Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 162–171
http://www.floc-conference.org/ICLP-home.html

TABLING AND ANSWER SUBSUMPTION FOR REASONING ON LOGIC PROGRAMS WITH ANNOTATED DISJUNCTIONS

FABRIZIO RIGUZZI [1] AND TERRANCE SWIFT [2]

[1] ENDIF – Università di Ferrara, Via Saragat 1, Ferrara, Italy
E-mail address: fabrizio.riguzzi@unife.it

[2] CENTRIA – Universidade Nova de Lisboa, Quinta da Torre 2829-516, Caparica, Portugal
E-mail address: tswift@cs.suysb.edu

Abstract. The paper presents the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) for computing the probability of queries from Logic Programs with Annotated Disjunctions. PITA is based on a program transformation techniques that adds an extra argument to every atom. PITA uses tabling for saving intermediate results and answer subsumption for combining different answers for the same subgoal. PITA has been implemented in XSB and compared with the ProbLog, `cplint` and CVE systems. The results show that in almost all cases, PITA is able to solve larger problems and is faster than competing algorithms.

Introduction

Languages that are able to represent probabilistic information have a long tradition in Logic Programming, dating back to [Sha83, van86]. With these languages, it is possible to model domains which contain uncertainty, situation often appearing in the real world. Recently, efficient systems have started to appear for performing reasoning with these languages [DR07, Kim08]

Logic Programs with Annotated Disjunction (LPADs) [Ven04] are a particularly interesting formalism because of the simplicity of their syntax and semantics, along with their ability to model causation [Ven09]. LPADs share with many other languages a distribution semantics [Sat95]: 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.

Various approaches have appeared for performing inference on LPADs. [Rig07] 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 [DR07]. [Rig08] presented SLGAD resolution that extends SLG resolution by repeatedly branching on disjunctive clauses. [Mee09] 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.

Key words and phrases: Probabilistic Logic Programming, Tabling, Answer Subsumption, Logic Programs with Annotated Disjunction, Program Transformation.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.162

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 recently been shown useful for probabilistic logic programming in [Kam00, Rig08, Kim09, Man09]. This is achieved by transforming 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. Moreover, we also exploit answer subsumption to combine different explanations for the same answer. PITA is tested on a number of datasets and compared with `cplint`, CVE and ProbLog [Kim08]. 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 1 briefly recalls tabling and answer subsumption. Section 2 illustrates syntax, semantics and inference for LPADs. Section 3 presents PITA, Section 4 describes the experiments and Section 5 concludes the paper.

1. 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) [van91]. 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 p(_,or/3 - zero/1)).* where a lattice is defined on the second argument by providing a bottom element (returned by $zero/1$) and a join operation ($or/3$). With the previous declaration, if a table contains an answer $p(a, E_1)$ and a new answer $p(a, E_2)$ were derived, the answer $p(a, E_1)$ is replaced by $p(a, E_3)$, where E_3 is obtained by $or(E_1, E_2, E_3)$. Answer subsumption over arbitrary upper semi-lattices is implemented in XSB for stratified programs [Swi99]; in addition, the mode-directed tabling of B-Prolog can also be seen as a form of answer subsumption.

2. Logic Programs with Annotated Disjunctions

A *Logic Program with Annotated Disjunctions* [Ven04] consists of a finite set of annotated disjunctive clauses of the form $h_1 : \alpha_1 \; ; \; \ldots \; ; \; h_n : \alpha_n \leftarrow b_1, \ldots, b_m$. 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 $\sum_{j=1}^{n} \alpha_j \leq 1$. $h_1 : \alpha_1 \; ; \; \ldots \; ; \; h_n : \alpha_n$ 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 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.

If LPAD T is ground, a clause represents a probabilistic choice between the non-disjunctive clauses obtained by selecting only one atom in the head. If T is not ground, it can be assigned a meaning by computing its grounding, $ground(T)$. The semantics of LPADs, given in [Ven04], requires the ground program to be finite, so the program must not contain function symbols if it contains variables.

By choosing a head atom for each ground clause of an LPAD we get a normal logic program called a *possible world* of the LPAD (*instance* in [Ven04]). A probability distribution is defined over the space of possible worlds by assuming independence between the choices made for each clause.

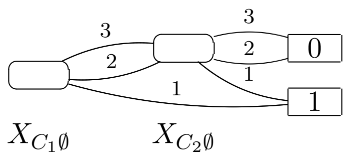
More specifically, an *atomic choice* is a triple (C, θ, i) where $C \in T$, θ is a substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for ground clause $C\theta$, the head $h_i(C)$ was chosen. A set of atomic choices κ is *consistent* if $(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, i.e., only one head is selected for a ground clause. A *composite choice* κ is a consistent set of atomic choices. 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. 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 defined as follows: $w_\sigma = \{(h_i(C) \leftarrow body(C))\theta | (C, \theta, i) \in \sigma\}$. 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: $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i) \in \sigma} \alpha_i(C)$.

