Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions

Fabrizio Riguzzi[1] and Terrance Swift[2]

[1] ENDIF – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
fabrizio.riguzzi@unife.it

[2] CENTRIA – Universidade Nova de Lisboa
tswift@cs.suysb.edu

Abstract

Probabilistic Logic Programming is an active field of research, with many proposals for languages, semantics and reasoning algorithms. One such proposal, Logic Programming with Annotated Disjunctions (LPADs) represents probabilistic information in a sound and simple way.

This paper presents the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) for computing the probability of queries. Answer subsumption is a feature of tabling that allows the combination of different answers for the same subgoal in the case in which a partial order can be defined over them. We have applied it in our case since probabilistic explanations (stored as BDDs in PITA) possess a natural lattice structure.

PITA has been implemented in XSB and compared with ProbLog, `cplint` and CVE. The results show that, in almost all cases, PITA is able to solve larger problems and is faster than competing algorithms.

1 Introduction

Languages that are able to represent probabilistic information have a long tradition in Logic Programming, dating back to [14, 17]. With these languages it is possible to model domains which contain uncertainty, as many real world domains do. Recently, efficient systems have started to appear for performing reasoning with these languages [3, 6]

Logic Programs with Annotated Disjunction (LPADs) [20] are a particularly interesting formalism because of the simplicity of their syntax and semantics along with their ability to model causation [19]. LPADs share with many other languages a distribution semantics [13]: a theory defines a probability distribution over logic programs and the probability of a query is given by the sum of the probabilities of the programs where the query is true. In LPADs the distribution over logic programs is defined by means of disjunctive clauses in which the atoms in the head are annotated with a probability.

Proceedings of the 17th International RCRA workshop (RCRA 2010):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Bologna, Italy, June 10–11, 2010

Various approaches have appeared for performing inference on LPADs. [11] proposed `cplint` that first finds all the possible explanations for a query and then makes them mutually exclusive by using Binary Decision Diagrams (BDDs), similarly to what has been proposed for the ProbLog language [3]. [12] presented SLGAD resolution that extends SLG resolution by repeatedly branching on disjunctive clauses. [8] discusses the CVE algorithm that first transforms an LPAD into an equivalent Bayesian network and then performs inference on the network using the variable elimination algorithm.

In this paper, we present the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) for computing the probability of queries from LPADs. PITA builds explanations for every subgoal encountered during a derivation of the query. The explanations are compactly represented using BDDs that also allow an efficient computation of the probability. Since all the explanations for a subgoal must be found, it is very useful to store such information so that it can be reused when the subgoal is encountered again. We thus propose to use tabling, which has already been shown useful for probabilistic logic programming in [4, 12, 5, 7]. This is achieved by transforming the input LPAD into a normal logic programs in which the subgoals have an extra argument storing a BDD that represents the explanations for its answers. Moreover, we also exploit answer subsumption to combine explanations coming from different clauses. PITA is tested on a number of datasets and compared with `cplint`, CVE and ProbLog [6]. The algorithm was able to successfully solve more complex queries than the other algorithms in most cases and it was also almost always faster.

The paper is organized as follows. Section 2 briefly recalls tabling and answer subsumption. Section 3 illustrates syntax, semantics and inference for LPADs. Section 4 gives an introduction to BDDs. Section 5 presents PITA and shows its correctness. Section 6 describes the experiments and Section 7 concludes the paper and presents directions for future works.

