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

No Institute Given

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. SLGAD is compared with Cilog2 and SLDNFAD, an algorithm based on SLDNF, on the programs that are modularly acyclic. The results show that SLGAD deals correctly with cyclic programs and, even if it is more expensive than SLDNFAD on problems where SLDNFAD succeeds, is faster than Cilog2 when the query is true in an exponential number of instances.

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 [2]. 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, ProbLog, Stochastic Logic Programs, Bayesian Logic Programs, PRISM and CLP($\mathcal{BN}$). 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.

When we are reasoning about actions and effects and we have causal independence among different causes for the same effect, Logic Programs with Annotated Disjunctions (LPADs) [15] 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

Proceedings of the 15th International RCRA workshop (RCRA 2008):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Udine, Italy, 12–13 December 2008

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 the well founded model [13] 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. [14] 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.

[14] also showed that acyclic LPADs can be converted to Independent Choice Logic (ICL) programs. Thus inference can be performed by using the Cilog2 system [9]. An algorithm for performing inference directly with LPADs was proposed in [10]. 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 [8] for the ProbLog language. Both Cilog2 and SLDNFAD are complete and correct for programs for which the Clark's completion semantics [7] and the well founded semantics coincide, as for acyclic [1] and modularly acyclic programs [12].

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 [4] 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 is able to deal with cycles and, 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, Section 4 presents the experiments and, finally, Section 5 concludes and presents directions for future work.

2 Preliminaries

A Logic Program with Annotated Disjunctions [15] 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. For example, the formula

$$C = itching(X, strong) : 0.3 \vee itching(X, moderate) : 0.5 : -measles(X).$$

is an annotated disjunctive clause in which

$$\begin{aligned} head(C) &= \{(itching(X, strong) : 0.3), (itching(X, moderate) : 0.5), \\ &\quad (null, 0.2)\}, \\ body(C) &= \{measles(X)\} \end{aligned}$$

Moreover, $H_1(C) = itching(X, strong)$, $\alpha_1(C) = 0.3$ and so on.

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 LPAD T, we must generate the grounding T' of T. Each ground annotated disjunctive clause represents a probabilistic choice among 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 [15], requires the grounding to be finite, so the program must not contain function symbols.

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 among 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 $H_i\theta$ 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. For different groundings of the same clause, different heads can be chosen: it is possible for a consistent choice κ to be such that $(C, \theta, i) \in \kappa$, $(C, \theta', j) \in \kappa$ with $\theta \neq \theta'$ and $i \neq j$¿

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 of a program T by $\mathcal{S}_T$. Note that, since the program T does not contain function symbols, the grounding of T is finite and so is σ.

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* P_κ *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. Note that if T contains function symbols, σ would be infinite and so P_σ would always be 0, being an infinite product of numbers all smaller than one.

The semantics of the instances of an LPAD is given by the well founded semantics (WFS) [13]. Given a normal program T, we call $WFM(T)$ its *well founded partial model*. For each instance T_σ, we require that the $WFM(T_\sigma)$ 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:

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

Note that this semantics is well defined because, since the instances are finite, the probabilities P_σ are not all 0. A semantics for LPADs with function symbols must take a different approach and is subject of future work.

Example 1. *Consider the dependency of a person's itching from him having allergy or measles:*
$C_1 = itching(X, strong) : 0.3 \vee itching(X, moderate) : 0.5 : -measles(X).$
$C_2 = itching(X, strong) : 0.2 \vee itching(X, moderate) : 0.6 : -allergy(X).$
$C_3 = allergy(david).$
$C_4 = measles(david).$
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* $1 - 0.2 - 0.6 = 0.2$.

For example, $itching(david, strong)$ *is true in 5 of the 9 instances of the program and its probability is*

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

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*

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

plus facts for the move predicate.

If move represents an acyclic relation[1]*, 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.*

An LPAD is (modularly) acyclic [12] 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.

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 [6, 4] for dealing with LPADs.

