Speeding Up Inference for Probabilistic Logic Programs

Fabrizio Riguzzi

Dipartimento di Matematica e Informatica – University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy
Email: fabrizio.riguzzi@unife.it

Probabilistic Logic Programming (PLP) allows to represent domains containing many entities connected by uncertain relations and has many applications in particular in Machine Learning. PITA is a PLP algorithm for computing the probability of queries that exploits tabling, answer subsumption and Binary Decision Diagrams (BDDs). PITA does not impose any restriction on the programs. Other algorithms, such as PRISM, reduce computation time by imposing restrictions on the program, namely that subgoals are independent and that clause bodies are mutually exclusive. Another assumption that simplifies inference is that clause bodies are independent. In this paper we present the algorithms PITA(IND,IND) and PITA(OPT). PITA(IND,IND) assumes that subgoals and clause bodies are independent. PITA(OPT) instead first checks whether these assumptions hold for subprograms and subgoals: if they do, PITA(OPT) uses a simplified calculation, otherwise it resorts to BDDs. Experiments on a number of benchmark datasets show that PITA(IND,IND) is the fastest on datasets respecting the assumptions while PITA(OPT) is a good option when nothing is known about a dataset.

Keywords: Logic Programming, Probabilistic Logic Programming, Distribution Semantics, Logic Programs with Annotated Disjunctions, PRISM, ProbLog

Received ; revised ; accepted

1. INTRODUCTION

Probabilistic Logic Programming (PLP) studies how to represent uncertain information in logic programming. PLP is receiving an increasing attention due to its applications in particular in the Machine Learning field [1], where many domains present complex and uncertain relations among the entities.

The distribution semantics [2] is probably the most used in PLP. Many languages adopt this semantics, such as Probabilistic Logic Programs [3], Independent Choice Logic [4], PRISM [5], pD [6], Logic Programs with Annotated Disjunctions (LPADs) [7] and ProbLog [8]. All these languages have the same expressive power as a theory in one language can be translated into another [9, 10].

An important problem in PLP is computing the probability of queries or the problem of inference. Solving this problem in a fast way is fundamental especially for Machine Learning applications, where a large number of queries have be to answered. PRISM [5] is a system that performs inference but restricts the class of programs it can handle: subgoals in the body of clauses must be independent and bodies of clauses with the same head must be mutually exclusive. These restrictions allow PRISM to be very fast. Recently, other algorithms have been proposed that lift these restrictions, such as Ailog2 [11], ProbLog [12], cplint [13, 14], SLGAD [15, 16] and PITA [17, 18]. All these systems are able to perform inference without making any assumption on the form of the probabilistic program.

In particular, PITA associates to each subgoal an extra argument used to store a Binary Decision Diagram (BDD) that encodes the explanations for the subgoal. BDDs allow to quickly manipulate explanations and represent them so that computing the probability of the query is fast. PITA uses a Prolog library that exposes the functions of a highly efficient BDD package: the conjunction of BDDs is used for handling conjunctions of subgoals in the body while the disjunction of BDDs is used for combining explanations for the same subgoal coming from different clauses. PITA exploits tabling and answer subsumption to effectively combine answers for the same subgoal and to store them for a fast retrieval. PITA was recently shown [19] to be highly customizable for specific settings in order to increase its efficiency. When the modeling assumptions of PRISM hold (independence of subgoals and exclusiveness of clauses), PITA can be

specialized to PITA(IND,EXC) obtaining a system that is comparable or superior to PRISM in speed.

In this paper, we consider another special case that can be treated by specializing PITA, one where the bodies of clause with the same head are independent, a situation first considered in [20]. This requires a different modification of PITA, and PITA(IND,IND), the resulting system, is much faster than PITA and ProbLog on programs where these assumptions hold.

In order to generalize these results, we propose PITA(OPT), a version of PITA that performs conjunctions and disjunctions of explanations by checking whether the independence or exclusiveness assumptions hold. Then, depending on the results of these tests, PITA(OPT) uses a simplified calculation or, if no assumption holds, it falls back on using BDDs. PITA(OPT) is useful when we have no knowledge regarding the assumptions that the program satisfies, when only parts of the program satisfy the assumptions or when the program satisfies only one assumption (independence of subgoals, exclusiveness of clauses or independence of clauses).

In order to investigate the performances of PITA(IND,IND) and PITA(OPT) in comparison with the other systems, including the specialized ones, we performed a number of experiments on real and artificial datasets. The results show that PITA(IND,IND) is the fastest when the corresponding assumptions hold. When nothing is known about the program, PITA(OPT) can be considered as a valid alternative to general purpose algorithms such as PITA and ProbLog since it is faster than them when the assumptions effectively hold.

The present paper is a revised version of [21] and extends it with experiments on the Cora citation database, on biological graphs and on hidden Markov models, with the results of profiling PITA(OPT) and with a thorough discussion of the experiments.

After presenting the basic concepts of PLP, we illustrate PITA. Then we present the modeling assumptions that simplify probability computations and the system PITA(IND,IND). The description of PITA(OPT) follows, together with the experiments performed. We then conclude and present directions for future work.

2. PROBABILISTIC LOGIC PROGRAMMING

The integration of probability in logic programming has been pursued by many authors. The distribution semantics [2] is one of the most widely used semantics for PLP. In the distribution semantics, a probabilistic logic program defines a probability distribution over a set of normal logic programs (called *worlds*). The distribution is extended to a joint distribution over worlds and a ground query and the probability that the query is true is obtained from this distribution by marginalization.

A different approach is followed in Stochastic Logic Programs (SLP) [22] where the focus is on a proof-theoretic integration of probability and logic programming: each clause is assigned a label that represents the probability with which an SLD-resolution procedure chooses the clause in the case of a choice point. The distribution over clause choices defines a distribution over instantiations of the top-level goal. Thus, while the distribution semantics defines a distribution over the values true and false of a ground query, in SLP the query must have at least one variable argument and a distribution is defined over the answer substitutions for the variables.

We here concentrate on the distribution semantics for its widespread adoption. The languages based on the distribution semantics differ in the way they define the distribution over logic programs. Each language allows probabilistic choices among atoms in clauses. Here we mainly focus on LPADs for their general syntax but we will briefly discuss also ProbLog and PRISM. LPADs are sets of disjunctive clauses in which each atom in the head is annotated with a probability.

Formally a *Logic Program with Annotated Disjunctions* [7] consists of a finite set of annotated disjunctive clauses. An annotated disjunctive clause C_i is of the form

$$h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} \leftarrow b_{i1}, \ldots, b_{im_i}.$$

In such a clause the semicolon stands for disjunction, $h_{i1}, \ldots h_{in_i}$ are logical atoms and $b_{i1}, \ldots, b_{im_i}$ are logical literals, $\Pi_{i1}, \ldots, \Pi_{in_i}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. Note that, if $n_i = 1$ and $\Pi_{i1} = 1$, the clause corresponds to a non-disjunctive clause. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, 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 $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T, i.e., the results of replacing variables with constants in T.

Example 1. The following LPAD T encodes a very simple model of the development of an epidemic or a pandemic:

$C_1 = epidemic : 0.6; pandemic : 0.3 \leftarrow flu(X), cold.$
$C_2 = cold : 0.7.$
$C_3 = flu(david).$
$C_4 = flu(robert).$

The program models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic or a pandemic arises. We are uncertain about whether the climate is cold but we know for sure that David and Robert have the flu.

We now present the distribution semantics for the case in which the program does not contain function symbols so that its Herbrand base is finite[1].

[1]However, the distribution semantics for programs with function symbols has been defined as well [2, 11, 18].

An *atomic choice* is a selection of the k-th atom for a grounding $C_i\theta_j$ of a probabilistic clause C_i and is represented by the triple (C_i, θ_j, k), where θ_j is a substitution (a set of couples $Var/constant$) and $k \in \{1, \ldots, n_i\}$. An atomic choice represents an equation of the form $X_{ij} = k$ where X_{ij} is a random variable associated to $C_i\theta_j$. A set of atomic choices κ is *consistent* if $(C_i, \theta_j, k) \in \kappa, (C_i, \theta_j, m) \in \kappa$ implies $k = m$, i.e., only one head is selected for a ground clause.

A *composite choice* κ is a consistent set of atomic choices. The probability of a composite choice κ is $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$. A *selection* σ is a total composite choice (one atomic choice for every grounding of each probabilistic clause). Let us call S_T the set of all selections. A selection σ identifies a logic program w_σ called a *world*. The probability of w_σ is $P(w_\sigma) = P(\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$. Since the program does not contain function symbols, the set of worlds is finite $W_T = \{w_1, \ldots, w_m\}$ and $P(w)$ is a distribution over worlds: $\sum_{w\in W_T} P(w) = 1$.

We can define the conditional probability of a query Q given a world w as: $P(Q|w) = 1$ if Q is true in w and 0 otherwise. The probability of the query can then be obtained by marginalizing over the query

$$P(Q) = \sum_w P(Q, w) = \sum_w P(Q|w)P(w) = \sum_{w\models Q} P(w) \tag{1}$$

EXAMPLE 2. For the LPAD T of Example 1, clause C_1 has two groundings, $C_1\theta_1$ with $\theta_1 = \{X/david\}$ and $C_1\theta_2$ with $\theta_2 = \{X/robert\}$, while clause C_2 has a single grounding $C_2\emptyset$. Since C_1 has three head atoms and C_2 two, T has $3 \times 3 \times 2$ worlds. The query *epidemic* is true in 5 of them and its probability is $P(epidemic) = 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = 0.588$

Inference in probabilistic logic programming is performed by finding a covering set of explanations for queries.

