Fundamenta Informaticae XXI (2001) 1–38
IOS Press

SLGAD Resolution 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 the well-founded models of the normal logic programs obtained by selecting one disjunct from each ground LPAD clause.

Inference on LPADs can be performed using either the system Ailog2, that was developed for the Independent Choice Logic, or SLDNFAD, an algorithm based on SLDNF. However, both of these algorithms run the risk of going into infinite loops and of performing redundant computations.

In order to avoid these problems, we present SLGAD resolution that computes the (conditional) probability of a ground query from a range-restricted LPAD and is based on SLG resolution for normal logic programs. As SLG, it uses tabling to avoid some infinite loops and to avoid redundant computations.

The performances of SLGAD are evaluated on classical benchmarks for normal logic programs under the well-founded semantics, namely a 2-person game and the ancestor relation, and on games of dice.

SLGAD is compared with Ailog2 and SLDNFAD on the problems in which they do not go into infinite loops, namely those that are described by a modularly acyclic program.

The results show that SLGAD is sometimes slower than Ailog2 and SLDNFAD but, if the program requires the repeated computations of the same goals, as for the dice games, then SLGAD is faster than both.

Keywords: Probabilistic Logic Programming, Well-Founded Semantics, Logic Programs with Annotated Disjunctions, SLG Resolution.

1. Introduction

Effectively combining logic and probability would allow the exploitation of the advantages of both: on the one side, the possibility, offered by logic, of representing in a simple way complex relationships

among the entities of the domain, on the other side, the possibility, offered by probability theory, of reasoning with uncertain and incomplete knowledge.

Many languages have been recently proposed that combine relational and statistical aspects, such as the Independent Choice Logic [17], ProbLog [9], PRISM [25] and CLP($\mathcal{BN}$) [23]. These languages have different semantics that make them suitable for different domains: the identification of the best setting for each language is currently under study [15, 14].

When we are reasoning about actions and effects and we have causal independence [32] among different causes for the same effect, Logic Programs with Annotated Disjunctions (LPADs) [31] seem particularly suitable. They extend logic programs by allowing clauses to be disjunctive and by annotating each atom in the head with a probability. A clause can be causally interpreted by supposing that 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 the well-founded models [28] of the normal logic programs obtained by selecting one head for each ground disjunctive clause.

In order to compute the (conditional) probability of queries, various options are possible. [30] 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 for all but trivial programs.

[30] also showed that acyclic LPADs can be converted to Independent Choice Logic programs. Thus inference can be performed by using the Ailog2 system [18]. An algorithm for performing inference directly with LPADs was proposed in [21]. The algorithm, that will be called SLDNFAD in the following, is an extension of SLDNF resolution and uses Binary Decision Diagrams, similarly to what is presented in [9] for the ProbLog language. Both Ailog2 and SLDNFAD are sound for programs for which the Clark's completion semantics [7] and the well-founded semantics coincide, as for acyclic [1] and modularly acyclic programs [22].

For programs that are not modularly acyclic, Ailog2 and SLDNFAD may go into infinite loops. Moreover, they both run the risk of computing solutions to the same or similar queries more than once. Therefore, we present SLGAD resolution that is able to perform inference on non-modularly acyclic LPADs and to avoid redundant computations. SLGAD resolution is based on SLG resolution [6] for normal logic programs under the well-founded semantics. We will present SLGAD resolution both at a declarative level, as a set of operations to be applied to a particular data structure, and at a procedural level, by presenting the procedures that implement the algorithm.

SLGAD is evaluated on classical benchmarks for inference algorithms under the well-founded semantics, namely a 2-person game and the ancestor relation, and on games of dice. In the first two cases, various extensional databases are considered, encoding linear, cyclic or tree-shaped relations. Of these problems, the 2-person game with linear and tree-shaped relation, ancestor with a linear relation and the games of dice are modularly acyclic.

The results show that SLGAD is able to deal with all of these problems. In order to compare it with Ailog2 and SLDNFAD, we applied them to the programs that are modularly acyclic and right recursive. On the 2-person game SLGAD was faster than SLDNFAD and Ailog2 except for SLDNFAD on the tree-shaped relation. On ancestor, SLGAD was slower than Ailog2 and SLDNFAD.

On the games of dice, Ailog2 and SLDNFA perform redundant computations and SLGAD outperformed them by a large margin, thanks to its use of tabling to store the answers of already computed subgoals.

The paper is organized as follows. Section 2 presents some preliminary notions on normal logic programs and on LPADs while Section 3 describes SLG resolution. Section 4 provides the declarative

definition of SLGAD resolution. Section 5 contains its proof of soundness and Section 6 its procedural implementation. Sections 7 and 8 discuss related works and experiments respectively. Section 9 presents directions for future work and Section 10 concludes.

2. Preliminaries

2.1. Normal Logic Programs

A *first order alphabet* Σ is a set of predicate symbols and function symbols (or functors) together with their arity. A functor with arity 0 is called a *constant*.

A *term* is either a variable or a functor applied to a tuple of terms of length equal to the arity of the functor. An *atom* A is a predicate symbol applied to a tuple of terms of length equal to the arity of the predicate. A *literal* L is either an atom A or its negation $\neg A$. In the latter case it is called a *negative literal*. In logic programming, Prolog conventions are common practice, and also in this work predicates and constants are indicated with alphanumeric strings starting with a lowercase character while variables are indicated with alphanumeric strings starting with an uppercase character.

A *normal logic program* T is a set of of formulas of the form

$$H : -B_1, \ldots, B_b$$

called *clauses* where H is an atom and all the B_is are literals. H is called the *head* of the clause and $B_1, \ldots, B_b$ is called the *body*. If the body is empty the clause is called a *fact*. In the following, by *program* we mean a normal logic program. Programs containing only functors with arity 0 will be called *functor-free programs*.

A term, atom, literal or clause is *ground* if it does not contain variables. A *substitution* θ is an assignment of variables to terms: $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$. The *application of a substitution to a term atom, literal or clause C*, indicated with $C\theta$, is the replacement of the variables appearing in C and in θ with the terms specified in θ. $C\theta$ is called an *instance* of C.

A normal logic program is *range-restricted* if all the variables appearing in the head of clauses also appear in positive literals in the body.

The *Herbrand universe* $H_U(T)$ is the set of all the ground terms that can be built with function symbols appearing in T. The *Herbrand base* $H_B(T)$ of a program T is the set of all the ground atoms that can be built with predicates appearing in T and terms of $H_U(T)$. If T is functor-free, then $H_B(T)$ is finite, otherwise it is infinite. A *grounding* of a clause C is obtained by replacing all the variables of C with terms from $H_U(T)$. Let $g(C)$ be the set of groundings of clause C. The grounding $g(T)$ of a program T is the program obtained by replacing each clause C with $g(C)$. If the program is functor-free, $g(T)$ is finite, otherwise it is infinite. A *Herbrand interpretation over $H_B(T)$* (or just *interpretation*) is a set of ground atoms, i.e. a subset of $H_B(T)$. Let $\mathcal{I}_T$ be the set of all the possible interpretations of T.

An *interpretation I for a set of predicates S* is a subset of $H_B(T)$ that contains only atoms whose predicate is in S.

Let $\mathbf{f}, \mathbf{u}, \mathbf{t}$ be truth values where $\mathbf{u}$ is intended as the truth value "undefined". A *partial Herbrand interpretation I* over $H_B(T)$ (or just *partial interpretation*) is a mapping from $H_B(T)$ to $\{\mathbf{f}, \mathbf{u}, \mathbf{t}\}$. I can be represented by means of two sets, $Pos(I)$ and $Neg(I)$, the set of atoms of $H_B(T)$ taking values $\mathbf{t}$ and $\mathbf{f}$ respectively. If $Pos(I) \cup Neg(I) = H_B(T)$ we say that I is *total*.

Given a partial interpretation I and a ground atom A, A ($\neg A$) is *true* (*false*) in I if $A \in Pos(I)$, is *false* (*true*) in I if $A \in Neg(I)$ and is *undefined* if $A \notin Pos(U)$ and $A \notin Neg(A)$.

A partial interpretation I is a *model* of a program T iff for all the ground instances

$$H : -B_1, \ldots, B_b$$

of its clauses, if all B_is are true in I then H is true in I and if H is false in I then at least one of the B_is is false in I.

If I and J are partial interpretations, there are two natural orderings between them:

- Fitting ordering: $I \preceq J$ if $Pos(I) \subseteq Pos(J)$ and $Neg(I) \supseteq Neg(J)$. Models that are least in the $\preceq$ ordering are called *least* models.
- Information ordering: $I \subseteq J$ if $Pos(I) \subseteq Pos(J)$ and $Neg(I) \subseteq Neg(J)$. Models that are least in the $\subseteq$ ordering are called *smallest* models.

Various semantics have been proposed for normal logic programs. In this paper we consider the 3-valued stable models semantics [19], the stable models semantics [13], the well-founded semantics [28] and the Clark's completion semantics [7].

Let us define 3-valued stable models.

Definition 2.1. ([19])
Let T be a program and let I be a partial interpretation. Then $\mathcal{T}_T(I)$ is a partial interpretation such that:

- $A \in Pos(\mathcal{T}_T(I))$ iff there is a clause $A : -B_1, \ldots, B_b$ in $g(T)$ and all the B_is are true in I;
- $A \in Neg(\mathcal{T}_T(I))$ iff for every clause $A : -B_1, \ldots, B_b$ in $g(T)$, some B_i is false in I

Let $\emptyset$ be the partial interpretation in which all the ground atoms are false. The powers of $\mathcal{T}_T$ are defined as follows:

$$\begin{array}{lcll} \mathcal{T}_T \uparrow 0 & = & \emptyset & \\ \mathcal{T}_T \uparrow n & = & \mathcal{T}_T(\mathcal{T}_T \uparrow (n-1)) & \text{if } n \text{ is a successor ordinal} \\ & = & \sqcup\{\mathcal{T}_T \uparrow k : k < n\} & \text{if } n \text{ is a limit ordinal} \end{array}$$

where $\sqcup$ is the least upper bound operation of interpretations with respect to the Fitting ordering $\preceq$.

A *non-negative program* is a finite set of clauses whose bodies do not contain any negative literals but may contain the special ground atom $\mathbf{u}$ which is always undefined (i.e., $\mathbf{u} \notin Pos(I)$ and $\mathbf{u} \notin Neg(I)$ for any partial interpretation I).

Theorem 2.1. ([19])
Let T be a non-negative program. Then T has a unique least 3-valued model, denoted by $LPM(T)$. Furthermore, $\mathcal{T}_T$ has a least fixed point which coincides with $\mathcal{T}_T \uparrow \omega$ and with $LPM(T)$.

Definition 2.2. ([19])
Let T be a program and let I be a partial interpretation. The *quotient of T modulo I*, denoted by $\frac{T}{I}$, is the non-negative program obtained from $g(T)$ by

- deleting every clause with a negative literal in the body that is false in I, and

- deleting a negative literal L in the body of a clause if L is true in I, and
- replacing a negative literal L in the body of a clause with $\mathbf{u}$ if L is undefined in I.

I is a *3-valued stable model* of T if I is the least 3-valued model $LPM(\frac{T}{I})$.

Every program has at least one 3-valued stable model but may have more than one.

The stable model semantics [13] is 2-valued, i.e., its models are Herbrand interpretations. A program can have any number of stable models, including the case of no stable models.

The well-founded semantics [28] is 3-valued and it assigns every program T a single partial interpretation $WF(T)$ called the *well-founded partial model*. To indicate that an atom A is true in $WF(T)$ we write $T \models_{WF} A$.

The following theorem shows the relationship between the stable models semantics and the well-founded semantics via 3-valued stable models.

Theorem 2.2. ([19])
Let T be a normal logic program. Then $WF(T)$ is the smallest 3-valued stable model of T. If a 3-valued stable model is total, then it coincides with a stable model as defined in [13].

Thus, if $WF(T)$ is total, then T has a single stable model equal to $WF(T)$.

We report here the definition of an acyclic [1] and modularly acyclic [22] program. A *level mapping* for a program T is a function $|\ | : H_B(T) \to N$ from ground atoms to natural numbers. For $A \in H_B(T)$, $|A|$ denotes the *level* of A. If $L = \neg A$ where $A \in H_B(T)$, we define $|L| = |A|$. A program T is called *acyclic with respect to a level mapping* if for every ground instance $A : -B$ of a clause of T, the level of A is greater then the level of each literal in B. A program T is called *acyclic* if there is some level mapping such that T is acyclic with respect to it.

A predicate p *directly depends* on a predicate q if q appears in the body of a rule that has p in the head. The relation "*depends*" is the transitive closure of the relation "directly depends". A predicate p is *recursive* on a predicate q if $p = q$ or if p depends on q and q depends on p.

Recursiveness is an equivalence relation between predicates: it partitions the set of predicates of a program T into equivalence classes $Q_1, \ldots, Q_M$. For each equivalence class Q_i, consider the set V_i containing the clauses of T whose predicate of the atom in the head belongs to Q_i. $V_1, \ldots, V_M$ is a partition of T and the sets $V_1, \ldots, V_M$ are called *components* of T. We write $V_j \sqsubset V_i$ if there is a predicate of Q_i that directly depends on a predicate of Q_j. We denote with $\lhd$ the transitive closure of $\sqsubset$. The predicates of $S_i = \bigcup_{V_j \sqsubset V_i} Q_j$ are called the *predicates used by* V_i.

Let V_i be a component of a program T and let S_i be the set of predicates used by V_i. Consider an interpretation I for S_i.

The *reduction of* V_i *modulo* I, denoted with $R_I(V_i)$, is obtained in the following way:

- ground in all possible ways the rules of V_i obtaining $g(V_i)$;
- delete from $g(V_i)$ all the rules having a literal in the body whose predicate is in S_i, but which is false in I;
- delete from the bodies of the remaining rules all the literals having predicates in S_i (which are true);
- set $R_I(V_i)$ to the set of remaining ground rules.

A normal logic program T is *modularly acyclic* if for every component V_i of T 1)there exists a total well-founded model M_i for the union of all the components $V_j \triangleleft V_i$, and 2) the reduction of V_i modulo M_i is acyclic.

If a program is acyclic or modularly acyclic, the unique Herbrand model of Clark's completion and the well-founded partial model coincide [1, 22], so queries can be answered in the well-founded semantics by means of SLDNF. If a program is not modularly acyclic, then SLG resolution [6] has to be employed for answering queries.

Example 2.1. Consider a two-person game in which the players alternate and a position X is winning for a player if there is a move from X to a position Y that is not winning for the opponent. Such a game can be modeled with the famous normal logic program [10, 13, 12]:

$win(X) : -move(X, Y), \neg win(Y).$

plus facts for the $move$ relation, where $move(a, b)$ means that there is a legal move from position a to position b. This is one of the examples that lead to the formulation of both the well-founded and the stable model semantics.

