The PITA System for Logical-Probabilistic Inference

Fabrizio Riguzzi[1] and Terrance Swift[2]

[1] ENDIF – University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy
fabrizio.riguzzi@unife.it
[2] CENTRIA – Universidade Nova de Lisboa
tswift@cs.suysb.edu

1 Introduction

Probabilistic Inductive Logic Programming (PILP) is gaining interest due to its ability to model domains with complex and uncertain relations among entities. Since PILP systems generally must solve a large number of inference problems in order to perform learning, they rely critically on the support of efficient inference systems.

PITA [6] is a system for reasoning under uncertainty on logic programs. While PITA includes frameworks for reasoning with possibilistic logic programming, and for reasoning on probabilistic logic programs with special exclusion and independence assumptions, we focus here on PITA's framework for reasoning on general probabilistic logic programs following the distribution semantics: one of the most prominent approaches to combining logic programming and probability. Syntactically, PITA targets Logic Programs with Annotated Disjunctions (LPADs) [8] but can be used for other languages that follow the distribution semantics, such as ProbLog [2], PRISM [7] and ICL [5], as there are linear transformation from one language to the others [1].

PITA is distributed as a package of XSB Prolog and uses tabling along with an XSB feature called *answer subsumption* that allows the combination of different explanations for the same atom in a fast and simple way. PITA works by transforming an LPAD into a normal program and then querying the program.

In this paper we provide an overview of PITA and an experimental comparison of it with ProbLog, a state of the art system for probabilistic logic programming. The experiments show that PITA has very good performances.

2 Probabilistic Logic Programming

The distribution semantics [7] is one of the more widely used semantics for probabilistic logic programming. 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 queries; the probability of a query is obtained from this distribution by marginalization.

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:. As stated above, PITA uses LPADs because of their general syntax. LPADs are sets of disjunctive clauses in which each atom in the head is annotated with a probability.

Example 1. The following LPAD T_1 captures a Markov model with three states of which state 3 is an end state

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

The predicate $s(T, S)$ models the fact that the system is in state S at time T. As state 3 is the end state, if $s(T, 3)$ is selected at time T, no state follows.

We now present the distribution semantics for the case in which a program does not contain function symbols so that its Herbrand base is finite[3].

An *atomic choice* is a selection of the i-th atom for a grounding $C\theta$ of a probabilistic clause C and is represented by the triple (C, θ, i). 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 of composite choice κ is $P(\kappa) = \prod_{(C,\theta,i)\in\kappa} P_0(C, i)$ where $P_0(C, i)$ is the probability annotation of head i of clause C. A *selection* σ is a total composite choice (one atomic choice for every grounding of each probabilistic statement/clause). A selection σ identifies a logic program w_σ called a *world*. The probability of w_σ is $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} P_0(C, i)$. Since the program does not have 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: $P(Q|w) = 1$ if Q is true in w and 0 otherwise. The probability of the query can then be obtained by marginalizing over the query $P(Q) = \sum_w P(Q, w) = \sum_w P(Q|w)P(w) = \sum_{w\models Q} P(w)$

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 to manipulate Binary Decision Diagrams (BDDs). The idea is to add an extra argument to each literal to store a BDD encoding the explanations for the answers of the goal. The extra arguments of these literals are combined using a set of general library functions:

[3] However, the distribution semantics for programs with function symbols has been defined as well [7,5].

- *init, end*: initialize and terminate the extra data structures necessary for manipulating BDDs;
- *zero(-D), one(-D), and(+D1,+D2,-DO), or(+D1,+D2, -DO), not(+D1,-DO)*: Boolean operations between BDDs;
- *get_var_n(+R,+S,+Probs,-Var)*: returns the multi-valued random variable associated to rule R with grounding substitution S and list of probabilities *Probs*;
- *equality(+Var,+Value,-D)*: D is the BDD representing *Var=Value*, i.e. that the random variable *Var* is assigned *Value* in D;
- *ret_prob(+D,-P)*: returns the probability of the BDD D.