A composite choice κ identifies a set ω_κ that contains all the worlds associated to a selection that is a superset of κ: i.e., $\omega_\kappa = \{w_\sigma | \sigma \in S_T, \sigma \supseteq \kappa\}$. 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 Q, a composite choice κ is an *explanation* for Q if Q is true in every world of ω_κ. In Example 1, the composite choice $\kappa_1 = \{(C_2, \emptyset, 1), (C_1, \{X/david\}, 1)\}$ is an explanation for *epidemic*. A set of composite choices K is *covering* with respect to Q if every world w_σ in which Q is true is such that $w_\sigma \in \omega_K$. In Example 1, a covering set of explanations for *epidemic* is $K = \{\kappa_1, \kappa_2\}$ where $\kappa_1 = \{(C_2, \emptyset, 1), (C_1, \{X/david\}, 1)\}$ and $\kappa_2 = \{(C_2, \emptyset, 1), (C_1, \{X/robert\}, 1)\}$.

Two composite choices κ_1 and κ_2 are *exclusive* if their union is inconsistent, i.e., if there exists a clause C_i and a substitution θ_j grounding C_i such that $(C_i, \theta_j, k) \in \kappa_1, (C_i, \theta_j, m) \in \kappa_2$ and $k \neq m$. A set K of composite choices is *mutually exclusive* if for all $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2$ implies that κ_1 and κ_2 are exclusive. As an illustration, the set of composite choices

$$\begin{aligned} K_2 = \{&\{(C_2, \emptyset, 1), (C_1, \{X/david\}, 1)\}, \\ &\{(C_2, \emptyset, 1), (C_1, \{X/david\}, 0), \\ &(C_1, \{X/robert\}, 1)\}\} \end{aligned}$$

is mutually exclusive for the theory of Example 1.

Given a covering set of explanations for a query, the query is true if the disjunction of the explanations in the covering set is true, where each explanation is interpreted as the conjunction of all its atomic choices. In this way we obtain a formula in Disjunctive Normal Form (DNF). In fact, the formula evaluates to true exactly on the worlds where the query is true [11]. In Example 1, if we associate the variables X_{11} to $C_1\{X/david\}$, X_{12} to $C_1\{X/robert\}$ and X_{21} to $C_2\emptyset$, the query is true if the following formula is true:

$$f(\mathbf{X}) = (X_{21} = 1 \wedge X_{11} = 1) \vee (X_{21} = 1 \wedge X_{12} = 1). \tag{2}$$

The covering set of explanations that is found for a query is not necessarily mutually exclusive, so the probability of the query can not be computed by a summation as in Formula (1). The explanations have first to be made mutually exclusive so that a summation can be computed. This problem, known as disjoint sum, is #P-complete [23]. One of the most effective ways to date of solving the problem makes use of Decision Diagrams.

Since the random variables that are associated to atomic choices can assume multiple values, we need to use Multivalued Decision Diagrams (MDDs) [24]. 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. Since MDDs split paths on the basis of the values of a variable, the branches are mutually exclusive so a dynamic programming algorithm [8] can be applied for computing the probability. Figure 1(a) shows the MDD corresponding to Formula (2).

Most packages for the manipulation of decision diagrams are however restricted to work on Binary Decision Diagrams (BDD), i.e., decision diagrams where all the variables are Boolean. These packages offer Boolean operators between BDDs and apply simplification rules to the results of operations in order to reduce as much as possible the size of the BDD, obtaining a reduced BDD.

A node n in a BDD has two children: the 1-child and the 0-child. When drawing BDDs, rather than using

edge labels, the 0-branch, the one going to the 0-child, is distinguished from the 1-branch by drawing it with a dashed line.

To work on MDDs with a BDD package we must represent multivalued variables by means of binary variables. The following encoding, used in [25], gives good performances. For a multi-valued variable X_{ij}, corresponding to ground clause $C_i\theta_j$, having n_i values, we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$ and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ by means of the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$. The BDD corresponding to the MDD of Figure 1(a) is shown in Figure 1(b). BDDs obtained in this way can be used as well for computing the probability of queries by associating to each Boolean variable X_{ijk} a parameter π_{ik} that represents $P(X_{ijk} = 1)$. The parameters are obtained from those of multivalued variables in this way: $\pi_{i1} = \Pi_{i1}$, $\ldots$ $\pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})}$, $\ldots$, up to $k = n_i - 1$.

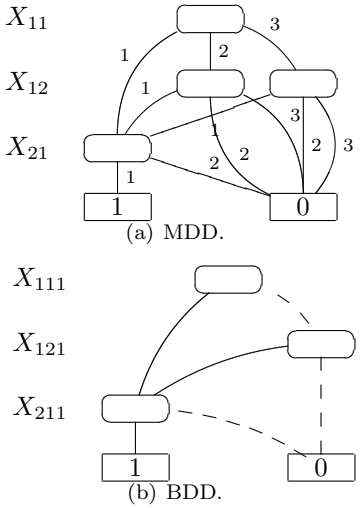


FIGURE 1. Decision diagrams for Example 1.

ProbLog [8] programs differs from LPADs as they allow probabilistic clauses only in the form of facts with two alternatives, one of which is implicit, so the program of Example 1 becomes

$epidemic \leftarrow flu(X), cold, epid(X).$
$pandemic \leftarrow flu(X), cold, not(epid(X)), pand(X).$
$0.6 :: epid(X).$
$0.75 :: pand(X).$
$0.7 :: cold.$
$flu(david).$
$flu(robert).$

where $0.6 :: epid(X).$ means that each grounding of $epid(X)$ is true with probability 0.6. In order to encode the three alternatives of the first clause of Example 1, two probabilistic atoms are used, $epid(X)$ and $pand(X)$, with the same encoding as the one adopted for representing MDDs with BDDs.

PRISM [5], similarly to ProbLog, allows probabilities only on facts but the alternatives can be more than two. In particular, PRISM offers the special predicate $msw(switch, value)$ that encodes a random switch (i.e., a random variable) and that can be used in the body of clauses to check that the random switch named *switch* takes the value named *value*. The possible value of each switch are defined by facts for the $values/2$ predicate, while the probability of each value is set by calling the predicate $set\_sw/2$. The program of Example 1 in PRISM is

$epidemic \leftarrow flu(X), msw(cold, 1), msw(event(X), epid).$
$pandemic \leftarrow flu(X), msw(cold, 1), msw(event(X), pand).$
$flu(david).$
$flu(robert).$
$values(event(X), [epid, pand, none]).$
$values(cold, [1, 0]).$
$\leftarrow set\_sw(event(X), [0.6, 0.3, 0.1]).$
$\leftarrow set\_sw(cold, [0.7, 0.3]).$

3. THE PITA SYSTEM

PITA computes the probability of a query from a probabilistic program in the form of an LPAD by first transforming the LPAD into a normal program containing calls for manipulating BDDs. The idea is to add an extra argument to each subgoal to store a BDD encoding the explanations for the answers of the subgoal. The values of the extra argument of subgoals are combined using a set of general library functions:

- *init, end*: initialize and terminate the data structures for manipulating BDDs;
- *bdd_zero(-D), bdd_one(-D)*: return BDD representing the Boolean constant 0 and 1;
- *bdd_and(+D1,+D2,-DO), bdd_or(+D1,+D2,-DO), bdd_not(+D1,-DO)*: Boolean operations between BDDs;
- *get_var_n(+R,+S,+Probs,-Var)*: returns the multivalued random variable associated to rule R with grounding substitution S and list of probabilities *Probs*;
- *bdd_equality(+Var,+Value,-D)*: D is the BDD representing *Var=Value*, i.e. that the multivalued random variable *Var* is assigned *Value*;
- *ret_prob(+D,-P)*: returns the probability of the BDD *D*.

These functions are implemented in C as an interface to the CUDD[2] library for manipulating BDDs. A BDD is represented in Prolog as an integer that is a pointer in memory to the root node of the BDD.

The PITA transformation applies to atoms, literals and clauses. The transformation for an atom h and a variable D, $PITA(h, D)$, is h with the variable D added as the last argument. The transformation for a negative literal $b_j = not(a_j)$ and a variable D_j, $PITA(b_j, D_j)$, is the Prolog conditional $(PITA(a_j, DN_j) \rightarrow not(DN_j, D_j); one(D_j))$. In other words, the data structure DN_j is negated if a_j has some explanations; otherwise the data structure for the constant function 1 is returned.

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

The disjunctive clause $C_r = h_1 : \Pi_1 \vee \ldots \vee h_n : \Pi_n \leftarrow b_1, \ldots, b_m.$ where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$:

$PITA(C_r, i) = PITA(h_i, D) \leftarrow one(DD_0),$
$PITA(b_1, D_1), and(DD_0, D_1, DD_1), \ldots,$
$PITA(b_m, D_m), and(DD_{m-1}, D_m, DD_m),$
$get\_var\_n(r, V, [\Pi_1, \ldots, \Pi_n], Var),$
$equality(Var, i, \Pi_i, DD), and(DD_m, DD, D).$

for $i = 1, \ldots, n$, where V is a list containing all the variables appearing in C_r. If the parameters do not sum up to 1, then $n-1$ rules are generated as the last head atom, *null*, does not influence the query since it does not appear in any body. In the case of empty bodies or non-disjunctive clauses (a single head with probability 1), the transformation can be optimized.

The PITA transformation applied to Example 1 yields

$PITA(C_1, 1) = epidemic(D) \leftarrow$
$one(DD_0), flu(X, D_1), and(DD_0, D_1, DD_1),$
$cold(D_2), and(DD_1, D_2, DD_2),$
$get\_var\_n(1, [X], [0.6, 0.3, 0.1], Var),$
$equality(Var, 1, 0.6, DD), and(DD_2, DD, D).$
$PITA(C_1, 2) = pandemic(D) \leftarrow$
$one(DD_0), flu(X, D_1), and(DD_0, D_1, DD_1),$
$cold(D_2), and(DD_1, D_2, DD_2),$
$get\_var\_n(1, [X], [0.6, 0.3, 0.1], Var),$
$equality(Var, 2, 0.3, DD), and(DD_2, DD, D).$
$PITA(C_2, 1) = cold(D) \leftarrow one(DD_0),$
$get\_var\_n(2, [], [0.7, 0.3], Var),$
$equality(Var, 1, 0.7, DD), and(DD_0, DD, D).$
$PITA(C_3, 1) = flu(david, D) \leftarrow one(D).$
$PITA(C_4, 1) = flu(robert, D) \leftarrow one(D).$

Clause C_1 has three alternatives in the head but the last one is the *null* atom so only two clauses are generated. Clauses C_3 and C_4 are definite facts so their transformation is optimized as shown above.