2 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 of programs with the *bounded term size property*. A program P has the bounded term size property if there is a finite function $f : N \rightarrow N$ such that if, a query term Q to P has size $size(Q)$, then no term used in the derivation of Q has size greater than $f(size(Q))$. This makes it easier to reason about termination than in basic Prolog. Second, tabling can be used to evaluate programs with negation according to the Well-Founded Semantics (WFS) [18]. 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. In the case of a lattice, answer subsumption may be specified by means of a declaration such as *table arc(_,_,or/3 - zero/1)).* which indicates that if a table contains an answer $arc(Arg_1, Arg_2, Arg_{1,3})$, and a new answer $arc(Arg_1, Arg_2, Arg_{2,3})$ is derived, then $arc(Arg_1, Arg_2, Arg_{1,3})$ is replaced by $arc(Arg_1, Arg_2, or(Arg_{1,3}, Arg_{2,3}))$ ($zero/1$ is the bottom element of the lattice). In the PITA algorithm for LPADs presented in Section 5, if a table had an answer $arc(a, b, E_1)$ and a new answer $arc(a, b, E_2)$ were derived, where E_1 and E_2 are probabilistic explanations, the answer $arc(a, b, E_1)$ is replaced by $arc(a, b, E_3)$, where E_3 is the logical disjunction of the first two explanations, as stored in a BDD[1]. Answer subsumption over arbitrary upper semi-lattices is implemented in XSB for stratified programs [15]; in addition, the mode-directed tabling of B-Prolog can also be seen as a form of answer subsumption.

Section 5 uses the SLG resolution [1] extended with answer subsumption in its proof of Theorem 2, although similar results could be extended to other tabling formalisms that support negation and answer subsumption.

3 Logic Programs with Annotated Disjunctions

A *Logic Program with Annotated Disjunctions* [20] consists of a finite set of annotated disjunctive clauses of the form

$$h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow b_1, \ldots, b_m$$

In such a clause $h_1, \ldots h_n$ are logical atoms and $b_1, \ldots, b_m$ are logical literals, $\{\alpha_1, \ldots, \alpha_n\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{j=1}^{n} \alpha_j \leq 1$. $h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n$ is called the *head* and $b_1, \ldots, b_m$ is called the *body*. Note that if $n = 1$ and $\alpha_1 = 1$ a clause corresponds to a normal program clause, sometimes called a *non-disjunctive* clause. If $\sum_{j=1}^{n} \alpha_j < 1$, the head of the

[1] The logical disjunction E_3 can be seen as subsuming E_1 and E_2 over the partial order af implication defined on logical formulas.

annotated disjunctive 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 of the form above, 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 $\{b_i|1 \leq i \leq m\}$, $h_i(C)$ as h_i and $\alpha_i(C)$ as α_i.

The semantics of LPADs, given in [20], requires the ground program to be finite, so the program must not contain function symbols if it contains variables. 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 can be 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 *possible world* of the LPAD (an *instance* of the LPAD in [20]). A probability distribution is defined over the space of possible 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 substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for ground clause $C\theta$, the head $h_i(C)$ 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.

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. We denote the set of all selections σ of a program T by $\mathcal{S}_T$. 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 identifies a normal logic program w_σ defined as follows $w_\sigma = \{(h_i(C)\theta \leftarrow body(C))\theta|(C, \theta, i) \in \sigma\}$. w_σ is called a *possible world* (or simply *world*) of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} \alpha_i(C)$.

We consider only *sound* LPADs in which every possible world has a total well-founded model. In this way, the uncertainty is modeled only by means of the disjunctions in the head and not by the features of the semantics. In the following we will write $w_\sigma \models \phi$ to mean that the closed formula ϕ is true in the well-founded model of the program w_σ.

The probability of a closed formula ϕ according to an LPAD T is given by the sum of the probabilities of the possible worlds where the formula is true according to the WFS:

$$P(\phi) = \sum_{\sigma\in\mathcal{S}_T, w_\sigma\models\phi} P(\sigma)$$

It is easy to see that P satisfies the axioms of probability.

Example 1. *Consider the dependency of a person's sneezing on his 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. Flu causes strong sneezing with probability 0.3, moderate sneezing with probability 0.5 and no sneezing with probability $1-0.3-0.5 = 0.2$*; hay fever causes strong sneezing with probability 0.4, moderate sneezing with probability 0.3 and no sneezing with probability* $1-0.2-0.6 = 0.2$*.* $strong\_sneezing(david)$ *is true in 5 of the 9 instances of the program and its probability is*

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

If the LPAD contains function symbols, its semantics can be given by following the approach proposed in [10] for assigning a semantics to ICL programs with function symbols. A similar result can be obtained using the approach of [13]. In the extended version of this paper[2] we discuss how this can be done.

In order to compute the probability of a query, we can first find a set of *covering explanations* and then compute the probability from them.