In this game, a position is surely losing if there are no moves from it. If a position leads to a surely losing position, it is winning for the opponent and so on. An example of such a game is checkers.

If $move$ is acyclic[1], the program has a single stable model and a total well-founded model and the two models coincide. If $move$ is cyclic, the program can have no stable models or multiple stable models and the well-founded model is in general not total.

Consider for example the following definition for $move$:

$move(a, b).\ move(b, a).\ move(a, c).$

Such a definition is cyclic but the program has the total well-founded model

$\{win(a), move(a, b), move(b, a), move(a, c)\}$

SLG resolution in this case assigns the value true to the query $win(a)$. This program is neither acyclic nor modularly acyclic, so SLNDF resolution is not able to answer this query. In particular, SLDNF resolution would go into an infinite loop for such a query.

2.2. Logic Programs with Annotated Disjunctions

A Logic Program with Annotated Disjunctions [31] T consists of a finite set of formulas of the form

$(H_1 : \alpha_1) \vee (H_2 : \alpha_2) \vee \ldots \vee (H_h : \alpha_h) : -B_1, B_2, \ldots B_b$

called *annotated disjunctive clauses*. In such clauses, the H_is are logical atoms, the B_is are logical literals and the α_is are real numbers in the interval $[0, 1]$ such that $\sum_{i=1}^{h} \alpha_i \leq 1$. If $\sum_{i=1}^{h} \alpha_i < 1$, the head of the clause implicitly contains an extra atom $null$ that does not appear in the body of any clause and whose annotation is $1 - \sum_{i=1}^{h} \alpha_i$.

If $h = 1$ and $\alpha_h = 1$, we write the clause as a definite clause of the form

$H_1 : -B_1, B_2, \ldots B_b.$

For an annotated disjunctive clause C, we define $head(C)$ as $\{(H_i : \alpha_i)|1 \leq i \leq h\}$ if $\sum_{i=1}^{h} \alpha_i = 1$ and as $\{(H_i : \alpha_i)|1 \leq i \leq h\} \cup \{(null : 1 - \sum_{i=1}^{h} \alpha_i)\}$ otherwise. Moreover, we define $body(C)$ as $\{B_i|1 \leq i \leq b\}$, $H_i(C)$ 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 T.

[1]A binary relation is *acyclic* if its transitive closure is not reflexive.

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

In order to define the semantics of a non-ground T, we must generate its grounding $g(T)$. Each ground annotated disjunctive clause represents a probabilistic choice among the ground non-disjunctive clauses obtained by selecting only one head atom. The intuitive interpretation of a ground clause is that the body represents an event that, when happening (i.e. when 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 [31], requires the grounding to be finite, so the program must be functor-free. In the following we will thus consider only functor-free programs.

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.

An *atomic choice* χ is a triple (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 set of atomic choices 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 set of atomic choices is called a *composite choice*.

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

A composite 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 composite 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 $\sigma \supseteq \kappa$.

The *probability P_κ of a composite choice κ* is the product of the probabilities of the individual atomic choices, i.e., $P_\kappa = \prod_{(C,\theta,i)\in\kappa} \alpha_i(C)$. The *probability of instance T_σ* is P_σ.

The meaning of the instances of an LPAD is given by the well-founded semantics. For each instance T_σ, we require that $WF(T_\sigma)$ is total, since we want to model uncertainty solely by means of disjunctions. An LPAD T is called *sound* iff, for each selection σ in $\mathcal{R}_T$, $WF(T_\sigma)$ is total. In the following we consider only sound LPADs.

The probability of a formula Q is given by the sum of the probabilities of the instances in which the formula is true according to the well-founded semantics: $P_T(Q) = \sum_{\sigma\in\mathcal{R}_T, T_\sigma \models_{WF} Q} P_\sigma$. The conditional probability of a formula Q given another formula E can be defined as usual as $P_T(Q|E) = \frac{P_T(Q\wedge E)}{P_T(E)}$. From these definition, it is clear that, if $g(T)$ is infinite, then the semantics is not well-defined. In fact, the probability P_σ for an instance would be an infinite product of numbers all smaller than 1, i.e., it would be 0. Therefore, also $P_T(Q)$ would be 0. An extension of the semantics to handle LPADs with function symbols of arity greater than 0 is subject of future work.

Example 2.2. Consider the dependency of a person's itching from him having allergy or measles:

$C_1 : strong_itching(X) : 0.3 \vee moderate_itching(X) : 0.5 : -measles(X).$

$C_2 : strong_itching(X) : 0.2 \vee moderate_itching(X) : 0.6 : -allergy(X).$

$C_3 : allergy(david).$

$C_4 : measles(david).$

Clauses C_1 and C_2 have three alternatives in the head, while clauses C_3 and C_4 have only a single alternative. 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 $1 - 0.2 - 0.6 = 0.2$.

In order to provide a semantics to the program, we must generate its grounding. The only constant is *david* so the above program has the following grounding:

$C'_1 : strong_itching(david) : 0.3 \vee moderate_itching(david) : 0.5 : -measles(david).$
$C'_2 : strong_itching(david) : 0.2 \vee moderate_itching(david) : 0.6 : -allergy(david).$
$C'_3 : allergy(david).$
$C'_4 : measles(david).$

By picking in all possible ways one head atom from C'_1 and one from C'_2 we get 9 instances, one of which is

$strong_itching(david) : -measles(david).$
$moderate_itching(david) : -allergy(david).$
$allergy(david).$
$measles(david).$

whose probability is $0.3 \cdot 0.6 \cdot 1 \cdot 1 = 0.18$.

$strong_itching(david)$ is true in 5 of the 9 instances of the program and its probability is

$$P_T(strong_itching(david)) = 0.3 \cdot 0.2 + 0.3 \cdot 0.6 + 0.3 \cdot 0.2 + 0.5 \cdot 0.2 + 0.2 \cdot 0.2 = 0.44$$

LPADs show patterns of *causal independence* [32]: 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 others so they combine with the noisy-or law [16]. 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 happening of the causes (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 $1 - \prod_{i=1}^{n}(1 - p_i)$.

In the above example, if only one cause of strong itching 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 $strong_itching(david)$, there are two causes, namely $allergy(david)$ and $measles(david)$. The probability computed by noisy-or is $1 - (1 - 0.3) \cdot (1 - 0.2) = 0.44$.

Example 2.3. Consider the program encoding the 2-person game of Example 2.1. Suppose that the game is probabilistic: a position X is winning with 80% probability for a player if there is a move from X to a position Y that is not winning for the opponent. This game can be modeled with the LPAD

$win(X) : 0.8 : -move(X, Y), \neg win(Y).$

plus facts for the $move$ predicate. If $move$ is acyclic, then the program is sound. Otherwise, there may be instances that do not have a total well-founded model.

Let us now see other properties of LPADs.

Lemma 2.1. Given an LPAD T and a composite choice κ, P_κ is the sum of the probability of the instances of $U(\kappa)$ i.e. $P_\kappa = \sum_{T_\sigma \in U(\kappa)} P_\sigma$

Proof:
Let $g(T)$ be $\{C_1, \ldots, C_p\}$, let $\kappa = \{(C_1, \emptyset, i_1), \ldots, (C_k, \emptyset, i_k)\}$ with $k \leq p$ and let a generic σ such that $T_\sigma \in U(\kappa)$ be $\{(C_1, \emptyset, i_1), \ldots, (C_k, \emptyset, i_k), (C_{k+1}, \emptyset, i_{\sigma,k+1}), \ldots, (C_p, \emptyset, i_{\sigma,p})\}$. We can write

$$\sum_{T_\sigma \in U(\kappa)} P_\sigma = \sum_{T_\sigma \in U(\kappa)} \prod_{l=1}^{k} \alpha_{i_l}(C_l) \prod_{m=k+1}^{p} \alpha_{i_{\sigma,m}}(C_m)$$

The set of instances of $U(\kappa)$ is obtained by selecting in all possible ways the head atoms of the clauses $C_{k+1}, \ldots, C_p$, so, if $N_m = \{1 \ldots |head(C_m)|\}$ and $\mathcal{N}_m = N_{k+1} \times \ldots \times N_m$ for $m = k+1, \ldots, p$, then

$$\begin{aligned}
\sum_{T_\sigma \in U(\kappa)} P_\sigma &= \sum_{(n_{k+1},\ldots,n_p)\in\mathcal{N}_p} \prod_{l=1}^{k} \alpha_{i_l}(C_l) \prod_{m=k+1}^{p} \alpha_{n_m}(C_m) \\
&= \left(\prod_{l=1}^{k} \alpha_{i_l}(C_l)\right) \sum_{(n_{k+1},\ldots,n_p)\in\mathcal{N}_p} \prod_{m=k+1}^{p} \alpha_{n_m}(C_m) \qquad (1) \\
&= \left(\prod_{l=1}^{k} \alpha_{i_l}(C_l)\right) \sum_{(n_{k+1},\ldots,n_{p-1})\in\mathcal{N}_{p-1}} \sum_{n_p \in N_p} \left(\prod_{m=k+1}^{p-1} \alpha_{n_m}(C_m)\right) \alpha_{n_p}(C_p) \\
&= \left(\prod_{l=1}^{k} \alpha_{i_l}(C_l)\right) \sum_{(n_{k+1},\ldots,n_{p-1})\in\mathcal{N}_{p-1}} \left(\prod_{m=k+1}^{p-1} \alpha_{n_m}(C_m)\right) \sum_{n_p \in N_p} \alpha_{n_p}(C_p) \qquad (2) \\
&= \left(\prod_{l=1}^{k} \alpha_{i_l}(C_l)\right) \sum_{(n_{k+1},\ldots,n_{p-1})\in\mathcal{N}_{p-1}} \left(\prod_{m=k+1}^{p-1} \alpha_{n_m}(C_m)\right) \cdot 1 \qquad (3) \\
&= \left(\prod_{l=1}^{k} \alpha_{i_l}(C_l)\right) \sum_{(n_{k+1},\ldots,n_{p-1})\in\mathcal{N}_{p-1}} \prod_{m=k+1}^{p-1} \alpha_{n_m}(C_m) \\
&= \left(\prod_{l=1}^{k} \alpha_{i_l}(C_l)\right) \sum_{(n_{k+1},\ldots,n_{p-2})\in\mathcal{N}_{p-2}} \prod_{m=k+1}^{p-nt} \alpha_{n_m}(C_m) \\
&\vdots \\
&= \prod_{l=1}^{k} \alpha_{i_l}(C_l) = \qquad (4) \\
&= P_\kappa
\end{aligned}$$

Formula 1 is obtained because $\prod_{l=1}^{k} \alpha_{i_l}(C_l)$ does not depend on the index of the summation. Formula 2 is obtained because $\prod_{m=k+1}^{p-1} \alpha_{n_m}(C_m)$ does not depend on the index of the innermost summation. Since the probabilities in the head of an LPAD clause sum up to 1 we get Formula 3. By repeating the above process $p - k$ times, we get Formula 4 which is P_κ by definition. □

A composite choice κ is an *explanation for a goal* Q if $T_\sigma \models_{WF} Q$ for all $T_\sigma \in U(\kappa)$.

For the case of Example 2.2, the following composite choices
$\kappa_1 = \{(C_1, \{X/david\}, 1)\}$
$\kappa_2 = \{(C_2, \{X/david\}, 1)\}$
$\kappa_3 = \{(C_1, \{X/david\}, 1), (C_2, \{X/david\}, 1)\}$
are explanation for $strong_itching(david)$.

A set of explanations $K = \{\kappa_1, \ldots, \kappa_n\}$ is *covering with respect to a query* Q if, for every instance T_σ such that $T_\sigma \models_{WF} Q$, $T_\sigma \in \bigcup_{i=1}^n U(\kappa_i)$. The sets of explanations
$K_1 = \{\kappa_1, \kappa_2\}$
$K_2 = \{\kappa_1, \{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 1)\}, \{(C_1, \{X/david\}, 3), (C_2, \{X/david\}, 1)\}\}$
are covering for $strong_itching(david)$ but $K_3 = \{\kappa_1\}$ is not.

Two composite 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 $\sum_{T_\sigma \in U(\kappa_1) \cup U(\kappa_2)} P_\sigma = P_{\kappa_1} + P_{\kappa_2}$ A set of composite choices $K = \{\kappa_1, \ldots, \kappa_n\}$ is *mutually incompatible* if every couple of composite choices κ_i and κ_j of K is incompatible. For example, K_1 above is not mutually incompatible while K_2 is.

If K is mutually incompatible, then $\sum_{T_\sigma \in \bigcup_{i=1}^n U(\kappa_i)} P_\sigma = \sum_{i=1}^n P_{\kappa_i}$ If a set of explanations K is covering for a query Q and is mutually incompatible, then $P_T(Q) = \sum_{\kappa \in K} P_\kappa$. For the case of Example 2.2, $P_T(strong_itching(david)) = \sum_{\kappa \in K_2} P_\kappa = 0.3 + 0.5 \cdot 0.2 + 0.2 \cdot 0.2 = 0.44$ An LPAD is *(modularly) acyclic* if all of its instances are (modularly) acyclic.

3. SLG Resolution Algorithm

SLG resolution [5, 4, 6] is a partial deduction procedure for query evaluation under the well-founded semantics. SLG resolution repeatedly applies operations to a data structure obtaining a final structure that contains all the answers to the query. The most distinctive feature of SLG resolution in comparison with SLDNF resolution is its ability to avoid going into some infinite loops and the possibility of avoiding redundant computations. SLG achieves this by using tabling.

In this section we will describe SLG resolution using the formulation presented in [27] that is clearer and easier to understand than the original formulation in [6].

SLG resolution takes as input a query in the form of an atom and produces a derivation that is a sequence of forests of trees. In the following, *subgoal* will be a synonym for atom. Each forest is obtained from the previous one by applying an operation. The nodes of the trees have the form $AnswerTemplate : -DelaySet|GoalList$ that is called *X-clause*. In it, $AnswerTemplate$ is an atom that is used to store bindings for a subgoal that have been obtained during the derivation, $DelaySet$ is a set of literals that have been "delayed", i.e., whose evaluation has been suspended, while $GoalList$ is the list of literals yet to be selected for resolution. The selection of literals in $GoalList$ is performed using an arbitrary but fixed computations rule, such as "left to right".