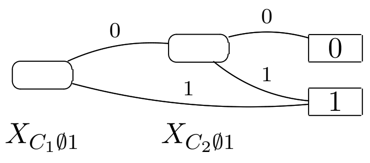
We consider only *sound* LPADs in which every possible world has a total well-founded model. In this way, the uncertainty is modeled only by means of the disjunctions in the head and not by the features of the semantics. In the following we write $w_\sigma \models \phi$ to mean that the closed formula 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 2.1. The following LPAD encodes the dependency of a person's sneezing on his having the flu or hay fever:



(a) MDD.



(b) BDD.

Figure 1: Decision diagrams for Example 2.1.

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

If the LPAD contains function symbols, its semantics can be given by following the approach proposed in [Poo00] for assigning a semantics to ICL programs with function symbols. A similar result can be obtained using the approach of [Sat95]. In a forthcoming extended version of this paper we discuss how this can be done.

In order to compute the probability of a query, we can first find a *covering set of 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 2.1, 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 2.1, the set of composite choices $L_1 = \{\{(C_1, \{X/david\}, 1)\}, \{(C_2, \{X/david\}, 1)\}\}$ 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 $strong\_sneezing(david)$.

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 L_1 translates into the function $f(\mathbf{X}) = (X_{C_1\emptyset} = 1) \vee (X_{C_2\emptyset} = 1)$. 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 [DR07].

A Multivalued Decision Diagram (MDD) [Tha78] 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. For example, the MDD corresponding to the function for L_1 is shown in Figure 1(a). MDDs represent a Boolean function $f(\mathbf{X})$ by means of a sum of disjoint terms, 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 [Bry86]. 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 [DR08], 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 for L_1 is given in Figure 1(b). The Boolean variables are associated with the following parameters: $P(X_1) = P(X = 1), \ldots, P(X_i) = \frac{P(X=i)}{\prod_{j=1}^{i-1}(1-P(X_j))}$.

3. 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[1] C library and use the following predicates[2]

- *init, end*: for the 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 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 *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.

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

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

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 BDD BN_j for a is negated if it exists (i.e. $PITA'_b(a_j)$ succeeds); 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 $C_r = h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n$. where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$

$$\begin{array}{ll} PITA(C_r, 1) = PITA_h(h_1) \leftarrow & get\_var\_n(r, [], [\alpha_1, \ldots, \alpha_n], Var), \\ & equality(Var, 1, BDD). \\ \ldots & \\ PITA(C_r, n) = PITA_h(h_n) \leftarrow & get\_var\_n(r, [], [\alpha_1, \ldots, \alpha_n], Var), \\ & equality(Var, n, BDD). \end{array}$$

In the case where the parameters do not sum to one, the clause is first transformed into $h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n \ ; \ null : 1 - \sum_1^n \alpha_i$. 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

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

The disjunctive clause
$C_r = h_1 : \alpha_1 \ ; \ \ldots \ ; \ h_n : \alpha_n \leftarrow b_1, b_2, \ldots, b_m$.
where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$

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

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 can be applied.

Example 3.1. Clause C_1 from the LPAD of Example 2.1 is translated into

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

while clause C_3 is translated into
$flu(david, BDD) \quad \leftarrow one(BDD)$.

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

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

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.

PITA is correct for range restricted, bounded term-size and fixed-order dynamically stratified LPADs. A formal presentation with all proofs and supporting definitions will be reported in a forthcoming extended version of this paper.

4. Experiments

PITA was tested on programs encoding biological networks from [DR07], a game of dice from [Ven04] and the four testbeds of [Mee09]. PITA was compared with the exact version of ProbLog [DR07] available in the git version of Yap as of 19/12/2009, with the version of `cplint` [Rig07] available in Yap 6.0 and with the version of CVE [Mee09] available in ACE-ilProlog 1.2.20. All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 processor (2333 MHz) and 4 GB of RAM.

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 deterministic 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.

