Well-Definedness and Efficient Inference for Probabilistic Logic Programming under the Distribution Semantics

FABRIZIO RIGUZZI

ENDIF – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
E-mail: fabrizio.riguzzi@unife.it

TERRANCE SWIFT

CENTRIA – Universidade Nova de Lisboa
E-mail: tswift@cs.suysb.edu

submitted 22 November 2010; revised 15 April 2011; accepted 14 June 2011

Abstract

The distribution semantics is one of the most prominent approaches for the combination of logic programming and probability theory. Many languages follow this semantics, such as Independent Choice Logic, PRISM, pD, Logic Programs with Annotated Disjunctions (LPADs) and ProbLog.

When a program contains functions symbols, the distribution semantics is well-defined only if the set of explanations for a query is finite and so is each explanation. Well-definedness is usually either explicitly imposed or is achieved by severely limiting the class of allowed programs. In this paper we identify a larger class of programs for which the semantics is well-defined together with an efficient procedure for computing the probability of queries. Since LPADs offer the most general syntax, we present our results for them, but our results are applicable to all languages under the distribution semantics.

We present the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) that computes the probability of queries by transforming a probabilistic program into a normal program and then applying SLG resolution with answer subsumption. PITA has been implemented in XSB and tested on six domains: two with function symbols and four without. The execution times are compared with those of ProbLog, `cplint` and CVE. PITA was almost always able to solve larger problems in a shorter time, on domains with and without function symbols.

KEYWORDS: Probabilistic Logic Programming, Tabling, Answer Subsumption, Logic Programs with Annotated Disjunction, Program Transformation

1 Introduction

Many real world domains can only be represented effectively if we are able to model uncertainty. Accordingly, there has been an increased interest in logic languages representing probabilistic information, stemming in part from their successful use in Machine Learning. In particular, languages that follow the distribution semantics

(Sato 1995) have received much attention in the last few years. In these languages a theory defines a probability distribution over logic programs, which is extended to a joint distribution over programs and queries. The probability of a query is then obtained by marginalizing out the programs.

Examples of languages that follow the distribution semantics are Independent Choice Logic (Poole 1997), PRISM (Sato and Kameya 1997), pD (Fuhr 2000), Logic Programs with Annotated Disjunctions (LPADs) (Vennekens et al. 2004) and ProbLog (De Raedt et al. 2007). All these languages have the same expressive power as a theory in one language can be translated into another (Vennekens and Verbaeten 2003; De Raedt et al. 2008). LPADs offer the most general syntax as the constructs of all the other languages can be directly encoded in LPADs.

When programs contain functions symbols, the distribution semantics has to be defined in a slightly different way: as proposed in (Sato 1995) and (Poole 1997): the probability of a query is defined with reference to a covering set of explanations for the query. For the semantics to be well-defined, both the covering set and each explanation it contains must be finite. To ensure that the semantics is well-defined, (Poole 1997) requires programs to be acyclic, while (Sato and Kameya 1997) directly imposes the condition that queries must have a finite covering set of finite explanations.

Since acyclicity is a strong requirement ruling out many interesting programs, in this paper we propose a looser requirement to ensure the well-definedness of the semantics. We introduce a definition of *bounded term-size* programs and queries, which are based on a characterization of the Well-Founded Semantics in terms of an iterated fixpoint (Przymusinski 1989). A bounded term-size program is such that in each iteration of the fixpoint the size of true atoms does not grow indefinitely. A bounded term-size query is such that the portion of the program relevant to the query is bounded term-size. We show that if a query is bounded term-size, then it has a finite set of finite explanations that are covering, so the semantics is well-defined.

We also present the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) that builds explanations for every subgoal encountered during a derivation of a query. The explanations are compactly represented using Binary Decision Diagrams (BDDs) that also allow an efficient computation of the probability. Specifically, PITA transforms the input LPAD into a normal logic program in which the subgoals have an extra argument storing a BDD that represents the explanations for its answers. As its name implies, PITA uses tabling to store explanations for a goal. Tabling has already been shown useful for probabilistic logic programming in (Kameya and Sato 2000; Riguzzi 2008; Kimmig et al. 2009; Mantadelis and Janssens 2010; Riguzzi and Swift 2011). However, PITA is novel in its exploitation of a tabling feature called answer subsumption to combine explanations coming from different clauses.

PITA draws inspiration from (De Raedt et al. 2007), which first proposed to use BDDs for computing the probability of queries for the ProbLog language, a minimalistic probabilistic extension of Prolog; and from (Riguzzi 2007) which applied BDDs to the more general LPAD syntax. Other approaches for reasoning on LPADs include (Riguzzi 2008), where SLG resolution is extended by repeatedly branching on disjunctive clauses, and the CVE system (Meert et al. 2009) which transforms LPADs into an equivalent Bayesian network and then performs inference on the network using the variable elimination algorithm.

PITA was tested on a number of datasets, both with and without function symbols, in order to evaluate its efficiency. The execution times of PITA were compared with those of `cplint` (Riguzzi 2007), CVE (Meert et al. 2009) and ProbLog (Kimmig et al. 2011). PITA was able to solve successfully more complex queries than the other algorithms in most cases and it was also almost always faster both on datasets with and without function symbols.

The paper is organized as follows. Section 2 illustrates the syntax and semantics of LPADs over finite universes. Section 3 discusses the semantics of LPADs with function symbols. Section 4 defines dynamic stratification for LPADs, provides conditions for the well-definedness of the LPAD semantics with function symbols, and discusses related work on termination of normal programs. Section 5 gives an introduction to BDDs. Section 6 briefly recalls tabling and answer subsumption. Section 7 presents PITA and Section 8 shows its correctness. Section 9 discusses related work. Section 10 describes the experiments and Section 11 discusses the results and presents directions for future works.

2 The Distribution Semantics for Function-free Programs

In this section we illustrate the distribution semantics for function-free program using LPADs as the prototype of the languages following this semantics.

A *Logic Program with Annotated Disjunctions* (Vennekens et al. 2004) consists of a finite set of annotated disjunctive clauses of the form

$$H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n \leftarrow L_1, \ldots, L_m.$$

In such a clause $H_1, \ldots H_n$ are logical atoms, $B_1, \ldots, B_m$ logical literals, and $\alpha_1, \ldots, \alpha_n$ real numbers in the interval $[0, 1]$ such that $\sum_{j=1}^{n} \alpha_j \leq 1$. The term $H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n$ is called the *head* and $L_1, \ldots, L_m$ is called the *body.* Note that if $n = 1$ and $\alpha_1 = 1$ a clause corresponds to a normal program clause, also called a *non-disjunctive* clause. If $\sum_{j=1}^{n} \alpha_j < 1$, 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 $1 - \sum_{j=1}^{n} \alpha_j$. For a clause C, we define $head(C)$ as $\{(H_i : \alpha_i)|1 \leq i \leq n\}$ if $\sum_{i=1}^{n} \alpha_i = 1$; and as $\{(H_i : \alpha_i)|1 \leq i \leq n\} \cup \{(null : 1 - \sum_{i=1}^{n} \alpha_i)\}$ otherwise. Moreover, we define $body(C)$ as $\{L_i|1 \leq i \leq m\}$, $H_i(C)$ as H_i and $\alpha_i(C)$ as α_i.

If the LPAD is ground, a clause represents a probabilistic choice between the non-disjunctive clauses obtained by selecting only one atom in the head. As usual, if the LPAD T is not ground, T is assigned a meaning by computing its grounding, $ground(T)$.

By choosing a head atom for each ground clause of an LPAD we get a normal logic program called a *world* of the LPAD (an *instance* of the LPAD in (Vennekens et al. 2004)). A probability distribution is defined over the space of worlds by assuming independence between the choices made for each clause.

More specifically, an *atomic choice* is a triple (C, θ, i) where $C \in T$, θ is a minimal substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for the ground clause $C\theta$, the head $H_i(C)$ 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 *composite choice* κ is a consistent set of atomic choices. The *probability* $P(\kappa)$ *of a composite choice* κ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C,\theta,i)\in\kappa} \alpha_i(C)$.