The predicates $one/1$, $not/2$, $and/3$ and $equality/4$ are defined by

$one(D) \leftarrow bdd\_one(D).$
$not(A, B) \leftarrow bdd\_not(A, B).$
$and(A, B, C) \leftarrow bdd\_and(A, B, C).$
$equality(V, I, \_P, D) \leftarrow bdd\_equality(V, I, D).$

PITA uses tabling, a logic programming technique that reduces computation time and ensures termination for a large class of programs [26]. The idea of tabling is simple: keep a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, the answers are retrieved from the store rather than recomputing them. Besides saving time, tabling ensures termination for programs without function symbols under the Well-Founded Semantics (WFS) [27].

PITA also uses a feature of XSB Prolog called *answer subsumption* [26] that, when a new answer for a tabled subgoal is found, combines old answers with the new one according to a partial order or lattice. For example, if the lattice is on the second argument of a binary predicate p, answer subsumption may be specified by means of the declaration *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. Thus if a table has an answer $p(a, d_1)$ and a new answer $p(a, d_2)$ is derived, the answer $p(a, d_1)$ is replaced by $p(a, d_3)$, where d_3 is obtained by calling $or(d_1, d_2, d_3)$.

In PITA, various predicates 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 disjunction of BDDs, with:

$zero(D) \leftarrow bdd\_zero(D).$
$or(A, B, C) \leftarrow bdd\_or(A, B, C).$

At a minimum, the predicate of the goal and all the predicates appearing in negative literals should be tabled with answer subsumption. If these predicates are not tabled with answer subsumption, PITA is not correct as it does not collect all answers for the subgoals of these predicates. It is usually useful to table every predicate whose answers have multiple explanations and are going to be reused often since in this way repeated computations are avoided.

4. MODELING ASSUMPTIONS

PRISM makes the following modeling assumptions [28]:

1. the probability of a conjunction (A, B) is computed as the product of the probabilities of A and B (*independent-and assumption*),
2. the probability of a disjunction $(A; B)$ is computed as the sum of the probabilities of A and B (*exclusive-or assumption*).

These assumptions can be stated more formally by referring to explanations. Given an explanation κ, let $RV(\kappa) = \{C_i\theta_j | (C_i, \theta_j, k) \in \kappa\}$. Given a set of explanations K, let $RV(K) = \bigcup_{\kappa \in K} RV(\kappa)$. Two sets of explanations, K_1 and K_2, are *independent* if $RV(K_1) \cap RV(K_2) = \emptyset$ and *exclusive* if, $\forall \kappa_1 \in K_1, \kappa_2 \in K_2$, κ_1 and κ_2 are exclusive.

The independent-and assumption means that, when deriving a covering set of explanations for a goal, the covering sets of explanations K_i and K_j for two ground subgoals in the body of a clause are independent.

The exclusive-or assumption means that, when deriving a covering set of explanations for a goal, two sets of explanations K_i and K_j obtained for a ground subgoal h from two different ground clauses are exclusive. This implies that the atom h is derived using clauses that have mutually exclusive bodies, i.e., that their bodies are not both true in any world.

PRISM [5] and PITA(IND,EXC) [19] exploit these assumptions to speed up the computation. In fact, these assumptions make the computation of probabilities "truth-functional" [29] (the probability of conjunction/disjunction of two propositions depends only on the probabilities of those propositions), while in the general case this is false. PITA(IND,EXC) differs from PITA in the definition of the $one/1$, $zero/1$, $not/2$, $and/3$, $or/3$ and $equality/4$ predicates that now work on probabilities P rather than on BDDs. Their definitions are

$zero(0)$.
$one(1)$.
$not(A, B) \leftarrow B \ is \ 1 - A$.
$and(A, B, C) \leftarrow C \ is \ A * B$.
$or(A, B, C) \leftarrow C \ is \ A + B$.
$equality(V, \_N, P, P)$.

Instead of the *exclusive-or assumption*, a program may satisfy the following assumption:

3. the probability of a disjunction $(A; B)$ is computed as if A and B were independent (*independent-or assumption*).

This means that, when deriving a covering set of explanations for a goal, two sets of explanations K_i and K_j obtained for a ground subgoal h from two different ground clauses are independent. If A and B are independent, the probability of their disjunction is

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B) = P(A) + P(B) - P(A)P(B)$$

by the laws of probability theory. PITA(IND,EXC) can be used for programs respecting this assumption by changing the $or/3$ predicate in this way

$or(A, B, P) \leftarrow P \ is \ A + B - A * B$.

We call PITA(IND,IND) the resulting system.

The exclusiveness assumption for conjunctions of literals means that the conjunction is true in 0 worlds and thus has probability 0 so it does not make sense to consider a PITA(EXC,_) system.

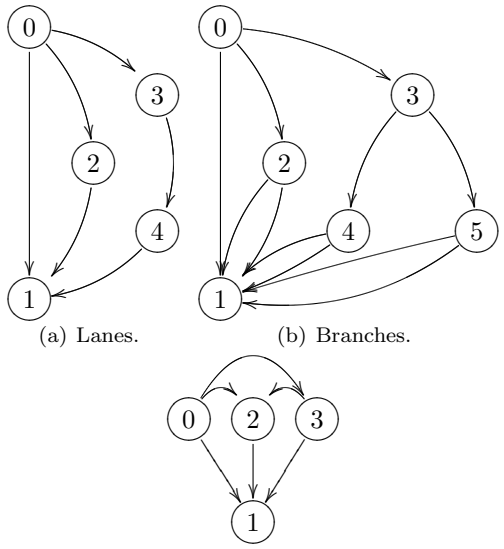
The following program

$path(Node, Node)$.
$path(Source, Target) : 0.3 \leftarrow edge(Source, Node), path(Node, Target)$.
$edge(0, 1) : 0.3$.
...

satisfies the independent-and and independent-or assumptions depending on the structure of the graph. For example, the graphs in Figures 2(a) and 2(b) respect these assumptions for the query $path(0, 1)$. Similar graphs of increasing sizes can be obtained with the procedures presented in [30]. We call the first graph type a "lanes" graph and the second a "branches" graph. The graphs of the type of Figure 2(c), called "parachutes" graphs, instead, satisfy only the independent-and assumption for the query $path(0, 1)$.

All three types of graphs respect the independent-and assumption because, when deriving the goal $path(0, 1)$, paths are built incrementally starting from node 0 and adding one edge at a time with the second clause of the definition of $path/2$. Since the edge that is added does not appear in the rest of the path, the assumption is respected.

Lanes and branches graphs respect the independent-or assumption because, when deriving the goal $path(0, 1)$, ground instantiations of the second path clause have $path(i, 1)$ in the head and originate atomic choices of the form $(C_2, \{Source/i, Target/1, Node/j\}, 1)$. Explanations for $path(i, 1)$ contain also atomic choices $(E_{i,j}, \emptyset, 1)$ for



(a) Lanes. (b) Branches.

(c) Parachutes.

FIGURE 2. Examples of graphs satisfying some of the assumptions.

every fact $edge(i, j) : 0.3$. in the path. Each explanation corresponds to a path. In the lanes graph each node except 0 and 1 lies on a single path, so the explanations for $path(i, 1)$ do not share random variables. In the branches graphs, each explanation for $path(i, 1)$ depends on a disjoint set of edges. In the parachutes graph instead this is not true: for example, the path from 2 to 1 shares the edge from 2 to 1 with the path 3, 2, 1.

Another program satisfying the independent-and and independent-or assumptions is the following

$sametitle(A, B) : 0.3 \leftarrow haswordtitle(A, word\_10), haswordtitle(B, word\_10)$.
$sametitle(A, B) : 0.3 \leftarrow haswordtitle(A, word\_1321), haswordtitle(B, word\_1321)$.
...

which computes the probability that the titles of two different citations are the same on the basis of the words that are present in the titles. The dots stand for clauses differing from the ones above only for the words considered. The $haswordtitle/2$ predicate is defined by a set of certain facts. This is part of the program that is used in Section 6 to test PITA on the Cora citation database [31]. The program satisfies the independent-and assumption for the query $sametitle(tit1, tit2)$ because $haswordtitle/2$ has no uncertainty. It satisfies the independent-or assumption because each clause for $sametitle/2$ defines a different random variable.

5. PITA(OPT)

PITA(OPT) differs from PITA because, before applying BDD logical operations between sets of explanations, it checks for the truth of the assumptions. If they hold, then the simplified probability computations are used.

The data structures used to store probabilistic information in PITA(OPT) are couples (P, T) where P is a real number representing a probability and T is a term formed with the functors $zero/0, one/0, c/2, or/2, and/2, not/1$ and the integers. If T is an integer, it represents a pointer to the root node of a BDD. If T is not an integer, it represents a Boolean expression of which the terms of the form $zero$, one, $c(var, val)$ and the integers represent the base case: $c(var, val)$ indicates the equation $var = val$ while an integer indicates a BDD. In this way, we are able to represent Boolean formulas by means of a BDD, by means of a Prolog term or a combination thereof.

For example, *or(0x94ba008, and(c(1,1),not(c(2,3)))* represents the expression: $B \vee (X_1 = 1 \wedge \neg(X_2 = 3))$ where B is the Boolean function represented by the BDD whose root node address in memory is the integer *0x94ba008* in Prolog hexadecimal notation.

PITA(OPT) differs from PITA also in the definition of $zero/1$, $one/1$, $not/2$, $and/3$, $or/3$ and $equality/4$ that now work on couples (P, T) rather than on BDDs. $equality/4$ is defined as

```
equality(V,N,P,(P,c(V,N))).
```

The $one/1$ and $zero/1$ predicates are defined as

```
zero((0,zero)).
one((1,one)).
```

The $or/3$ and $and/3$ predicates first check whether one of their input argument is (an integer pointing to) a BDD. If so, they convert also the other input argument to a BDD and test for independence using the library function $bdd\_ind(B1, B2, I)$. Such a function is implemented in C and uses the CUDD function `Cudd_SupportIndex` that returns an array indicating which variables appear in a BDD (the support variables). $bdd\_ind(B1, B2, I)$ checks whether there is an intersection between the set of support variables of $B1$ and $B2$ and returns $I = 1$ if the intersection is empty. If the two BDDs are independent, then the value of the resulting probability is computed using a formula and a compound term is returned.