A composite choice κ identifies a set of possible worlds ω_κ that contains all the worlds relative to a selection that is a superset of κ, i.e.,

$$\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_T, \sigma \supseteq \kappa\}$$

Similarly we can define the set of possible worlds associated to a set of composite choices K:

$$\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$$

Given a closed formula ϕ, we define the notion of explanation and of covering set of composite choices. A finite composite choice κ is an *explanation* for ϕ if ϕ 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 choices K is *covering* with respect to ϕ if every world w_σ in which ϕ is true is such that $w_\sigma \in \omega_K$. In Example 1, the set of composite choices

$$L_1 = \{\{(C_1, \{X/david\}, 1)\}, \{(C_2, \{X/david\}, 1)\}\} \tag{1}$$

is covering for $strong\_sneezing(david)$. Moreover, both elements of L_1 are explanations, so L_1 is a covering set of explanations for the query $strong\_sneezing(david)$.

[2] `http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/RigSwi09-TR.pdf`.

We associate to each ground clause $C\theta$ appearing in a covering set of explanations a multivalued variable $X_{C\theta}$ with values $\{1, \ldots, head(C)\}$. Each atomic choice (C, θ, i) can then be represented by the propositional equation $X_{C\theta} = i$. If we conjoin equations for a single explanation and disjoin expressions for the different explanations we obtain a Boolean function that assumes value 1 if the values assumed by the multivalued variables correspond to an explanation for the goal.

Thus, if K is a covering set of explanations for a query ϕ, the probability of the Boolean formula

$$f(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(C,\theta,i) \in \kappa} X_{C\theta} = i$$

taking value 1 is the probability of the query, where $\mathbf{X}$ is the set of all ground clause variables.

For example, the covering set of explanations of Equation (1) translates into the function

$$f(\mathbf{X}) = (X_{C_1\emptyset} = 1) \vee (X_{C_2\emptyset} = 1) \tag{2}$$

Computing the probability of $f(\mathbf{X})$ taking value 1 is equivalent to computing the probability of a DNF formula which is an NP-hard problem. In order to solve it as efficiently as possible we use Decision Diagrams, as proposed by [3].

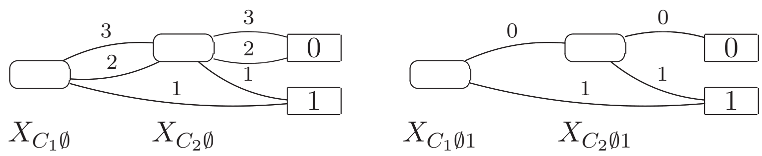
4 Representing Explanations by Means of Decision Diagrams

In order to compute the probability of Boolean expressions over multi-valued variables we can use Multivalued Decision Diagrams [16]. 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 has one child for each possible value of the multivalued variable associated to the level of the node. The leaves store either 0 or 1. Given values for all the variables $\mathbf{X}$, an MDD can 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. For example, the MDD corresponding to Function 2 is shown in Figure 1(a).

The advantage of MDDs is that they represent a Boolean function $f(\mathbf{X})$ by means of a generalization of the Shannon's expansion

$$f(\mathbf{X}) = [(X_1 = 1) \wedge f_{X_1=1}(\mathbf{X})] \vee \ldots \vee [(X_1 = n) \wedge f_{X_1=n}(\mathbf{X})]$$

where X_1 is the variable associated to the root node of the diagram and $f_{X_1=i}(\mathbf{X})$ is the function associated to the i-th child of the root node. The



$X_{C_1\emptyset}$ $X_{C_2\emptyset}$
(a) MDD.

$X_{C_1\emptyset 1}$ $X_{C_2\emptyset 1}$
(b) BDD.

Figure 1: Decision diagrams for Example 1.

expansion can be applied recursively to the functions $f_{X_1=1}(\mathbf{X})$. This expansion allows the probability of $f(\mathbf{X})$ to be expressed by means of the following recursive formula

$$P(f(\mathbf{X})) = [P(X_1 = 1) \cdot P(f_{X_1=1}(\mathbf{X}))] + \ldots + [P(X_1 = n) \cdot P(f_{X_1=n}(\mathbf{X}))]$$

because the disjuncts are mutually exclusive due to the presence of the $X_1 = i$ equations. Thus the probability of $f(\mathbf{X})$ can be computed by means of a dynamic programming algorithm that traverses the MDD and sums up probabilities.