Each tree in the forest has a root of the form $A : -|A$ where A is a subgoal. X-clauses are then resolved with clauses from the program to obtain new nodes for the trees. When a node with an empty $GoalList$ is found, we have an answer for the subgoal in the root node. Answers for a subgoal may be returned to other nodes in which the selected literal is built over that subgoal. When a positive literal is selected, if we have an answer for the literal, resolution is performed between the X-clause and the answer. When a ground negative literal $\neg A$ is selected, the literal is removed from the clause if A has

been completely evaluated and no answer has been found, as predicated by negation as failure. If A has been completely evaluated and has been found true, the node is considered as failed. However, in the presence of loops through negation, it may be necessary to proceed with the computation even if A has not yet been completely evaluated. In this case SLG resolution chooses to "delay" the selected literal, in the hope that its truth value can be ascertained later.

Definition 3.1. (X-Clause)
An *X-clause* is a clause of the form $AnswerTemplate :- DelaySet|GoalList$ where $AnswerTemplate$ is an atom, $DelaySet$ is a sequence of delayed literals (see Definition 3.3) and $GoalList$ is a sequence of literals. If $GoalList$ is empty, the X-clause is called an *answer* clause. If the $DelaySet$ of an answer is empty it is termed an *unconditional answer*, otherwise, it is a *conditional answer*.

Definition 3.2. (SLG Trees and Forest)
An *SLG forest* is a set of *SLG trees*. The nodes of an SLG tree are either an X-clause or *fail*. The latter form is called a *failure node*. The root node of an SLG tree may be marked with the token *completed*. In this case, we also say that the subgoal A in the root $A :- |A$ of the tree is *completed*.

We call a node N an *answer* (*unconditional answer, conditional answer*) when the corresponding X-clause is an answer (unconditional answer, conditional answer).

If an SLG forest $\mathcal{F}$ has an SLG tree with root $A :- |A$, we call it the *tree for* A and we say that A *belongs* to $\mathcal{F}$. If A belongs to $\mathcal{F}$, let $\mathcal{F}(A)$ be the SLG tree for A.

Definition 3.3. (Delayed Literals)
A *negative delayed literal* in the $DelaySet$ of a node N has the form $\neg A$ where A is a ground atom. *Positive delayed literals* have the form D^{Call}_{Answer}, where D is an atom whose truth value depends on the truth value of some answer $Answer$ for the subgoal $Call$. If θ is a substitution, then $(D^{Call}_{Answer})\theta = (D\theta)^{Call}_{Answer}$. A delayed literal of the form D^{Call}_{Answer} is *ground* if D is ground.

Delayed literals are used in order to store information regarding suspended computations so that they can be later simplified away. We now define resolution between an X-clause and an answer so that delayed literals are taken into account.

Definition 3.4. (SLG Answer Resolution)
Let N be a node of the form $A :- D|L_1, \ldots, L_n$ where $n > 0$ and let L_j be the selected atom. Let $Ans = A' :- D'|$ be an answer whose variables have been standardized apart from N. N is *SLG answer resolvable with* Ans if L_j and A' are unifiable with an mgu θ. The *SLG answer resolvent of* N *and* Ans *on* L_j has the form $(A :- D|L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_n)\theta$ if D' is empty and $(A :- D, \overline{D}|L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_n)\theta$ otherwise, where $\overline{D} = L_j$ if L_j is negative, and $\overline{D} = {L_j}^{L_j}_{A'}$ otherwise.

The following definition states when no more answers can be produced for a subgoal.

Definition 3.5. (Completely evaluated)
A set $\mathcal{A}$ of subgoals belonging to a forest $\mathcal{F}$ is *completely evaluated* if at least one of the following conditions holds for each $A \in \mathcal{A}$:

1. The tree for A contains an answer $A :- |$; or

2. For each node N in the tree for A:

   (a) The selected literal L of N is completed or in $\mathcal{A}$; or

   (b) There are no applicable NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN, DELAYING or NEGATIVE RETURN operations (see Definition 3.9).

In certain cases the propagation of delayed answers may lead to a set of unsupported answers, i.e. conditional answers of completely evaluated goals that can be removed.

Definition 3.6. (Supported answer)
Let $\mathcal{F}$ be an SLG forest, A a subgoal belonging to $\mathcal{F}$ and $Answer$ an atom that occurs in the head of some answer of A. Then *$Answer$ is supported by A in $\mathcal{F}$* if and only if:

1. A is not completely evaluated; or

2. There exists an answer node $Answer : -DelaySet|$ of A such that, for every positive delayed literal D_{Ans}^{Call}, Ans is supported by $Call$.

Therefore, $Answer$ is *unsupported by A in $\mathcal{F}$* if and only if A is completely evaluated and, for each answer node $Answer : -DelaySet|$ of A, there is a positive delayed literal D_{Ans}^{Call} such that Ans is unsupported by $Call$.

An SLG derivation consists of a possibly transfinite sequence of SLG forests. Since we consider only functor-free programs, an SLG derivation is a finite sequence of SLG forests.

Definition 3.7. (SLG Derivation)
Given a program T, an atomic query Q and a set of tabling operations (from Definition 3.9), an *SLG derivation $\mathcal{D}$ for Q in T* is a sequence of SLG forests $\mathcal{F}_0, \ldots, \mathcal{F}_n$ such that:

- $\mathcal{F}_0$ is the forest containing the only tree $Q : -|Q$.
- For each integer $m < n$, $\mathcal{F}_{m+1}$ is obtained by $\mathcal{F}_m$ by the application of an operation from Definition 3.9.

If no operation is applicable to $\mathcal{F}_n$, $\mathcal{F}_n$ is called a *final forest* of $\mathcal{D}$. If $\mathcal{F}_m$ contains a leaf node with a non-ground selected negative literal, the derivation is *floundered*. If a derivation $\mathcal{D}$ is not floundered and $\mathcal{F}_n$ is a final forest, we say that $\mathcal{D}$ is *complete* and that $\mathcal{F}_n$ is *complete*.

Definition 3.8. Given an SLG forest $\mathcal{F}$, an atom A is *successful* in $\mathcal{F}$ if the tree for A has an unconditional answer A. A is *failed* in $\mathcal{F}$ if A is completely evaluated in $\mathcal{F}$ and the tree for A contains no answer. A negative delayed literal $\neg D$ is *successful* (*failed*) in $\mathcal{F}$ if D is failed (successful) in $\mathcal{F}$. A positive delayed literal D_{Ans}^{Call} is *successful* in $\mathcal{F}$ if $Call$ has an unconditional answer $Ans : -|$ in $\mathcal{F}$ and is *failed* if $Call$ is completed and it has no answers.

In the following we define the set of operations that can be applied to SLG forests.

Definition 3.9. (SLG Operations)
Given a forest $\mathcal{F}_n$ of an SLG derivation for a query Q in a program T, $\mathcal{F}_{n+1}$ is produced by one of the following operations:

1. NEW SUBGOAL: Let $\mathcal{F}_n$ contain a non-root node $N = Ans :-DelaySet|G, GoalList$ where G is the selected literal A or $\neg A$. Assume $\mathcal{F}_n$ contains no tree with root A. Then add the tree $A :-|A$ to $\mathcal{F}_n$.

2. PROGRAM CLAUSE RESOLUTION: Let $\mathcal{F}_n$ contain a root node $N = A :-|A$ and let C be a program clause $Head :-Body$ such that $Head$ unifies with A with mgu θ. Assume that, in $\mathcal{F}_n$, N does not have a child $N_{child} = (A :-|Body)\theta$. Then add N_{child} as a child of N.

3. POSITIVE RETURN: Let $\mathcal{F}_n$ contain a non-root node N whose selected literal L is positive. Let Ans be an answer node for L in $\mathcal{F}_n$ and N_{child} be the SLG answer resolvent of N and Ans on L. Assume that, in $\mathcal{F}_n$, N does not have a child N_{child}. Then add N_{child} as a child of N.

4. NEGATIVE RETURN: Let $\mathcal{F}_n$ contain a leaf node $N = Ans :-DelaySet|\neg A, GoalList$ whose selected literal $\neg A$ is ground. Then apply one of the following operations:

   (a) NEGATION SUCCESS: If A is failed in $\mathcal{F}_n$, then let N_{child} be $Ans :-DelaySet|GoalList$ and add N_{child} as a child for N.

   (b) NEGATION FAILURE: If A is succeessful in $\mathcal{F}_n$, then create a child for N of the form *fail*.

5. DELAYING: Let $\mathcal{F}_n$ contain a leaf node $N = Ans :-DelaySet|\neg A, GoalList$ such that A is ground but A is neither successful nor failed in $\mathcal{F}_n$. Then create a child for N of the form $Ans :-DelaySet, \neg A|GoalList$

6. SIMPLIFICATION: Let $\mathcal{F}_n$ contain a leaf node $N = Ans :-DelaySet|$, and let $L \in DelaySet$:

   (a) if L is failed in $\mathcal{F}_n$, then create a child *fail* for N;

   (b) if L is successful in $\mathcal{F}_n$, then let $N_{child} = Ans :-DelaySet'|$ where $DelaySet' = DelaySet - L$. If N does not have a child N_{child}, then add N_{child} as a child of N.

7. COMPLETION: Given a completely evaluated set $\mathcal{A}$ of subgoals (Definition 3.5), mark the roots of the trees for all subgoals in $\mathcal{A}$ as completed.

8. ANSWER COMPLETION: Given a set of unsupported answer $\mathcal{UA}$, add a failure node as a child of each answer $Ans \in \mathcal{UA}$.

Let us illustrate the various operations by means of an example.

Example 3.1. Consider an extension of the 2-person game from Example 2.1 in which a position X is winning for a player if there is a move from X to a position Y that is not winning for the opponent and Y satisfies a certain property p:

$C_1 = win(X) :-move(X, Y), \neg win(Y), p(Y).$

Suppose $move$ and p have the following definitions:

$C_2 = move(a, b).$ $C_3 = move(b, a).$ $C_4 = move(a, c).$

$C_5 = p(b).$ $C_6 = p(c).$

Suppose the query is $win(a)$ and that the literals are selected in the body of clauses from left to right. Figure 1 shows the forest that is built by SLG. In it, the number associated to nodes indicate the step at which the node is added.

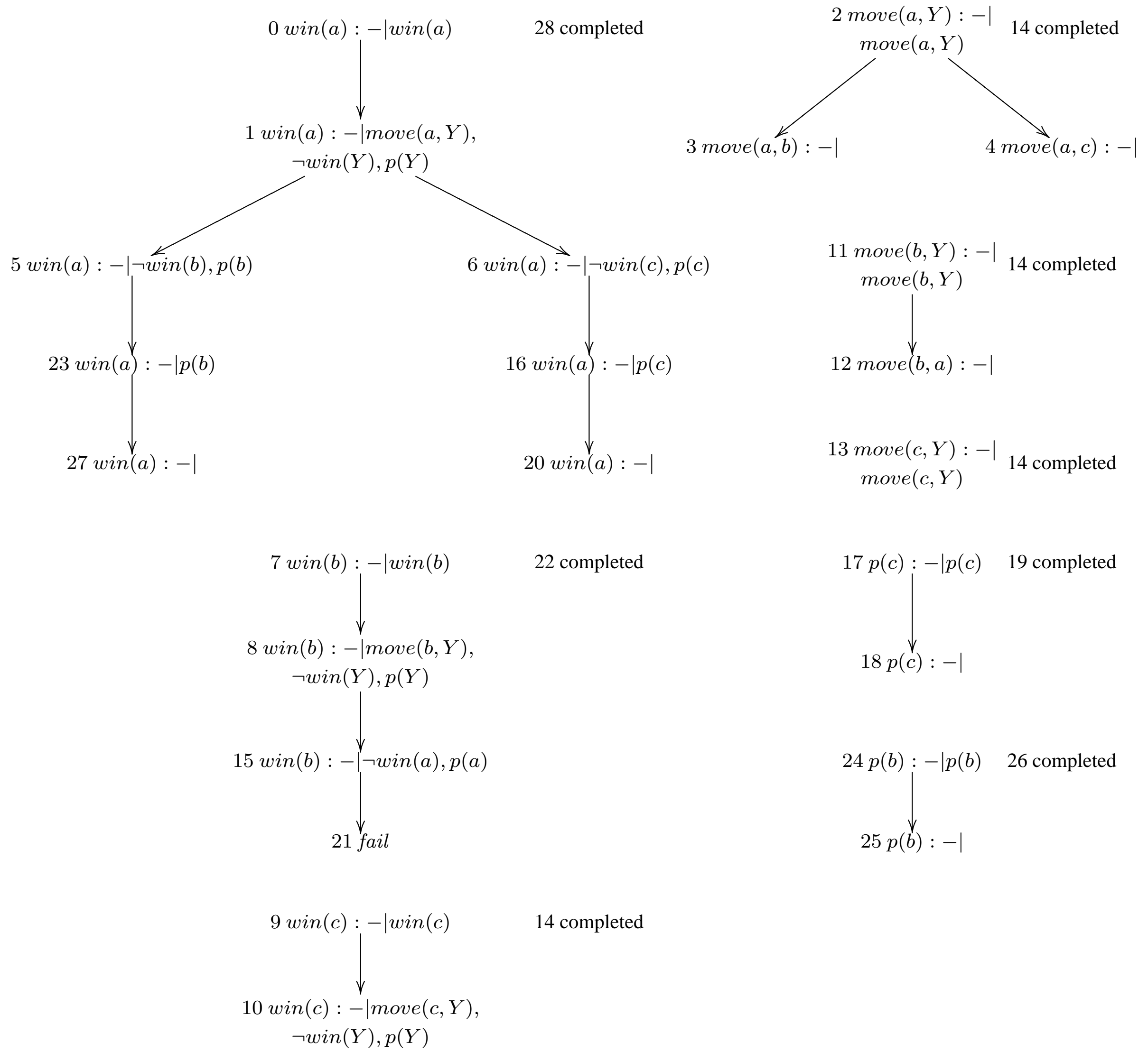


Figure 1. SLG derivation tree for Example 2.3.

In step 0, the tree for $win(a)$ is created with root node $win(a) :- | win(a)$. Then PROGRAM CLAUSE RESOLUTION is applied with clause C_1 obtaining the only child $win(a) :- | move(a, Y), \neg win(Y), p(Y)$ of $win(a) :- | win(a)$. In step 2 NEW SUBGOAL is applied creating a new tree for $move(a, Y)$ with root node $move(a, Y) :- | move(a, Y)$. In steps 3 and 4 PROGRAM CLAUSE RESOLUTION is applied with clauses C_2 and C_4 respectively, producing the two children of $move(a, Y) :- | move(a, Y)$. These are answers for the subgoal $move(a, Y)$. At this point POSITIVE RETURN can be applied twice to $win(a) :- | move(a, Y), \neg win(Y), p(Y)$ obtaining the two children marked with 5 and 6. In step 7 NEW SUBGOAL is applied creating the root node $win(b) :- | win(b)$. The application of PROGRAM CLAUSE RESOLUTION leads to node $win(b) :- | move(b, Y), \neg win(Y), p(Y)$.