A *selection* σ is a composite choice that, for each clause $C\theta$ in $ground(T)$, contains an atomic choice (C, θ, i) in σ. Since T does not contain function symbols, $ground(T)$ is finite and so is each σ. We denote the set of all selections σ of a program T by $\mathcal{S}_T$. A selection σ identifies a normal logic program w_σ, called a *world* of T, defined as: $w_\sigma = \{(H_i(C)\theta \leftarrow body(C))\theta | (C, \theta, i) \in \sigma\}$. $\mathcal{W}_T$ denotes the set of all the worlds of T. Since selections are composite choices, we can assign a probability to worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} \alpha_i(C)$.

Throughout this paper, we consider only *sound* LPADs, in which every world has a total model according to the Well-Founded Semantics (WFS) (Van Gelder et al. 1991). In this way, uncertainty is modeled only by means of the disjunctions in the head and not by the semantics of negation. Thus in the following, $w_\sigma \models A$ means that the ground atom A is true in the well-founded model of the program w_σ[1].

In order to define the probability of an atom A being true in an LPAD T, note that the probability distribution over possible worlds induces a probability distribution over Herbrand interpretations by assuming $P(I|w) = 1$ if I is the well-founded model of w ($I = WFM(w)$) and 0 otherwise. We can thus compute the probability of an interpretation I as

$$P(I) = \sum_{w\in\mathcal{W}_T} P(I, w) = \sum_{w\in\mathcal{W}_T} P(I|w)P(w) = \sum_{w\in\mathcal{W}_T, I=WFM(w)} P(w).$$

We can extend the probability distribution on interpretation to ground atoms by assuming $P(a_j|I) = 1$ if A_j belongs to I and 0 otherwise, where A_j is a ground atom of the Herbrand base $\mathcal{H}_T$ and a_j stands for $A_j = true$. Thus the probability of a ground atom A_j being true, according to an LPAD T can be obtained as

$$P(a_j) = \sum_I P(a_j, I) = \sum_I P(a_j|I)P(I) = \sum_{I\subseteq\mathcal{H}_T, A_j\in I} P(I).$$

Alternatively, we can extend the probability distribution on programs to ground atoms by assuming $P(a_j|w) = 1$ if A_j is true in w and 0 otherwise. Thus the probability of A_j being true is

$$P(a_j) = \sum_{w\in\mathcal{W}_T} P(a_j, w) = \sum_{w\in\mathcal{W}_T} P(a_j|w)P(w) = \sum_{w\in\mathcal{W}_T, w\models A_j} P(w).$$

The probability of A_j being false is defined similarly.

[1] We sometimes abuse notation slightly by saying that an atom A is true in a world w to indicate that A is true in the (unique) well-founded model of w.

Example 1
Consider the dependency of sneezing on having the flu or hay fever:

$$\begin{array}{lll} C_1 = & strong\_sneezing(X) : 0.3 \vee moderate\_sneezing(X) : 0.5 & \leftarrow \quad flu(X). \\ C_2 = & strong\_sneezing(X) : 0.2 \vee moderate\_sneezing(X) : 0.6 & \leftarrow \quad hay\_fever(X). \\ C_3 = & flu(david). & \\ C_4 = & hay\_fever(david). & \end{array}$$

This program models the fact that sneezing can be caused by flu or hay fever. The query $moderate\_sneezing(david)$ is true in 5 of the 9 worlds of the program and its probability of being true is
$P_T(moderate\_sneezing(david)) = 0.5{\cdot}0.2{+}0.5{\cdot}0.6{+}0.5{\cdot}0.2{+}0.3{\cdot}0.6{+}0.2{\cdot}0.6 = 0.8$

Even if we assumed independence between the choices for individual ground clauses, this does not represents a restriction, in the sense that this still allows to represent all the joint distributions of atoms of the Herbrand base that are representable with a Bayesian network over those variables. Details of the proof are omitted for lack of space.

3 The Distribution Semantics for Programs with Function Symbols

If a non-ground LPAD T contains function symbols, then the semantics given in the previous section is not well-defined. In this case, each world w_σ is the result of an infinite number of choices and the probability $P(w_\sigma)$ is 0 since it is given by the product of an infinite number of factors all smaller than 1. Thus, the probability of a formula is 0 as well, since it is a sum of terms all equal to 0. The distribution semantics with function symbols was defined in (Sato 1995) and (Poole 2000). Here we follow the approach of (Poole 2000).

A composite choice κ identifies a set of worlds ω_κ that contains all the worlds associated to a selection that is a superset of κ: i.e., $\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_T, \sigma \supseteq \kappa\}$ We define the set of worlds identified by a set of composite choices K as $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$

Given a ground atom A, we define the notion of explanation, covering set of composite choices and mutually incompatible set of explanations. A composite choice κ is an *explanation* for A if A is true in every world of ω_κ. In Example 1, the composite choice $\{(C_1, \{X/david\}, 1)\}$ is an explanation for $strong\_sneezing(david)$. A set of composite choices K is *covering* with respect to A if every world w_σ in which A is true is such that $w_\sigma \in \omega_K$. In Example 1, the set of composite choices

$$K_1 = \{\{(C_1, \{X/david\}, 2)\}, \{(C_2, \{X/david\}, 2)\}\} \tag{1}$$

is covering for $moderate\_sneezing(david)$. Two composite choices κ_1 and κ_2 are *incompatible* if their union is inconsistent, i.e., if there exists a clause C and a substitution θ grounding C such that $(C, \theta, j) \in \kappa_1, (C, \theta, k) \in \kappa_2$ and $j \neq k$. A set K of composite choices is *mutually incompatible* if for all $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2 \Rightarrow \kappa_1$ and κ_2 are incompatible. As illustration, the set of composite choices

$$\begin{array}{rcl} K_2 & = & \{\{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 1)\}, \\ & & \{(C_1, \{X/david\}, 2), (C_2, \{X/david\}, 3)\}, \\ & & \{(C_2, \{X/david\}, 2)\}\} \end{array} \tag{2}$$

is mutually incompatible for the theory of Example 1. (Poole 2000) proved the following results

- Given a finite set K of finite composite choices, there exists a finite set K' of mutually incompatible finite composite choices such that $\omega_K = \omega_{K'}$.
- If K_1 and K_2 are both mutually incompatible finite sets of finite composite choices such that $\omega_{K_1} = \omega_{K_2}$ then $\sum_{\kappa \in K_1} P(\kappa) = \sum_{\kappa \in K_2} P(\kappa)$

Thus, we can define a unique probability measure $\mu : \Omega_T \rightarrow [0,1]$ where Ω_T is defined as the set of sets of worlds identified by finite sets of finite composite choices: $\Omega_T = \{\omega_K | K \text{ is a finite set of finite composite choices}\}$. It is easy to see that Ω_T is an algebra over $\mathcal{W}_T$. Then μ is defined by $\mu(\omega_K) = \sum_{\kappa \in K'} P(\kappa)$ where K' is a finite mutually incompatible set of finite composite choices such that $\omega_K = \omega_{K'}$. As is the case for ICL, $\langle \mathcal{W}_T, \Omega_T, \mu \rangle$ is a probability space (Kolmogorov 1950).

Definition 1
The probability of a ground atom A is given by $P(A) = \mu(\{w | w \in \mathcal{W}_T \wedge w \models A\}$

If A has a finite set K of finite explanations such that K is covering then $\{w | w \in \mathcal{W}_T \wedge w \models A\} = \omega_K$ and $\mu(\{w | w \in \mathcal{W}_T \wedge w \models A\} = \mu(\omega_K)$ so $P(A)$ is well-defined. In the case of Example 1, K_2 shown in equation 2 is a finite covering set of finite explanations for $moderate\_sneezing(david)$ that is mutually incompatible, so

$$P(moderate\_sneezing(david)) = 0.5 \cdot 0.2 + 0.5 \cdot 0.2 + 0.6 = 0.8.$$

4 Dynamic Stratification of LPADs

One of the most important formulations of stratification is that of *dynamic* stratification. (Przymusinski 1989) shows that a program has a 2-valued well-founded model iff it is dynamically stratified, so that it is the weakest notion of stratification that is consistent with the WFS. As presented in (Przymusinski 1989), dynamic stratification computes strata via operators on *3-valued interpretations* – pairs of the form $\langle Tr; Fa \rangle$, where Tr and Fa are subsets of the Herbrand base $\mathcal{H}_P$ of a normal program P.