If none of the input arguments of $or/3$ and $and/3$ are BDDs, then these predicates test if the independence or the exclusiveness assumption holds. If so, they update the value of the probability using a formula and return a compound term. If not, they convert the terms to BDDs, apply the corresponding operation and return the resulting BDD together with the probability it represents:

```
or((PA,TA),(PB,TB),(PC,TC)) ←
  ((integer(TA);integer(TB)) →
    ev(TA,BA),ev(TB,BB),
    bdd_ind(BA,BB,I),
    (I = 1 →
      PC is PA + PB - PA * PB,
      TC = or(BA,BB)
    ;
      bdd_or(BA,BB,TC),ret_prob(TC,PC)
    )
  ;
    (ind(TA,TB) →
      PC is PA + PB - PA * PB,
      TC = or(BA,BB)
    ;
      (exc(TA,TB) →
        PC is PA + PB,
        TC = or(BA,BB)
      ;
        ev(TA,BA),ev(TB,BB),
        bdd_or(BA,BB,TC),ret_prob(TC,PC)
      )
    )
  ).
and((PA,TA),(PB,TB),(PC,TC)) ←
  ((integer(TA);integer(TB)) →
    ev(TA,BA),ev(TB,BB),
    bdd_ind(A,BB,I),
    (I = 1 →
      PC is PA * PB,
      TC = and(BA,BB)
    ;
      bdd_and(BA,BB,TC),ret_prob(TC,PC)
      (bdd_zero(TC) →
        fail
      ;
        true
      )
    )
  ;
    (ind(TA,TB) →
      PC is PA * PB,
      TC = and(BA,BB)
    ;
      (exc(TA,TB) →
        fail
      ;
        ev(TA,BA),ev(TB,BB),
        bdd_and(BA,BB,TC),ret_prob(TC,PC)
      )
    )
  ).
```

In these predicate definitions, $ev/2$ evaluates a term returning a BDD. In $and/3$, after the first $bdd\_and/3$ operation, a test is made to check whether the resulting BDD represent the 0 constant. If so, the derivation fails as this branch contributes with a 0 probability. These predicates make sure that, once a BDD has been built, it is used in the following operations, avoiding the manipulation of terms and exploiting the work performed as much as possible.

The $not/2$ predicate is very simple: it complements the probability and returns a new term:

```
not((P,B),(P1,not(B))) ← P1 is 1 - P.
```

The predicate $exc/2$ checks for the exclusiveness between two terms with a recursion through the structure of the terms. The following code defines $exc/2$:

$exc(zero, \_) \leftarrow !.$
$exc(\_, zero) \leftarrow !.$
$exc(c(V, N), c(V, N1)) \leftarrow !, N \backslash = N1.$
$exc(c(V, N), or(X, Y)) \leftarrow !, exc(c(V, N), X),$
$exc(c(V, N), Y).$
$exc(c(V, N), and(X, Y)) \leftarrow !, (exc(c(V, N), X);$
$exc(c(V, N), Y)).$
$exc(or(A, B), or(X, Y)) \leftarrow !, exc(A, X), exc(A, Y),$
$exc(B, X), exc(B, Y).$
$exc(or(A, B), and(X, Y)) \leftarrow !, (exc(A, X); exc(A, Y)),$
$(exc(B, X); exc(B, Y)).$
$exc(and(A, B), and(X, Y)) \leftarrow !, exc(A, X); exc(A, Y);$
$exc(B, X); exc(B, Y).$
$exc(and(A, B), or(X, Y)) \leftarrow !, (exc(A, X); exc(B, X)),$
$(exc(A, Y); exc(B, Y)).$
$exc(not(A), A) \leftarrow !.$
$exc(not(A), and(X, Y)) \leftarrow !, exc(not(A), X);$
$exc(not(A), Y).$
$exc(not(A), or(X, Y)) \leftarrow !, exc(not(A), X),$
$exc(not(A), Y).$
$exc(A, or(X, Y)) \leftarrow !, exc(A, X), exc(A, Y).$
$exc(A, and(X, Y)) \leftarrow exc(A, X); exc(A, Y).$

For example, the goal $exc(or(c(1,1), c(2,1)), and(c(1,2), c(2,2)))$ matches with the 7th clause and calls $exc(c(1,1), c(1,2))$, $exc(c(1,1), c(2,2))$, $exc(c(2,1), c(1,2))$ and $exc(c(2,1), c(2,2))$. Of the first two calls, $exc(c(1,1), c(1,2))$ succeeds, thus satisfying the first conjunct in the body. Of the latter two calls, $exc(c(2,1), c(2,2))$ succeeds, thus satisfying the second conjunct in the body and proving the goal.

The $ind/2$ predicate checks for independence between two terms. It visits the structure of the first term until it reaches an atomic choice. Then it checks for the absence of the variable in the second term with the predicate $absent/2$. $ind/2$ and $absent/2$ are defined as:

$ind(one, \_) \leftarrow !.$
$ind(zero, \_) \leftarrow !.$
$ind(\_, one) \leftarrow !.$
$ind(\_, zero) \leftarrow !.$
$ind(c(V, \_N), B) \leftarrow !, absent(V, B).$
$ind(or(X, Y), B) \leftarrow !, ind(X, B), ind(Y, B).$
$ind(and(X, Y), B) \leftarrow !, ind(X, B), ind(Y, B).$
$ind(not(A), B) \leftarrow ind(A, B).$
$absent(V, c(V1, \_N1)) \leftarrow !, V \backslash = V1.$
$absent(V, or(X, Y)) \leftarrow !, absent(V, X), absent(V, Y).$
$absent(V, and(X, Y)) \leftarrow !, absent(V, X), absent(V, Y).$
$absent(V, not(A)) \leftarrow absent(V, A).$

For example, the goal $ind(or(c(1,1), c(2,1)), and(c(3,2), c(4,2)))$ matches with the 6th clause and calls $ind(c(1,1), and(c(3,2), c(4,2)))$ and $ind(c(2,1), and(c(3,2), c(4,2)))$. The first call matches with the 5th clause and calls $absent(1, and(c(3,2), c(4,2)))$ which, in turn, calls $absent(1, c(3,2))$ and $absent(1, c(4,2))$. Since they both succeed, $ind(c(1,1), and(c(3,2), c(4,2)))$ succeeds as well. The second call, $ind(c(2,1), and(c(3,2), c(4,2)))$, matches with the 5th clause and calls $absent(2, and(c(3,2), c(4,2)))$ which, in turn, calls $absent(2, c(3,2))$ and $absent(2, c(4,2))$. They both succeed so $ind(c(2,1), and(c(3,2), c(4,2)))$ and the original goal are proved.

The predicates $exc/2$ and $ind/2$ define sufficient conditions for exclusion and independence respectively. If the arguments of $exc/2$ and $ind/2$ do not contain integer terms representing BDDs, then the conditions are also necessary. The evaluation of a term, i.e., its transformation into a BDD, is defined as

$ev(B, B) \leftarrow integer(B), !.$
$ev(zero, B) \leftarrow !, bdd\_zero(B).$
$ev(one, B) \leftarrow !, bdd\_one(B).$
$ev(c(V, N), B) \leftarrow !, bdd\_equality(V, N, B).$
$ev(and(A, B), C) \leftarrow !, ev(A, BA), ev(B, BB),$
$bdd\_and(BA, BB, C).$
$ev(or(A, B), C) \leftarrow !, ev(A, BA), ev(B, BB),$
$bdd\_or(BA, BB, C).$
$ev(not(A), C) \leftarrow ev(A, B), bdd\_not(B, C).$

When the program satisfies the (IND,EXC) or (IND,IND) assumptions, PITA(OPT) answers the query using the probability formulas without building BDDs: terms are combined in progressively larger terms that are used to check the assumptions, while the probabilities of the combinations are computed only from the probabilities of the operands without considering their structure.

When the program does not satisfy any of the assumptions, PITA(OPT) can still be beneficial since it delays the construction of BDDs as much as possible and may lead to the construction of less intermediate BDDs than PITA. While in PITA the BDD for each intermediate subgoal must be kept in memory because it is stored in the table and has to be available for future use, in PITA(OPT) BDDs are built only when necessary, leading to a smaller memory footprint and a leaner memory management.

6. EXPERIMENTS

In this section, we compare the systems PITA, PITA(IND,EXC), PITA(IND,IND), PITA(OPT), PRISM and ProbLog on the following datasets: the graphs of Figure 2, biological networks [8], the Cora[3] [31] citation database, the four benchmarks[4] of [32] and a program modeling a hidden Markov model. We consider here version 1 of ProbLog and leave for future work the comparison with version 2 [33].

We first consider the graphs of Figure 2 and the path program shown in Section 4. The number of edges of the graphs for size N is $N(N+1)/2$ for lanes and parachutes and $2^{N+1} - N - 2$ for branches. The execution times of PITA(OPT), PITA(IND,IND), PITA and ProbLog for graphs of increasing sizes are shown in Figure 3 for lanes, Figure 4 for branches and Figure 5 for parachutes. The x-axis indicates N. A point missing from the graph for a system in these and in the following figures means that the system was not able to solve the corresponding problem due to a 24-hour time-out or an out-of-memory error.

Figure 3 clearly shows the advantage of the (IND,IND) modeling assumptions that allow PITA(IND,IND) to achieve high speed and scalability. The performances of PITA(OPT) are inferior but still much better than PITA and ProbLog. Figure 4 again shows the good performances of PITA(IND,IND).

