values for $\kappa$ after a successful call of
    SLG_SUBGOAL(E,PosMin,NegMin) such that $\mathcal{T}$ contains $A$ as an answer;
    if $\sum_{\kappa \in \psi_E} P_\kappa = 0$ then
        return undefined
    else begin
        Initialize Count, $\mathcal{T}$, $\mathcal{S}$, DFN, PosMin and NegMin as in SLG;
        let $\psi_A$ be the set of all the values of $\kappa$ after a successful execution of
        begin
            pick a choice $\kappa'$ from $\psi_E$;
            $\kappa = \kappa'$;
            SLG_SUBGOAL(E,PosMin,NegMin);
            $\mathcal{T}$ contains $A$ as an answer;
        end
        return $P(A|E) = \frac{\sum_{\kappa' \in \psi_A} P_{\kappa'}}{\sum_{\kappa \in \psi_E} P_\kappa}$
    end
end
```

– *if $\mathcal{T}$ contains an answer X-clause that has A in the head and an empty set of delayed literals then A is true in $WFM(P)$; if $\mathcal{T}$ does not contain an answer X-clause that has A in the head then A is false in $WFM(T)$; otherwise A is undefined;*
– *if A is true in $WFM(T)$, then there exists in $\mathcal{T}$ an answer X-clause with an empty set of delayed literals; if A is false in $WFM(P)$ then there does not exist in $\mathcal{T}$ an answer X-clause with A in the head; if A is undefined in $WFM(T)$ then $\mathcal{T}$ contains at least one answer X-clause with A in the head and all answer X-clauses for A have a non-empty set of delayed literals.*

Lemma 1. *If $G = (A : -D|B, C, \theta, i)$ appears anywhere in $\mathcal{T}$, the variables appearing in $C\theta$ are those appearing in A:-D—B.*

Proof. We will prove the lemma by induction on the sequence of operations on XD-clauses. G is inserted into a table by SLGAD goal resolution: for the definition of the operation the lemma holds.

SLGAD answer resolution and SLGAD factoring keep the property. □

Lemma 2. *If T is range restricted, all XD-answers in the table $\mathcal{T}$ after a call to SLGAD are ground.*

Proof. The lemma can be proved by induction on the number of answers added to the table. □

Lemma 3. *If T is range restricted and (C, θ, i) belongs to choice κ after a successful execution of SLG_SUBGOAL (Figure 2), then $C\theta$ is ground.*

Proof. Each triple (C, θ, i) is inserted into κ only in SLG_ANSWER called with an answer XD-clause $G = (A : -D|, C, \theta, i)$. Since $A : -D|$ is ground, by Lemma 1 $C\theta$ is ground. □

Theorem 1. *If T is sound and range restricted, A is ground and no floundering occurs in the call of SLGAD(A, T), then SLGAD(A, T) returns $P_T(A)$.*

Proof (Sketch). Let $\mathcal{T}$ and κ be the values of the table and of the choice after the execution of SGL_SUBGOAL. Then κ is a consistent choice, since ADD_CLAUSE adds a triple (C, θ, i) only if (C, θ, j) with $j \neq i$ is not present in κ.

SLG_SUBGOAL considers all the clauses that are directly relevant to the subgoal A and it may call SLG_NEWCLAUSE with XD-clauses that are inconsistent with each other or with the current choice. Thus, the sets of resolvents waiting for answers for a subgoal in the table $\mathcal{T}$ may refer to incompatible choices. However, the set of answers is updated only by SLG_ANSWER, that adds an element to κ only if the answer is not already present in the table and if κ is consistent with the previous choices.

Since resolutions and factorings are performed only with program clauses or answers, given a successful execution of SLGAD, a successful execution of SLG can be identified that contains a subset of the steps performed by SLGAD and that uses program clauses only from T_κ. Thus, if A appears in the set of answers of $\mathcal{T}$ it can be derived by SLG in T_κ, .

Given that SLG is correct and T is sound, if A appears as an answer in $\mathcal{T}$ then it does not have delayed literals and so is true in the well founded model of T_κ. Since the addition of other clauses to T_κ can not alter the truth of A then A is true also in all the instances of $U(\kappa)$.

Since SLG is complete and SLG_SUBGOAL considers all the clauses that are directly relevant to the subgoal, if there exists a sub-instance T_δ such that A is derivable by SLG in T_δ, in backtracking SLGAD will find it.