Decision diagrams can be built with various software packages that provide highly efficient implementation of Boolean operations. However, most packages are restricted to work on Binary Decision Diagram (BDD), i.e., decision diagrams where all the variables are Boolean. To work on MDD 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 [2], gives the best performance. For a variable X_1 having n values, we use $n-1$ Boolean variables $X_{11}, \ldots, X_{1n-1}$ and we represent the equation $X_1 = i$ for $i = 1, \ldots n-1$ by means of the conjunction $\overline{X_{11}} \wedge \overline{X_{12}} \wedge \ldots \wedge \overline{X_{1i-1}} \wedge X_{1i}$, and the equation $X_1 = n$ by means of the conjunction $\overline{X_{11}} \wedge \overline{X_{12}} \wedge \ldots \wedge \overline{X_{1n-1}}$. The BDD representation of the function in Equation 2 is given in Figure 1(b). The Boolean variables are associated with the following parameters: $P(X_{11}) = P(X_1 = 1) \ldots P(X_{1i}) = \frac{P(X_1=i)}{\prod_{j=1}^{i-1}(1-P(X_{1i-1}))}$.

5 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[3] C library and use the following predicates[4]

- *init, end*: for allocation and deallocation of a BDD manager, a data structure used to keep track of the memory for storing BDD nodes;

[3] `http://vlsi.colorado.edu/~fabio/`

[4] BDDs are represented in CUDD as pointers to their root node.

- *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 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 auxiliary predicate *get_var_n/4* is used 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 grounding *S*. The auxiliary predicates has the following definition

$get\_var\_n(R, S, Probs, Var) \leftarrow$
$\quad (var(R, S, Var) \rightarrow true;$
$\quad length(Probs, L), add\_var(L, Probs, Var), assert(var(R, S, Var))).$

where *Probs* is a list of floats that stores the parameters in the head of the rule. *R*, *S* and *Probs* are input arguments while *Var* is an output argument.

The PITA transformation applies to clauses, literals and atoms.

- If h is an atom, $PITA_h(h)$ is h with the variable BDD added as the last argument.
- If b_j is an atom, $PITA_b(b_j)$ is b_j with the variable B_j added as the last argument.

In either case for an atom a, $BDD(\mathrm{PITA}(a))$ is the value of the last argument of $PITA(a)$,

- If b_j is negative literal $\neg a_j$, $PITA_b(b_j)$ 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_k 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
$PITA(C_r) = PITA_h(h) \leftarrow one(BDD)$.

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 $PITA(C_r)$

$$PITA(C_r,1) = PITA_h(h_1) \leftarrow \quad get\_var\_n(i,[],[\alpha_1,\ldots,\alpha_n],Var),$$
$$equality(Var,1,BDD).$$

...

$$PITA(C_r,n) = PITA_h(h_n) \leftarrow \quad get\_var\_n(r,[],[\alpha_1,\ldots,\alpha_n],Var),$$
$$equality(Var,n,BDD).$$

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 $C_r = h \leftarrow b_1, b_2, \ldots, b_m$. is transformed into the clause

$$PITA(C_r) = PITA_h(h) \leftarrow \quad PITA_b(b_1), PITA_b(b_2),$$
$$and(B_1,B_2,BB_2),\ldots,$$
$$PITA_b(b_m), and(BB_{m-1},B_m,BDD).$$

The disjunctive clause

$$C_r = h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow b_1, b_2, \ldots, b_m.$$

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

$$PITA(C_r,1) = PITA_h(h_1) \leftarrow \quad PITA_b(b_1), PITA_b(b_2),$$
$$and(B_1,B_2,BB_2),\ldots,$$
$$PITA_b(b_m), and(BB_{m-1},B_m,BB_m),$$
$$get\_var\_n(r,VC,[\alpha_1,\ldots,\alpha_n],Var),$$
$$equality(Var,1,B),$$
$$and(BB_m,B,BDD).$$