The PITA transformation applies to clauses, literals and atoms. The transformation for a head atom H, $PITA_H(H)$, is H with the variable D added as the last argument. Similarly, the transformation for a body atom A_j, $PITA_B(A_j)$, is A_j with the variable D_j added as the last argument. The transformation for a negative body literal $L_j = \neg A_j$, $PITA_B(L_j)$, is the Prolog conditional $(PITA'_B(A_j) \rightarrow not(DN_j, D_j); one(D_j))$, where $PITA'_B(A_j)$ is A_j with the variable DN_j added as the last argument. In other words, the input data structure, DN_j, is negated if it exists; otherwise the data structure for the constant function 1 is returned.

The disjunctive clause $C_r = H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n \leftarrow L_1, \ldots, L_m.$ where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$:

$$\begin{aligned} PITA(C_r, i) = PITA_H(H_i) \leftarrow\ & one(DD_0), \\ & PITA_B(L_1), and(DD_0, D_1, DD_1), \ldots, \\ & PITA_B(L_m), and(DD_{m-1}, D_m, DD_m), \\ & get\_var\_n(r, VC, [\alpha_1, \ldots, \alpha_n], Var), \\ & equality(Var, i, DD), and(DD_m, DD, D). \end{aligned}$$

for $i = 1, \ldots, n$, where VC is a list containing each variable appearing in C_r. PITA uses tabling and a feature called *answer subsumption* available in XSB that, when a new answer for a tabled subgoal is found, it combines old answers with the new one accoring to a partial order or (upper semi-)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 of the transformed program 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. At a minimum, the predicate of the goal and all the predicates appearing in negative literals should be tabled with answer subsumption. However, it is usually better to table every predicate whose answers have multiple explanations and are going to be reused often.

In Prolog systems that do not have answer subsumption, such as Yap, its behavior can be simulated on acyclic programs by using the transformation

$$PITA(C_r, i) = PITA_H(H_i) \leftarrow bagof(DB, EV\hat{}(one(DD_0),$$
$$PITA_B(L_1), and(DD_0, D_1, DD_1), \ldots,$$
$$PITA_B(L_m), and(DD_{m-1}, D_m, DD_m),$$
$$get\_var\_n(r, VC, [\alpha_1, \ldots, \alpha_n], Var),$$
$$equality(Var, i, DD), and(DD_m, DD, DB)), L),$$
$$or\_list(L, D).$$

where EV is the list of variables appearing only in the body except DB and $or\_list/2$ computes the *or*-join of all BDDs in the list passed as the first argument.

4 Experiments

PITA was tested on six datasets: a Markov model from [8], the biological networks from [2] and the four testbeds of [4]. PITA was compared with the exact version of ProbLog [2] available in the git version of Yap as of 15 June 2011[4]. This version of Problog can exploit tabling, but as mentioned above, it cannot exploit answer subsumption which is not available in Yap.

The first problem is modeled by the program in Example 1. For this experiment, we query the probability of the model being in state 1 at time N for increasing values of N. For both PITA and ProbLog, we did not use reordering of BDDs variables[5] and we tabled the $s/2$ predicate. The graph of the execution times (Figure 1) shows that PITA achieves a large speedup with respect to ProbLog.

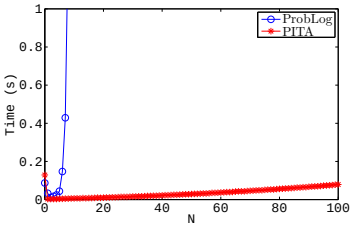


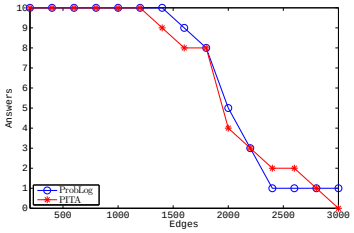
Fig. 1. Hidden Markov model.

