Probabilistic Inference in SWI-Prolog

Fabrizio Riguzzi[1], Jan Wielemaker[2], and Riccardo Zese[3]

[1] Dipartimento di Matematica e Informatica – University of Ferrara, Ferrara, Italy
[2] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[3] Dipartimento di Ingegneria – University of Ferrara, Ferrara, Italy
`j.wielemaker@cs.vu.nl,[fabrizio.riguzzi,riccardo.zese]@unife.it`

Abstract. Probabilistic Logic Programming (PLP) emerged as one of the most prominent approaches to cope with real-world domains. The distribution semantics is one of most used in PLP, as it is followed by many languages, such as Independent Choice Logic, PRISM, pD, Logic Programs with Annotated Disjunctions (LPADs) and ProbLog. A possible system that allows performing inference on LPADs is PITA, which transforms the input LPAD into a Prolog program containing calls to library predicates for handling Binary Decision Diagrams (BDDs). In particular, BDDs are used to compactly encode explanations for goals and efficiently compute their probability. However, PITA needs mode-directed tabling (also called tabling with answer subsumption), which has been implemented in SWI-Prolog only recently. This paper shows how SWI-Prolog has been extended to include correct answer subsumption and how the PITA transformation has been changed to use SWI-Prolog implementation.

Keywords: Probabilistic Logic Programming, Tabling, Answer Subsumption, Logic Programs with Annotated Disjunctions, Program Transformation

1 Introduction

Probabilistic Programming (PP) [13] allows users to define complex probabilistic models and perform inference and learning on them. In fact, many real-world domains can only be represented effectively by exploiting uncertainty. To model complex domains containing many uncertain relationships among their entities, Probabilistic Logic Programming (PLP) [8,16] has emerged among the PP proposals as one of the most prominent approaches to cope with such relationships.

In PLP, the distribution semantics [22] is at the basis of many languages. Examples of languages that follow the distribution semantics are Independent Choice Logic [14], PRISM [22], Logic Programs with Annotated Disjunctions (LPADs) [28] and ProbLog [9]. All these languages have the same expressive power as a theory in one language can be translated into another [7]. LPADs offer a general syntax as the constructs of all the other languages can be directly encoded in this language.

The PITA algorithm (for "Probabilistic Inference with Tabling and Answer subsumption"), presented in [18,19,20], takes as input an LPAD and builds Binary Decision Diagrams (BDDs) for every subgoal encountered during the derivation of the query. The use of BDDs is due to the fact that they allow an efficient computation of the probability of the query. Specifically, PITA transforms the input LPAD into a normal logic program in which the subgoals have an extra argument storing a BDD that represents the explanations for their answers. As its name implies, PITA uses *tabling* with *answer subsumption* to store explanations for a goal.

Tabling is a logic programming technique for saving time and ensuring termination for programs without function symbols. With tabling, the Prolog interpreter keeps a store of the subgoals encountered in a derivation together with answers to these subgoals. The stored answers are then retrieved in successive calls of the subgoals, to avoid their re-computation and infinite loops.

PITA is also used in machine learning systems: EMBLEM [4,3] and SLIPCOVER [5], performing respectively parameter and structure learning, use PITA for computing the probability of examples.

Initially, PITA was implemented in the XSB and YAP systems, which allow the use of answer subsumption. We decided to port it to SWI-Prolog in order to exploit the SWISH framework for developing web applications [29]. This allowed us to develop cplint on SWISH [1], a web application for probabilistic logic programming available at `http://cplint.eu`.

The porting of PITA to the SWI-Prolog system led to the extension of its tabling implementation to feature answer subsumption as well. With answer subsumption, or *mode-directed* tabling, the answers for a goal are aggregated when the individual answers are not needed.

In this paper, we discuss the extension implemented in SWI-Prolog's tabling and present how we modified PITA to cope with the new SWI-Prolog implementation of mode-directed tabling.

The paper is organized as follows. Section 2 briefly introduces PLP and in particular the LPAD language. Section 3 discusses tabling in SWI-Prolog, while Section 4 illustrates the implementation of answer subsumption. Section 5 presents the PITA algorithm and transformation, and Section 6 its adaptation to SWI-Prolog. Finally, Section 7 concludes the paper.