In step 9 NEW SUBGOAL is applied to node $win(a) :- | \neg win(c), p(c)$ obtaining the new root node $win(c) :- | win(c)$. By PROGRAM CLAUSE RESOLUTION we get $win(c) :- | move(c, Y), \neg win(Y), p(Y)$.

Then NEW SUBGOAL is applied to $win(b) :- | move(b, Y), \neg win(Y), p(Y)$ leading to root node $move(b, Y) :- | move(b, Y)$ that is resolved with clause C_3 by PROGRAM CLAUSE RESOLUTION. In step 13 NEW SUBGOAL is applied to $win(c) :- | move(c, Y), \neg win(Y), p(Y)$ obtaining the root node $move(c, Y) :- | move(c, Y)$. At this point the subgoals $move(a, Y), move(b, Y), move(c, Y)$ and $win(c)$ are completely evaluated, the first three because no operation is applicable and the fourth because the selected literal of the only child in its tree is $move(c, Y)$. Therefore COMPLETION is applied and the root of the trees for these subgoals are marked as completed.

In step 15, the operation POSITIVE RETURN is applied to node $win(b) :- | move(b, Y), \neg win(Y), p(Y)$ with answer $move(b, a) :- |$ leading to $win(b) :- | \neg win(a), p(a)$. In step 16 NEGATIVE RETURN is applied to node $win(a) :- | \neg win(c), p(c)$ and, since $win(c)$ is failed, the node $win(a) :- | p(c)$ is obtained.

By NEW SUBGOAL the root node $p(c) :- | p(c)$ is added. Then PROGRAM CLAUSE RESOLUTION is applied obtaining the answer $p(c) :- |$. In step 19 $p(c)$ can be completed. The answer $p(c) :- |$ is then used by POSITIVE RETURN on clause $win(a) :- | p(c)$ leading to the answer $win(a) :- |$.

In step 21 NEGATIVE RETURN adds the child *fail* to $win(b) :- | \neg win(a), p(a)$. $win(b)$ can now be completed because there is no applicable operation.

The child $win(a) :- | p(b)$ is produced from $win(a) :- | \neg win(b), p(b)$ by NEGATIVE RETURN since $win(b)$ is failed. In step 24 NEW SUBGOAL adds the root node $p(b) :- | p(b)$ which is then resolved with C_5 leading to the answer $p(b) :- |$. $p(b)$ can now be completed since there is no applicable operation.

In step 27 the answer $p(b) :- |$ is returned to $win(a) :- | p(b)$ by POSITIVE RETURN obtaining the answer $win(a) :- |$. At this point $win(a)$ can be completed leading to the final forest shown in Figure 1.

The operations that are not illustrated in this example, namely DELAYING, SIMPLIFICATION and ANSWER COMPLETION, are those that deal with delayed literals. When a negative literal $\neg B$ is selected in an active clause and it is neither successful nor failed, it is moved to the set of delayed literals with a DELAYING operation. Later, if and when the truth value of B becomes known, a SIMPLIFICATION operation is applied. Example 4.1 in Section 4 will show an application of the DELAYING operation.

SLG resolution is sound with respect to the well-founded semantics.

Theorem 3.1. (Theorem 5.8 in [6])
Let T be a finite program, R be an arbitrary but fixed computation rule, Q be an atomic query and $\mathcal{F}$ be a final forest for Q that is complete. Then, for every subgoal A in the root of a tree in $\mathcal{F}$ and every ground

instance B of A:

- $B \in WF(T)$ if and only if B is an instance of the head of an unconditional answer of A in $\mathcal{F}$, and
- $\neg B \in WF(T)$ if and only if B is not an instance of the head of any answer of A in $\mathcal{F}$.

Moreover, SLG resolution is search space complete with respect to the well-founded semantics [6]: if the terms appearing in the forests of a derivation do not grow indefinitely, then a final forest is always achieved. For the case of functor-free programs, all the terms appearing in a forest have size 1 so SLG resolution always terminates.

4. SLGAD Resolution Algorithm

In this section we present *SLGAD resolution* (Linear resolution with Selection function for General logic programs with Annotated Disjunctions) that extends SLG resolution for dealing with LPADs. In the following, let T be an LPAD.

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

Definition 4.1. (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 *answer*; if D and B are empty, the XD-clause is called an *unconditional answer*, if B is empty and D is not empty, the XD-clause is called a *conditional answer*.

In SLGAD, SLG forests and trees are replaced by SLGAD systems, forests and trees.

Definition 4.2. (SLGAD Systems, Forests and Trees)
An *SLGAD system* $\mathcal{S}$ is a couple $(\mathcal{F}, \kappa)$ where $\mathcal{F}$ is an SLGAD forest and κ is a composite choice. An *SLGAD forest* is a set of *SLGAD trees*.

The root node of an SLGAD tree is an X-clause of the form $A : -|A$ while the other nodes are either XD-clauses or *fail*. The second form is called a *failure node*. The root node of an SLGAD tree may be marked with the token *completed*.

We call a node N an *answer* (*unconditional answer, conditional answer*) when the corresponding XD-clause is an answer (unconditional answer, conditional answer).

If an SLGAD system $\mathcal{S}$ (forest $\mathcal{F}$) has an SLGAD tree with root $A : -A$, we call it the *tree for* A and we say that A *belongs* to $\mathcal{S}$ ($\mathcal{F}$). If A belongs to $\mathcal{F}$, let $\mathcal{F}(A)$ be the SLG tree for A.

Given an SLGAD tree $\mathcal{F}(A)$ and a set of LPAD clauses $\mathcal{C}$, let $\mathcal{F}(A) \cap \mathcal{C}$ be the tree containing only the nodes (X, C, θ, i) such that $C \in \mathcal{C}$. Given an SLGAD forest $\mathcal{F}$ (tree $\mathcal{F}(A)$), let $s(\mathcal{F})$ ($s(\mathcal{F}(A))$) be the SLG forest (tree) obtained by replacing each XD-clause (X, C, θ, i) with the X-clause X. Given an SLGAD system $\mathcal{S} = (\mathcal{F}, \kappa)$, let $s(\mathcal{S}) = s(\mathcal{F})$.

The resolution between a root node $A : -|A$ and a program clause in PROGRAM CLAUSE RESOLUTION is replaced by SLGAD goal resolution.

Definition 4.3. (SLGAD Goal Resolution)
Let $A : -|A$ be an X-clause and let C be a clause of T such that A is unifiable with an atom H_i' in the head of C', where C' is a variant of C with variables renamed so that A and C' have no variables in common.

We say that $A : -|A$ 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 mgu 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.

SLG answer resolution between an X-clause and an answer X-clause in POSITIVE RETURN is replaced by SLGAD answer resolution.

Definition 4.4. (SLGAD Answer Resolution)
Let G be an XD-clause $(A : -D|L_1, \ldots, L_n, C, \theta, i)$ with $n > 0$, and let L_j be the selected atom. Let $Ans = (A' : -D'|, E', \theta', i')$ be an answer XD-clause whose variables have been standardized apart from G. If L_j and A' are unifiable with an mgu δ then we say that G is *SLGAD answer resolvable with Ans*. The *SLGAD answer resolvent of G with Ans* on L_j has the form $((A : -D|L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_n)\delta, C, \theta\delta, i)$ if D' is empty, and $((A : -D, \overline{D}|L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_n)\delta, C, \theta\delta, i)$ otherwise, where $\overline{D} = L_j$ if L_j is negative and $\overline{D} = L_j{}^{L_j}_{A'}$ otherwise.

The definitions of delayed literals, completely evaluated set of subgoals and supported answer are a simple adaptation of those for SLG (definitions 3.3, 3.5 and 3.6).

We now provide a definition for an SLGAD derivation.

Definition 4.5. (SLGAD Derivation)
Given an LPAD T, a ground atomic query Q and a set of tabling operations (from Definition 4.6), an *SLGAD derivation $\mathcal{D}$ for Q in T* is a tree of SLGAD systems such that: 1) the root system $\mathcal{S}_0 = (\mathcal{F}_0, \kappa_0)$ is such that $\mathcal{F}_0$ contains a single tree $Q : -|Q$ and $\kappa_0 = \emptyset$; 2) the children of a system $\mathcal{S}_m$ are obtained from $\mathcal{S}_m$ by the application of one of the operations from Definition 4.6.

If no operation is applicable to a system $\mathcal{S}_n$, $\mathcal{S}_n$ is called a *final system* of $\mathcal{D}$. If $\mathcal{S}_n$ contains a leaf node with a non-ground selected negative literal, the derivation is *floundered*. If a derivation $\mathcal{D}$ is not floundered and a final system is reached in every branch of $\mathcal{D}$, we say that $\mathcal{D}$ is *complete* and we call *complete* also each final system.

The definition of successful and failed atom and of successful and failed delayed literal are a simple generalization of those for SLG (Definition 3.8).

SLGAD resolution is defined by a number of operations that are applied to SLGAD systems to produce one or more new systems. The initial system $\mathcal{S}_0 = (\mathcal{F}_0, \kappa_0)$ is such that $\mathcal{F}_0$ is obtained as in SLG resolution while κ_0 is empty.

Definition 4.6. (SLGAD Operations)
SLGAD resolution contains the same operations of SLG.

NEW SUBGOAL, NEGATION FAILURE, DELAYING, SIMPLIFICATION, COMPLETION and ANSWER COMPLETION modify the SLGAD forest in the same way as they modify the SLG forest and leave the composite choice unchanged.

PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN and NEGATION SUCCESS are modified in the following way: if N is the node to which they are applied and $\mathcal{S}_m = (\mathcal{F}_m, \kappa_m)$ is the system to which N belongs, the generated clause N_{child} is tested to see if it is answer. If not, then N_{child} is added to N as a child if it is not already a child. Otherwise, let N_{child} be $(Ans : -DelaySet|, C, \theta, i)$ and let $A : -A$ be the root ancestor of N. Then one of the following operations is performed:

1. if $Ans :-|$ is already present in $\mathcal{F}_m(A)$ then add $fail$ as a child of N and leave κ_m unchanged; otherwise
2. if $(C, \theta, j) \in \kappa_m$ with $i \neq j$ then add *fail* as a child of N and leave κ_m unchanged; otherwise
3. if $(C, \theta, i) \in \kappa_m$ then add N_{child} as a child of N if it is not already present and leave κ_m unchanged; otherwise
4. generate h branches, one for each atom in the head of C. In the ith branch, add (C, θ, i) to κ_m and N_{child} as a child of N. In the jth branch with $j \neq i$, add (C, θ, j) to κ_m and *fail* as a child of N.

SLGAD resolution for an atomic query Q proceeds by building a tree of systems until a final system is reached in every branch. When a new answer $Ans :-DelaySet|$ is found, SLGAD resolution checks for the presence of an unconditional answer $Ans :-|$ in $\mathcal{F}_m(Ans)$. If it is present, the current answer is redundant and the child *fail* is added. Otherwise, SLGAD resolution considers the atomic choice (C, θ, i) that originated the current answer. If there is already an atomic choice for $C\theta$ in κ_m, SLGAD resolution either fails the tree branch, in case a different head has been selected, or adds the answer to the tree and leaves κ_m unchanged. If $C\theta$ does not appear in κ_m, we have a branching: SLGAD resolution generates a different derivation branch for each atom in the head of $C\theta$. In the ith branch it adds the answer to the tree, while in the other branches it adds *fail*. Moreover, in the jth branch it adds the choice (C, θ, j) to the composite choice.

We will prove in Section 5 that, if T is range-restricted, each answer in an SLGAD forest is ground.

Let $L(Q)$ be the set of final systems of a complete SLGAD derivation, i.e., those associated to the leaves of the derivation tree. For each system in $L(Q)$, SLGAD resolution checks whether there are only conditional answers for Q. If so, SLGAD resolution returns the message “unsound” to the user.

Otherwise, SLGAD resolution builds the set $K(Q)$ of the composite choices of the systems in $L(Q)$ that contain the unconditional answer $Q :-|$ and returns the probability given by $\sum_{\kappa \in K(Q)} P_\kappa$.

Example 4.1. Let us now show the application of SLGAD resolution to the program of Example 2.3:
$C_1 = win(X) : 0.8 :- move(X, Y), \neg win(Y), p(Y).$
$C_2 = move(a, b).\ \ C_3 = move(b, a).\ \ C_4 = move(a, c).$
$C_5 = p(b).\ \ C_6 = p(c).$
Let the query be $win(a)$. The first 19 steps are the same as those of SLG resolution (see Example 2.1) with X-clauses replaced by XD-clauses, obtaining the system $\mathcal{S}_{19}$ shown in Figure 2 where the triples $(Clause, Substitution, Index)$ are omitted for definite clauses. The system $\mathcal{S}_{19}$ has an empty composite choice.

Then POSITIVE RETURN is applied to $(win(a) :-|p(c), C_1, \{X/a, Y/c\}, 1)$ with answer $p(c) :-|$. Since the result of the SLGAD answer resolution is the answer $win(a) :-|$, branching is performed obtaining two branches D_1 and D_2 shown respectively in Figures 3 and 4. In branch D_1, the child $(win(a) :-|, C_1, \{X/a, Y/c\}, 1)$ is obtained and $(C_1, \{X/a, Y/c\}, 1)$ is added to the composite choice of the system. In branch D_2 the child *fail* is obtained and $(C_1, \{X/a, Y/c\}, 2)$ is added to the composite choice of the system.

Let us now consider branch D_1. In step 21 NEGATIVE RETURN is applied obtaining the child *fail* of $(win(b) :-|\neg win(a), p(a), C_1, \{X/b, Y/a\}, 1)$. $win(b)$ can now be completed because there is no applicable operation.