The biological network programs compute the probability of a path in a large graph in which the nodes encode biological entities and the links represents conceptual relations among them. Each program in this dataset contains a non-probabilistic definition of path plus a number of links represented by probabilistic facts. The programs have been sampled from a very large graph and contain 200, 400, ..., 10000 edges. Sampling was repeated ten times, to obtain ten series of programs of increasing size. In each program we query the probability that the two genes HGNC_620 and HGNC_983 are related. For PITA, we used the

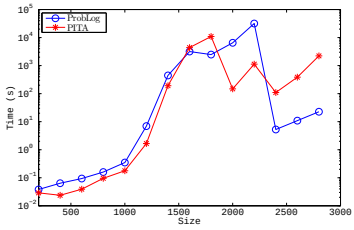
[4] All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.

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

definition of path from [3] that performs loop checking explicitly by keeping the list of visited nodes. For ProbLog we used a definition of path in which tabling is exploited for performing loop checking. $path/2$, $edge/2$ and $arc/2$ are tabled in ProbLog, while only $path/2$ is tabled in PITA. We found these to be the best performing settings for the two systems. 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 both algorithm succeeded. Here there is no clear winner, with PITA faster for smaller graphs and ProbLog solving slightly more graphs and faster for larger graphs.



(a) Number of successes.



(b) Average execution times on the graphs on which both algorithms succeeded.

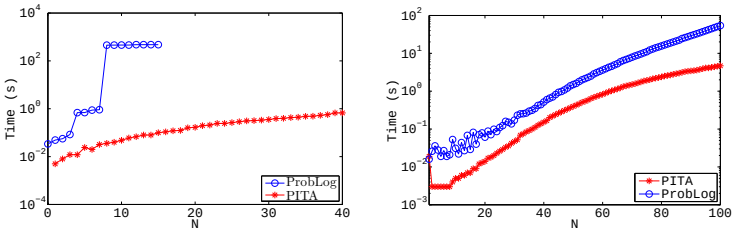
Fig. 2. Biological graph experiments.

The four datasets of [4] are: `bloodtype`, that encodes the genetic inheritance of blood type; `growingbody`, that contains programs with growing bodies; `growinghead` that contains programs with growing heads, and `uwcse`, that encodes a university domain. The best results for ProbLog were obtained by using tabling in all experiments except `growinghead`. For PITA, all the predicates are tabled. The execution times of PITA and ProbLog are shown in Figures 3(a) and 3(b), 4(a) and 4(b)[6]. In `bloodtype`, `growingbody` and `growinghead` variable reordering was turned off for both systems, while in `uwcse` group sift automatic reordering was used. In these experiments PITA is faster and more scalable than ProbLog.

References

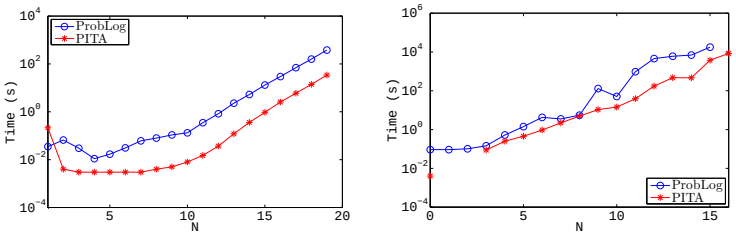
1. De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., Vennekens, J.: Towards digesting the alphabet-soup of statistical relational learning. In: NIPS*2008 Workshop on Probabilistic Programming (2008)

[6] 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.



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

Fig. 3. Datasets from (Meert et al. 2009).



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

Fig. 4. Datasets from (Meert et al. 2009).

2. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: International Joint Conference on Artificial Intelligence. pp. 2462–2467 (2007)
3. Kimmig, A., Demoen, B., De Raedt, L., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language problog. Theor. Pract. of Log. Prog. 11(Special Issue 2-3), 235–262 (2011)
4. Meert, W., Struyf, J., Blockeel, H.: CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In: International Conference on Inductive Logic Programming. KU LEuven (2009)
5. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. J. Log. Prog. 44(1-3), 5–35 (2000)
6. Riguzzi, F., Swift, T.: Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In: International Conference on Logic Programming. LIPIcs, vol. 7, pp. 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)
7. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: International Conference on Logic Programming. pp. 715–729. MIT Press (1995)
8. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: International Conference on Logic Programming. LNCS, vol. 3131, pp. 195–209. Springer (2004)