2 Probabilistic Logic Programming

PLP languages under the distribution semantics [22] have been used to represent a wide variety of domains [1,2,17]. A program in a language adopting the distribution semantics defines a probability distribution over normal logic programs called *worlds*. Each normal program is assumed to have a total well-founded model [25]. Then, the distribution is extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs.

Logic Programs with Annotated Disjunctions (LPADs) [27] have the most general syntax among PLP languages under the distribution semantics.

In LPADs, heads of clauses are disjunctions in which each atom is annotated with a probability. An LPAD T is a finite set of clauses: $T = \{C_1, \ldots, C_n\}$. Each clause C_i takes the form: $h_{i1} : \Pi_{i1}; \ldots; h_{iv_i} : \Pi_{iv_i} :- b_{i1}, \ldots, b_{iu_i}$, where $h_{i1}, \ldots, h_{iv_i}$ are logical atoms, $b_{i1}, \ldots, b_{iu_i}$ are logical literals and $\Pi_{i1}, \ldots, \Pi_{iv_i}$ are real numbers in the interval $[0, 1]$ that sum to 1. $b_{i1}, \ldots, b_{iu_i}$ is indicated with $body(C_i)$. Note that if $v_i = 1$ the clause corresponds to a non-disjunctive clause. We also allow clauses where $\sum_{k=1}^{v_i} \Pi_{ik} < 1$: in this case 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 $1 - \sum_{k=1}^{v_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T.

We present here the semantics of LPADs for the case of no function symbols, for the case of function symbols see [15].

Each grounding $C_i\theta_j$ of a clause C_i corresponds to a random variable X_{ij} with values $\{1, \ldots, v_i\}$, i.e., it indicates which head literal is chosen. The random variables X_{ij} are independent of each other. An *atomic choice* [14] is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, v_i\}$ identifies one of the head atoms. In practice (C_i, θ_j, k) corresponds to an assignment $X_{ij} = k$.

A *selection* σ is a set of atomic choices that, for each clause $C_i\theta_j$ in $ground(T)$, contains an atomic choice (C_i, θ_j, k). A selection σ identifies a normal logic program l_σ defined as $l_\sigma = \{(h_{ik} :- body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. l_σ is called a *world* of T. Since the random variables associated to ground clauses are independent, we can assign a probability to instances: $P(l_\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$.

We consider only *sound* LPADs where, for each selection σ, the well-founded model of the program l_σ chosen by σ is two-valued. We write $l_\sigma \models q$ to mean that the query q is true in the well-founded model of the program l_σ. Since the well-founded model of each world is two-valued, q can only be true or false in l_σ.

We denote the set of all instances by L_T. Let $P(l)$ be the distribution over instances. The probability of a query q given an instance l is $P(q|l) = 1$ if $l \models q$ and 0 otherwise. The probability of a query q is given by

$$P(q) = \sum_{l \in L_T} P(q, l) = \sum_{l \in L_T} P(q|l)P(l) = \sum_{l \in L_T : l \models q} P(l) \tag{1}$$

Example 1. The following LPAD models the appearance of medical symptoms as a consequence of disease. A person may sneeze if he has the flu or if he has hay fever:

$C_1 = strong\_sneezing(X) : 0.3 \; ; \; moderate\_sneezing(X) : 0.5 \leftarrow$
$flu(X).$
$C_2 = strong\_sneezing(X) : 0.2 \; ; \; moderate\_sneezing(X) : 0.6 \leftarrow$
$hay\_fever(X).$
$C_3 = flu(bob).$
$C_4 = hay\_fever(bob).$

Here clauses C_1 and C_2 have three alternatives in the head of which the one associated to atom *null* is left implicit. This program has 9 worlds, the query

$strong\_sneezing(bob)$ is true in 5 of them, and $P(strong\_sneezing(bob)) = 0.3 \times 0.2 + 0.3 \times 0.6 + 0.3 \times 0.2 + 0.2 \times 0.5 + 0.2 \times 0.2 = 0.44$.