In the following we give a brief sketch of the implementation of SLG resolution as presented in [4]. 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,

[1] A binary relation R is acyclic if the graph containing an edge from node a to node b if $R(a, b)$ holds is acyclic

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 3.1 (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 3.2 (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 (some of) 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 3.3 (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 call the XD-clause*

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

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.*

Definition 3.4 (SLGAD Answer Resolution)**.** *Let* G *be an 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 call the XD-clause*

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

Figure 1: Procedure SLGAD

```
1  function SLGAD(A,T)
2  begin
3      Initialize Count, 𝒯, 𝒮, DFN, PosMin and NegMin as in SLG;
4      κ := ∅;
5      let ψ be the set of all the values for κ after an execution of
6      SLG_SUBGOAL(A,PosMin,NegMin) such that 𝒯 contains
7          A as an answer;
8      return ∑_{κ∈ψ} P_κ
9  end
```

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 3.5 (SLGAD Factoring)**.** *Let G be an 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, 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*

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

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 heads 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 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 [4], plus procedure ADD_CLAUSE. We refer to

Figure 2: Procedures SLG_SUBGOAL

```
1  procedure SLG_SUBGOAL(A,PosMin,NegMin)
2  begin
3      for each  SLGAD goal resolvent G of A with some clause C ∈ T
4          on the head H_i do begin
5          SLG_NEWCLAUSE(A, G,PosMin,NegMin);
6      end;
7      SLG_COMPLETE(A,PosMin,NegMin,κ);
8  end;
```

[4] 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 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.

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 Boolean value to SLG_ANSWER in the variable Derivable. If $G = (X, C, \theta, i)$[2], ADD_CLAUSE may add a new triple (C, θ, j) to the current κ set. If the program is range restricted, $C\theta$ is ground. 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 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.

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.

[2] 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

Figure 3: Procedure SLG_ANSWER

```
1  procedure SLG_ANSWER(A, G,PosMin,NegMin)
2  begin
3     if G is not subsumed by any answer in Anss(A) in T then
3     begin
4        ADD_CLAUSE(G,Derivable);
5        if Derivable then begin
6           insert G into Anss(A);
7           if G has no delayed literals then begin;
8              reset Negs(A) to empty;
9              let L be the list of all pairs (B, H'), where
10                (B, H) ∈ Poss(A) and
11                H is the SLGAD answer resolvent of H with G;
12             for each (B, H') in L do begin
13                SLG_NEWCLAUSE(B,H',PosMin,NegMin);
14             end;
15          end else begin /* G has a non empty delay */
16             if no other answer in Anss(A) has the same head
17                as G does then
18             begin
19                let L be the list of all pairs (B, H'), where
20                   (B, H) ∈ Poss(A)
21                   and H is the SLGAD factor of H with G;
22                for each (B, H') in L do begin
23                   SLG_NEWCLAUSE(B, H',PosMin,NegMin);
24                end;
25             end;
26          end;
27       end;
28    end;
29 end;
```

Figure 4: Procedure ADD_CLAUSE

```
1  procedure ADD_CLAUSE(G,Derivable)
2  begin
3      let G be (X, C, θ, i);
4      if ∃k : (C, θ, k) ∈ κ, k ≠ i then begin
5          fail;
6      end else begin
7          choose an index j from {1, ..., |head(C)|} (choice point);
8          if i = j then begin
9              Derivable:= true;
10         end else begin
11             Derivable:= false;
12         end
13         κ := κ ∪ {(C, θ, j)};
14     end
15 end
```

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, modifies the one of SLG by replacing SLG resolution with SLGAD answer resolution and SLG factoring with SLGAD factoring. The other SLG procedure are modified simply by replacing X-clauses with XD-clauses.

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.

SLGAD is sound and complete with respect to the LPAD semantics and the proof is based on the theorem of partial correctness of SLG [6, 5]: SLG is sound and complete given an arbitrary but fixed computation rule when it does not flounder. The proof of soundness and completeness can be found in [11].