Each consistent choice returned by SLGAD is incompatible with the others because ADD_CLAUSE considers a ground clause only once and generates a different search branch for each head. Therefore,

$$P_T(A) \sum_{\sigma \in \mathcal{S}_T, T_\sigma \models A} P_\sigma = \sum_{\sigma \in \mathcal{S}_T, \sigma \supseteq \kappa, \kappa \in \psi} P_\sigma = \sum_{\kappa \in \psi} P_\kappa$$

□

Note that SLGAD can be used to check the unsoundness of an LPAD T: if all the answers for A in the table after one of the successful calls to SLG_SUBGOAL contains delayed literals, then T is unsound. The implemented system actually returns unsound in such a case.

The proof of correctness of SLGAD_COND is similar and is omitted for brevity.

5 Experiments

We tested SLGAD on some synthetic problems that were used as benchmarks for SLG [19, 24]: `win`, `ranc` and `lanc`. `win` is an implementation of the stalemate game and contains the clause

```
win(X):0.8 :- move(X,Y),\+ win(Y).
```

`ranc` and `lanc` model the ancestor relation with right and left recursion respectively:

```
rancestor(X,Y):0.8 :- move(X,Y).
rancestor(X,Y):0.8 :- move(X,Z),rancestor(Z,Y).
lancestor(X,Y):0.8 :- move(X,Y).
lancestor(X,Y):0.8 :- lancestor(Z,Y),move(X,Z).
```

Various definitions of `move` are considered: a linear and acyclic relation, containing the tuples $(1,2),\ldots,(N-1,N)$, a linear and cyclic relation, containing the tuples $(1,2),\ldots,(N-1,N),(N,1)$, and a tree relation, that represents a complete binary tree of height N, containing $2^{N+1}+1$ tuples. For `win`, all the `move` relations are used, while for `ranc` and `lanc` only the linear ones.

SLDAG was compared with Cilog2 and SLDNFAD. Cilog2 [14] computes probabilities by identifying consistent choices on which the query is true then it makes them mutually incompatible with an iterative algorithm. SLDNFAD [15] extends SLDNF in order to store choices and computes the probability with an algorithm based on Binary Decision Diagrams.