3 Tabling

Tabling is a logic programming technique for saving time and ensuring termination for programs without function symbols.

With tabling, the Prolog interpreter keeps a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, its answers are retrieved from the store rather than recomputing them. Tabling is implemented in the Prolog systems XSB [24], YAP [21] and SWI-Prolog [30].

Tabling is implemented in SWI-Prolog using delimited control [10]. Delimited control [6,11] was originally introduced in functional programming and is based on two operators, implemented in SWI-Prolog with the predicates `reset(Goal,Cont,Term1)` and `shift(Term2)`. The first executes the goal in `Goal` and unifies the other two arguments on the basis of the results of calls to `shift/1` during the execution of the goal. If `Goal` calls `shift/1`, the execution of the goal is interrupted, the rest of its code up to the nearest call to `reset/3`, called *delimited continuation*, is represented as a Prolog term and unified with `Cont` in `reset/3`, while the value `Term2` in `shift/1` is unified with `Term1` in `reset/3`. Finally, the execution restarts from the code just after the call to `reset/3`.

Example 2. We report here the example shown in [10]. Consider the following program:

```
p :-  reset(q,Cont,Term1),
      writeln(Term1),
      writeln(Cont),
      writeln('end').
q :-  writeln('before shift'),
      shift('return value'),
      writeln('after shift').
```

`shift/1` instantiates `Cont` with the `writeln('after shift')` goal and `Term1` with the term `'return value'` in `reset/3`. The output of this program is:

```
?- p.
before shift
return value
[$cont$(785488,[])]
end
```

As one can see, when entering in `q` the execution is interrupted by the call to `shift/1`. The continuation in this case is not called, therefore what follows the call to `shift/1` is not executed.

If we modify `p` by replacing `writeln(Cont)` with `call(Cont)`, then the input would be

```
?- p.
before shift
after shift
end
```

In this case, continuation is called and, therefore, the goal `writeln('after shift')` is executed.

Predicates are declared as tabled using the `table/1` directive. Tabled predicates are transformed in order to collect answers. This transformation makes use of the `table/2` predicate, which retrieves the *table* data structure containing the answers to the tabled predicate.

Example 3. The program on the left is transformed in that shown on the right

```
:- table p/2.                  p(X,Y) :- table(p(X,Y),p_aux(X,Y)).
p(X,Y) :- p(X,Z), e(Z,Y).  →  p_aux(X,Y) :- p(X,Z), e(Z,Y).
p(X,Y) :- e(X,Y).              p_aux(X,Y) :- e(X,Y).
```

When a tabled predicate is called, the execution enters in a delimited answer computation starting the `reset` phase. If this phase succeeds normally, the answer is added to the table of the tabled predicate. If the tabled predicate calls a predicate that is tabled as well, then the computation enters in the `shift` phase without producing an answer and the first predicate is suspended, capturing the reminder in `Cont`. At this point the so-called `completion` phase starts, collecting all the possible continuation, to find answers for the tabled predicate in the `reset` phase.

A call to a tabled predicate can be either a leader or a follower: a *leader* has only non-tabled ancestors in the call graph, while a *follower* has a tabled ancestor in the call graph. The leader and the followers that are his descendants make up a *scheduling component*. `completion` is performed on one component at a time.

Each component is associated to a *global worklist*, i.e., a queue of tables. There is a table for each subgoal for the tabled predicates, called *call variants*, mapping it to a data structure containing its answers, in the form of an answer *trie*.

Each table is also associated to a *local worklist* that is a dequeue containing answers and dependencies. A *dependency* is a triple formed by a *source*, a *continuation* and a *target*. If collecting answers for a tabled call p requires the answers for a tabled call q (q may be p itself), then p is the target and q is the source. A dependency indicates that, given an answer for the source call q, we can obtain an answer for the target call p by resuming the suspended continuation. The continuation's answer is then unified with p.