[3] http://alchemy.cs.washington.edu/data/cora
[4] http://dtai.cs.kuleuven.be/cplve/ilp09/

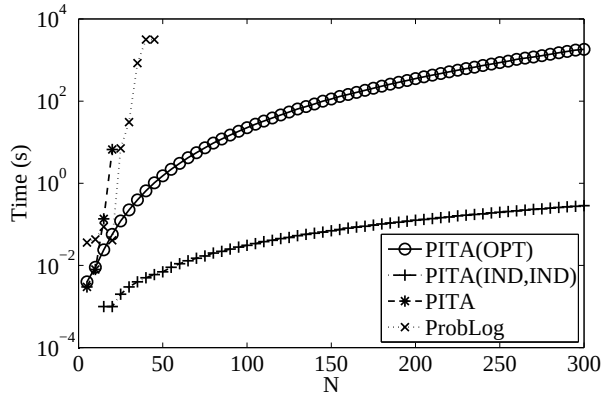


FIGURE 3. Execution times on the lanes graphs.

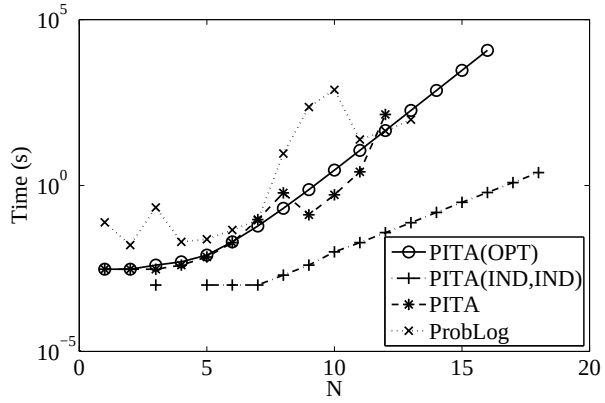


FIGURE 4. Execution times on the branches graphs.

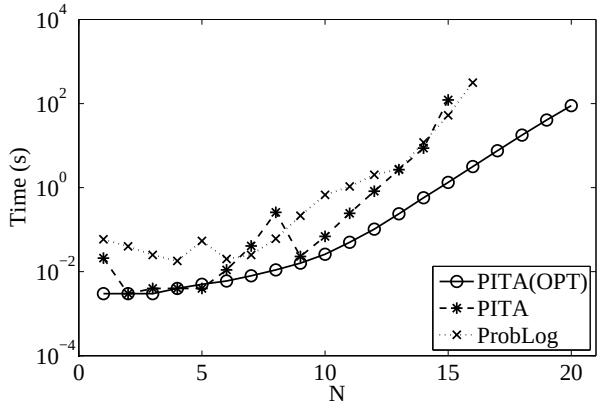


FIGURE 5. Execution times on the parachutes graphs.

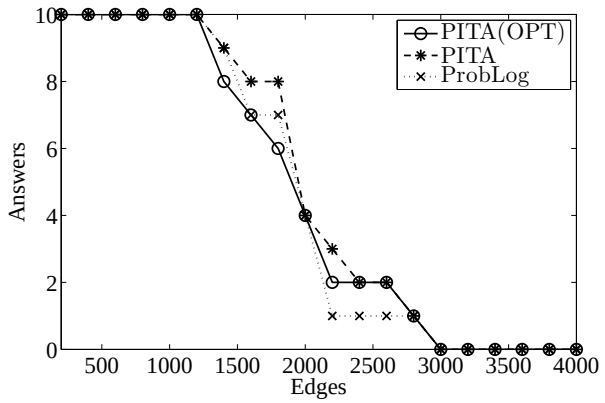


FIGURE 6. Solved biological networks.

Here PITA(OPT) is faster and more scalable than PITA and ProbLog as well.

The parachute graphs do not respect the (IND,IND) modeling assumption so PITA(IND,IND) has not been applied. Figure 5 compares PITA, PITA(OPT) and ProbLog and shows that PITA(OPT) is the fastest and most scalable.

We also consider another graph dataset containing biological networks [8] in which the nodes encode biological entities and the links represents conceptual relations among them. Each program in this dataset contains the following non-probabilistic definition of $path/2$:

$path(Node, Node).$
$path(Source, Target) \leftarrow edge(Source, Node),$
$\quad path(Node, Target).$

plus a number of probabilistic facts representing edges. The programs in the dataset 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. In these graphs, no assumption holds. For PITA, we used the definition of path from [34] that performs loop checking explicitly by keeping the list of visited nodes. For ProbLog we used the definition of path above and we used tabling for loop checking. The path predicate is tabled also in PITA. We found these to be the best performing settings for the two systems. Figure 6 shows the number of subgraphs for which each algorithm was able to answer the query (i.e., incurred in no time-out and no memory error) as a function of the size of the subgraphs, while Figure 7 shows the execution times averaged over all and only the subgraphs for which all the algorithms succeeded. Here PITA and ProbLog perform better than PITA(OPT) that however is very close to PITA.

The Cora [31] database contains citations to computer science research papers. For each citation we know the title, the authors, the venue and the words that appear in them. The task is to determine which citations are referring to the same paper, by predicting the predicate $samebib(cit1, cit2)$. We used the program:

$samebib(A, B) : 0.3 \leftarrow$
$\quad venue(A, C), venue(B, D), samevenue(C, D).$
$samebib(A, B) : 0.3 \leftarrow$
$\quad author(A, C), author(B, D), sameauthor(C, D).$
$samebib(A, B) : 0.3 \leftarrow$
$\quad title(A, C), title(B, D), sametitle(C, D).$
$samevenue(A, B) : 0.3 \leftarrow$
$\quad haswordvenue(A, word\_06),$
$\quad haswordvenue(B, word\_06).$
...

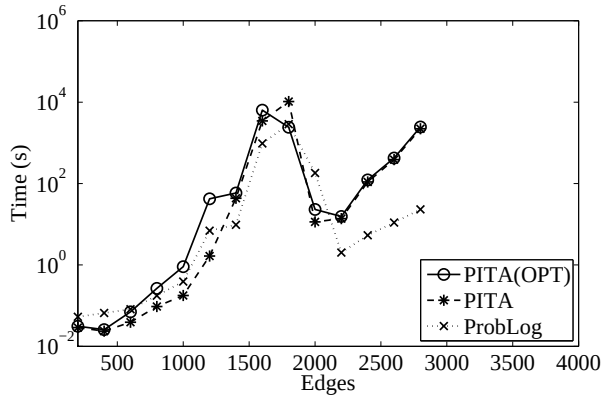


FIGURE 7. Average time on biological networks.

PITA(IND,IND)	PITA(OPT)	PITA	ProbLog
5.753	23.751	33.445	42.423

TABLE 1. Execution Times on Cora

$sameauthor(A, B) : 0.3 \leftarrow$
$haswordauthor(A, word\_a)$,
$haswordauthor(B, word\_a)$.
...
$sametitle(A, B) : 0.3 \leftarrow$
$haswordtitle(A, word\_10)$,
$haswordtitle(B, word\_10)$.
...

where there is a clause for $samevenue/2$, $sameauthor/2$ and $sametitle/2$ for each word that appear in the venue, author and title respectively. Overall there are 556 clauses for $samevenue/2$, $sameauthor/2$ and $sametitle/2$. The program also contains non-probabilistic facts for the predicates $venue(cit, ven)$, $author(cit, aut)$, $title(cit, tit)$, $haswordvenue(ven, word)$, $haswordauthor(aut, word)$ and $haswordtitle(tit, word)$. This program is a simplified version of the probabilistic logic theory used for parameter learning in [35]. In that paper, the dataset is divided into five subsets for performing cross-validation machine learning experiments. We consider the first subset that contains 5231 positive examples for $samebib/2$.

This dataset satisfies the independent-and assumption, as every body contains at most one probabilistic atom, and the independent-or assumption, as different bodies for the same ground head do not share random variables. Therefore we applied PITA(IND,IND), PITA(OPT), PITA and ProbLog and we measured the total time needed to compute the probability of each positive example. All predicates are tabled for PITA(IND,IND), PITA(OPT) and PITA, while no predicate is tabled in ProbLog as this gives better results. The results are shown in Table 1. PITA(IND,IND) is by far the quickest, followed by PITA(OPT), PITA and ProbLog.

The blood type dataset [32] encodes the genetic inheritance of blood type in families of increasing size. The blood type of a person depends on her chromosomes that in turn depend on those of her parents. The blood type is given by clauses of the form

$bloodtype(P, a) : 0.90; bloodtype(P, b) : 0.03;$
$bloodtype(P, ab) : 0.03; bloodtype(P, null) : 0.04 \leftarrow$
$pchrom(P, a), mchrom(P, a)$.

where P stands for a person, $pchrom/2$ indicates the chromosome inherited from the father and $mchrom/2$ that inherited from the mother. There is one such clause for every combination of the values $\{a, b, null\}$ for the father and mother chromosomes, so 9 clauses overall. In turn, the chromosomes of a person depend on those of her parents, with clauses of the form

$mchrom(P, a) : 0.90; mchrom(P, b) : 0.05;$
$mchrom(P, null) : 0.05 \leftarrow$
$mother(Mother, P), pchrom(Mother, a)$,
$mchrom(Mother, a)$.

There is one such clause for every combination of the values $\{a, b, null\}$ for the father and mother chromosomes of the mother and similarly for the father chromosome of a person, 18 clauses in total. In this dataset we query the blood type of a person considering families with an increasing number of components: each program adds two persons to the previous one. The chromosomes of the parent-less ancestors are given by disjunctive facts of the form

$mchrom(p, a) : 0.3; mchrom(p, b) : 0.3;$
$mchrom(p, null) : 0.4.$
$pchrom(p, a) : 0.3; pchrom(p, b) : 0.3;$
$pchrom(p, null) : 0.4.$

For a dataset containing N persons, the program contains $2(N - 2^{\lfloor \log_2 N \rfloor} + 1)$ such clauses and $N - 1$ facts for $father/2$ and $mother/2$.

In this dataset, the (IND,EXC) assumption holds so PRISM is also used, with the default values for all parameters. Figure 8 shows the execution times of the algorithms where the x-axis indicates the number of family members. As can be seen, PITA(IND,EXC) is much faster than PRISM that exploits the same assumptions and PITA is slightly faster than PITA(OPT).