4 Experiments

We tested SLGAD on some synthetic problems similar to those that were used as benchmarks for SLG [4, 3]: `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-2 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 [9] computes probabilities by identifying consistent choices on which the query is true then it makes them mutually incompatible with an iterative algorithm. SLDNFAD [10] 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[3] available in the `cplint` suite[4]. SLGAD code is based on the SLG system[5]. For Cilog2 we ported the code available on the web[6] to Yap. All the experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2,333 MHz) processor and 4 GB of RAM.

The execution times for the query `win(1)` to the `win` program are shown in Figures 5(a), 5(b) and 6 as a function of N for linear, cyclic and tree `move` respectively.

The execution times for the query `ancestor(1,N)` to the `ranc` program are shown in Figures 7(a) and 7(b) as a function of N for linear and cyclic `move` respectively.

The execution times for the query `ancestor(1,N)` to the `lanc` program are shown in Figures 8(a) and 8(b) as a function of N for linear and cyclic `move` respectively.

`win` has an exponential number of instances where the query is true and the graphs show the combinatorial explosion. On `win` with tree `move` the graph in Figure 6 shows the execution times only up to $N=4$ because for $N=4$ SLGAD did not terminate after one day.

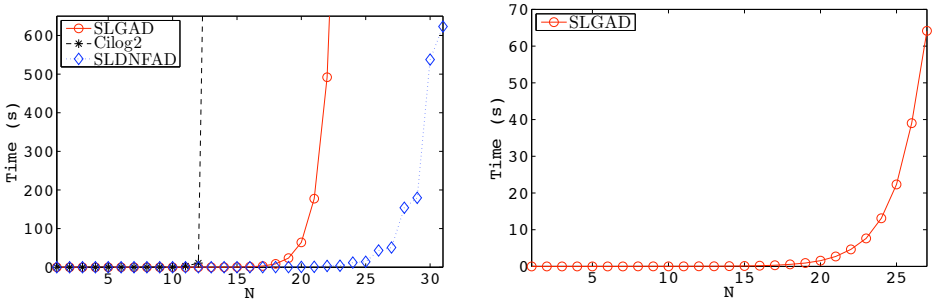
On the ancestor dataset, the proof tree has only one successful branch with a number of nodes proportional to N. However, the execution time

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

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

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

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



(a) Linear `move`. (b) Cyclic `move`.

Figure 5: Execution times for `win` with linear and cyclic `move`.

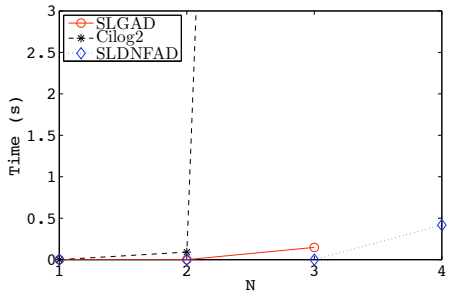
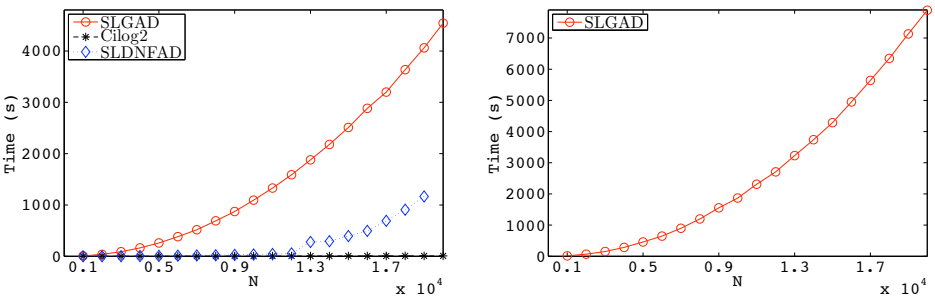
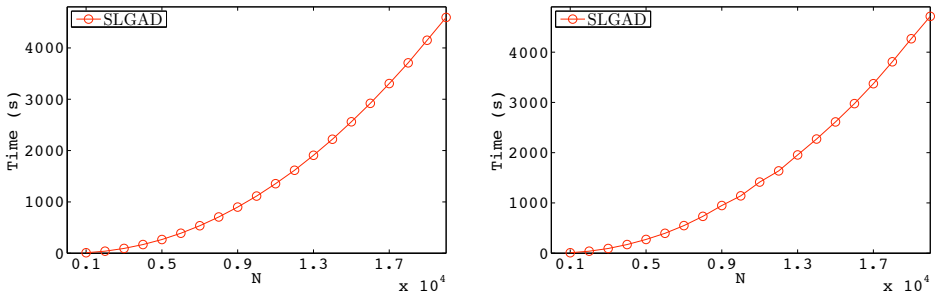