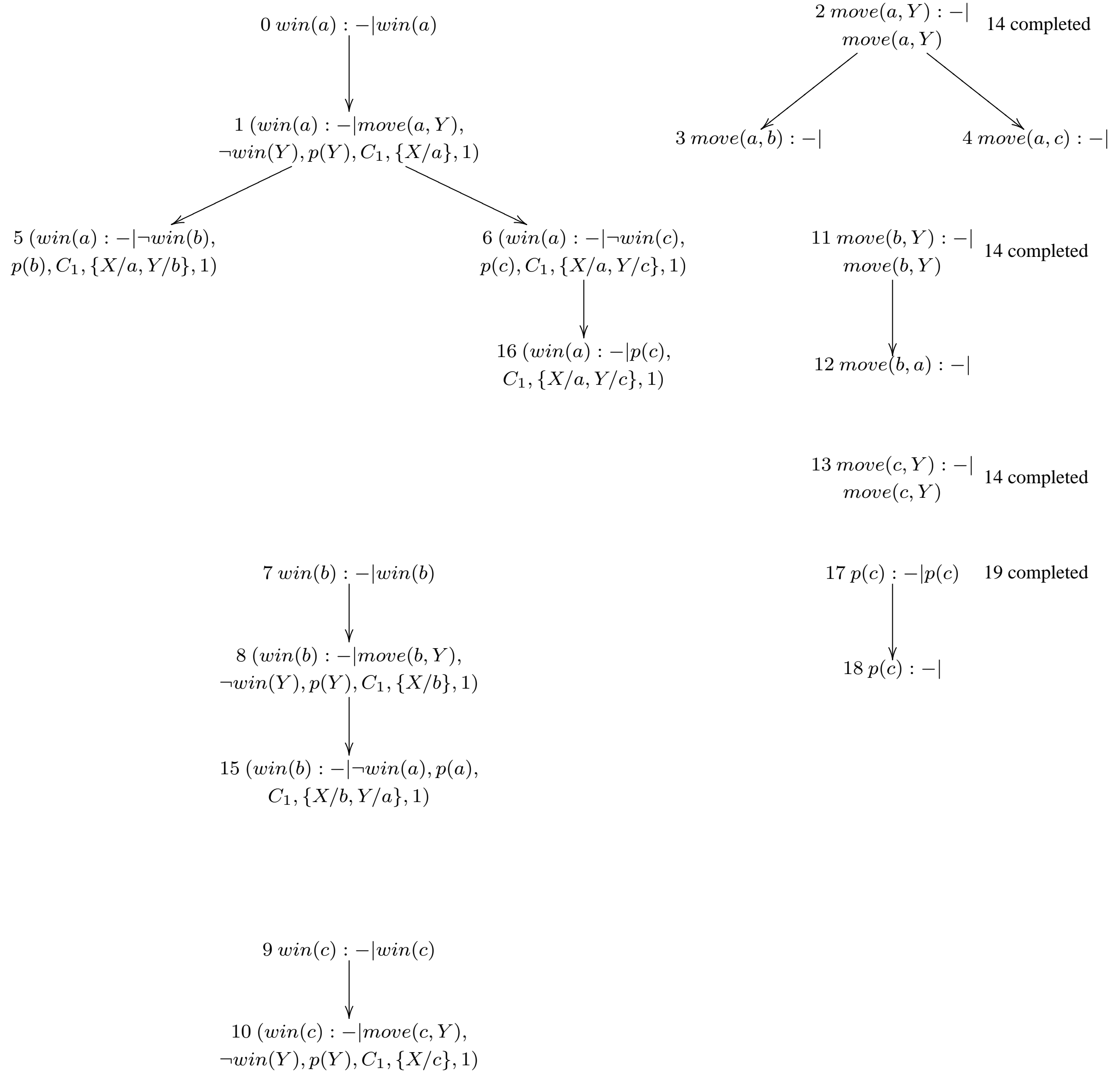


Figure 2. System $\mathcal{S}_{19}$ of the SLGAD derivation tree for Example 2.3.

NEGATIVE RETURN adds the child $(win(a) : -|p(b), C_1, \{X/a, Y/b\}, 1)$ since $win(b)$ is failed. NEW SUBGOAL adds the root node $p(b) : -|p(b)$ which is then resolved with C_5 leading to the answer $p(b) : -|$. $p(b)$ can now be completed since there is no applicable operation.

In step 27 POSITIVE RETURN is applied to $(win(a) : -|p(b), C_1, \{X/a, Y/b\}, 1)$. Since the result is an answer $win(a) : -|$ that is already present in the tree for $win(a)$, no branching is performed, the composite choice is left unaltered and the child *fail* is added. At this point $win(a)$ can be completed and the derivation along the branch ends.

In branch D_2, DELAYING is applied to $(win(b) : -|\neg win(a), p(a), C1, \{X/b, Y/a\}, 1)$ obtaining $(win(b) : -\neg win(a)|p(a), C1, X/b, Y/a, 1)$ in step 30.

Then, in step 31, by NEW SUBGOAL, the root $p(a) : -|p(a)$ is added and, since there is no applicable operation, it is marked as completed, together with $win(b)$. Since $win(b)$ is now failed, by NEGATIVE RETURN the node $(win(a) : -|p(b), C_1, \{X/a, Y/b\}, 1)$ is added.

In step 34 the root node $p(b) : -|p(b)$ is created by NEW SUBGOAL and is resolved with clause C_5 obtaining the answer $p(b) : -|$. COMPLETION is then applied to the set $\{p(b)\}$.

By POSITIVE RETURN applied to $(win(a) : -|p(b), C_1, \{X/a, Y/b\}, 1)$ we get an answer so branching is performed obtaining the branches $D_{2,1}$ and $D_{2,2}$.

In $D_{2,1}$, the answer $(win(a) : -|, C_1, \{X/a, Y/b\}, 1)$ is obtained and the atomic choice $(C_1, \{X/a, Y/b\}, 1)$ is added to composite choice. $win(a)$ can now be completed, the derivation along the branch ends and the forest in Figure 4 is obtained.

In $D_{2,2}$, $(win(a) : -|p(b), C_1, \{X/a, Y/b\}, 1)$ has child *fail* and the atomic choice $(C_1, \{X/a, Y/b\}, 2)$ is added to the composite choice. $win(a)$ can now be completed and the derivation ends. The forest that is obtained differs from the one in Figure 4 only in the composite choice and in the node associated to 37 which is *fail*.

The derivation tree that is built by SLGAD resolution for this example is shown in Figure 5. The set $L(Q)$ of final systems in the leaves of the tree is $\{\mathcal{S}_{28}, \mathcal{S}_{38}, \mathcal{S}_{40}\}$. None of these systems contains only conditional answers for $win(a)$ so "unsound" is not returned to the user. $win(a)$ is an answer in $\mathcal{S}_{28}$ and $\mathcal{S}_{38}$ but not in $\mathcal{S}_{40}$ so $K(Q) = \{\kappa_{28}, \kappa_{38}\}$ and $P_T(win(a)) = P_{\kappa_{28}} + P_{\kappa_{38}} = 0.8 + 0.2 \cdot 0.8 = 0.96$.

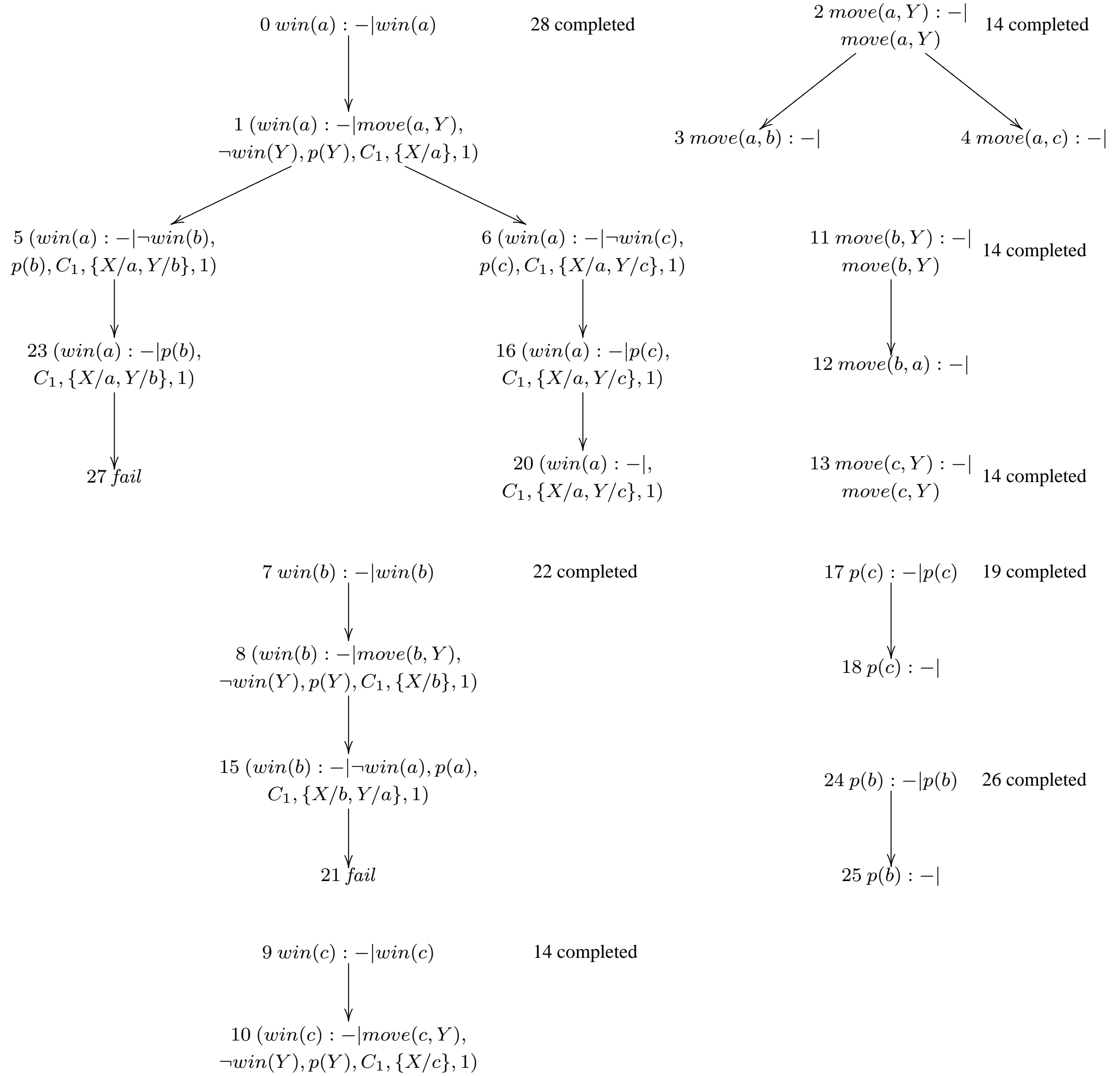
5. Proof of Soundness

The proof of soundness of SLDAG with respect to the LPAD semantics is based on Theorem 3.1. In order to prove the soundness we need the following definition and lemmas.

An XD-clause $H : -D|B$ is *range-restricted* if all the variables appearing in H also appear in positive literals of B. An answer XD-clause $A : -D|$ is *ground* if A and D are ground.

Lemma 5.1. Let T be a range-restricted LPAD, Q be a ground atom and $\mathcal{D}$ be an SLGAD derivation for Q in T that is not floundered. Then all nodes in every system of each SLGAD derivation branch for Q are range-restricted. Moreover, all answers in every system are ground.

Proof:
We will prove this lemma by induction on the systems in each derivation branch. For $n = 0$, $\mathcal{S}_0$ contains the only node $Q : -|Q$ which is range-restricted.



$\kappa = \{(C_1, \{X/a, Y/c\}, 1)\}$

Figure 3. Branch D_1 of the SLGAD derivation for Example 2.3.

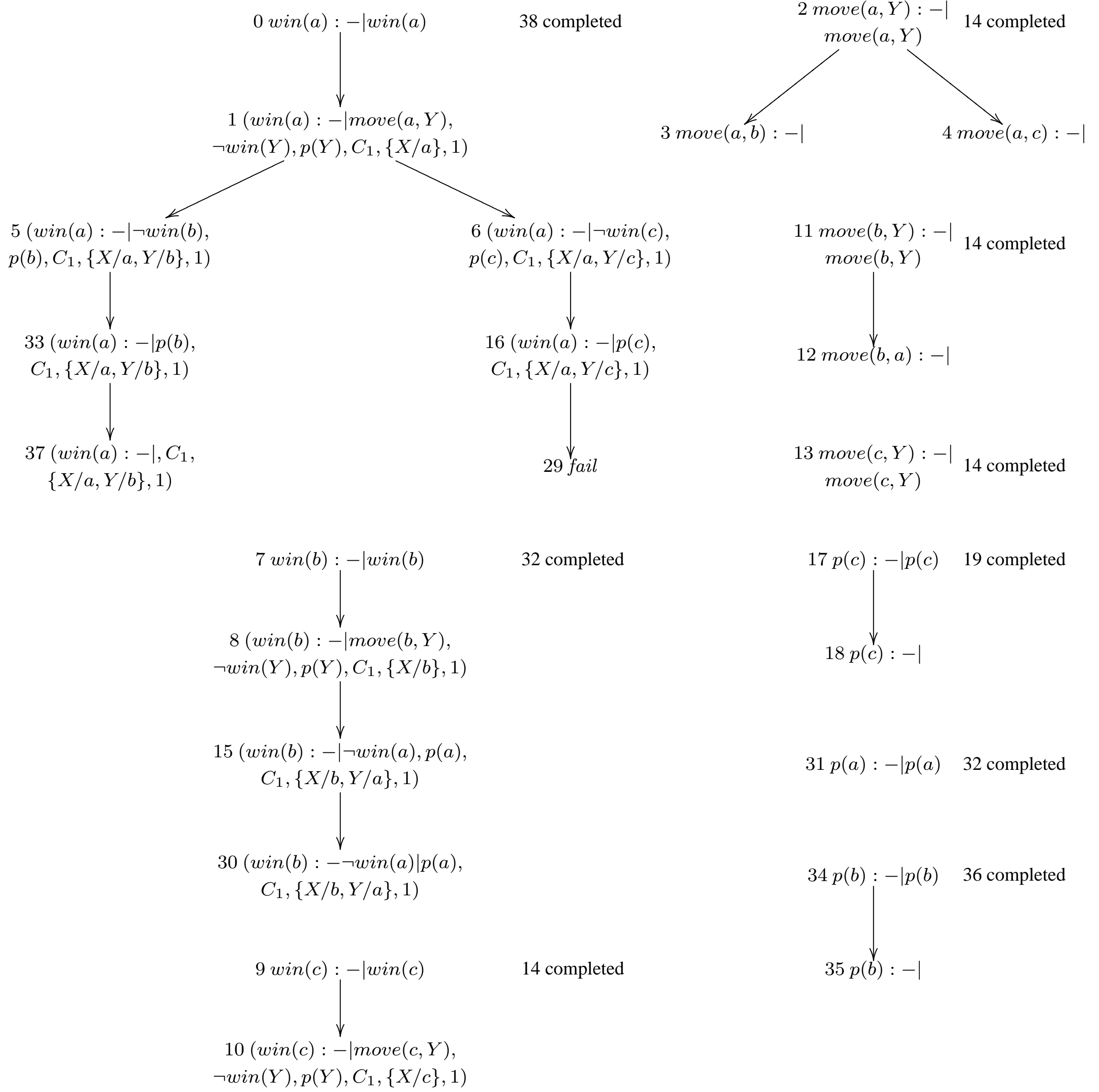


Figure 4. Branch $D_{2,1}$ of the SLGAD derivation for Example 2.3.