Definition 2
For a normal program P, sets Tr and Fa of ground atoms, and a 3-valued interpretation I we define

$True_I^P(Tr) = \{A | A$ is not true in I; and there is a clause $B \leftarrow L_1, ..., L_n$ in P, a ground substitution θ such that $A = B\theta$ and for every $1 \leq i \leq n$ either $L_i\theta$ is true in I, or $L_i\theta \in Tr\}$;

$False_I^P(Fa) = \{A | A$ is not false in I; and for every clause $B \leftarrow L_1, ..., L_n$ in P and ground substitution θ such that $A = B\theta$ there is some i $(1 \leq i \leq n)$ such that $L_i\theta$ is false in I or $L_i\theta \in Fa\}$.

(Przymusinski 1989) shows that $True_I^P$ and $False_I^P$ are both monotonic, and defines $\mathcal{T}_I^P$ as the least fixed point of $True_I^P(\emptyset)$ and $\mathcal{F}_I^P$ as the greatest fixed point of $False_I^P(\mathcal{H}_P)$ [2]. In words, the operator $\mathcal{T}_I$ extends the interpretation I to add the

[2] Below, we will sometimes omit the program P in these operators when the context is clear.

new atomic facts that can be derived from P knowing I; $\mathcal{F}_I$ adds the new negations of atomic facts that can be shown false in P by knowing I (via the uncovering of unfounded sets). An iterated fixed point operator builds up dynamic strata by constructing successive partial interpretations as follows.

Definition 3 (*Iterated Fixed Point and Dynamic Strata*)
For a normal program P let

$$\begin{array}{rcl} WFM_0 & = & \langle \emptyset; \emptyset \rangle; \\ WFM_{\alpha+1} & = & WFM_\alpha \cup \langle \mathcal{T}_{WFM_\alpha}; \mathcal{F}_{WFM_\alpha} \rangle; \\ WFM_\alpha & = & \bigcup_{\beta<\alpha} WFM_\beta, \text{ for limit ordinal } \alpha. \end{array}$$

Let $WFM(P)$ denote the fixed point interpretation WFM_δ, where δ is the smallest (countable) ordinal such that both sets $\mathcal{T}_{WFM_\delta}$ and $\mathcal{F}_{WFM_\delta}$ are empty. We refer to δ as the *depth* of program P. The *stratum* of atom A, is the least ordinal β such that $A \in WFM_\beta$ (where A may be either in the true or false component of WFM_β).

(Przymusinski 1989) shows that the iterated fixed point $WFM(P)$ is in fact the well-founded model and that any undefined atoms of the well-founded model do not belong to any stratum – i.e. they are not added to WFM_δ for any ordinal δ. Thus, a program is *dynamically stratified* if every atom belongs to a stratum.

Dynamic stratification captures the order in which recursive components of a program must be evaluated. Because of this, dynamic stratification is useful for modeling operational aspects of program evaluation. Fixed-order dynamic stratification (Sagonas et al. 2000), used in Section 7, models programs whose well-founded model can be evaluated using a fixed literal selection strategy. In this class, the definition of $False_I^P(Fa)$ in Definition 2 is replaced by[3]:

$False_I^P(F) = \{A|A$ is not false in I; and for every clause $B \leftarrow L_1, ..., L_n$ in P and ground substitution θ such that $A = B\theta$ there is some i $(1 \leq i \leq n)$ such that $L_i\theta$ is false in I or $L_i\theta \in Fa$, ***and for all*** j $(1 \leq j \leq i-1)$**,** $L_j\theta$ ***is true in*** $I\}$.

(Sagonas et al. 2000) describes how fixed-order dynamic stratification captures those programs that a tabled evaluation can evaluate with a fixed literal selection strategy (i.e. without the SLG operations of SIMPLIFICATION and DELAY).

Example 2
The following program has a 2-valued well-founded model and so is dynamically stratified, but does not belong to other stratification classes in the literature, such as local, modular, or weak stratification.

$s \leftarrow \neg s.$ $\quad$ $s \leftarrow \neg p, \neg q, \neg r.$
$p \leftarrow q, \neg r, \neg s.$ $\quad$ $q \leftarrow r, \neg p.$ $\quad$ $r \leftarrow p, \neg q.$

p, q, and r all belong to stratum 0, while s belongs to stratum 1. Note that the above program also meets the definition of fixed-order dynamically stratified as does the simple program

[3] Without loss of generality, we assume throughout that the fixed literal selection strategy is left-to-right as in Prolog.

$p \leftarrow \neg\, p.$ $\qquad\qquad$ $p.$

which is not locally, modularly, or weakly stratified. Fixed-order stratification is more general than local stratification, and than modular stratification (since modular stratified programs can be decidably rearranged so that they have failing prefixes). It is neither more nor less general than weak stratification.

As seen by the above examples, fixed-order dynamic stratification is a fairly weak property for a program to have. The above definitions of (fixed-order) dynamic stratification for normal programs can be straightforwardly adapted to LPADs – an LPAD T is *(fixed-order) dynamically stratified* if each $w \in \mathcal{W}_T$ is (fixed-order) dynamically stratified.

4.1 Conditions for Well-Definedness of the Distribution Semantics

When a given LPAD T contains function symbols there are two reasons why the distribution semantics may not be well-defined for T. First, a world of T may not have a two-valued well-founded model; and second, $\mathcal{H}_T$ may contain an atom that does not have a finite set of finite explanations that is covering (cf. Section 3). As noted in Section 2, we consider only sound LPADs in this paper and in this section address the problem of determining whether $\mathcal{H}_T$ may contain a atom that does not have a finite set of finite explanations that is covering.

As is usual in logic programming, we assume that a program P is defined over a language with a finite number of function and constant symbols. Given such an assumption, placing an upper bound on the size of terms in a derivation implies that the number of different terms in a derivation must be finite – and for certain methods of derivation, such as tabled or bottom-up evaluations, that the derivation itself is finite.

To motivate our definitions, consider the normal program T_{inf}:

$p(s(X)) \leftarrow p(X).$ $\qquad\qquad$ $p(0).$

This program does not have a model with a finite number of true or undefined atoms, and accordingly, there is no upper limit on the size of atoms produced either in a bottom-up derivation of the program (e.g. using the fixed-point characterization of Definition 3), or in a top-down evaluation of the query $p(Y)$. However, the superficially similar program, T_{fin}:

$p(X) \leftarrow p(f(X)).$ $\qquad\qquad$ $p(0).$

does have a model with a finite number of true and undefined atoms. Of course, the model for the program does not have a finite number of false atoms, but (default) false atoms are generally not explicitly represented in derivations. The model can in fact be produced by various derivation techniques, such as an alternating fixed point computation (van Gelder 1989) based on sets of true and of true or undefined atoms; or by tabling with term depth abstraction (Tamaki and Sato 1986).

From the perspective of the distribution semantics consider T'_{fin}, the extension of T_{fin} with the clause *q : 0.5 ← p(X).*

and T'_{inf}, the similar extension of T_{inf}. Recall from Definition 1 that the probability of an atom A in an LPAD is defined as a probability measure that is constructed from finite sets of finite composite choices: accordingly, the distribution semantics for A is well-defined if and only if it has a finite set of finite explanations that is covering. In T'_{fin}, q has such a finite set of finite explanations that is covering, and so its distribution semantics is well-defined. However, in T'_{inf}, q does not have a finite set of finite explanations that is covering, and so the distribution semantics is not well-defined for q, even though every world of T'_{inf} has a total well-founded model.

The following definition captures these intuitions, basing the notion of bounded term-size on the preceding definition of dynamic stratification.

Definition 4 (*Bounded Term-size Programs*)
Let P be the ground instantiation of a normal program. and $I, Tr \subseteq \mathcal{H}_P$. Then an application of $True^P_I(Tr)$ (Definition 2) has the *bounded term-size* property if there is a integer L such that the size of every ground substitution θ used to produce an atom in $True^P_I(Tr)$ is less than L. P itself has the *bounded term-size* property if every application of $True^P_I$ used to construct $WFM(P)$ has the bounded term-size property with the same bound L. Finally, an LPAD T has the *bounded term-size* property if each world of T has the bounded term-size property.

Note that T_{inf} does not have the bounded term-size property, but T_{fin} does. While determining whether a program P is bounded term-size is clearly undecidable in general, T_{fin} shows that *ground*(P) need not be finite if P is bounded term-size. However, the model of P may be characterized as follows[4].

Theorem 1
Let P be a normal program. Then $WFM(P)$ has a finite number of true atoms iff P has the bounded term-size property.