We used the definition of path of [Kim08] that performs loop checking explicitly by keeping the list of visited nodes:

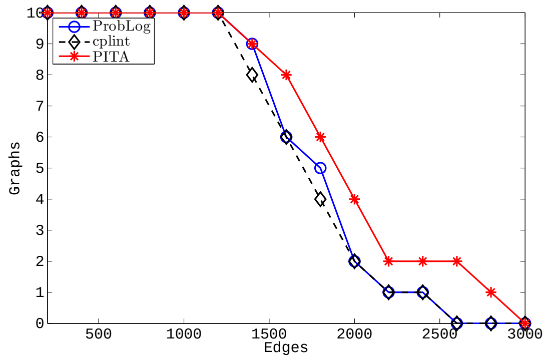
$$\begin{array}{lll} \mathit{path}(X,Y) & \leftarrow & \mathit{path}(X,Y,[X],Z).\\ \mathit{path}(X,Y,V,[Y|V]) & \leftarrow & \mathit{edge}(X,Y).\\ \mathit{path}(X,Y,V0,V1) & \leftarrow & \mathit{edge}(X,Z), \mathit{append}(V0,\_S,V1),\\ & & \neg \mathit{member}(Z,V0), \mathit{path}(Z,Y,[Z|V0],V1). \end{array}$$

This definition gave better results than the one without explicit loop checking. We are currently investigating the reasons for this unexpected behavior.

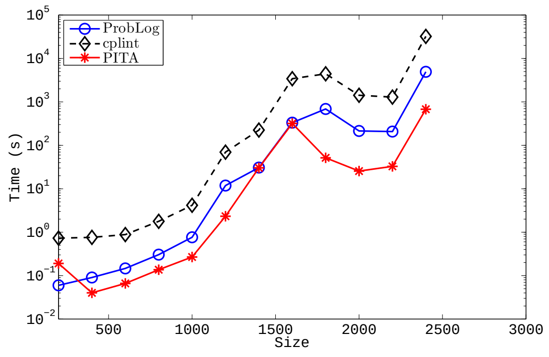
We ran PITA, ProbLog and `cplint` on the graphs in sequence starting from the smallest program and in each case we stopped after one day or at the first graph for which the program ended for lack of memory[3]. In PITA, we used the group sift method for automatic reordering of BDDs variables[4]. Figure 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

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

[4]For each experiment, we used either group sift automatic reordering or no reordering of BDDs variables depending on which gave the best results.



(a) Number of successes.



(b) Execution times.

Figure 2: Biological graph experiments.

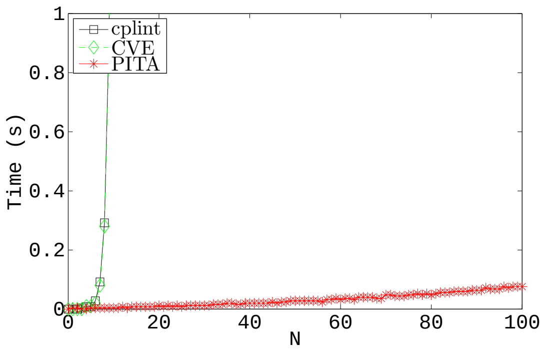


Figure 3: Three sided die.

than `cplint` and ProbLog. For PITA the vast majority of time for larger graphs was spent on BDD maintenance.

The second problem 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

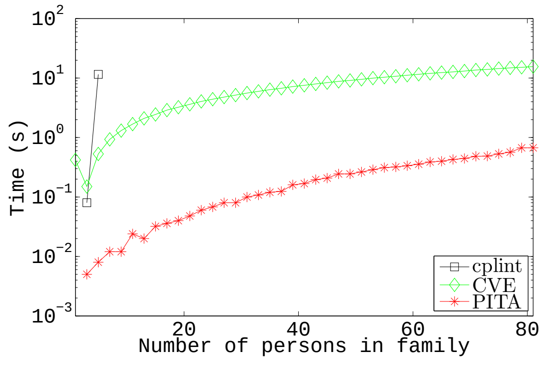
$on(0,1) : 1/3 \; ; \; on(0,2) : 1/3 \; ; \; on(0,3) : 1/3.$

$on(N,1) : 1/3 \; ; \; on(N,2) : 1/3 \; ; \; on(N,3) : 1/3 \leftarrow$
$\quad N1 \text{ is } N-1, N1 \geq 0, on(N1,F), \neg on(N1,3).$

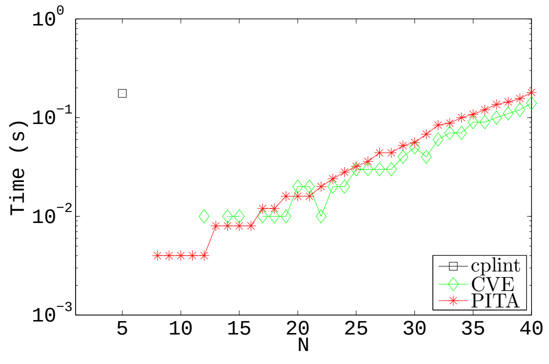
Form the above program, we query the probability of *on(N,1)* for increasing values of *N*. Note that this problem can also be seen as computing the probability that a Hidden Markov Model (HMM) is in state 1 at time N, where the HMM has three states of which 3 is an end state.