...

$$PITA(C_r,n) = PITA_h(h_n) \leftarrow \quad PITA_b(b_1), PITA_b(b_2),$$
$$and(B_1,B_2,BB_2),\ldots,$$
$$PITA_b(b_m), and(BB_{m-1},B_m,BB_m),$$
$$get\_var\_n(r,VC,[\alpha_1,\ldots,\alpha_n],Var),$$
$$equality(Var,n,B),$$
$$and(BB_m,B,BDD).$$

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 2. *Clause C_1 from the LPAD of Example 1 is translated into*

$$strong\_sneezing(X,BDD) \leftarrow flu(X,B_1),$$
$$get\_var\_n(1,[X],[0.3,0.5,0.2],Var),$$
$$equality(Var,1,B), and(B_1,B,BDD).$$
$$moderate\_sneezing(X,BDD) \leftarrow flu(X,B_1),$$
$$get\_var\_n(1,[X],[0.3,0.5,0.2],Var),$$
$$equality(Var,2,B), and(B_1,B,BDD).$$

while clause C_3 is translated into

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

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

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

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 should be tabled; as in normal programs, tabling may also be used for to ensure termination of recursive predicates, or to reduce the complexity of evaluations.

Correctness of PITA In this section we consider the correctness of the PITA transformation and its tabled evaluation[5]. For the purposes of our semantics, we consider the BDDs produced as ground terms, and do not specify them further. We first state the correctness of the PITA transformation with respect to the well-founded semantics of LPADs. Because we allow LPADs to have function symbols, care must be taken to ensure that explanations are finite. To accomplish this, we prove correctness for what we term finitary programs, essentially those for which a derivation in the well-founded semantics does not depend on an infinite unfounded set. In the statement of Theorem 1, for a ground atom a for a predicate p/n, $PITA(a)$ is an atom of predicate $p/(n+1)$ whose last argument is a variable for the BDD. In the well-founded model, an atom $PITA(a)\theta$ has its last argument instantiated to a given BDD: $BDD(PITA(a)\theta)$.

Theorem 1 (Correctness of PITA Transformation)**.** Let T be a sound finitary LPAD. Then σ is a finite explanation for a ground atom a iff there is some $PITA(a)\theta$ in $WFM(PITA(ground(T)))$, such that σ is a path to a 1 leaf in $BDD(PITA(a)\theta)$

Theorem 2 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 but that is restricted to fixed-order dynamically stratified programs [15], a formalism that models the implementation tested in Section 6. Note that unlike Theorem 1, Theorem 2 does not require the program T to be grounded. However, Theorem 2 does require T to be range restricted in order to ensure that tabled evaluation grounds answers. A normal program/LPAD is *range restricted* if all the

[5]Due to space limitations, our presentation is somewhat informal: a formal presentation with all proofs and supporting definitions can be found at `http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/RigSwi09-TR.pdf`.

variables appearing in the head of each clause appear also in the body. If a normal program is range restricted, every successful SLDNF-derivation for G completely grounds G [9], a result that can be straightforwardly extended to tabled evaluations. In addition, Theorem 2 requires T to have the bounded term size property (cf. Section 2).

Theorem 2 (Correctness of PITA Evaluation)**.** Let T be a range restricted, bounded term depth, fixed-order dynamically stratified LPAD and a a ground atom. Let $\mathcal{E}$ be an SLG evaluation of $PITA(a)$ against $PITA(T)$, such that answer subsumption is declared on $PITA(a)$ using BDD-disjunction. Then $\mathcal{E}$ terminates with an answer ans for $PITA(a)$ and $BDD(ans)$ represents a set of covering explanations for a.