Theorem 1 gives a clear model-theoretic characterization of bounded term-size normal programs: note that if *ground*(P) is infinite, then $WFM(P)$ may have an infinite number of false or undefined atoms. In the context of LPADs, the bounded term-size property ensures the well-definedness of the distribution semantics.

Theorem 2
Let T be a sound bounded term-size LPAD, and let $A \in \mathcal{H}_T$. Then A has a finite set of finite explanations that is covering.

The proof of Theorem 2 is presented in the online Appendix; here we indicate the intuition behind the proof. First, we note that it is straightforward to show that since each world of an LPAD T has a finite number of true atoms by Theorem 1, explanations are finite. On the other hand, showing that a query has a finite covering set of explanations is less obvious, as T could have an infinite number of worlds.

[4] The proof of this and other theorems is given in the online Appendix to this paper.

The proof addresses this by showing that T has a finite number of models, in turn shown by demonstrating the existence of a bound L_T on the maximal size of any true atom in any world of T. The existence of L_T is shown by contradiction by demonstrating that if no bound existed, a world could be constructed that was not bounded term-size. The idea is explained in the following example.

Example 3
Consider the program

$$q : 0.5 \vee p(f(X)) : 0.5 \leftarrow p(X). \qquad\qquad p(0).$$

This program has an infinite number of finite models, which consist of true atoms

$$\{q, p(0)\}, \{q, p(0), p(f(0))\}, \{q, p(0), p(f(0)), p(f(f(0)))\}, \ldots$$

depending on the selections made for instantiations of the first clause, and so no finite bound L_T exists for this program. However such a program also has a selection that gives rise to an infinite model

$$\{p(0), p(f(0)), p(f(f(0))), p(f(f(f(0)))), \ldots\}$$

and so is not bounded term-size.

Although bounded term-size programs have appealing properties, such programs can make only weak use of function symbols. For instance, a program containing the Prolog predicate *member/2* would not be bounded term-size, although as any Prolog programmer knows, a query to *member/2* will terminate whenever the second argument of the query is ground. We capture this intuition with *bounded term-size queries*. The definition of such queries relies on the notion of an atom dependency graph, whose definition we state for LPADs.

Definition 5 (*Atom Dependency Graph*)
Let T be a ground LPAD. Then the *atom dependency graph* of T is a graph (V, E) such that $V = \mathcal{H}_T$ and an edge $(v_1, v_2) \in E$ iff there is a clause $C \in T$ such that

1. $(v_1 : \alpha_1) \in head(C)$ and if v_2 or $\neg v_2 \in body(C)$; or
2. $(v_1 : \alpha_1), (v_2 : \alpha_2) \in head(C)$.

Definition 5 includes dependencies among atoms in the head of a disjunctive LPAD clause, similar to how dependencies are defined in disjunctive logic programs. Given a ground LPAD T, the atom dependency graph of T is used to bound the search space of a (relevant) derivation in a world of T under the WFS.

Definition 6 (*Bounded Term-size Queries*)
Let T be a ground LPAD, and Q an atomic query to T (not necessarily ground). Then the *atomic search space* of Q consists of the union of all ground instantiations of Q in $\mathcal{H}_T$ together with all atoms reachable in the atom dependency graph of T from any ground instantiation of Q. Let

$$T_Q = \{C | C \in T \text{ and a head atom of } C \text{ is in the atomic search space of } Q\}$$

The query Q is *bounded term-size* if T_Q is a bounded term-size program.

The notion of a bounded-term size query will be used in Section 6 to characterize termination of the SLG tabling approach, and in Section 8 to characterize correctness and termination of our tabled PITA implementation.

4.2 Comparisons of Termination Properties

We next consider how the concepts of bounded term-size programs and queries relate to some other classes of programs for which termination has been studied. Since the definitions of the previous section are based on LPADs, and other work in the literature is often based on disjunctive logic programs, we restrict our attention to normal programs, for which the semantics coincide.

(Baselice et al. 2009) studies the class of *finitely recursive* programs, which is a superset of *finitary* programs previously introduced into the literature by the authors. The paper first defines a dependency graph, which for normal programs is essentially the same as Definition 5. A finitely recursive normal program, then, is one for which in its atom dependency graph, only a finite number of vertices are reachable from any vertex. It is easy to see that neither bounded term-size programs nor finitely recursive programs are a subclass of each other. A program containing simply *member/2* (and a constant) is finitely recursive, but is not bounded term-size. However, the program

$$p(X) \leftarrow p(f(X)).$$

has bounded term-size, as does the program

$$p(s(X)) \leftarrow q(X),p(X). \qquad\qquad p(0).$$

although neither is finitely recursive (for the last program, the failure of $q(X)$ means that all applications of $True_I$ have bounded term-size). However, note that for any program P that is finitely recursive, all ground atomic queries to P will have bounded term-size. Therefore, if P is finitely recursive, every ground atomic query to P will be bounded term-size, even if P itself isn't bounded term-size.

Another recent work (Calimeri et al. 2008) defines the class *finitely-ground* programs. We do not present its formalism here, but Corollary 1 of (Calimeri et al. 2008) states that if a program is finitely-ground, it will have a finite number of answer sets and each answer set will be finite (as represented by the set of true atoms in the model). By Theorem 1 of this paper, such a program will have bounded term-size, so that finitely-ground programs may be co-extensive with bounded term-size programs. On the other hand, (Calimeri et al. 2008) notes that finitely-ground programs and finitely recursive programs are incompatible. Non-range restricted programs are not finitely-ground, although they can be finitely recursive. As discussed above, any ground atomic query to a finitely recursive program will have bounded term-size, so that finitely-ground programs must be a proper subclass of those programs for which all ground atomic queries have bounded term-size.

To summarize for normal programs:

- Finitely recursive and bounded term-size programs are incompatible, but

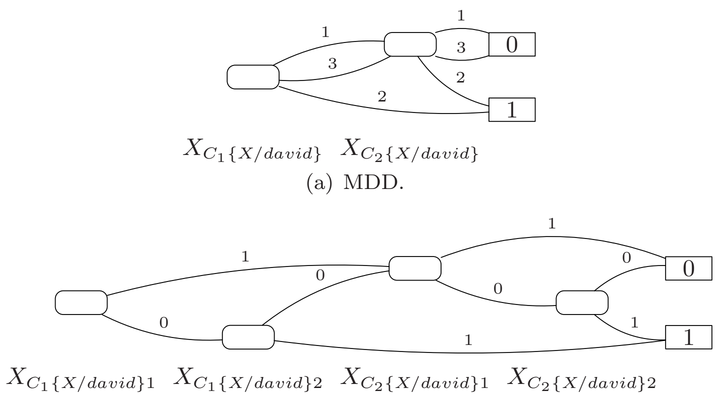


Fig. 1. Decision diagrams for Example 1.

finitely recursive programs are a proper subclass of those programs for which all ground atomic queries are bounded term-size.

- Finitely-ground and bounded term-size programs appear to be co-extensive, but finitely-ground programs are a proper subclass of those programs for which all ground atomic queries are bounded term-size.

5 Representing Explanations by Means of Decision Diagrams

In order to represent explanations we can use Multivalued Decision Diagrams (MDDs) (Thayse et al. 1978). An MDD represents a function $f(\mathbf{X})$ taking Boolean values on a set of multivalued variables $\mathbf{X}$ by means of a rooted graph that has one level for each variable. Each node N has one child for each possible value of the multivalued variable associated to N. The leaves store either 0 or 1. Given values for all the variables $\mathbf{X}$, an MDD can be used to compute the value of $f(\mathbf{X})$ by traversing the graph starting from the root and returning the value associated to the leaf that is reached.

Given a set of explanations K, we obtain a Boolean function f_K in the following way. Each ground clause $C\theta$ appearing in K is associated to a multivalued variable $X_{C\theta}$ with as many values as atoms in the head of C. In other words, each atomic choice (C, θ, i) is represented by the propositional equation $X_{C\theta} = i$. Equations for a single explanation are conjoined and the conjunctions for the different explanations are disjoined. The set of explanations in Equation (1) can be represented by the function $f_{K_1}(\mathbf{X}) = (X_{C_1\{X/david\}} = 2) \vee (X_{C_2\{X/david\}} = 2)$. The MDD shown in Figure 1(a) represents $f_{K_1}(\mathbf{X})$.