Figure 6: Execution times for `win` with tree `move`



(a) Linear `move`. (b) Cyclic `move`.

Figure 7: Execution times for `ranc`



(a) Linear `move`. (b) Cyclic `move`.

Figure 8: Execution times for `lanc`.

of SLGAD increases more than linearly as a function of N because each derivation step requires a lookup and an insert into the table $\mathcal{T}$ that is implemented as a tree-like data structure (2-3 tree in the SLG system). Each insert and lookup take logarithmic time.

SLGAD is compared with Cilog2 and SLDNFAD on the problems that are modularly acyclic and right recursive, i.e. `win` with linear and tree `move` (Figures 5(a) and 6) and `ranc` with linear `move` (Figure 7(a)). On the other problems a comparison was not possible because Cilog2 and SLDNFAD would go into a loop. In `win` all the algorithms show the combinatorial explosion, with SLGAD performing better than Cilog2 and worse than SLDNFAD. On `ranc` with linear `move` (Figure 7(a)), SLGAD takes longer than Cilog2 and SLDNFAD, with Cilog2, SLDNFAD and SLGAD taking 8.3, 1165.4 and 4726.8 seconds for $N = 20000$ respectively.

5 Conclusions and Future Works

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

In the future, we plan to study the possibility of answering queries in an approximate way by returning an upper and a lower bound of the correct probability. Moreover, we plan to consider extensions of the language such as aggregates in the body and probabilities in the head that depend on literals in the body.

References

[1] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Comput.*, 9(3/4):335–364, 1991.

[2] R. Carnap. *Logical Foundations of Probability*. University of Chicago Press, 1950.

[3] L. F. Castro, T. Swift, and D. S. Warren. Suspending and resuming computations in engines for SLG evaluation. In *Practical Aspects of Declarative Languages*, volume 2257 of *LNCS*, pages 332–350. Springer, 2002.

[4] W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Log. Program.*, 24(3):161–199, 1995.

[5] W. Chen and D. Warren. Towards effective evaluation of general logic programs. Technical report, State University of New York at Stony Brook, 1993.

[6] W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *Symposium on Principles of Database Systems*, pages 168–179. ACM Press, 1993.

[7] K. L. Clark. Negation as failure. In *Logic and Databases*. Plenum Press, 1978.

[8] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.

[9] D. Poole. Abducing through negation as failure: stable models within the independent choice logic. *J. Log. Program.*, 44(1-3):5–35, 2000.

[10] F. Riguzzi. A top down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence*, volume 4733 of *LNAI*, pages 109–120. Springer, 2007.

[11] F. Riguzzi. The SLGAD procedure for inference on logic programs with annotated disjunctions. Technical Report CS-2008-01, University of Ferrara, 2008. http://www.unife.it/dipartimento/ingegneria/informazione/informatica/rapporti-tecnici-1/cs-2008-01.pdf/view.

[12] K. A. Ross. Modular acyclicity and tail recursion in logic programs. In *Symposium on Principles of Database Systems*, pages 92–101. ACM Press, 1991.

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

[14] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven, 2003. http://www.cs.kuleuven. ac.be/∼joost/techrep.ps.

[15] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*. Springer, 2004.