For SLGAD and SLDNFAD we used the implementations in Yap Prolog[2] available in the `cplint` suite[3]. SLGAD code is based on the SLG system[4]. For Cilog2 we ported the code available on the web[5] to Yap. All the experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.

The execution times for the query `win(1)` to the `win` program are shown in Figures 7, 8 and 9 as a function of N for linar, cyclic and tree `move` respectively.

The execution times for the query `ancestor(1,N)` to the `ranc` program are shown in Figures 12 and 13 as a function of N for linar and cyclic `move` respectively.

The execution times for the query `ancestor(1,N)` to the `lanc` program are shown in Figures 10 and 11 as a function of N for linar and cyclic `move` respectively.

`win` has an exponential number of instances where the query is true and the graphs show the combinatorial explosion.

[2] `http://www.ncc.up.pt/~vsc/Yap/`

[3] `http://www.ing.unife.it/software/cplint/`, also included in the CVS version of Yap

[4] `http://engr.smu.edu/~wchen/slg.html`

[5] `http://www.cs.ubc.ca/spider/poole/aibook/code/cilog/CILog2.html`

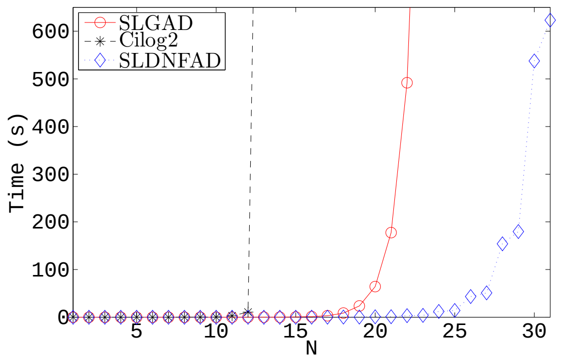

Fig. 7. Execution times for `win` with linear `move`

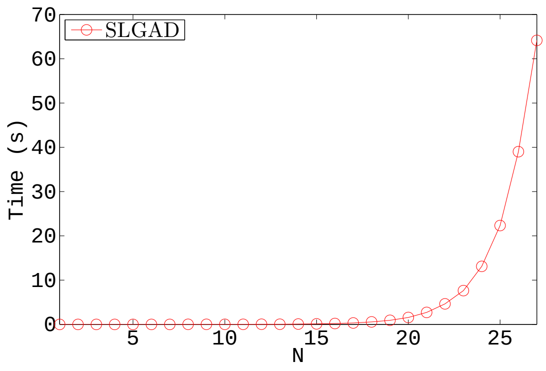

Fig. 8. Execution times for `win` with cyclic `move`

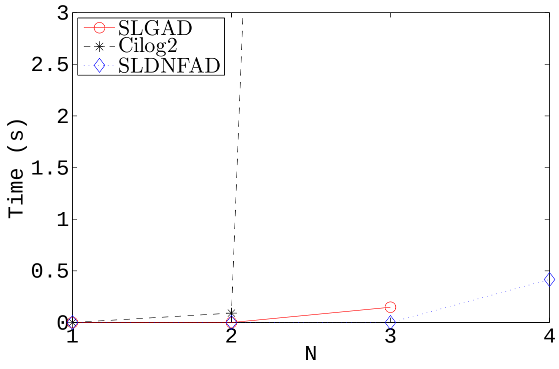

Fig. 9. Execution times for `win` with tree `move`

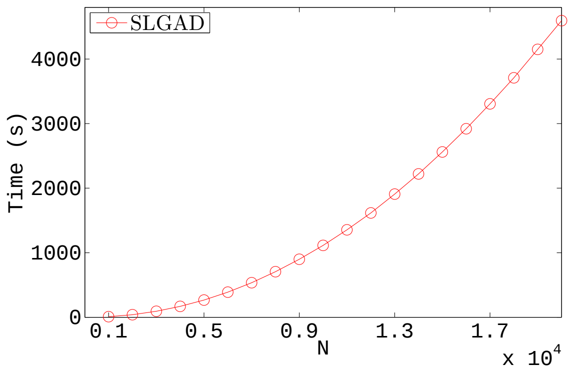

Fig. 10. Execution times for `lanc` with linear `move`

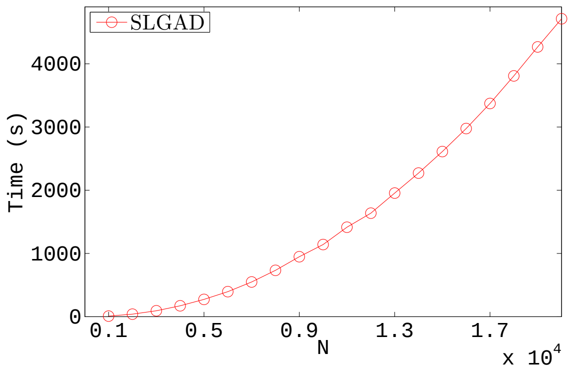

Fig. 11. Execution times for `lanc` with cyclic `move`

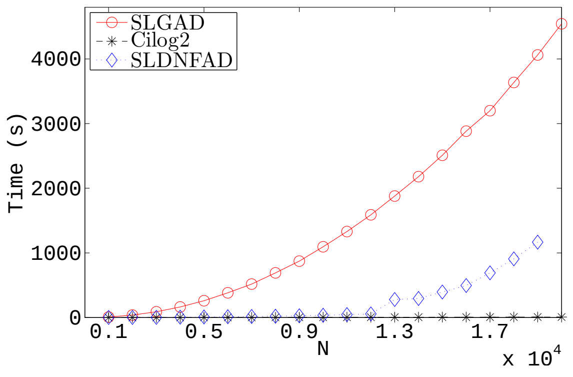

Fig. 12. Execution times for `ranc` with linear `move`

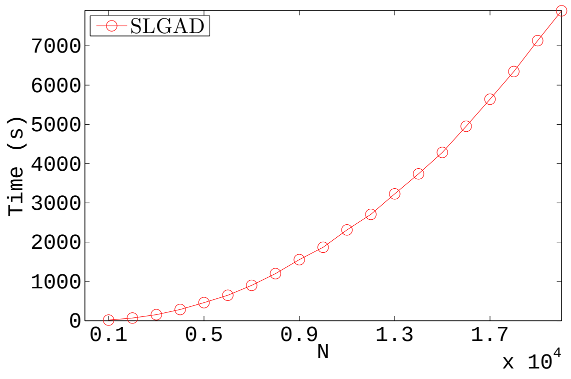

Fig. 13. Execution times for `ranc` with cyclic `move`

On the ancestor dataset, the proof tree has only one branch with a number of nodes proportional to N. However, the execution time of SLGAD increases more than linearly as a function of N because each derivation step requires a lookup and an insert in the table $\mathcal{T}$ that is implemented as a tree-like data structure (2-3 tree [25] in the SLG system). Each insert and lookup take logarithmic time.