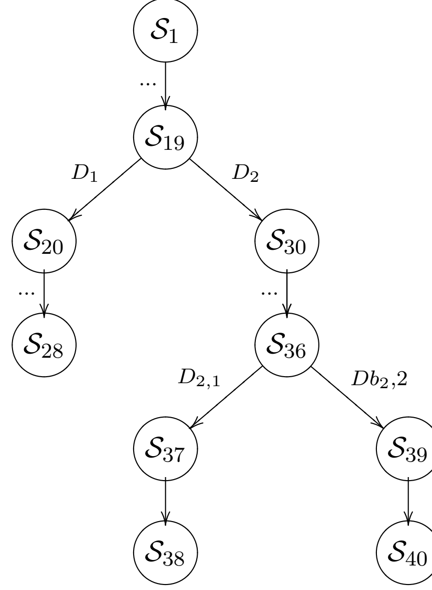


Figure 5. SLGAD derivation tree for Example 2.3.

In the inductive case, we will prove that all the operations that can be applied to $\mathcal{S}_{n-1}$ preserve the property. For PROGRAM CLAUSE RESOLUTION, since the node and the program clauses are range--restricted, the result of the resolution is range-restricted. Moreover, if an answer is obtained, it has an empty set of delayed literal and, since it is range-restricted, it is ground.

For NEW SUBGOAL we get a new tree whose root is range-restricted. COMPLETION and ANSWER COMPLETION do not have influence on the property. NEGATIVE RETURN deletes a literal from the body only if is ground. For POSITIVE RETURN, by the inductive hypothesis the answer $Ans = H : -D|$ is ground, therefore the child node has the property.

DELAYING moves a ground negative literal to the delay set, so it keeps the property.

SIMPLIFICATION removes a literal L from the delay set if it is successful. If L is negative, then it is ground. If L is a positive literal A_{Ans}^{Call}, $Call$ has an unconditional answer $Ans : -|$ that, by the inductive hypothesis, is ground. □

Lemma 5.2. If $G = (H : -D|B, C, \theta, i)$ appears anywhere in a system of a non-floundered SLGAD derivation branch for a ground atom Q, the variables appearing in $C\theta$ are those appearing in $H : -D|B$.

Proof:
We will prove this lemma by induction on the systems in a derivation branch. For $n = 0$, no XD-clause $G = (H : -D|B, C, \theta, i)$ appears in $\mathcal{S}_0$ so the property holds.

In the inductive case, if $\mathcal{S}_n$ is obtained by NEGATIVE RETURN, DELAYING, COMPLETION or ANSWER COMPLETION the property holds trivially because no variables are removed from $H : -D|B$. If PROGRAM CLAUSE RESOLUTION is applied to $\mathcal{S}_{n-1}$, for the definition of the operation the lemma holds. For POSITIVE RETURN, SLGAD answer resolution keeps the property because the mgu substitution δ

is composed with θ in the result of SLGAD answer resolution. For SIMPLIFICATION, if L is negative it must be ground so no variables are removed from $H:-D|B$. If L is D^{Call}_{Ans} and is successful, this means that $Ans:-|$ is an answer for $Call$ and D is an instance of Ans. Since answers are ground, so is D and no variables are removed from $H:-D|B$.

□

Lemma 5.3. If T is a range-restricted LPAD and (C,θ,i) belongs to the composite choice κ in a leaf of a branch of a non-floundered SLGAD derivation for a ground atom Q, then $C\theta$ is ground.

Proof:
Each triple (C,θ,i) is inserted into κ only in PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN or NEGATIVE RETURN that produce an answer XD-clause $G=(H:-D|,C,\theta,i)$. Since $H:-D|$ is ground, by Lemma 5.1, $C\theta$ is ground by Lemma 5.2. □

Lemma 5.4. Given a ground atom Q and a ground LPAD T, for every complete SLG derivation for Q in an instance T_σ with final forest $\mathcal{F}$, there exists a branch of a complete SLGAD derivation for Q in T with final system $(\mathcal{F}',\kappa)$ such that $\kappa\subseteq\sigma$ and the set of answers for Q in $\mathcal{F}$ and $\mathcal{F}'$ is the same. Vice-versa, for every branch of a complete SLGAD derivation for Q in T with final system $(\mathcal{F}',\kappa)$, there exists a selection σ with $\kappa\subseteq\sigma$ and a complete SLG derivation for Q in T_σ with final forest $\mathcal{F}$ such that the set of answers for Q in $\mathcal{F}'$ and $\mathcal{F}$ is the same.

Proof:
Consider the first part. Let $\mathcal{D}$ be a complete SLG derivation for Q in T_σ and let R_{n-1} be the operation applied to forest $\mathcal{F}_{n-1}$ in $\mathcal{D}$ to get forest $\mathcal{F}_n$. We will build a sequence of systems D that is a branch of the SLGAD derivation for Q in T and in which operation R'_{n-1} is applied to the system $\mathcal{S}_{n-1}$ to get $\mathcal{S}_n$.

We will prove by induction that, for every $\mathcal{F}_n\in\mathcal{D}$, $\mathcal{S}_n=(\mathcal{F}'_n,\kappa_n)\in D$ is such that $s(\mathcal{F}'_n)=\mathcal{F}_n$ and $\kappa_n\subseteq\sigma$. For $n=0$, let $\mathcal{F}_0$ be equal to $\mathcal{F}'_0$ and let $\kappa_0=\emptyset$. Suppose now that the property holds for $n-1$. If $\mathcal{F}_n$ is obtained by an operation different from PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN and NEGATION SUCCESS, the same operation is applied also to $\mathcal{S}_{n-1}$, producing a $\mathcal{S}_n$ such that $s(\mathcal{S}_n)=\mathcal{F}_n$.

If R_{n-1} is PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN or NEGATION SUCCESS and an answer is not obtained, the same operation is applied to $\mathcal{S}_{n-1}$ producing a $\mathcal{S}_n$ such that $s(\mathcal{S}_n)=\mathcal{F}_n$.

If an answer is obtained, suppose the answer is a descendant of a child of a root obtained by PROGRAM CLAUSE RESOLUTION with the clause $H_i(C):-Body(C)$ of T_σ. There exists one branch of the SLGAD derivation in which the answer is obtained as well, unless $(C,\emptyset,j)\in\kappa_{n-1}$ with $j\neq i$, but this case can be ruled out because $\kappa_{n-1}\subseteq\sigma$, the clause $(H_i(C):-body(C))$ is in T_σ and σ is consistent.

Since $s(\mathcal{F}'_n)=\mathcal{F}_n$ for all n, the final system $(\mathcal{F}',\kappa)$ of D contains every answer for Q that the final forest $\mathcal{F}$ of $\mathcal{D}$ contains.

Consider the second part. Let D be a SLGAD derivation branch of Q in T, let R'_{n-1} be the operation applied to the system $\mathcal{S}_{n-1}$ in D to get system $\mathcal{S}_n$. Consider a selection σ such that $\sigma\supseteq\kappa$. We will build a derivation $\mathcal{D}$ that is a valid SLG derivation for Q in T_σ provided that we add a NO OP operation to SLG resolution that keeps the forest unaltered. Let $\mathcal{H}$ be the set of atoms that appear in the head of rules of T_σ.

We will prove by induction that, for every $\mathcal{S}_n\in D$ where $\mathcal{S}_n=(\mathcal{F}'_n,\kappa_n)$, $\mathcal{F}_n\in\mathcal{D}$ is such that $\mathcal{F}_n(A)=s(\mathcal{F}'{}_n(A)\cap T_\sigma)$ for every $A\in\mathcal{H}$ and that only clauses in T_σ are used in operations of type

PROGRAM CLAUSE RESOLUTION on $\mathcal{F}_n$. For $n = 0$, let $\mathcal{F'}_0 = \mathcal{F}_0$. Suppose now that the property holds for $n-1$. If $\mathcal{S}_n$ is obtained by an operation that is applied to a node $(H : -D|B, C, \emptyset, i)$ such that $(C, \emptyset, i) \notin \sigma$ let R_{n-1}=NO OP. In this case the property is kept.

Otherwise, if R'_{n-1} is PROGRAM CLAUSE RESOLUTION, suppose SLGAD goal resolution is performed with program clause C on head $H_i(C)$. If $(C, \emptyset, i) \notin \sigma$, let R_{n-1} =NO OP. If $(C, \emptyset, i) \in \sigma$, let R_{n-1} =PROGRAM CLAUSE RESOLUTION with the clause $H_i(C) : -Body(C)$. In both cases, the operation keeps the property.

If R'_{n-1} is applied to an XD-clause $(H : -D|B, C, \emptyset, i)$ such that $(C, \emptyset, i) \in \sigma$ let $R_{n-1} = R'_{n-1}$. If R'_{n-1} is POSITIVE RETURN, the answer that is used is obtained from an XD-clause $(H : -D|, C, \emptyset, i)$ such that $(C, \emptyset, i) \in \kappa$ so the operation keeps the property. Similarly for NEGATION SUCCESS.

If $\mathcal{S}_n$ is obtained by COMPLETION applied to a set $\mathcal{A}$, then let R_{n-1} be COMPLETION with the set $\mathcal{A} \cap \mathcal{H}$. If the tree for $A \in \mathcal{A} \cap \mathcal{H}$ contains an answer $A : -|$ in $\mathcal{F}'_{n-1}(A)$, then $\mathcal{F}_{n-1}(A)$ also contains the answer $A : -|$. Otherwise, consider an XD-clause G in $\mathcal{F}'_{n-1}(A)$ and let L be the selected atom.

If $L \in \mathcal{H}$, then the tree for L belongs to $\mathcal{F}_{n-1}$. If L is completed in $\mathcal{S}_{n-1}$ or L is in $\mathcal{A}$, then L is completed in $\mathcal{F}_{n-1}$ or L is in $\mathcal{A} \cap \mathcal{H}$. If there are no applicable NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN, DELAYING or NEGATIVE RETURN operations to every node N in the tree for L in $\mathcal{F}'_{n-1}$, then these operations are not applicable as well to N in $\mathcal{F}_{n-1}$.

If $L \notin \mathcal{H}$, then there is no clause for L in T_σ, so L is completed in $\mathcal{F}_{n-1}$.

If $\mathcal{S}_n$ is obtained by ANSWER COMPLETION applied to a set $\mathcal{UA}$, then let R_{n-1} be ANSWER COMPLETION applied to the set $\mathcal{UA} \cap \mathcal{H}$.

Let us prove that $\mathcal{UA} \cap \mathcal{H}$ is unsupported in $\mathcal{F}_{n-1}$ by induction on the levels of unsupportedness: the atoms of level 0 are those that are completed and that have no answers, the atoms of level n are those that are completed and whose answer have an unsupported delayed literal L_L^L such that L belongs to levels $0, 1, \ldots$ or $n-1$. In the base case, A does not have any answer in $\mathcal{F}'_{n-1}$. This means that A is unsupported also in $\mathcal{F}_{n-1}$.

In the recursive case, for very answer $A : -DelaySet'|$, there exists a delayed literal L_L^L such that L is unsupported by L in $\mathcal{F}'_{n-1}$. If $L \in \mathcal{UA} \cap \mathcal{H}$, then L is unsupported by the inductive hypothesis. If $L \in \mathcal{UA} \setminus \mathcal{H}$, then L cannot be an answer, against the hypothesis that L_L^L belongs to the delay set.

So $\mathcal{D}$ is a valid SLG derivation in T_σ. Moreover, since $s(\mathcal{F}'_n(A) \cap T_\sigma) = \mathcal{F}_n(A)$ for all n and all $A \in \mathcal{H}$, then the final forest $\mathcal{F}$ of $\mathcal{D}$ contains every answer for Q that the final system $(\mathcal{F}', \kappa)$ of D contains. □

Lemma 5.5. Let T be a range-restricted LPAD, $g(T)$ be its grounding and Q be a ground atom. If the SLGAD derivation for Q in T is not floundered, the probabilities of Q returned by SLGAD resolution in T and in $g(T)$ are the same.

Proof:
Let $\mathcal{D}$ be an SLGAD derivation for Q in T and let $\mathcal{D}'$ be an SLGAD derivation for Q in $g(T)$. Since only ground answers are obtained both in $\mathcal{D}$ and in $\mathcal{D}'$ and branching is performed only when a new answer is obtained, there is a one to one correspondence between the branches of $\mathcal{D}$ and $\mathcal{D}'$. Moreover, the set of answers in a branch D of $\mathcal{D}$ is the same as that in the corresponding branch D' of $\mathcal{D}'$. □

Theorem 5.1. If T is a range-restricted LPAD, Q is a ground atom and the SLGAD derivation for Q in T is not floundered, then the derivation returns $P_T(Q)$.

Proof:
By lemmas 5.4 and 5.5, for every instance T_σ such that Q is true in T_σ, there exists an SLGAD derivation branch for Q whose final system has a composite choice κ such that $\sigma \supseteq \kappa$, so $K(Q)$ is covering.

Also by lemmas 5.4 and 5.5, for every composite choice κ such that $\kappa \in K(Q)$ and for every instances T_σ such that $\sigma \supseteq \kappa$, we have that Q is true in T_σ. So $K(Q)$ is a set of explanations for Q.

Moreover $K(Q)$ is mutually incompatible because, given two composite choices κ_1 and κ_2 in $K(Q)$, we can identify a ground clause $C\theta$ for which $(C, \theta, i) \in \kappa_1$, $(C, \theta, j) \in \kappa_2$ and $i \neq j$ by going back to the node of the derivation tree that is the least common ancestor of the leaves associated with κ_1 and κ_2. Therefore, the probability of the query can be obtained by summing the probability of all the composite choices in $K(Q)$. □

6. SLGAD Implementation

In this section we present a description of an algorithm for performing SLGAD resolution that is based on the SLG resolution algorithm described in [4].

SLGAD resolution, as SLG resolution, only specifies the set of possible operations, it does not specify in what order they should be applied. In particular, DELAYING delays ground negative literals so that the computation can proceed even if there are loops through negation. However, if we delay a literal too early, we may incur in the computation of irrelevant subgoal that can slow down the procedure. Moreover, COMPLETION has to detect subgoals that have been completely evaluated in order to resolve their negative counterparts.