Given a MDD M, we can identify a set of explanations K_M associated to M that is obtained by considering each path from the root to a 1 leaf as an explanation. It is easy to see that if K is a set of explanations and M is obtained from f_K, K and K_M represent the same set of worlds, i.e., that $\omega_K = \omega_{K_M}$.

Note that K_M is mutually incompatible because at each level we branch on a variable so that the explanations associated to the leaves that are descendants of a child of a node N are incompatible with those of any other children of N.

By converting a set of explanations into a mutually incompatible set of explanations, MDDs allow the computation of $\mu(\omega_K)$ (Section 3) given any K. This is equivalent to computing the probability of a DNF formula which is #P-complete (**?**). Decision diagrams offer a practical solution for this problem and were shown better than other methods (De Raedt et al. 2007).

Decision diagrams can be built with various software packages that provide highly efficient implementation of Boolean operations. However, most packages are restricted to work with Binary Decision Diagrams, i.e., decision diagrams where all the variables are Boolean. To manipulate MDDs with a BDD package, we must represent multivalued variables by means of binary variables. Various options are possible, we found that the following, proposed in (De Raedt et al. 2008), gives the best performance. For a variable X having n values, we use $n-1$ Boolean variables $X_1, \ldots, X_{n-1}$ and we represent the equation $X = i$ for $i = 1, \ldots n-1$ by means of the conjunction

$$\overline{X_1} \wedge \overline{X_2} \wedge \ldots \wedge \overline{X_{i-1}} \wedge X_i$$

and the equation $X = n$ by means of the conjunction

$$\overline{X_1} \wedge \overline{X_2} \wedge \ldots \wedge \overline{X_{n-1}}$$

The BDD representation of the function f_{K_1} is given in Figure 1(b). The Boolean variables are associated with the following parameters:

$$\begin{array}{l} P(X_1) = P(X \ = \ 1) \\ \ldots \\ P(X_i) = \frac{P(X=i)}{\prod_{j=1}^{i-1}(1-P(X_{j-1}))} \end{array}$$

.

6 Tabling and Answer Subsumption

The idea behind tabling is to maintain in a table both subgoals encountered in a query evaluation and answers to these subgoals. If a subgoal is encountered more than once, the evaluation reuses information from the table rather than re-performing resolution against program clauses. Although the idea is simple, it has important consequences. First, tabling ensures termination for a wide class of programs, and it is often easier to reason about termination in programs using tabling than in basic Prolog. Second, tabling can be used to evaluate programs with negation according to the WFS. Third, for queries to wide classes of programs, such as datalog programs with negation, tabling can achieve the optimal complexity for query evaluation. And finally, tabling integrates closely with Prolog, so that Prolog's familiar programming environment can be used, and no other language is required to build complete systems. As a result, a number of Prologs now support tabling including XSB, YAP, B-Prolog, ALS, and Ciao. In these systems, a predicate p/n is evaluated using SLDNF by default: the predicate is made to use tabling by a declaration such as *table* p/n that is added by the user or compiler.

This paper makes use of a tabling feature called *answer subsumption.* Most formulations of tabling add an answer A to a table for a subgoal S only if A is a not a variant (as a term) of any other answer for S. However, in many applications it may be useful to order answers according to a partial order or (upper semi-)lattice. As an example, consider the case of a lattice on the values of the second argument of a binary predicate $p/2$. Answer subsumption may be specified by a declaration such as *table p(_,or/3 - zero/1).* where $zero/1$ is the bottom element of the lattice and $or/3$ is the join operation of the lattice. For example, if a table had an answer $p(a, b_1)$ and a new answer $p(a, b_2)$ were derived, the answer $p(a, b_1)$ is replaced by $p(a, b_3)$, where b_3 is the join of b_1 and b_2 obtained by calling $or(b_1, b_2, b_3)$. In the PITA algorithm for LPADs presented in Section 7 the last argument of an atom is used to store explanations for the atom in the form of BDDs and the $or/3$ operation is the logical disjunction of two explanations [5]. Answer subsumption over arbitrary upper semi-lattices is implemented in XSB for stratified programs (Swift 1999b).

For formal results in this section and Section 8 we use SLG resolution (Chen and Warren 1996), under the forest-of-trees representation (Swift 1999a); this framework is extended with answer subsumption in the proof of Theorem 4. However, first we present a theorem stating that bounded term-size queries (Definition 6) to normal programs are amenable to top-down evaluation using tabling. Although SLG has been shown to finitely terminate for other notions of bounded term-size queries, the concept as presented in Definition 6 is based on a bottom-up fixed-point definition of WFS, and only bounds the size of substitutions used in $True_I^P$ of Definition 2, but not of $False_I^P$. In fact, to prove termination of SLG with respect to bounded term-size queries, SLG must be extended so that its New Subgoal operation performs what is called *term-depth abstraction* (Tamaki and Sato 1986), explained informally as follows. An SLG evaluation can be formalized as a forest of trees in which each tree corresponds to a unique (up to variance) subgoal. The SLG New Subgoal operation checks to see if a given selected subgoal S is the root of any tree in the current forest. If not, then a new tree with root S is added to the forest. Without term-depth abstraction, an SLG evaluation of the query *p(a)* and the program consisting of the single clause

$$p(X) \leftarrow p(f(X)).$$

would create an infinite number of trees. However, if the New Subgoal operation uses term-depth abstraction, any subterm in S over a pre-specified maximal depth would be replaced by a new variable. For example, in the above program if the maximal depth were specified as 3, the subgoal *p(f(f(f(a))))* would be rewritten to *p(f(f(f(X))))* for the purposes of creating a new tree. The subgoal *p(f(f(f(a))))* would consume any answer from the tree for *p(f(f(f(X))))* where the binding for X unified with a. In this manner it can be ensured that only a finite number of trees were created in the forest. This fact, together with the size bound on the derivation of answers provided by Definition 6 ensures the following theorem, where

[5] The logical disjunction b_3 can be seen as subsuming b_1 and b_2 over the partial order af implication defined on propositional formulas that represent explanations.

a finitely terminating evaluation may terminate normally or may terminate through floundering.

Theorem 3
Let P be fixed-order dynamically stratified normal program, and Q a bounded term-size query to P. Then there is an SLG evaluation of Q to P using term-depth abstraction that finitely terminates.

By the discussion of Section 4.2, Theorem 3 shows that there is an SLG evaluation with term-depth abstraction will finitely terminate on any ground query to a finitely recursive (Baselice et al. 2009) or finitely-ground (Calimeri et al. 2008) program that is fixed-order stratified [6]. While SLG itself is ideally complete for all normal programs, the PITA implementation is restricted to fixed-order stratified programs, so that Theorem 3 is used in the proof of the termination results of Section 8.

7 Program Transformation

The first step of the PITA algorithm is to apply a program transformation to an LPAD to create a normal program that contains calls for manipulating BDDs. In our implementation, these calls provide a Prolog interface to the CUDD[7] C library and use the following predicates[8]

- *init, end*: for allocation and deallocation of a BDD manager, a data structure used to keep track of the memory for storing BDD nodes;
- *zero(-BDD), one(-BDD), and(+BDD1,+BDD2,-BDDO), or(+BDD1,+BDD2, -BDDO), not(+BDDI,-BDDO)*: Boolean operations between BDDs;
- *add_var(+N_Val,+Probs,-Var)*: addition of a new multi-valued variable with *N_Val* values and parameters *Probs*;
- *equality(+Var,+Value,-BDD)*: *BDD* represents *Var=Value*, i.e. that the random variable *Var* is assigned *Value* in the BDD;
- *ret_prob(+BDD,-P)*: returns the probability of the formula encoded by *BDD*.

add_var(+N_Val,+Probs,-Var) adds a new random variable associated to a new instantiation of a rule with *N_Val* head atoms and parameters list *Probs*. The PITA transformation uses the auxiliary predicate *get_var_n(+R,+S,+Probs,-Var)* to wrap *add_var/3* and avoid adding a new variable when one already exists for an instantiation. As shown below, a new fact *var(R,S,Var)* is asserted each time a new random variable is created, where R is an identifier for the LPAD clause, S is a list of constants, one for each variable of the clause, and *Var* is an integer that identifies the random variable associated with clause R under the grounding represented by *S*. The auxiliary predicate has the following definition