During the `completion` phase, tables from the global worklist are extracted one at a time and the local worklist of the considered table is used to find all the answers for the corresponding tabled call. During the `reset` phase, each time an answer is found for a call p, it is added to the list of answers in the table for p and to the left of the dequeue of the local worklist of subgoals calling p, while each time the execution enters in the `shift` phase a new dependency for p is added to the right of its worklist. Then, pairs (*answer*, *dependency*) are extracted from the dequeue of the local worklist to try to find new answers. Note that the *answer* in the pair is an answer for the *source* predicate. Pairs are created by associating an *answer* to the dependency that is immediately to its right in the dequeue. After the combination, the *answer* and the *dependency* just combined are swapped, moving the *answer* to the right of the *dependency*, meaning that their pair have been already tested. Then, *answer* and *dependency* from the pair are combined using values in *answer* to instantiate variables in *source*, *continuation* and, eventually, in *target*, and the predicate in *continuation* is called to find new answers for the target, i.e., instantiate all the remaining free variables in *target*. The new answer for *target* is then added to the answers list in its table and to the left of the dequeue of the local worklists where the predicate is the source of some dependencies. The `completion` phase stops when all the answers in all the local worklists are on the left of all the dependencies, meaning that all the combinations have been tested and no more answers can be found.

Note that in the real implementation there are some minor differences with this description due to performance reasons. For example, answers and dependencies in local worklists are considered in homogeneous batches instead of one at a time and the combination of the two batches is performed by means of Cartesian product. Moreover, the batches containing answers contain also answers for predicates different from that in *source*, however, during the combination if *answer* and *dependency* do not match then their combination is discarded.

When all the answers for the subogoal in the `reset` phase are found, they are returned by `table/2` one at a time to continue the computation of the query.

4 Mode-Directed Tabling and Answer Subsumption

As seen in the previous section, plain tabling creates an answer table per *call variant* and guarantees termination if the created data structures are finite. For example, this allows us to prove whether a graph connects the nodes A and B while the graph is undirected or cyclic. However, it does *not* allow us to return the path between A and B because an infinite number of such paths can be constructed by going back and forth or following cycles. This limitation also blocks us from obtaining all proofs for a logical theory.

The above problem is resolved using tabling with *answer subsumption*, also called *mode-directed tabling* [24,26]. In *mode-directed tabling*, a subset of the predicate arguments defines the call variant while answers for the remaining arguments are *aggregated*. A classical aggregation is, following the example above, computing the minimum or shortest path. As the minimum can be considered to *subsume* higher values this technique is also called *Answer Subsumption*.

To facilitate PITA, we extended SWI-Prolog's original tabling implementation with mode-directed tabling. The specification was inherited from XSB, B-Prolog and YAP and includes the most generic aggregation function called *lattice* that allows a user defined predicate to determine the subsumer for the aggregated answer so far and a new answer. The implementation is straight forward. The modified `table/1` directive is used to determine the term that is used for call variant detection and compile a combined aggregation predicate that is called for each answer that is added to the table. The answer table has been extended such that each answer in the answer *trie* can be assigned an aggregated value. The above mentioned generated predicate is called on each answer to maintain the aggregated answer.

Note that tabling does *not* guarantee a particular order in which suspended computations are resumed and thus requires the aggregation function to produce the correct result regardless of the order.

Furthermore, if one mode-directed tabled goal is the *follower* of another as in the example below where, given the goal `p(A)`, `p/1` is the *leader* and `s/1` the *follower* we get incorrect results because `s/1` may succeed multiple times with partial answers.

```
:- table
    p(lattice(or/3)),
    s(lattice(or/3)).
or(A,B,A-B).
p(A) :- s(A).
s(1).
s(2).
```

In the initial implementation of mode-directed tabling in SWI-Prolog, the query `p(A)` succeeded with answer `A = 1-2-(1-2)` instead of the desired `A = (1-2)`.

This has been highlighted in [26] that showed that many implementations of mode-directed tabling produce unsound results. The authors of [26] thus define a formal semantics for mode-directed tabling that allows the evaluation of the soundness of implementations. For the program above, the semantics returns `A = (1-2)`.