The growing head dataset [32] contains propositional programs with heads of increasing size. For example, the program for size 3 is

$a0 \leftarrow a1.$
$a1 : 0.5.$
$a0 : 0.5; a1 : 0.5 \leftarrow a2.$
$a2 : 0.5.$

The program for size N contains $2(N - 1)$ clauses. The query to be answered is $a0$. This dataset respect the independent-and assumption (trivially as bodies are all at most singletons) but neither the independent-or assumption nor the exclusive-or assumption, so we compare PITA(OPT) with PITA and ProbLog. Figure 9 shows that PITA(OPT) is faster than ProbLog and than PITA for sizes larger than 12 and is able to solve 7 more programs.

The growing negated body dataset [32] contains propositional programs with bodies of increasing size. For example, the program for size 4 is

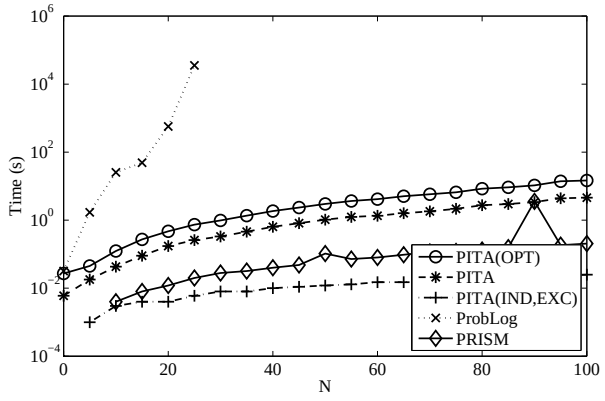


FIGURE 8. Execution times on the blood type dataset.

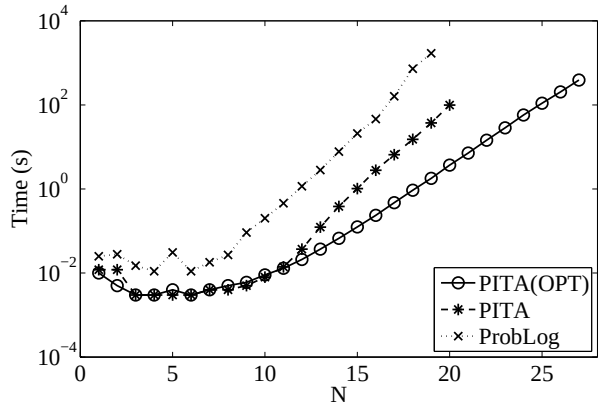


FIGURE 9. Execution times on the growing head dataset.

$a0 : 0.5 \leftarrow a1.$
$a0 : 0.5 \leftarrow not(a1), a2.$
$a0 : 0.5 \leftarrow not(a1), not(a2), a3.$
$a1 : 0.5 \leftarrow a2.$
$a1 : 0.5 \leftarrow not(a2), a3.$
$a2 : 0.5 \leftarrow a3.$
$a3 : 0.5.$

The number of clauses of the program for size N is $(N-1)N/2+1$. The query to be answered is $a0$. This dataset respects the exclusive-or assumptions but not the independent-and assumption. Therefore we compare PITA(OPT) with PITA and ProbLog. Figure 10 shows that in this case the overhead of PITA(OPT) makes it slower than PITA and ProbLog.

The UWCSE dataset [32] encodes a university domain with 16 predicates such as $taught\_by/2$, $advised\_by/2$, $course\_level/2$, $phase/2$, $position/2$, $course/1$, $professor/1$ and $student/1$. The definitions of these predicates take 36 clauses. Programs of increasing size are considered by adding facts for the $student/1$ predicate, i.e., by considering an increasing number of students. The program for one student is shown in the Appendix. The runtime of the query $taught\_by(c1, p1)$ as a function of the number of students is shown in Figure 11. As can be seen from the Appendix, each clause for the same predicate

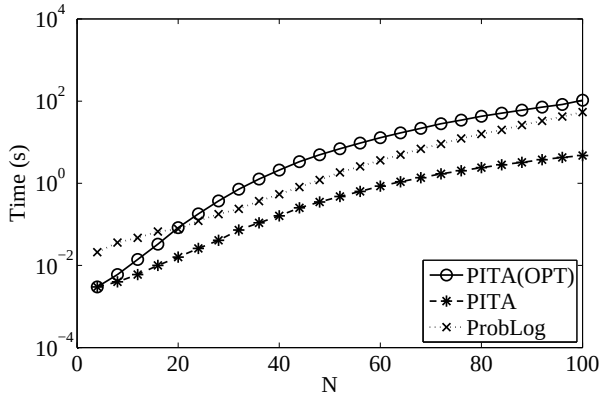


FIGURE 10. Execution times on the growing negated body dataset.

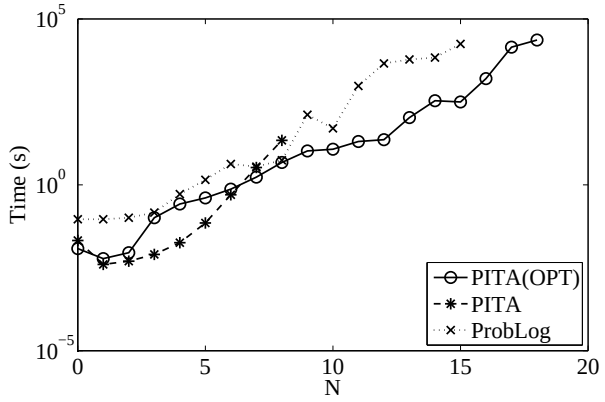


FIGURE 11. Execution times on the UWCSE dataset.

differs from the others for the sign of at least one literal, thus the program satisfies the exclusive-or assumption. However, it does not satisfy the independent-and assumption, so we compare PITA(OPT) with PITA and ProbLog. PITA(OPT) is able to solve more problems than PITA and ProbLog and is faster than ProbLog.

The last experiment involves the Hidden Markov model for DNA sequences from [36]: biochemical bases are the output symbols and three states are assumed, of which one is the end state. The following program generates base sequences:

$hmm(O) \leftarrow hmm1(\_, O).$
$hmm1(S, O) \leftarrow hmm(q1, [], S, O).$
$hmm(end, S, S, []).$
$hmm(Q, S0, S, [L|O]) \leftarrow Q \backslash = end,$
$next\_state(Q, Q1, S0),$
$letter(Q, L, S0), hmm(Q1, [Q|S0], S, O).$
$next\_state(q1, q1, \_S) : 1/3; next\_state(q1, q2, \_S) : 1/3;$
$next\_state(q1, end, \_S) : 1/3.$
$next\_state(q2, q1, \_S) : 1/3; next\_state(q2, q2, \_S) : 1/3;$
$next\_state(q2, end, \_S) : 1/3.$
$letter(q1, a, \_S) : 0.25; letter(q1, c, \_S) : 0.25;$
$letter(q1, g, \_S) : 0.25; letter(q1, t, \_S) : 0.25.$
$letter(q2, a, \_S) : 0.25; letter(q2, c, \_S) : 0.25;$
$letter(q2, g, \_S) : 0.25; letter(q2, t, \_S) : 0.25.$

The algorithms are used to compute the probability of *hmm(O)* for sequences O of increasing length. We consider two types of sequences: "repeating" ones, in

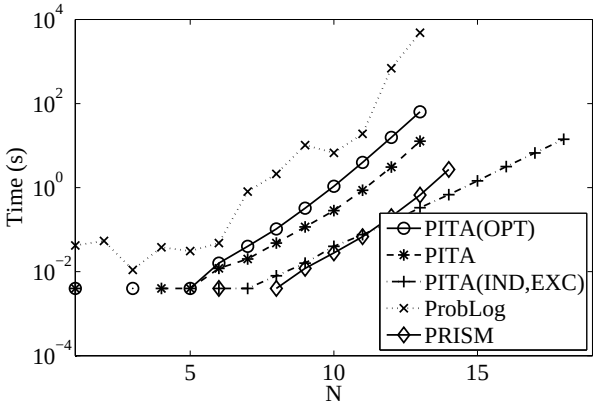


FIGURE 12. Execution times on the hidden Markov model, repeating sequences.

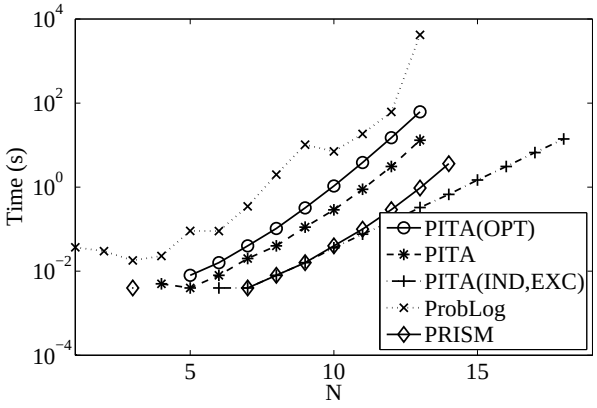


FIGURE 13. Execution times on the hidden Markov model, random sequences.

which the pattern *a,c,g,t* is repeated, and random ones.

In this dataset, the (IND,EXC) assumption holds and Figures 12 and 13 show the times taken by PITA, PITA(IND,EXC), PITA(OPT), ProbLog and PRISM as a function of the sequence length for repeating and random sequences respectively. PRISM is used with the default values for all parameters.

PITA(IND,EXC) is clearly the best performing one, achieving lower times and higher scalability. PITA(OPT) here is slightly slower than PITA.