[6] The proof of Theorem 3 relies on a delay-minimal evaluation of Q that does not produced any conditional answers – that is, an evaluation that does not explore the space of atoms that are undefined in $WFM(P)$.
[7] `http://vlsi.colorado.edu/~fabio/`
[8] BDDs are represented in CUDD as pointers to their root node.

$$\begin{array}{l} get\_var\_n(R, S, Probs, Var) \leftarrow \\ \quad (var(R, S, Var) \rightarrow true; \\ \quad\; length(Probs, L), add\_var(L, Probs, Var), assert(var(R, S, Var))). \end{array}$$

The PITA transformation applies to atoms, literals and clauses. If H is an atom, $PITA_H(H)$ is H with the variable BDD added as the last argument. If A_j is an atom, $PITA_B(A_j)$ is A_j with the variable B_j added as the last argument. In either case for an atom A, $BDD(PITA(A))$ is the value of the last argument of $PITA(A)$, If L_j is negative literal $\neg A_j$, $PITA_B(L_j)$ is the conditional

$$(PITA'_B(A_j) \rightarrow not(BN_j, B_j); one(B_j)),$$

where $PITA'_B(A_j)$ is A_j with the variable BN_j added as the last argument. In other words the input BDD, BN_j, is negated if it exists; otherwise the BDD for the constant function 1 is returned.

A non-disjunctive fact $C_r = H$ is transformed into the clause

$$PITA(C_r) = \quad PITA_H(H) \leftarrow one(BDD).$$

A disjunctive fact $C_r = H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n$. where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$[9]

$$\begin{array}{lll} PITA(C_r, 1) = & PITA_H(H_1) \leftarrow & get\_var\_n(r, [], [\alpha_1, \ldots, \alpha_n], Var), \\ & & equality(Var, 1, BDD). \\ \ldots & & \\ PITA(C_r, n) = & PITA_H(H_n) \leftarrow & get\_var\_n(r, [], [\alpha_1, \ldots, \alpha_n], Var), \\ & & equality(Var, n, BDD). \end{array}$$

In the case where the parameters do not sum to one, the clause is first transformed into $H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n \vee null : 1 - \sum_1^n \alpha_i$. and then into the clauses above, where the list of parameters is $[\alpha_1, \ldots, \alpha_n, 1 - \sum_1^n \alpha_i]$ but the $(n+1)$-th clause (the one for $null$) is not generated.
The definite clause $C_r = H \leftarrow L_1, \ldots, L_m$. is transformed into the clause

$$\begin{array}{lll} PITA(C_r) = & PITA_H(H) \leftarrow & one(BB_0), \\ & & PITA_B(L_1), and(BB_0, B_1, BB_1), \\ & & \ldots, \\ & & PITA_B(L_m), and(BB_{m-1}, B_m, BDD). \end{array}$$

The disjunctive clause

$$C_r = H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n \leftarrow L_1, \ldots, L_m.$$

where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$

[9] The second argument of get_var_n is the empty list because a fact does not contain variables since the program is bounded term-size.

$$\begin{array}{lll}
PITA(C_r,1) = & PITA_H(H_1) \leftarrow & one(BB_0),\\
& & PITA_B(L_1), and(BB_0,B_1,BB_1),\\
& & \ldots,\\
& & PITA_B(L_m), and(BB_{m-1},B_m,BB_m),\\
& & get\_var\_n(r,VC,[\alpha_1,\ldots,\alpha_n],Var),\\
& & equality(Var,1,B), and(BB_m,B,BDD).\\
\ldots & & \\
PITA(C_r,n) = & PITA_H(H_n) \leftarrow & one(BB_0),\\
& & PITA_B(L_1), and(BB_0,B_1,BB_1),\\
& & \ldots,\\
& & PITA_B(L_m), and(BB_{m-1},B_m,BB_m),\\
& & get\_var\_n(r,VC,[\alpha_1,\ldots,\alpha_n],Var),\\
& & equality(Var,n,B), and(BB_m,B,BDD).
\end{array}$$

where VC is a list containing each variable appearing in C_r. If the parameters do not sum to 1, the same technique used for disjunctive facts is used.

Example 4
Clause C_1 from the LPAD of Example 1 is translated into

$$\begin{array}{ll}
strong\_sneezing(X,BDD) & \leftarrow one(BB_0), flu(X,B_1), and(BB_0,B_1,BB_1),\\
& get\_var\_n(1,[X],[0.3,0.5,0.2],Var),\\
& equality(Var,1,B), and(BB_1,B,BDD).\\
moderate\_sneezing(X,BDD) & \leftarrow one(BB_0), flu(X,B_1), and(BB_0,B_1,BB_1),\\
& get\_var\_n(1,[X],[0.3,0.5,0.2],Var),\\
& equality(Var,2,B), and(BB_1,B,BDD).
\end{array}$$

while clause C_3 is translated into

$$flu(david,BDD) \quad \leftarrow one(BDD).$$

In order to answer queries, the goal *prob(Goal,P)* is used, which is defined by

$$\begin{array}{ll}
prob(Goal,P) & \leftarrow init, retractall(var(\_,\_,\_)),\\
& add\_bdd\_arg(Goal,BDD,GoalBDD),\\
& (call(GoalBDD) \rightarrow ret\_prob(BDD,P); P = 0.0),\\
& end.
\end{array}$$

where $add\_bdd\_arg(Goal,BDD,GoalBDD)$ implements $PITA_H(Goal)$. Moreover, various predicates of the LPAD should be declared as tabled. For a predicate p/n, the declaration is *table p(_1,...,_n,or/3-zero/1)*, which indicates that answer subsumption is used to form the disjunct of multiple explanations. At a minimum, the predicate of the goal and all the predicates appearing in negative literals should be tabled with answer subsumption. As shown in Section 10, it is usually better to table every predicate whose answers have multiple explanations and are going to be reused often.

8 Correctness of PITA Evaluation

In this section we show a result regarding the PITA transformation and its tabled evaluation on bounded term-size queries: this result takes as a starting point the well-definedness result of Theorem 2.

The main result of this section, Theorem 4, makes explicit mention of BDD data structures, which are considered to be ground terms for the purposes of formalization and are not specified further. Accordingly, the BDD operations used in the PITA transformation: *and/3*, *or/3*, *not/2*, *one/1*, *zero/1*, and *equality/3*, are all taken as (infinite) relations on terms, so that these predicates can be made part of a program's ground instantiation in the normal way. As a result, the ground instantiation of $PITA(T)$ instantiates all variables in T with all BDD terms. Similarly, for the purposes of proving correctness, a ground program is assumed to be extended with the relation *var(RuleName,[],Var)* to associate a random variable with the identifier of each clause (see Appendix C for more details). Note that since Theorem 4 assumes a bounded term-size query, the semantics is well-defined so the BDD and *var/3* terms are finite. In other words, the representation of each explanation of each atom are finite, and each atom has a finite covering set of explanations.

Lemma 1 shows that the PITA transformation does not affect the property of a query being bounded term-size. a result that is used in the proof of Theorem 4.

Lemma 1
Let T be an LPAD and Q a bounded term-size query to T. Then the query $PITA_H(Q)$ to $PITA(T)$ has bounded term-size.

Theorem 4 below states the correctness of the tabling implementation of PITA, since the BDD returned for a tabled query is the disjunction of a covering set of explanations for that query. The proof uses an extension of SLG evaluation that includes answer subsumption to collect explanations by disjoining BDDs, but that is restricted to the fixed-order dynamically stratified programs of Section 4. This formalism models the programs and implementation tested in Section 10.

Theorem 4 (*Correctness of PITA Evaluation*)
Let T be a fixed-order dynamically stratified LPAD and Q a ground bounded term-size atomic query. Then there is an SLG evaluation $\mathcal{E}$ of $PITA_H(Q)$ against $PITA(T_Q)$, such that answer subsumption is declared on $PITA_H(Q)$ using BDD-disjunction where $\mathcal{E}$ finitely terminates with an answer Ans for $PITA_H(Q)$ and $BDD(Ans)$ represents a covering set of explanations for Q.