In the semantics, aggregation is a post-processing step. Real systems aggregate intermediate results during resolution for efficiency and to avoid loops. The authors of [26] discuss conditions for this greedy strategy to be sound with respect to the theoretical semantics.

In order to make SWI-Prolog sound, we modified its tabling implementation by creating a new *component* for every fresh mode-directed tabled goal we encounter. This component is completed before execution of the parent component is resumed with the complete aggregated result. This ensures soundness with respect to the theoretical semantics of [26], provided that within the subcomponent we do not encounter a variant of a tabled goal that was started before the subcomponent but has not yet been completed.

5 PITA

The PITA system [18,19,20] applies a program transformation to an LPAD to create a normal program that contains calls for manipulating BDDs. In the implementation, these calls provide a Prolog interface to the CUDD[4] [23] C library and use the following predicates[5]

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

As said above, *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 a particular grounding. The auxiliary predicate has the following definition

```
get_var_n(R,S,Probs,Var):-
     (var(R,S,Var) ->
        true
     ;
        length(Probs,L),
        add_var(L,Probs,Var),
        assert(var(R,S,Var))
     ).
```

where `Probs` is a list of floats that stores the parameters in the head of rule `R`. `R`, `S` and `Probs` are input arguments while `Var` is an output argument. `assert/1` is a builtin Prolog predicate that adds its argument to the program, allowing its dynamic extension.

The PITA transformation applies to atoms, literals, conjunction of literals and clauses. The transformation for an atom a and a variable D, $PITA(a, D)$, is a with the variable D added as the last argument. The transformation for a negative literal $b = \textbf{not}\, a$, $PITA(b, D)$, is the expression

$$(PITA(a, DN) \rightarrow not(DN, D); one(D))$$

which is an if-then-else construct in Prolog: if $PITA(a, DN)$ evaluates to true, then $not(DN, D)$ is called, otherwise $one(D)$ is called.

[4] `http://vlsi.colorado.edu/~fabio/`
[5] BDDs are represented in CUDD as pointers to their root node.

A conjunction of literals $b_1, \ldots, b_m$ becomes:

$PITA(b_1, \ldots, b_m, D) = one(DD_0),$
$\quad PITA(b_1, D_1), and(DD_0, D_1, DD_1), \ldots,$
$\quad PITA(b_m, D_m), and(DD_{m-1}, D_m, D).$

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 $PITA(C_r)$:

$PITA(C_r, i) = PITA(h_i, D) \leftarrow PITA(b_1, \ldots, b_m, DD_m),$
$\quad get\_var\_n(r, S, [\Pi_1, \ldots, \Pi_n], Var), equality(Var, i, DD),$
$\quad and(DD_m, DD, D).$

for $i = 1, \ldots, n$, where S is a list containing all the variables appearing in r.

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

$PITA(C_r) = PITA_h(h, D) \leftarrow one(D).$

A disjunctive fact $C_r = h_1 : \Pi_1 \vee \ldots \vee h_n : \Pi_n$. where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r, i)$

$PITA(C_r, i) = get\_var\_n(r, S, [\Pi_1, \ldots, \Pi_n], Var),$
$\quad equality(Var, i, DD), and(DD_m, DD, D).$

for $i = 1, \ldots, n$.

In the case where the parameters do not sum to one, the clause is first transformed into $null : 1 - \sum_1^n \Pi_i \vee h_1 : \Pi_1 \vee \ldots \vee h_n : \Pi_n$. and then into the clauses above, where the list of parameters is $[1 - \sum_1^n \Pi_i, \Pi_1, \ldots, \Pi_n,]$ but the 0-th clause (the one for *null*) is not generated.