SLGAD is compared with Cilog2 and SLDNFAD om the problems that are modularly acyclic and right recursive, i.e. `win` with linear and tree `move` and `ranc` with linear `move`. On the other problems a comparison was not possible because Cilog2 and SLDNFAD would go into a loop. In `win` all the algorithm show the combinatorial explosion, with SLGAD performing better than Cilog2 and worse than SLDNFAD. On `ranc` with linear `move`, SLGAD takes longer than Cilog2 and SLDNFAD, with Cilog2, SLDNFA and SLGAD taking 8.3, 1165.4 and 4726.8 seconds for $N = 20000$ respectively.

6 Conclusions and Future Works

We have presented the SLGAD top-down procedure for computing the probability of LPADs queries that is based on SLG [19] and we have experimentally compared it with Cilog2 and SLDNFAD.

In the future, we plan to consider the possibility of answering queries in an appoximate way, similary to what is done in [3], and we plan to extend the interpreter by considering also aggregates and the possibility of having the probabilities in the head depend on literals in the body.

References

1. Carnap, R.: Logical Foundations of Probability. University of Chicago Press (1950)
2. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. **94**(1–2) (1997) 7–56

3. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic prolog and its application in link discovery. In: International Joint Conference on Artificial Intelligence. (2007) 2462–2467
4. Cussens, J.: Stochastic logic programs: Sampling, inference and applications. In: Uncertainty in Artificial Intelligence, Morgan Kaufmann (2000) 115–122
5. Kersting, K., Raedt, L.D.: Towards combining inductive logic programming and Bayesian networks. In: Inductive Logic Programming. Volume 2157 of LNAI., Springer-Verlag (2001)
6. Sato, T., Kameya, Y.: PRISM: A language for symbolic-statistical modeling. In: International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1997) 1330–1339
7. Santos Costa, V., Page, D., Qazi, M., Cussens, J.: CLP($\mathcal{BN}$): Constraint logic programming for probabilistic knowledge. In: Uncertainty in Artificial Intelligence, Morgan Kaufmann (2003)
8. Jaeger, M., Lidman, P., Mateo, J.L.: Comparative evaluation of pl languages. In: Mining and Learning with Graphs. (2007)
9. Jaeger, M.: Model-theoretic expressivity analysis. In: Probabilistic Inductive Logic Programming - Theory and Applications. Volume 4911 of LNCS., Springer (2008) 325–339
10. Zhang, N.L., Poole, D.: Exploiting causal independence in Bayesian network inference. J. Artif. Intell. Res. **5** (1996) 301–328
11. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: International Conference on Logic Programming. Volume 3131 of LNCS., Springer (2004)
12. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. of the ACM **38**(3) (1991) 620–650
13. Vennekens, J., Verbaeten, S.: Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven (2003) http://www.cs.kuleuven.ac.be/~joost/techrep.ps.
14. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. J. Log. Program. **44**(1-3) (2000) 5–35
15. Riguzzi, F.: A top down interpreter for LPAD and CP-logic. In: Congress of the Italian Association for Artificial Intelligence. Volume 4733 of LNAI., Springer (2007) 109–120
16. Clark, K.L.: Negation as failure. In: Logic and Databases. Plenum Press (1978)
17. Apt, K.R., Bezem, M.: Acyclic programs. New Generation Comput. **9**(3/4) (1991) 335–364
18. Ross, K.A.: Modular acyclicity and tail recursion in logic programs. In: Symposium on Principles of Database Systems, ACM Press (1991) 92–101
19. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. J. Log. Program. **24**(3) (1995) 161–199
20. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc. (1988)
21. Alferes, J.J., Pereira, L.M.: Reasoning with logic programming. Volume 1111 of LNAI. Springer-Verlag (1996)
22. Chen, W., Warren, D.S.: Query evaluation under the well founded semantics. In: Symposium on Principles of Database Systems, ACM Press (1993) 168–179
23. Chen, W., Warren, D.: Towards effective evaluation of general logic programs. Technical report, State University of New York at Stony Brook (1993)

24. Castro, L.F., Swift, T., Warren, D.S.: Suspending and resuming computations in engines for SLG evaluation. In: Practical Aspects of Declarative Languages. Volume 2257 of LNCS., Springer (2002) 332–350
25. Bratko, I.: PROLOG Programming for Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc. (1990)