In PITA, we disabled automatic variable reordering. 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 [Mee09], containing programs of increasing size. served as a final suite of benchmarks. `bloodtype` encodes genetic inheritance of blood type, `growingbody` and `growinghead` contains programs with growing bodies and heads respectively, and `uwcse` encodes a university domain. In PITA we disabled automatic reordering of BDDs variables

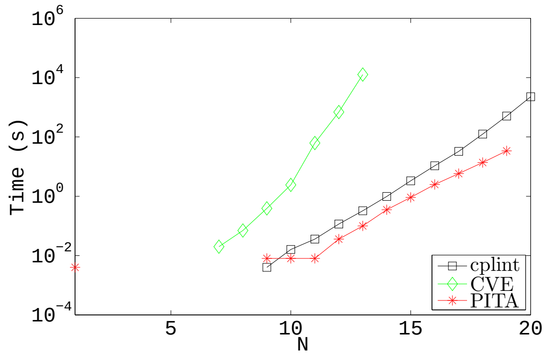


(a) `bloodtype`.

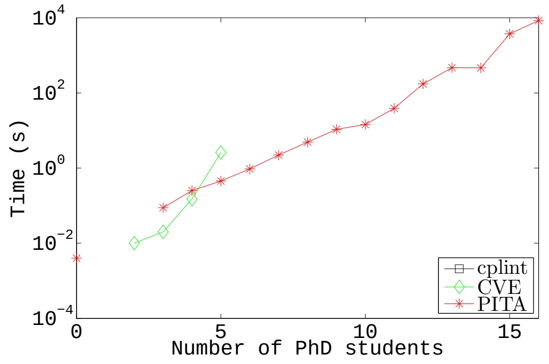


(b) `growingbody`.

Figure 4: Datasets from [Mee09].



(a) `growinghead`.



(b) `uwcse`.

Figure 5: Datasets from [Mee09].

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

5. 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 normal program clauses.

The experiments substantiate the PITA approach which uses BDDs together with tabling with answer subsumption. PITA outperformed `cplint`, CVE and ProbLog in scalability or speed in almost all domains considered.

[5]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.

References

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comput.*, 35(8):677–691, 1986.

[DR07] 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*, pp. 2462–2467. 2007.

[DR08] 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.

[Kam00] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. In *Computational Logic*, *LNCS*, vol. 1861, pp. 269–284. Springer, 2000.

[Kim08] 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*, *LNCS*, vol. 5366, pp. 175–189. Springer, 2008.

[Kim09] A. Kimmig, B. Gutmann, and V. Santos Costa. Trading memory for answers: Towards tabling ProbLog. In *International Workshop on Statistical Relational Learning*. KU Leuven, Leuven, Belgium, 2009.

[Man09] 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.

[Mee09] 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*. KU LEuven, Leuven, Belgium, 2009.

[Poo00] D. Poole. Abducing through negation as failure: stable models within the independent choice logic. *J. Log. Program.*, 44(1-3):5–35, 2000.

[Rig07] F. Riguzzi. A top down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence*, *LNAI*, vol. 4733, pp. 109–120. Springer, 2007.

[Rig08] F. Riguzzi. Inference with logic programs with annotated disjunctions under the well founded semantics. In *International Conference on Logic Programming*, *LNCS*, vol. 5366, pp. 667–771. Springer, 2008.

[Sat95] T. Sato. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, pp. 715–729. 1995.

[Sha83] E. Y. Shapiro. Logic programs with uncertainties: a tool for implementing rule-based systems. In *International Joint conference on Artificial intelligence*, pp. 529–532. Morgan Kaufmann Publishers Inc., 1983.

[Swi99] T. Swift. Tabling for non-monotonic programming. *Ann. Math. Artif. Intell.*, 25(3-4):201–240, 1999.

[Tha78] A. Thayse, M. Davio, and J. P. Deschamps. Optimization of multivalued decision algorithms. In *International Symposium on Multiple-Valued Logic*, pp. 171–178. IEEE Computer Society Press, Los Alamitos, CA, USA, 1978.

[van86] M H van Emden. Quantitative deduction and its fixpoint theory. *J. Log. Program.*, 30(1):37–53, 1986.

[van91] A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[Ven04] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, *LNCS*, vol. 3131, pp. 195–209. Springer, 2004.

[Ven09] 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.