In [36] the authors proposed a technique for speeding up query answering by removing *non-discriminating arguments*. These are arguments that play no role in determining the control flow of a logic program with respect to goals satisfying given mode and sharing restrictions. The authors show that the removal of non-discriminating arguments is very useful with tabling because the calls to a tabled predicate differing only in the non-discriminating arguments will merge into a single table that is much smaller and has a higher chance of reuse. Since non-discriminating arguments do not influence the control flow of a logic program, the probability of the query is the same as that computed

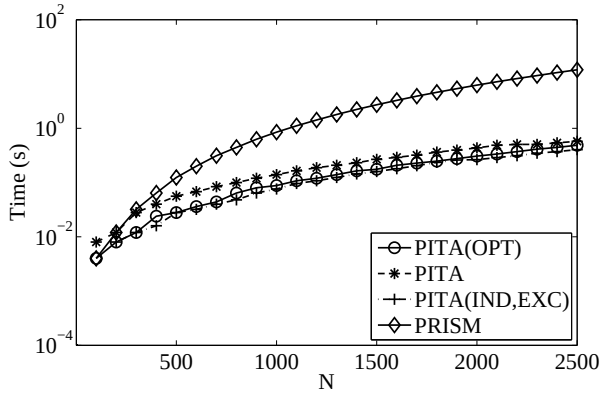


FIGURE 14. Execution Times on the optimized hidden Markov model, repeating sequences.

from the original program. After removing non-discriminating arguments, the HMM program above becomes

$hmm(O) \leftarrow hmm(q1, O).$
$hmm(end, []).$
$hmm(Q, [L|O]) \leftarrow Q \backslash = end, next\_state(Q, Q1, S0),$
$letter(Q, L, S0), hmm(Q1, O).$

plus the clauses from the previous program defining $next\_state/3$ and $letter/3$. In this dataset the (IND,EXC) assumptions hold as well. Figures 14 and 15 show the times taken by PITA, PITA(IND,EXC), PITA(OPT) and PRISM as a function of the sequence length for repeating and random sequences respectively. Results for ProbLog are not shown because this encoding of an HMM requires deriving non-ground probabilistic atoms which is forbidden by ProbLog[5].

PITA(IND,EXC) is the best performing algorithm for repeating sequences, while it is very close to PRISM for random sequences. This is due to the fact that XSB uses tries to store answers in the tables, while PRISM uses hashing: tries very compactly represent a term with repetitions, while hashing is not able to exploit this property. PITA(OPT) is very close to PITA(IND,EXC) on repeating sequences and close on random sequences. If compared to PITA, PITA(OPT) has better performances in both cases.

These experiments show that, if we know that the program respects either the (IND,EXC) or the (IND,IND) assumptions, using the corresponding algorithm gives the best results. If nothing is known about the program, PITA(OPT) is a good option since it gives very good results in datasets where these assumptions hold. Moreover, PITA(OPT) has good performances also when the assumptions hold individually: parachutes respect only the independent-or assumption, growing head the independent-and

[5]When $next\_state(Q, Q1, S0)$ and $letter(Q, L, S0)$ are called, $S0$ is not instantiated and is non-ground in the probabilistic facts defining these predicates. This is forbidden by ProbLog, while PITA and PRISM allow it and consider two atoms with different variables, e.g. $next\_state(q1, q1, \_1)$ and $next\_state(q1, q1, \_2)$, as different for the purpose of assigning random variables.

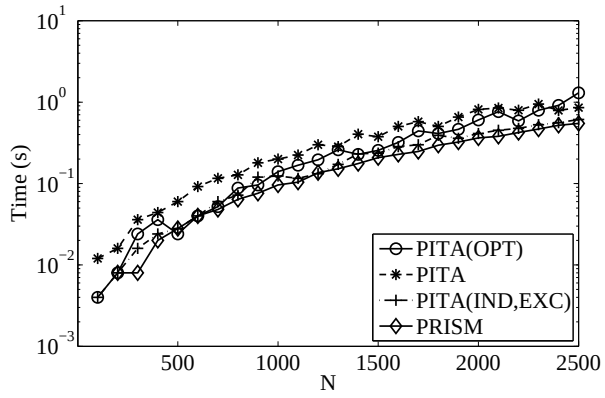


FIGURE 15. Execution times on the optimized hidden Markov model, random sequences.

assumption and UWCSE the exclusive-or assumption.

On these datasets PITA(OPT) has to resort to BDDs at last, since the independent-and assumption does not hold, but can delay the construction of BDDs thus avoiding to keep them in memory for some subgoals.

The overhead of PITA(OPT) with respect to PITA is limited on all datasets. This is confirmed by profiling PITA(OPT) on the biological network with 1200 edges from the first set of network samples: PITA(OPT) spends 59% of the time in the BDD functions, while the time spent in the $ind/2$ and $exc/2$ functions is negligible with respect to the total time (2% on $absent/2$ and less than 1% on $ind/2$ and $exc/2$).

When comparing PITA(OPT) to ProbLog, we can see that the advantages of PITA highlighted in [18] are still exploited by PITA(OPT): the use of tabling and answer subsumption for computing and storing the explanations of the individual subgoals allows a larger amount of explanation (and BDD) reuse during the computation, leading to lower execution times.

7. CONCLUSIONS

We have discussed how assumptions on the program can much simplify the computation of the probability of queries. When subgoals in the body of clauses and the bodies of clauses for the same atom are independent, PITA(IND,IND) is faster than general purpose inference algorithms. When we do not know whether these assumptions hold, PITA(OPT) applies simplified probability computations when the corresponding assumptions hold on the program while resorting to the general probability computations otherwise.

In the future, we plan to compare these techniques with (lifted) weighted model counting [33, 37] that allow to answer multiple conditional queries at the same time. Moreover, we plan to apply these techniques to learning systems such as ALLPAD [38, 39], RIB [40], EMBLEM [41] and SLIPCASE [42].

REFERENCES

[1] De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S. (eds.) (2008) *Probabilistic Inductive Logic Programming - Theory and Applications*, LNCS, **4911**. Springer, Berlin.

[2] Sato, T. (1995) A statistical learning method for logic programs with distribution semantics. *Proceedings of ICLP*, Tokyo, 13-16 June, pp. 715–729. MIT Press, Cambridge, MA.

[3] Dantsin, E. (1991) Probabilistic logic programs and their semantics. *Proceedings of RLP*, Irkutsk, Russia, St. Petersburg, Russia, 14-18 September 1990, 11-16 September 1991, LNCS, **592**, pp. 152–164. Springer, Berlin.

[4] Poole, D. (1997) The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, **94**, 7–56.

[5] Sato, T. and Kameya, Y. (1997) PRISM: A language for symbolic-statistical modeling. *Proceedings of IJCAI*, Nagoya, Japan, 23-29 August, pp. 1330–1339. Morgan Kaufmann, Burlington, MA.

[6] Fuhr, N. (2000) Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *J. Am. Soc. Inf. Sci.*, **51**, 95–110.

[7] Vennekens, J., Verbaeten, S., and Bruynooghe, M. (2004) Logic programs with annotated disjunctions. *Proceedings of ICLP*, Saint-Malo, France, 6-10 September, LNCS, **3131**, pp. 195–209. Springer, Berlin.

[8] De Raedt, L., Kimmig, A., and Toivonen, H. (2007) ProbLog: A probabilistic Prolog and its application in link discovery. *Proceedings of IJCAI*, Hyderabad, India, 6-12 January, pp. 2462–2467. IJCAI/AAAI, Palo Alto, CA.

[9] Vennekens, J. and Verbaeten, S. (2003) Logic programs with annotated disjunctions. Technical Report CW386. K. U. Leuven, Leuven, Belgium.

[10] De Raedt, L. et al. (2008) Towards digesting the alphabet-soup of statistical relational learning. *Proceedings of the NIPS Workshop on Probabilistic Programming*, Whistler, Canada, 13-14 December. `http://probabilistic-programming.org/wiki/NIPS*2008_Workshop/Schedule#talk-deraedt`.

[11] Poole, D. (2000) Abducing through negation as failure: stable models within the Independent Choice Logic. *J. Log. Program.*, **44**, 5–35.

[12] Kimmig, A., Demoen, B., Raedt, L. D., Costa, V. S., and Rocha, R. (2011) On the implementation of the probabilistic logic programming language ProbLog. *Theory Pract. Log. Program.*, **11**, 235–262.

[13] Riguzzi, F. (2007) A top down interpreter for LPAD and CP-logic. *Proceedings of AI*IA*, Rome, Italy, 10-13 September, LNAI, **4733**, pp. 109–120. Springer, Berlin.

[14] Riguzzi, F. (2009) Extended semantics and inference for the Independent Choice Logic. *Log. J. IGPL*, **17**, 589–629.

[15] Riguzzi, F. (2008) Inference with logic programs with annotated disjunctions under the well founded semantics. *Proceedings of ICLP*, Udine, Italy, 9-13 December, LNCS, **5366**, pp. 667–771. Springer, Berlin.

[16] Riguzzi, F. (2010) SLGAD resolution for inference on Logic Programs with Annotated Disjunctions. *Fundam. Inform.*, **102**, 429–466.

[17] Riguzzi, F. and Swift, T. (2010) Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. *Technical Communications of ICLP*, Edinburgh, Scotland, 16-19 July, LIPIcs, **7**, pp. 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[18] Riguzzi, F. and Swift, T. (2013) Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory Pract. Log. Program.*, **13**, 279–302.

[19] Riguzzi, F. and Swift, T. (2011) The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory Pract. Log. Program.*, **11**, 433–449.

[20] Hommersom, A. and Lucas, P. (2011) Generalising the interaction rules in probabilistic logic. *Proceedings of IJCAI*, Barcelona, Spain, 16-22 July, pp. 912–917. IJCAI/AAAI, Palo Alto, CA.

[21] Riguzzi, F. (2012) Optimizing inference for probabilistic logic programs exploiting independence and exclusiveness. *Proceedings of CILC*, Rome, Italy, 6-7 June, CEUR Workshop Proceedings, **857**, pp. 206–220. Sun SITE Central Europe, Aachen, Germany.

[22] Cussens, J. (2000) Stochastic logic programs: Sampling, inference and applications. *Proceedings of UAI*, Stanford, CA, 30 June - 3 July, pp. 115–122. Morgan Kaufmann, Burlington, MA.

[23] Valiant, L. G. (1979) The complexity of enumeration and reliability problems. *SIAM J. Comp.*, **8**, 410–421.