6 Experiments

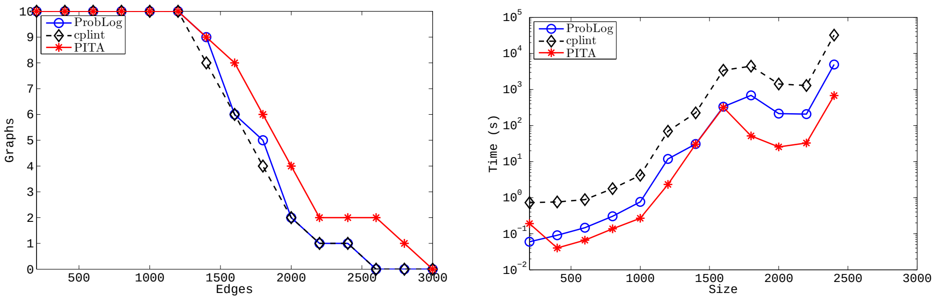
PITA was tested on programs encoding biological networks from [3], a game of dice from [20] and the four testbeds of [8]. PITA was compared with the exact version of ProbLog [3] available in the git version of Yap as of 19/12/2009, with the version of `cplint` [11] available in Yap 6.0 and with the version of CVE [8] available in ACE-ilProlog 1.2.20[6]. The biological network problems 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 programs in this dataset contains a 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, . . ., 5000 edges. Sampling has been repeated ten times, so overall we have 10 series of programs of increasing size. In each test we queried the probability that the two genes HGNC_620 and HGNC_983 are related[7].

We ran PITA, ProbLog and `cplint` on the graphs in sequence starting from the smallest program and in each case we stopped after one day or at the first graph for which the program ended for lack of memory[8]. In PITA, we used the group sift method for automatic reordering of BDDs variables. Figure 2(a) shows the number of subgraphs for which each algorithm was able to answer the query as a function of the size of the subgraphs, while Figure 2(b) shows the execution time averaged over all and only the subgraphs for which all the algorithms succeeded. PITA was able to solve more subgraphs and in a shorter time than `cplint` and ProbLog. For PITA the vast majority of time for larger graphs was spent on BDD maintenance.

[6]All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.

[7]The definition of path that is used is the one in [6] that performs loop checking explicitly by keeping the list of visited nodes.

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



(a) Number of successes.

(b) Execution times.

Figure 2: Biological graph experiments.

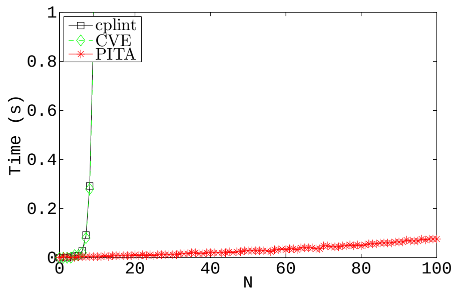


Figure 3: Three sided die.

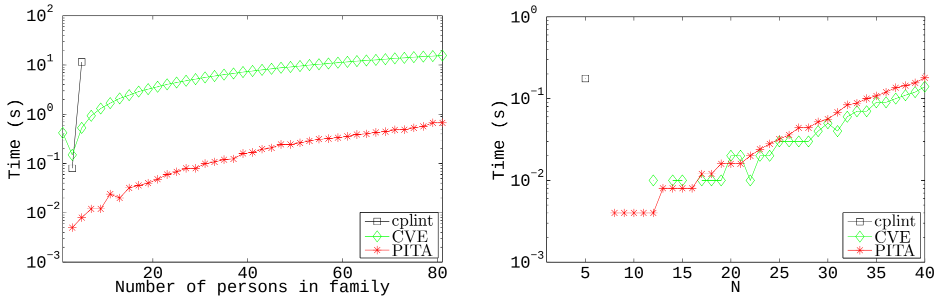
The second problem[9] models a game in which a die with three faces is repeatedly thrown until a 3 is obtained. This problem is encoded by the program

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).

For this problem, we query the probability of *on(N,F)* for increasing values of *N*. In PITA, we disabled automatic reordering of BDDs variables. The execution times of PITA, CVE and `cplint` are shown in Figure 3. In this problem, tabling provides an impressive speedup, since computations can be reused often.