In order to overcome these problems, [4] proposed an algorithm for keeping tracks of the dependencies among subgoals. The algorithm uses a number of global data structures for storing the state of the computation. It keeps a "table" $\mathcal{T}$ where it stores, for each subgoal A, the set of answers $Anss(A)$ found so far for A, a list of clauses $Poss(A)$ in which A is selected and that wait for its answers, a list of clauses $Negs(A)$ in which $\neg A$ is selected and that wait for its answers and a Boolean flag $Comp(A)$ that stores whether the subgoal has been completely evaluated. Moreover, it keeps a stack $\mathcal{S}$ on which new subgoals encountered during the search are pushed. The stack is used in order to keep track of the dependencies of the subgoals and to perform the COMPLETION operation correctly and efficiently.

The pseudocode for the SLGAD algorithm is very similar to that of SLG: it differs from it mainly because it adds non-deterministic choice points corresponding to cases in which a new answer is found by the operations PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN or NEGATION SUCCESS.

The main function of the algorithm is shown in Figure 6. It takes as input a ground atom Q and an LPAD 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, used to keep track of dependencies among subgoals. The fourth variable is specific to SLGAD and is a composite choice κ. We assume that the global variables are copied to the 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 SLGAD algorithm is composed of the same procedures as SLG plus procedure ADD_CHOICE that implements the operations specific to SLGAD resolution. We refer to [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 7) is called to take into account a new subgoal and differs from that of SLG because in line 3 each SLGAD goal resolvent is considered rather than each resolvent.

Figure 6. Procedure SLGAD

```
1  function SLGAD(Q,T)
2  begin
3      Initialize Count, 𝒯, 𝒮, DFN, PosMin and NegMin as in SLG;
4      κ := ∅;
5      let K(Q) be the set of all the values for κ after a call of
6          SLG_SUBGOAL(Q,PosMin,NegMin) such that 𝒯 contains Q as an answer;
8      if Q appears only in conditional answers in a derivation branch then return unsound;
9      else return Σ_{κ∈K(Q)} P_κ;
10 end;
```

Procedure SLG_NEWCLAUSE (see Figure 7) considers the selected literal and it 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 8) where a call to ADD_CHOICE is added in line 3.

ADD_CHOICE takes as input a subgoal A and an XD-clause G and returns a Boolean variable Derivable. If $G = (Ans : -D|, C, \theta, i)$, ADD_CHOICE checks whether the answer $Ans : -|$ is already contained in the table entry for A or if κ contains an atomic choice inconsistent with (C, θ, i). If so, it sets Derivable to false and leaves κ unchanged. Otherwise, it checks whether κ already contains (C, θ, i). If so, it sets Derivable to true and leaves κ unchanged. Otherwise it generates $|head(C)|$ search branches. In the jth branch, it adds (C, θ, j) to κ and, if $j = i$, it sets Derivable to true, otherwise it sets Derivable to false.

In SLG_ANSWER, if Derivable is set to true by ADD_CHOICE, G is added to the table as an answer. Otherwise nothing is done.

Procedure SLG_POSITIVE, that performs resolution on a positive literal, modifies the one of SLG by replacing SLG answer resolution with SLGAD answer resolution. The other SLG procedure are modified simply by replacing X-clauses with XD-clauses.

If the conditional probability of a ground atom Q given another ground atom E must be computed, rather then computing $P_T(Q \wedge E)$ and $P_T(E)$ separately, an optimization can be done: we first identify the explanations for E and then we look for the explanations for Q starting from an explanation for E, as shown in Figure 10.

7. Related Works

LPADs share with many other languages the basic approach for defining a probabilistic semantics: a theory in the language defines a probability distribution over normal logic programs and the probability of a query is given by the sum of the probabilities of the programs where the query is true. This approach was called "distribution semantics" in [24].

Other languages that follow a distribution semantics include: probabilistic logic programs [8], the Independent Choice Logic (ICL) [17], pD [11], PRISM [26] and ProbLog [9].

[8] introduced the distribution semantics for probabilistic logic programs. The paper discusses a functor-free language in which you can have normal clauses and probabilistic disjunctive clauses as in

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

LPADs. However, disjunctive clauses are restricted to have an empty body. The algorithm proposed for computing the probability of a query first finds all the explanations for the query and then computes the probability by using the inclusion-exclusion formula that returns the probability of a propositional formula given the probabilities of the individual propositions. However, the inclusion-exclusion formula works only for very small programs because it requires the computation of the probability of every possible conjunction of explanations.

ICL [17] allows normal clauses and disjunctive clauses with an empty body as probabilistic logic programs but it allows function symbols by defining the probability of a query in terms of its explanations rather than in terms of complete instances. However, the definite part of the program is required to be acyclic. ICL is equipped with a reasoning system called Ailog2 [18]. As [8], it first finds all the explanations for the query and then it computes the probability. It uses an iterative algorithm for making the explanations incompatible so that the probability can be computed as a sum of products. Even if the semantics of ICL has been defined for acyclic programs, it can be extended to modularly acyclic programs and Ailog2 already handles correctly such a semantics.

[30] showed that acyclic functor-free LPADs can be converted to ICL programs in a way that preserves the semantics. Thus inference on LPADs can be performed by first converting them to ICL and then using Ailog2. The semantics of ICL can be defined also for modularly acyclic programs and the mapping from LPADs can be applied also for modularly acyclic LPADs, so Ailog2 can be used for inference on this kind of LPADs.

pD [11] is a Datalog language very similar to LPADs, in which probability distributions are defined over the heads of rules. The proposed inference algorithm computes all the explanations for a goal and then uses the inclusion-exclusion formula for computing the probability of the disjunction of the

Figure 8. Procdure SLG_ANSWER

```
1  procedure SLG_ANSWER(A, G,PosMin,NegMin)
2  begin
3     ADD_CHOICE(A, 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 (B, H) ∈ Poss(A) and H′
10             is the SLGAD answer resolvent of H with G;
11          for each (B, H′) in L do begin
12             SLG_NEWCLAUSE(B,H’,PosMin,NegMin);
13          end;
14       end else begin /* G has a non empty delay */
15          if no other answer in Anss(A) has the same head as G does then
16          begin
17             let L be the list of all pairs (B, H′), where (B, H) ∈ Poss(A) and H′
18                is the SLGAD answer resolvent of H with G;
19             for each (B, H′) in L do begin
20                SLG_NEWCLAUSE(B, H′,PosMin,NegMin);
21             end;
22          end;
23       end;
24    end;
25 end;
```

Figure 9. Procedure ADD_CHOICE

```
1  procedure ADD_CHOICE(A, G,Derivable)
2  begin
3     let G be (Ans : −D|, C, θ, i);
4     if 𝒯 contains Ans : −| in Anss(A) or
5        (C, θ, j) ∈ κ with j ≠ i then begin
6        Derivable:= false;
7     end else begin
8        if (C, θ, i) ∈ κ then
9           Derivable:= true;
10       end else begin
11          choose an index j from {1, . . . , |head(C)|} (choice point);
12          if i = j then begin
13             Derivable:= true;
14          end else begin
15             Derivable:= false;
16          end
17          κ := κ ∪ {(C, θ, j)};
18       end
19    end
20 end
```

Figure 10. Procedure SLGAD_COND

```
1  procedure SLGAD_COND(Q, E,T)
2  begin
3     Initialize Count, 𝒯, 𝒮, DFN, PosMin and NegMin as in SLG;
4     κ := ∅;
5     let K(E) be the set of all the values for κ after a call of
6        SLG_SUBGOAL(E,PosMin,NegMin) such that 𝒯 contains E as an answer;
7     if E appears only in conditional answers in a derivation branch then return unsound;
8     else if Σ_{κ∈K(E)} P_κ = 0 then return undefined;
9     else begin
10       Initialize Count, 𝒯, 𝒮, DFN, PosMin and NegMin as in SLG;
11       let K(Q) be the set of all the values of κ after a call of
12       begin
13          pick a choice κ′ from K(E);
14          κ = κ′;
15          let K(Q) be the set of all the values for κ after a call of
16             SLG_SUBGOAL(Q,PosMin,NegMin) such that 𝒯 contains Q as an answer;
17       end;
18    if Q appears only in conditional answers in a derivation branch then return unsound;
19    else return P(Q|E) = (Σ_{κ∈K(Q)} P_κ) / (Σ_{κ∈K(E)} P_κ);
20 end;
```

explanations. As already noted for [8], this approach is infeasible in practice.

PRISM [26] is a language that follows the distribution semantics and assigns probabilities to ground facts. The algorithm proposed for inference requires the bodies of the rules for the same ground atom to be mutually exclusive, i.e., no couple of bodies can be true in the same instance, thus making it inapplicable to the problems considered in Section 8.

ProbLog [9] is a language in which each clause can be annotated with a probability p. A probability distribution over normal logic programs is defined by picking each clause with probability p and by leaving out the clause with probability $1 - p$. ProbLog differs from LPADs because an LPAD clause encodes more than two possibilities but especially because the selection is performed directly on the clauses of the program rather than on their grounding. [9] proposed an inference algorithm that first finds the explanations for queries and then computes the probability using Binary Decision Diagrams. In principle the ProbLog inference algorithm could be applied to LPAD but this would required the complete grounding of the LPAD which is too large for all but the smallest programs.

In order to avoid completely grounding an LPAD for performing inference, [21] proposed SLFNFAD that adopts an approach similar to the one of ProbLog to LPADs. SLDNFAD thus uses Binary Decision Diagrams for making the explanations incompatible. SLDNFAD is sound and complete for programs for which the Clark's completion semantics [7] and the well-founded semantics coincide, as for acyclic [1] and modularly acyclic programs [22].

SLGAD resolution generates explanations that are automatically mutually incompatible. Therefore, it can simply sum up the probabilities of individual explanations, differently from Ailog2, where an iterative algorithm is applied, and from SLDNFAD, where Binary Decision Diagrams are used.

[29] presents the CP-logic language that is syntactically a superset of LPADs but differs in the definition of the semantics: a CP-logic theory defines a probabilistic process that specifies a sequence of events in a way that respects a number of axioms on causal influence. [29] then shows that the CP-logic semantics, when it is defined, is equivalent to the LPAD semantics. However, there are LPADs that are not valid CP-logic theories, i.e., for which the CP-logic semantics is not defined.

8. Experiments

We tested SLGAD on some synthetic problems that were used as benchmarks for SLG [4, 3]: `win`, `lanc` and `ranc`. Moreover, we tested it on programs that encode games of dice similar to the one presented in [31].

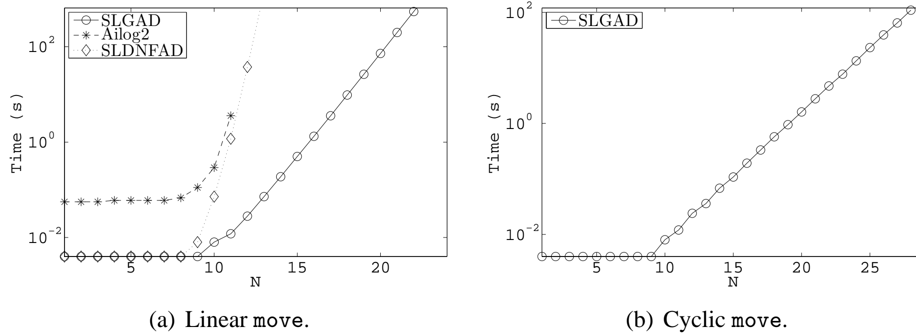
`win` is an implementation of the 2-person game of Example 2.3 and contains the clause

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

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

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

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



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

Figure 11. Execution times for `win`.

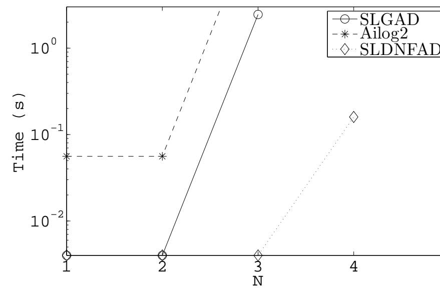


Figure 12. Execution times for `win` with tree `move`.

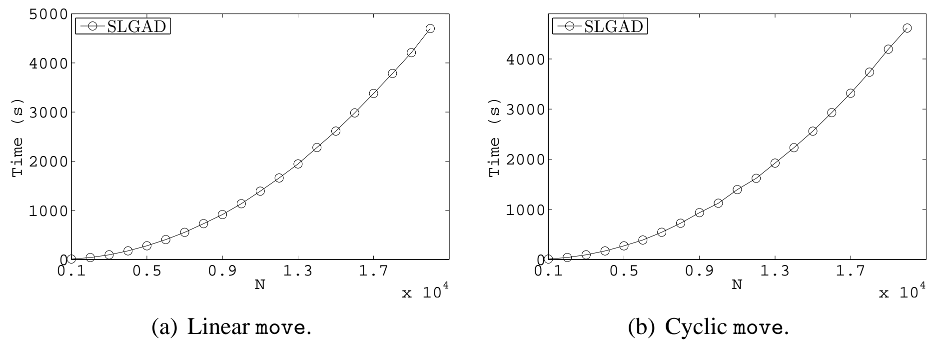
relation, that represents a complete binary tree of height N, containing $2^{N+1}-1$ tuples. For `win`, all the `move` relations are used, while for `lanc` and `ranc` only the linear ones.

SLGAD was compared with Ailog2 and SLDNFAD. For SLGAD and SLDNFAD we used the implementations in Yap Prolog[2] available in the `cplint` suite[3]. The SLGAD code was developed starting from the code of the SLG system[4]. For Ailog2 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 (2,333 MHz) processor and 4 GB of RAM. Yap version 6.0.0 was used in all cases except where indicated.

The execution times for the query `win(1)` to the `win` program are shown in Figures 11(a)[6], 11(b) and 12 as a function of N for linear, cyclic and tree `move` respectively. The time axis is logarithmic in these figures. Figures 13 and 14 show the execution time for the query `ancestor(1,N)` to the programs `lanc` and `ranc` respectively.

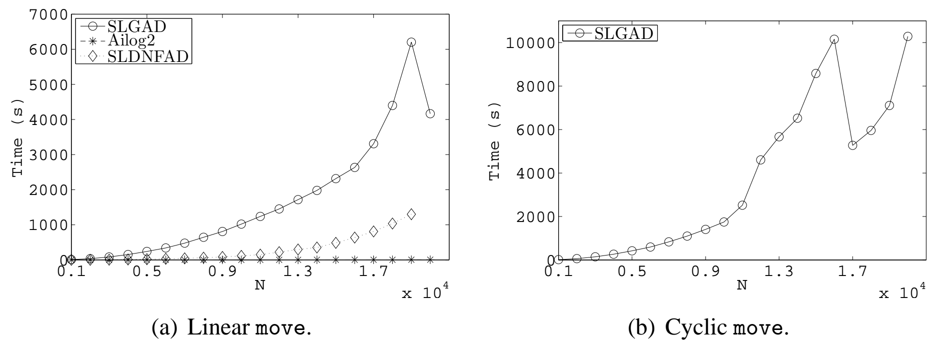
`win` has an exponential number of instances where the query is true and the graphs show the combinatorial explosion. On the ancestor dataset, there is only one instance where the query is true, the one obtained by always selecting the non-null head. In such an instance, a goal-oriented proof procedure

[2] `http://www.ncc.up.pt/~vsc/Yap/`
[3] `http://www.ing.unife.it/software/cplint/`, also included in the development version of Yap
[4] `http://engr.smu.edu/~wchen/slg.html`
[5] `http://www.cs.ubc.ca/~poole/aibook/code/ailog/ailog_man.html`
[6] Yap version 5.1.3 was used in this experiments for SLDNFAD because it gave better results than Yap version 6.0.0.