9 Related Work

(Mantadelis and Janssens 2010) presented an algorithm for answering queries to ProbLog programs that uses tabling. Our work differs from this in two important ways. The first is that we use directly XSB tabling with answer subsumption while (Mantadelis and Janssens 2010) use some user-defined predicates that manipulate extra tabling data structures. The second difference is that in (Mantadelis and

Janssens 2010) explanations are stored in trie data structures that are then translated into BDDs. When translating the tries into BDDs, the algorithm of (Mantadelis and Janssens 2010) finds shared substructures, i.e., sub-explanations shared by many explanations. By identifying shared structures the construction of BDDs is sped up since sub-explanations are transformed into BDD only once. In our approach, we similarly exploit the repetition of structures but we do it while finding explanations: by storing in the table the BDD representation of the explanations of each answer, every time the answer is reused its BDD does not have to be rebuilt. Thus our optimization is guided by the derivation of the query. Moreover, if a BDD is combined with another BDD that already contains the first as a subgraph, we rely on the highly optmized CUDD functions for the identification of the repetition and the simplification of the combining operation. In this way we exploit structure sharing as well without the intermediate pass over the trie data strucutres.

10 Experiments

PITA was tested on two datasets that contain function symbols: the first is taken from (Vennekens et al. 2004) and encodes a Hidden Markov Model (HMM) while the second from (De Raedt et al. 2007) encodes biological networks. Moreover, it was also tested on the four testbeds of (Meert et al. 2009) that do not contain function symbols. PITA was compared with the exact version of ProbLog (De Raedt et al. 2007) available in the git version of Yap as of 10 November 2010, with the version of `cplint` (Riguzzi 2007) available in Yap 6.0 and with the version of CVE (Meert et al. 2009) available in ACE-ilProlog 1.2.20[10].

The first problem models a hidden Markov model with states 1, 2 and 3, of which 3 is an end state. This problem is encoded by the program

$$
\begin{aligned}
&s(0,1){:}1/3 \vee s(0,2){:}1/3 \vee s(0,3){:}1/3.\\
&s(T,1){:}1/3 \vee s(T,2){:}1/3 \vee s(T,3){:}1/3 \leftarrow\\
&\qquad T1\ is\ T\text{-}1,\ T1\text{>=}0,\ s(T1,F),\ \backslash\text{+}\ s(T1,3).
\end{aligned}
$$

For this experiment, we query the probability of the HMM being in state 1 at time *N* for increasing values of *N*, i.e., we query the probability of *s(N,1)*. In PITA and ProbLog, we did not use reordering of BDDs variables[11]. In PITA we tabled *on*/2 and in ProbLog we tabled the same predicate using the technique described in (Mantadelis and Janssens 2010). The execution times of PITA, ProbLog, CVE and `cplint` are shown in Figure 2. In this problem tabling provides an impressive speedup, since computations can be reused often.

The biological network programs compute the probability of a path in a large graph in which the nodes encode biological entities and the links represents conceptual relations among them. Each program in this dataset contains a non-probabilistic

[10] All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.
[11] For each experiment with PITA and ProbLog, we used either group sift automatic reordering or no reordering of BDDs variables depending on which gave the best results.

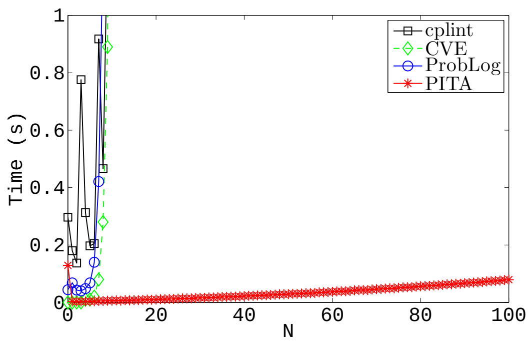


Fig. 2. Hidden Markov model.

definition of path plus a number of links represented by probabilistic facts. The programs have been sampled from a very large graph and contain 200, 400, ..., 10000 edges. Sampling was repeated ten times, to obtain ten series of programs of increasing size. In each program we query the probability that the two genes HGNC_620 and HGNC_983 are related. We used two definitions of path. The first, from (Kimmig et al. 2011), performs loop checking explicitly by keeping the list of visited nodes:

$$\begin{array}{lcl} path(X,Y) & \leftarrow & path(X,Y,[X],Z). \\ path(X,Y,V,[Y|V]) & \leftarrow & arc(X,Y). \\ path(X,Y,V0,V1) & \leftarrow & arc(X,Z), append(V0,\_S,V1), \\ & & \backslash + member(Z,V0), path(Z,Y,[Z|V0],V1). \\ arc(X,Y) & \leftarrow & edge(X,Y). \\ arc(X,Y) & \leftarrow & edge(Y,X). \end{array} \tag{3}$$

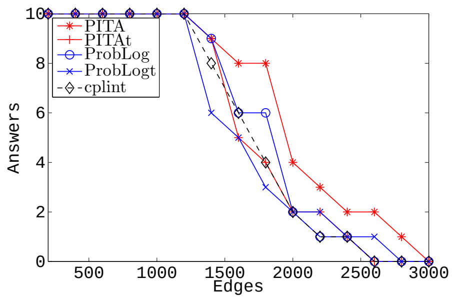
The second exploits tabling for performing loop checking:

$$\begin{array}{lcl} path(X,X). & & \\ path(X,Y,) & \leftarrow & path(X,Z), arc(Z,Y). \\ arc(X,Y) & \leftarrow & edge(X,Y). \\ arc(X,Y) & \leftarrow & edge(Y,X). \end{array} \tag{4}$$

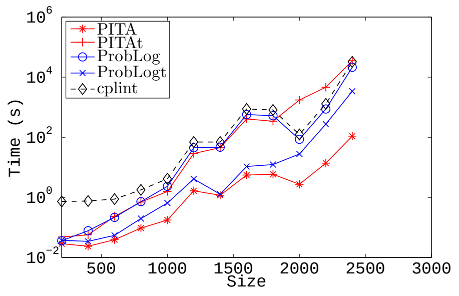
The possibility of using lists (that require function symbols) allowed in this case more modeling freedom. In PITA, the predicates $path/2$, $edge/2$ and $arc/2$ are tabled in both cases. For ProbLog we used its implementation of tabling for loop checking in the second program. As in PITA, $path/2$, $edge/2$ and $arc/2$ are tabled.

We ran PITA, ProbLog and `cplint` on the graphs starting from the smallest program. In each series we stopped after one day or at the first graph for which the program ended for lack of memory[12]. In `cplint`, PITA and ProbLog we used group sift reordering of BDDs variables. Figure 3(a) shows the number of subgraphs for

[12] CVE was not applied to this dataset because the current version can not handle graph cycles.



(a) Number of successes.



(b) Average execution times on the graphs on which all the algorithm succeeded.

Fig. 3. Biological graph experiments.

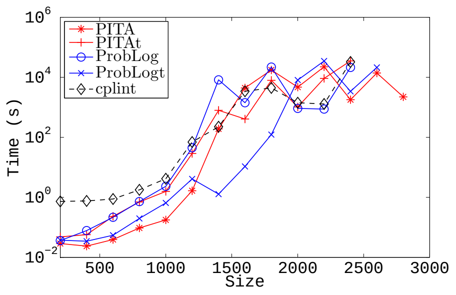
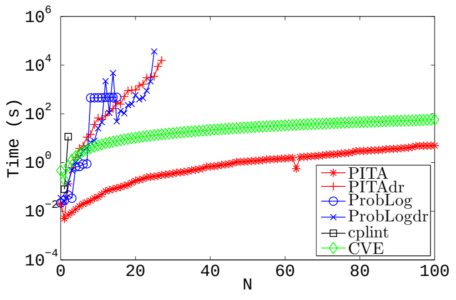


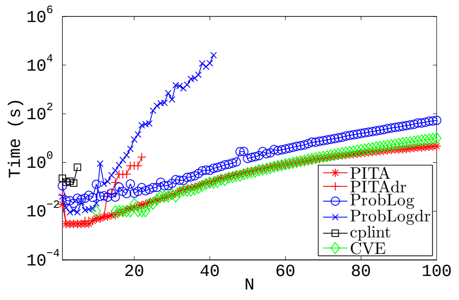
Fig. 4. Average exection times on the biological graph experiments.

which each algorithm was able to answer the query as a function of the size of the subgraphs, while Figure 3(b) shows the execution time averaged over all and only the subgraphs for which all the algorithms succeeded. Figure 4 alternately shows the execution times averaged, for each algorithm, over all the graphs on which the algorithm succeeded. In these Figures PITA and PITAt refers to PITA applied to path programs (3) and (4) respectively and similarly for ProbLog and ProbLogt.