[24] Thayse, A., Davio, M., and Deschamps, J. P. (1978) Optimization of multivalued decision algorithms. *Proceedings of MVL*, Rosemont, IL, January 1978, pp. 171–178. IEEE Computer Society Press, Los Alamitos, CA,.

[25] Sang, T., Beame, P., and Kautz, H. A. (2005) Performing bayesian inference by weighted model counting. *Proceedings of AAAI*, Pittsburgh, PA, 9-13 July, pp. 475–482. AAAI Press / The MIT Press, Palo Alto, CA.

[26] Swift, T. and Warren, D. S. (2012) XSB: Extending prolog with tabled logic programming. *Theory Pract. Log. Program.*, **12**, 157–187.

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

[28] Sato, T., Zhou, N.-F., Kameya, Y., and Izumi, Y. (2010). PRISM User's Manual (Version 2.0).

[29] Gerla, G. (2001) *Fuzzy Logic*, Trends in Logic, **11**. Springer Netherlands, Dordrecht, Netherlands.

[30] Bragaglia, S. and Riguzzi, F. (2011) Approximate inference for logic programs with annotated disjunctions. *Proceedings of ILP*, Florence, Italy, 27-30 June, LNAI, **6489**, pp. 30–37. Springer, Berlin.

[31] Singla, P. and Domingos, P. (2005) Discriminative training of Markov logic networks. *Proceedings of AAAI/IAAI*, Pittsburgh, PA, 9-13 July, pp. 868–873. AAAI Press/The MIT Press, Palo Alto, CA.

[32] Meert, W., Struyf, J., and Blockeel, H. (2009) CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. *Proccedings of ILP*, Leuven, Belgium, 2-4 July, LNCS, **5989**, pp. 96–109. Springer, Berlin.

[33] Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., and De Raedt, L. (2011) Inference in probabilistic logic programs using weighted CNF's. *Proceedings of UAI*, Barcelona, Spain, 14-17 July, pp. 211–220. AUAI Press, Corvallis, Oregon.

[34] Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., and Raedt, L. D. (2008) On the efficient execution of ProbLog programs. *Proceedings of ICLP*, Udine, Italy, 9-13 December, LNCS, **5366**, pp. 175–189. Springer, Berlin.

[35] Singla, P. and Domingos, P. (2006) Entity resolution with Markov logic. *Proceedings of ICDM*, Hong Kong, China, 18-22 December, pp. 572–582. IEEE Computer Society, Washington, DC.

[36] Christiansen, H. and Gallagher, J. P. (2009) Non-discriminating arguments and their uses. *Proceedings of ICLP*, Pasadena, CA, 14-17 July, LNCS, **5649**, pp. 55–69. Springer, Berlin.

[37] Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., and De Raedt, L. (2011) Lifted probabilistic inference by first-order knowledge compilation. *Proceedings of IJCAI*, Barcelona, Spain, 16-22 July, pp. 2178–2185. AAAI Press/IJCAI, Palo Alto, CA.

[38] Riguzzi, F. (2007) ALLPAD: Approximate learning of logic programs with annotated disjunctions. *Proceedings of ILP*, Santiago de Compostela, Spain, 24-27 August, LNCS, **4455**, pp. 43–45. Springer, Berlin.

[39] Riguzzi, F. (2008) ALLPAD: Approximate learning of logic programs with annotated disjunctions. *Mach. Learn.*, **70**, 207–223.

[40] Riguzzi, F. and Di Mauro, N. (2012) Applying the information bottleneck to statistical relational learning. *Mach. Learn.*, **86**, 89–114.

[41] Bellodi, E. and Riguzzi, F. (2013) Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intell. Data Anal.*, **17**, 343–363.

[42] Bellodi, E. and Riguzzi, F. (2012) Learning the structure of probabilistic logic programs. *Proceedings of ILP, Revised Papers*, London, UK, 31 July - 3 August, 2011, LNCS, **7207**, pp. 61–75. Springer, Berlin.

APPENDIX: UWCSE PROGRAM FOR ONE STUDENT

```
course(c1).
professor(p1).
student(s1).
advised_by(A,B):0.107 :-
    student(A),
    professor(B),
    position(B,faculty).
advised_by(A,B):0.027 :-
    student(A),
    professor(B),
    \+position(B,faculty).
course_level(A,level_300):0.066;
  course_level(A,level_400):0.318;
course_level(A,level_500):0.614 :-
    course(A).
phase(A,post_Generals):0.935; phase(A,post_Quals):0.032;
  phase(A,pre_Quals):0.0322 :-
    student(A),
    years_in_program(A,year_6_or_more).
phase(A,post_Generals):0.04; phase(A,post_Quals):0.04;
  phase(A,pre_Quals):0.92 :-
    student(A),
```

```
    \+years_in_program(A,year_6_or_more),
    years_in_program(A,year_1).
phase(A,post_Generals):0.566; phase(A,post_Quals):0.366;
  phase(A,pre_Quals):0.066 :-
    student(A),
    \+years_in_program(A,year_6_or_more),
    \+years_in_program(A,year_1),
    years_in_program(A,year_5).
phase(A,post_Generals):0.222; phase(A,post_Quals):0.666;
  phase(A,pre_Quals):0.111111111111111 :-
    student(A),
    \+years_in_program(A,year_6_or_more),
    \+years_in_program(A,year_1),
    \+years_in_program(A,year_5),
    years_in_program(A,year_4).
phase(A,post_Generals):0.047; phase(A,post_Quals):0.547;
 phase(A,pre_Quals):0.404 :-
    student(A),
    \+years_in_program(A,year_6_or_more),
    \+years_in_program(A,year_1),
    \+years_in_program(A,year_5),
    \+years_in_program(A,year_4).
position(A,faculty):0.732; position(A,faculty_adjunct):0.125;
  position(A,faculty_affiliate):0.071;
  position(A,faculty_emeritus):0.0714 :-
    professor(A).
aux1_prof_nb_publications(A) :-
    advised_by(B,A),
    student_nb_publications(B,three_or_more).
prof_nb_publications(A,none):0.041;
  prof_nb_publications(A,one_to_nine):0.416;
  prof_nb_publications(A,ten_or_more):0.541 :-
    professor(A),
    aux1_prof_nb_publications(A).
prof_nb_publications(A,none):0.5;
  prof_nb_publications(A,one_to_nine):0.294;
  prof_nb_publications(A,ten_or_more):0.205:-
    professor(A),
    \+aux1_prof_nb_publications(A).
shared_publication(A,B):0.001 :-
    professor(A),
    student(B),
    student_nb_publications(B,none).
shared_publication(A,B):0.956:-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    advised_by(B,A),
    years_in_program(B,year_6_or_more).
shared_publication(A,B):0.125 :-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    advised_by(B,A),
    \+years_in_program(B,year_6_or_more),
    prof_nb_publications(A,none).
shared_publication(A,B):0.631 :-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    advised_by(B,A),
    \+years_in_program(B,year_6_or_more),
    \+prof_nb_publications(A,none).
shared_publication(A,B):0.005 :-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    \+advised_by(B,A),
    prof_nb_publications(A,none).
shared_publication(A,B):0.099 :-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    \+advised_by(B,A),
    \+prof_nb_publications(A,none),
    prof_nb_publications(A,one_to_nine),
    student_nb_publications(B,one_or_two).
shared_publication(A,B):0.015 :-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    \+advised_by(B,A),
    \+prof_nb_publications(A,none),
    prof_nb_publications(A,one_to_nine),
    \+student_nb_publications(B,one_or_two).
shared_publication(A,B):0.146 :-
    professor(A),
    student(B),
    \+student_nb_publications(B,none),
    \+advised_by(B,A),
    \+prof_nb_publications(A,none),
    \+prof_nb_publications(A,one_to_nine).
student_nb_publications(A,none):0.594;
  student_nb_publications(A,one_or_two):0.195;
  student_nb_publications(A,three_or_more):0.209 :-
    student(A).
aux1_taught_by(A,B) :-
    teaching_assistant(A,C),
    advised_by(C,B).
aux2_taught_by(A,B) :-
    teaching_assistant(A,C),
    advised_by(C,B),
    shared_publication(B,C).
taught_by(A,B):0.777 :-
    course(A),
    professor(B),
    aux1_taught_by(A,B),
    aux2_taught_by(A,B).
taught_by(A,B):0.333 :-
    course(A),
    professor(B),
    aux1_taught_by(A,B),
    \+aux2_taught_by(A,B).
taught_by(A,B):0.266 :-
    course(A),
    professor(B),
    \+aux1_taught_by(A,B),
    position(B,faculty),
    course_level(A,level_300).
taught_by(A,B):0.102 :-
    course(A),
    professor(B),
    \+aux1_taught_by(A,B),
    position(B,faculty),
    \+course_level(A,level_300).
taught_by(A,B):0.029 :-
    course(A),
    professor(B),
    \+aux1_taught_by(A,B),
    \+position(B,faculty).
teaching_assistant(A,B):0.033 :-
    course(A),
    student(B),
    course_level(A,level_300),
    student_nb_publications(B,three_or_more).
teaching_assistant(A,B):0.158 :-
    course(A),
    student(B),
    course_level(A,level_300),
    \+student_nb_publications(B,three_or_more).
teaching_assistant(A,B):0.033 :-
    course(A),
    student(B),
    \+course_level(A,level_300),
    course_level(A,level_400).
teaching_assistant(A,B):0.012 :-
    course(A),
    student(B),
    \+course_level(A,level_300),
    \+course_level(A,level_400).
years_in_program(A,year_1):0.244;
  years_in_program(A,year_2):0.155;
  years_in_program(A,year_3):0.233;
  years_in_program(A,year_4):0.144;
  years_in_program(A,year_5):0.111;
  years_in_program(A,year_6_or_more):0.111 :-
    student(A),
    student_nb_publications(A,none).
years_in_program(A,year_1):0.032;
  years_in_program(A,year_2):0.064;
  years_in_program(A,year_3):0.064;
  years_in_program(A,year_4):0.209;
  years_in_program(A,year_5):0.306;
  years_in_program(A,year_6_or_more):0.322 :-
    student(A),
    \+student_nb_publications(A,none).
```