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

Figure 13. Execution times for `lanc`.



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

Figure 14. Execution times for `ranc`.

builds a search tree composed of a single 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 [2] in the SLG system). Each insert and lookup take logarithmic time.

SLGAD is compared with Ailog2 and SLDNFAD on 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 Ailog2 and SLDNFAD would go into an infinite loop. In `win` all the algorithm show the combinatorial explosion (see figures 11(a) and 12), with SLGAD performing better than Ailog2 and SLDNFAD for linear `move` and better than Ailog2 and worse than SLDNFAD for tree `move`. On `ranc` with linear `move` (Figure 14(a)), SLGAD takes longer than Ailog2 and SLDNFAD, with Ailog2 being particularly fast, probably due to the peculiarity of its incompatibility algorithm.

The peaks of SLGAD on `ranc` are due to an uneven behavior of garbage collection due to the fact that Yap 6.0.0 is still a development version.

SLGAD was tested also on programs encoding games of dice in which the player, starting from time 0, repeatedly throws a die at each time point and stops only when a certain subset of the faces comes up. We want to compute the probability that a certain face is obtained at a certain time point.

To model this problem, we use a predicate `on(T,F)` that states that the die was thrown at time `T` and face `F` was obtained. If we consider a three-sided die, the problem can be encoded with the following LPAD:

```
on(0,1):1/3 ; on(0,2):1/3 ; on(0,3):1/3.
on(T,1):1/3 ; on(T,2):1/3 ; on(T,3):1/3 :-
  T1 is T-1, T1>=0, on(T1,F), \+ on(T1,3).
```

The first clause states that, at time 0, one of the three faces is obtained with equal probability. The second clause states that, at time `T`, a face is obtained with equal probability if, at the previous time point, the die was thrown (`on(T1,F)`) and face 3 was not obtained (`\+ on(T1,3)`).

Note that this program uses integers and so its grounding is potentially infinite. For such programs the semantics of LPADs is not defined. To overcome this difficulty we generated ground programs from the one above by considering a finite set of integers, from 0 to N, and by pre-evaluating the built predicates in the body. So, for example, for $N = 2$ we get the program `die1`:

```
on(0,1):1/3 ; on(0,2):1/3 ; on(0,3):1/3.
on(1,1):1/3 ; on(1,2):1/3 ; on(1,3):1/3 :- on(0,F), \+ on(0,3).
on(2,1):1/3 ; on(2,2):1/3 ; on(2,3):1/3 :- on(1,F), \+ on(1,3).
```

In this way, we are able to answer only queries of the form `on(T,F)` with `T` smaller or equal to N.

We generated these programs for increasing values of N and, from each of them, we query the probability of `on(N,1)`. The execution times of SLGAD, SLDNFAD and Ailog2 for increasing values of N are shown in Figure 15, where the Y axis is logarithmic. In this case, SLGAD clearly outperforms the other systems, due to the use of tabling: when computing `on(T,1)`, the subgoal `on(T-1,F)` will be evaluated which will, in turn, compute `on(T-2,F)` and `on(T-2,3)`. Then `\+ on(T-1,3)` will be considered and resolved with the recursive clause giving a body of `on(T-2,F),\+ on(T-2,3)`: both of these subgoal have already been evaluated so their answers can be extracted from the table.

On the other hand, Ailog2 and SLDNFAD have to go back to time zero each time they evaluate `on(T-2,F),\+ on(T-2,3)`.

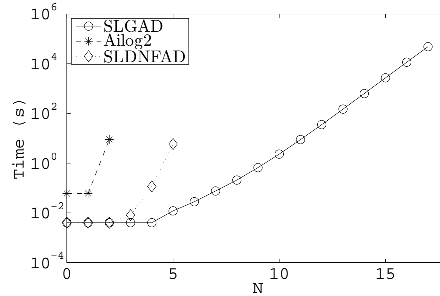
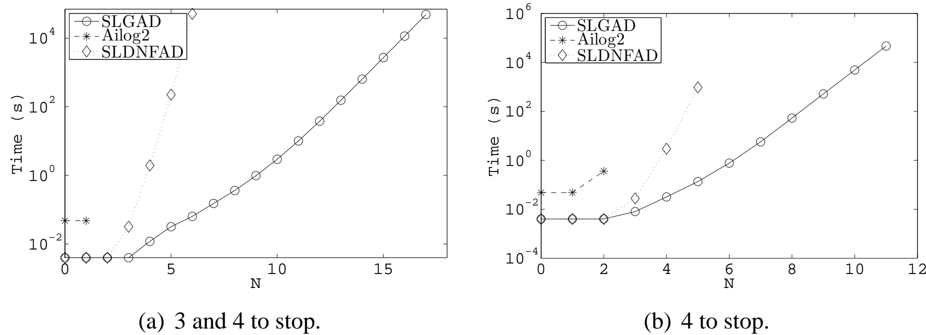


Figure 15. Execution times for a three sided die



(a) 3 and 4 to stop.

(b) 4 to stop.

Figure 16. Execution times for a four-sided die.

To further investigate the advantages of tabling, we considered two games in which a four-sided die is used. In the first, we stop as soon as we get 3 or 4 (`die2`), in the second we stop as soon as we get 4 (`die3`). The first program is

```
on(0,1):1/4 ; on(0,2):1/4 ; on(0,3):1/4 ; on(0,4):1/4.
on(T,1):1/4 ; on(T,2):1/4 ; on(T,3):1/4 ; on(T,4):1/4 :-
  T1 is T-1, T1>=0, on(T1,F), \+ on(T1,3), \+ on(T1,4).
```

Again, we generated the grounding of the programs for increasing values of N and we measured the time required to answer the query `on(N,1)`. Figure 16 shows the execution times of SLGAD, SLDNFAD and Ailog2 on the two problems. In this figure the Y axis is logarithmic These results confirm those of Figure 15.

9. Future Work

The version of SLG resolution that we have presented here is called $SLG_{variance}$ in [27] because, in the definition of the operations, it checks whether two formulas are variants of each other[7] to avoid redundant

[7] Two formulas are variants of each other if one can be obtained from the other by variable renaming only.

computations. In particular, the following operations are affected:

- NEW SUBGOAL adds a new tree if a tree for a variant of the selected literal is not already present;
- PROGRAM CLAUSE RESOLUTION and POSITIVE RETURN do not add a child node to the current node N if N has already a child that is a variant of the one to add;
- SIMPLIFICATION checks for success or failure of a variant of a delay literal and adds a child if a variant of it is not already present.

In [27] the author also proposes $SLG_{subsumption}$ that replaces the variance relation on atoms with a subsumption relation in order to avoid more redundant computations. Specifically:

- NEW SUBGOAL is applied only if the new subgoal is not subsumed by any subgoal in the forest;
- PROGRAM CLAUSE RESOLUTION and POSITIVE RETURN add a child node to the current node N only if N does not have a child that subsumes the one to add;
- subsumption can be employed also in SIMPLIFICATION to remove or fail a delay literal;
- COMPLETION can be applied to a subgoal if it is subsumed by a subgoal that is already completed.

At the moment SLGAD resolution, as $SLG_{variance}$, uses a variance relation to avoid redundant computations. In particular, NEW SUBGOAL is applied only if the new subgoal is not a variant of any subgoal in $\mathcal{F}_n$ and PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN and NEGATION SUCCESS, when a new answer $Ans:-|$ for a subgoal A is found, check for its presence in $\mathcal{F}_n(A)$.

In the future, we plan to improve SLGAD resolution by using subsumption instead of variance: for example, NEW SUBGOAL will not be applied if a subgoal that subsumes the one to be added is present. Moreover, in PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN and NEGATION SUCCESS, we will check whether $Ans:-|$ is already an answer for any subgoal in the SLGAD forest.

These two optimizations will avoid branching in cases in which it is not needed. The latter optimization will require to be able to access the table $\mathcal{T}$ on the basis of the answers present in the trees or the maintenance of a separate data structure for storing found answers.

Another optimization consists in completing ground subgoals as soon as an unconditional answer is found for them. In fact, new answers will not add any new information and, by completion, we can avoid branch exploration.

Other research directions for the future include the possibility of answering queries in an approximate way, similarly to what is done in [9] and the extension of the algorithm for considering also aggregates. Moreover, we plan to apply the techniques of [20] to LPADs and to investigate more closely the performances of SLGAD and the algorithms based on Binary Decision Diagrams.

10. Conclusions

Logic Programs with Annotated Disjunctions are a powerful language for representing probabilistic information in logic. However, all previously existing approaches for answering queries from LPADs are not able to deal with programs containing loops. Moreover, they run the risk of recomputing explanations for the same query.

For normal logic programs, SLG resolution uses tabling to avoid some loops involving positive and/or negative literals and to avoid recomputing answers for the same subgoal.

In this paper we have proposed an extension of SLG resolution, called SLGAD resolution, for performing inference on LPADs. SLGAD resolution is defined as a partial deduction approach in which a number of operations are repeatedly applied to a data structure called system. The operations are those of SLG resolution modified in order to take into account probabilistic disjunctive clauses. Once explanations for a goal have been obtained, the probability of the query can simply be computed by summing the probabilities of the individual explanations, since they are mutually incompatible by construction.

SLGAD resolution has been experimentally evaluated on a number of problems. On those that are modularly acyclic and right recursive, SLGAD has been compared with Ailog2 and SLDNFAD. The experimental results show that SLGAD is able to outperform the other systems when it is necessary to compute answers to queries more than once.

11. Acknowledgements

The author would like to thank Evelina Lamma, Paola Mello and Sergio Storari for many interesting discussions on the topics of this paper.

References

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

[2] Bratko, I.: *PROLOG Programming for Artificial Intelligence*, Addison-Wesley Longman, 1990.

[3] Castro, L. F., Swift, T., Warren, D. S.: Suspending and Resuming Computations in Engines for SLG Evaluation, *Practical Aspects of Declarative Languages*, 2257, Springer, 2002.

[4] 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.

[5] Chen, W., Warren, D. S.: Query Evaluation under the Well Founded Semantics, *Principles of Database Systems*, ACM Press, 1993.

[6] Chen, W., Warren, D. S.: Tabled Evaluation With Delaying for General Logic Programs, *J. ACM*, **43**(1), 1996, 20–74.

[7] Clark, K. L.: Negation as Failure, *Logic and Data Bases*, Plenum Press, 1977.

[8] Dantsin, E.: Probabilistic Logic Programs and their Semantics, *Russian Conference on Logic Programming*, 592, Springer, 1991.

[9] De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery., *International Joint Conference on Artificial Intelligence*, 2007.

[10] van Emden, M. H., Clark, K. L.: The logic of two-person games, in: *Micro-PROLOG: Programming in Logic*, Prentice-Hall, 1984, 320–340.

[11] Fuhr, N.: Probabilistic datalog: Implementing logical information retrieval for advanced applications, *J. of the Am. Soc. for Information Science*, **51**(2), 2000, 95–110.

[12] Gelder, A. V., Ross, K. A., Schlipf, J. S.: Unfounded Sets and Well-Founded Semantics for General Logic Programs, *Principles of Database Systems*, ACM Press, 1988.

[13] Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming, *International Conference on Logic Programming*, MIT Press, 1988.

[14] Jaeger, M.: Model-Theoretic Expressivity Analysis, *Probabilistic Inductive Logic Programming - Theory and Applications*, 4911, Springer, 2008.

[15] Jaeger, M., Lidman, P., Mateo, J. L.: Comparative Evaluation of PL languages, *Mining and Learning with Graphs*, 2007.

[16] Pearl, J.: *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 1988, ISBN 1558604790.

[17] Poole, D.: The Independent Choice Logic for Modelling Multiple Agents under Uncertainty, *Artif. Intell.*, **94**(1–2), 1997, 7–56.

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

[19] Przymusinski, T. C.: The Well-Founded Semantics Coincides with the Three-Valued Stable Semantics, *Fundam. Inform.*, **13**(4), 1990, 445–463.

[20] Riguzzi, F.: Extended Semantics and Inference for the Independent Choice Logic, *Log. J. IGPL*, , in press.

[21] Riguzzi, F.: A Top Down Interpreter for LPAD and CP-logic, *Congress of the Italian Association for Artificial Intelligence*, 4733, Springer, 2007.

[22] Ross, K. A.: Modular acyclicity and tail recursion in logic programs, *Principles of Database Systems*, ACM Press, 1991, ISBN 0-89791-430-9.

[23] Santos Costa, V., Page, D., Qazi, M., Cussens, J.: CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilistic Knowledge, *Uncertainty in Artificial Intelligence*, Morgan Kaufmann, 2003.

[24] Sato, T.: A Statistical Learning Method for Logic Programs with Distribution Semantics, *International Conference on Logic Programming*, MIT Press, 1995.

[25] Sato, T., Kameya, Y.: PRISM: A Language for Symbolic-Statistical Modeling, *International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1997.

[26] Sato, T., Kameya, Y.: Parameter Learning of Logic Programs for Symbolic-Statistical Modeling, *J. Artif. Intell. Res.*, **15**, 2001, 391–454.

[27] Swift, T.: A New Formulation of Tabled Resolution with Delay, *Portuguese Conference on Artificial Intelligence*, 1695, Springer, 1999.

[28] 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.

[29] Vennekens, J., Denecker, M., Bruynooghe, M.: Representing Causal Information about a Probabilistic Process, *European Conference on Logics in Artificial Intelligence*, 4160, Springer, September 2006.

[30] Vennekens, J., Verbaeten, S.: *Logic Programs With Annotated Disjunctions*, Technical Report CW386, K. U. Leuven, 2003.

[31] Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions, *International Conference on Logic Programming*, 3131, Springer, 2004.

[32] Zhang, N. L., Poole, D.: Exploiting Causal Independence in Bayesian Network Inference, *J. Artif. Intell. Res.*, **5**, 1996, 301–328.