PITA applied to program (3) was able to solve more subgraphs and in a shorter time than `cplint` and all cases of ProbLog. On path definition (4), on the other

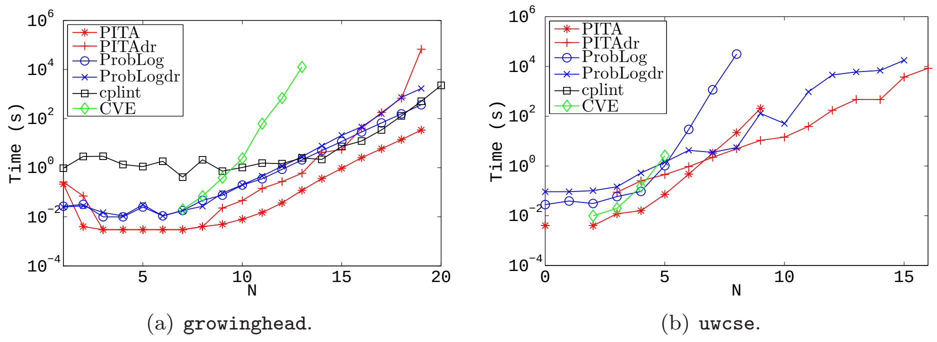


(a) `bloodtype`.



(b) `growingbody`.

Fig. 5. Datasets from (Meert et al. 2009).



(a) `growinghead`. (b) `uwcse`.

Fig. 6. Datasets from (Meert et al. 2009).

hand, ProbLogt was able to solve a larger number of problems than PITAt and in a shorter time. For PITA the vast majority of time for larger graphs was spent on BDD maintenance. This shows that, even if tabling consumes more memory when finding the explanations, BDDs are built faster and use less memory, probably due to the fact that tabling allows less redundancy (only one BDD is stored for an answer) and supports a bottom-up construction of the BDDs, which is usually better.

The four datasets of (Meert et al. 2009), served as a final suite of benchmarks. `bloodtype` encodes the genetic inheritance of blood type, `growingbody` contains programs with growing bodies, `growinghead` contains programs with growing heads and `uwcse` encodes a university domain. The best results for ProbLog were obtained by using ProbLog's tabling in all experiments except `growinghead`. The execution times of `cplint`, ProbLog, CVE and PITA are shown in Figures 5(a) and 5(b), 6(a) and 6(b)[13]. In the legend PITA means that dynamic BDD variable reordering was disabled, while PITAdr has group sift automatic reordering enabled. Similarly for ProbLog and ProbLogdr.

In `bloodtype`, `growingbody` and `growinghead` PITA without variable reordering was the fastest, while in `uwcse` PITA with group sift automatic reordering was the fastest. These results show that variable reordering has a strong impact on performances: if the variable order that is obtained as a consequence of the sequence of BDD operations is already good, automatic reordering severely hinders performances. Fully understanding the effect of variable reordering on performances is subject of future work.

11 Conclusion and Future Works

This paper has made two main contributions. The first is the identification of bounded term-size programs and queries as conditions for the distribution semantics to be well-defined when LPADs contain function symbols. As shown in Section 4.2,

[13] For the missing points at the beginning of the lines a time smaller than 10^{-6} was recorded. For the missing points at the end of the lines the algorithm exhausted the available memory.

bounded-term-size programs and queries sometimes include programs that other termination classes do not. Given the transformational equivalence of LPADs and other probabilistic logic programming formalisms that use the distribution semantics, these results may form a basis for determining well-definedness beyond LPADs.

As a second contribution, the PITA transformation provides a practical reasoning algorithm that was directly used in the experiments of Section 10. The experiments substantiate the PITA approach. Accordingly, PITA should be easily portable to other tabling engines such as that of YAP, Ciao and B Prolog if they support answer subsumption over general semi-lattices. PITA is available in XSB Version 3.3 and later, downloadable from `http://xsb.sourceforge.net`. A user manual is included in XSB manual and can also be found at `http://sites.unife.it/ml/pita`.

In the future, we plan to extend PITA to the whole class of sound LPADs by implementing the SLG DELAYING and SIMPLIFICATION operations for answer subsumption; an implementation of tabling with term-depth abstraction (Section 6) is also underway. Finally, we are developing a version of PITA that is able to answer queries in an approximate way, similarly to (Kimmig et al. 2011).

References

Baselice, S., Bonatti, P., and Criscuolo, G. 2009. On finitely recursive programs. *Theory and Practice of Logic Programming 9,* 2, 213–238.

Calimeri, F., Cozza, S., Ianni, G., and Leone, N. 2008. Computable functions in asp: Theory and implementation. In *International Conference on Logic Programming.* LNCS, vol. 5366. Springer, 407–424.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the Association for Computing Machinery 43,* 1, 20–74.

De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., and Vennekens, J. 2008. Towards digesting the alphabet-soup of statistical relational learning. In *NIPS2008 Workshop on Probabilistic Programming, 13 December 2008, Whistler, Canada.*

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *Internation Joint Conference on Artificial Intelligence.* IJCAI, 2462–2467.

Fuhr, N. 2000. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society of Information Sciences 51,* 2, 95–110.

Kameya, Y. and Sato, T. 2000. Efficient EM learning with tabulation for parameterized logic programs. In *International Conference on Computational Logic.* LNCS, vol. 1861. Springer, 269–284.

Kimmig, A., Demoen, B., De Raedt, L., Costa, V. S., and Rocha, R. 2011. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming 11,* Special Issue 2-3, 235–262.

Kimmig, A., Gutmann, B., and Santos Costa, V. 2009. Trading memory for answers: Towards tabling ProbLog. In *International Workshop on Statistical Relational Learning, 2-4 July 2009, Leuven, Belgium.* KU Leuven.

Kolmogorov, A. N. 1950. *Foundations of the Theory of Probability.* Chelsea Publishing Company.

Mantadelis, T. and Janssens, G. 2010. Dedicated tabling for a probabilistic setting. In *Technical Communications of the International Conference on Logic Programming.* LIPIcs, vol. 7. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 124–133.

Meert, W., Struyf, J., and Blockeel, H. 2009. CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *International Conference on Inductive Logic Programming.* LNCS, vol. 5989. Springer, 96–109.

Poole, D. 1997. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence 94,* 1-2, 7–56.

Poole, D. 2000. Abducing through negation as failure: stable models within the independent choice logic. *Journal of Logic Programming 44,* 1-3, 5–35.

Przymusinski, T. 1989. Every logic program has a natural stratification and an iterated least fixed point model. In *Symposium on Principles of Database Systems.* ACM Press, 11–21.

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

Riguzzi, F. 2008. Inference with logic programs with annotated disjunctions under the well founded semantics. In *International Conference on Logic Programming.* LNCS, vol. 5366. Springer, 667–771.

Riguzzi, F. and Swift, T. 2011. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming, International Conference on Logic Programming Special Issue 11,* 4-5, 433–449.

Sagonas, K., Swift, T., and Warren, D. S. 2000. The limits of fixed-order computation. *Theoretical Computer Science 254,* 1-2, 465–499.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming.* MIT Press, 715–729.

Sato, T. and Kameya, Y. 1997. Prism: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence.* IJCAI, 1330–1339.

Swift, T. 1999a. A new formulation of tabled resolution with delay. In *Recent Advances in Artifiial Intelligence.* LNAI, vol. 1695. Sringer, 163–177.

Swift, T. 1999b. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 201–240.

Tamaki, H. and Sato, T. 1986. OLDT resolution with tabulation. In *International Conference on Logic Programming.* LNCS, vol. 225. Springer, 84–98.

Thayse, A., Davio, M., and Deschamps, J. P. 1978. Optimization of multivalued decision algorithms. In *International Symposium on Multiple-Valued Logic.* IEEE Computer Society Press, 171–178.

van Gelder, A. 1989. The alternating fixpoint of logic programs with negation. In *Symposium on Principles of Database Systems.* ACM, 1–10.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery 38,* 3, 620–650.

Vennekens, J. and Verbaeten, S. 2003. Logic programs with annotated disjunctions. Tech. Rep. CW386, K. U. Leuven.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004. Logic programs with annotated disjunctions. In *International Conference on Logic Programming.* LNCS, vol. 3131. Springer, 195–209.