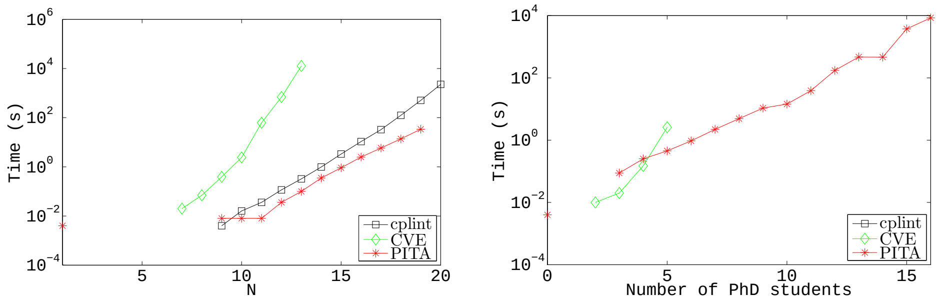
The four datasets of [8], containing programs of increasing size. served as a final suite of benchmarks. `bloodtype` encodes the genetic inheritance of the blood type, `growingbody` contains programs with growing bodies, `growinghead` contains programs with growing heads and `uwcse` encodes a university domain. In PITA we disabled automatic reordering of BDDs

[9] In the remaining problems, ProbLog was not considered because the publicly available version is not yet able to deal with non-binary variables



(a) `bloodtype`. (b) `growingbody`.

Figure 4: Datasets from (Meert et al. 2009).



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

Figure 5: Datasets from (Meert et al. 2009).

variables for all datasets except for `uwcse` where we used automatic reordering with the group sift method. The execution times of `cplint`, CVE and PITA are shown respectively in Figures 4(a), 4(b), 5(a) and 5(b)[10]. PITA was faster than `cplint` in all domains and faster than CVE in all domains except `growingbody`.

7 Conclusion and Future Works

This paper presents the algorithm PITA for computing the probability of queries from an LPAD. PITA is based on a program transformation approach in which LPAD disjunctive clauses are translated into definite clauses. We show that PITA is correct for finitary LPADs with the bounded term depth property. Such results may be used to guide termination and other analysis for tabled evaluation of LPADs with functions.

The experiments substantiate the PITA approach which uses BDDs to-

[10] 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.

gether with tabling with answer subsumption. PITA outperformed `cplint`, CVE and ProbLog in expressiveness, scalability or speed in almost all domains considered. The implementation of PITA is greatly simplified by its use of answer subsumption, which is a comparatively easy extension to an engine that already performs tabling. Accordingly PITA programs 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.

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. In addition, we plan to develop a version of PITA that is able to answer queries in an approximate way, similarly to [6].

References

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

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

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

[4] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. In *Computational Logic*, volume 1861 of *LNCS*, pages 269–284. Springer, 2000.

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

[6] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. On the efficient execution of ProbLog programs. In *International Conference on Logic Programming*, volume 5366 of *LNCS*, pages 175–189. Springer, 2008.

[7] T. Mantadelis and G. Janssens. Tabling relevant parts of sld proofs for ground goals in a probabilistic setting. In *Colloquium on Implementation of Constraint and Logic Programming Systems*, 2009.

[8] W. Meert, J. Struyf, and H. Blockeel. CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference

methods. In *International Conference on Inductive Logic Programming*, Leuven, Belgium, 2009. KU LEuven.

[9] S. Muggleton. Learning stochastic logic programs. *Electron. Trans. Artif. Intell.*, 4(B):141–153, 2000.

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

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

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

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

[14] E. Y. Shapiro. Logic programs with uncertainties: a tool for implementing rule-based systems. In *International Joint conference on Artificial intelligence*, pages 529–532. Morgan Kaufmann Publishers Inc., 1983.

[15] T. Swift. Tabling for non-monotonic programming. *Ann. Math. Artif. Intell.*, 25(3-4):201–240, 1999.

[16] A. Thayse, M. Davio, and J. P. Deschamps. Optimization of multi-valued decision algorithms. In *International Symposium on Multiple-Valued Logic*, pages 171–178, Los Alamitos, CA, USA, 1978. IEEE Computer Society Press.

[17] M. H. van Emden. Quantitative deduction and its fixpoint theory. *J. Log. Program.*, 30(1):37–53, 1986.

[18] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[19] J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, 9(3):245–308, 2009.

[20] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.