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

$PITA(C_r) = PITA(h, D) \leftarrow PITA(b_1, \ldots, b_m, D).$

Example 4 (Medical example - PITA). Clause C_1 from the LPAD of Example 1 is translated to

$strong\_sneezing(X, BDD) \leftarrow one(BB_0), flu(X, B_1),$
$\quad and(BB_0, B_1, BB_1),$
$\quad get\_var\_n(1, [X], [0.3, 0.5, 0.2], Var),$
$\quad equality(Var, 1, B), and(BB_1, B, BDD).$
$moderate\_sneezing(X, BDD) \leftarrow one(BB_0), flu(X, B_1),$
$\quad and(BB_0, B_1, BB_1),$
$\quad get\_var\_n(1, [X], [0.3, 0.5, 0.2], Var),$
$\quad equality(Var, 2, B), and(BB_1, B, BDD).$

while clause C_3 is translated to

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

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

$prob(Goal, P) \leftarrow init, retractall(var(\_, \_, \_)),$
$\quad add\_bdd\_arg(Goal, BDD, GoalBDD),$
$\quad (call(GoalBDD) \rightarrow ret\_prob(BDD, P); P = 0.0),$
$\quad end.$

Predicate *equality(+Var,+Value,-BDD)* returns a BDD representing the equality *Var=Value*. Since variables may be multi-valued, an encoding with Boolean variables must be chosen. The encoding used by PITA is the same as that used to translate LPADs into ProbLog proposed in [7].

Example 5 (Example 1 Cont.). If we associate the random variables X_{11} with $(C_1, \{X/bob\}, 1)$, X_{12} with $(C_1, \{X/bob\}, 2)$, X_{13} with $(C_1, \{X/bob\}, 3)$ X_{21} with $(C_2, \{X/bob\}, 1)$, X_{22} with $(C_2, \{X/bob\}, 2)$ and X_{23} with $(C_2, \{X/bob\}, 3)$, the BDD corresponding with the set L_T for query $strong\_sneezing(bob)$ is shown in Figure 1. The probability of the query is computed by following the BDD and

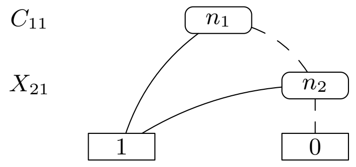

Fig. 1. BDD for query $strong\_sneezing(bob)$ in Example 1.

computing the probability of each node

$$\text{PROB}(n_2) = 0.2 \cdot 1 + 0.8 \cdot 0 = 0.2$$
$$\text{PROB}(n_1) = 0.3 \cdot 1 + 0.7 \cdot 0.2 = 0.44$$

so $P(strong\_sneezing(bob)) = \text{PROB}(n_1) = 0.44$ as shown in Example 1.

6 Extension of PITA for SWI-Prolog

PITA in SWI-Prolog adds the library predicate:

– *and_check(+D1,+D2,-DO)* fails if one of the input arguments is the BDD representing the Boolean constant 0, otherwise it succeeds returning the conjunction of the input arguments.

The tabling implementation in SWI-Prolog doesn't handle cut thus the if-then-else construct cannot be used to implement negation. In SWI-Prolog the transformation for a negative literal $b = \textbf{not}\, a$, $PITA(b, DN)$ is the conjunction $PITA(a, D), not(D, DN)$. A conjunction of literals $b_1, \ldots, b_m$ becomes:
$PITA(b_1, \ldots, b_m, D) = one(DD_0),$
$PITA(b_1, D_1), and\_check(DD_0, D_1, DD_1), \ldots,$
$PITA(b_m, D_m), and\_check(DD_{m-1}, D_m, D).$
Clauses are then transformed as in PITA for XSB. Moreover, for each predicate p/n, an extra clause of the form

$$p(X_1, \ldots, X_n, D) \leftarrow nonvar(X_1), \ldots, nonvar(X_n), zero(D).$$

is added to the program, where $nonvar/1$ is an extra-logical predicate that succeeds if its argument is not a variable. We call these *zero clauses.*

Such a transformation is used in combination with *answer subsumption.* Each predicate is tabled and *answer subsumption* is applied to the BDD argument added by the PITA transformation, defining as *lattice* the *or/3* predicate. This combines every BDD, which corresponds to a different explanation for the call variant of the predicate, in order to compute a final BDD representing the set of all the explanations. If the goal fails, the only BDD returned is the one representing the 0 constant, which leads to the fail of *and_check/3*, otherwise, the zero BDD is disjoint with other BDDs, maintaining unchanged their truth value.

In this way, negative literals $b = \textbf{not}\, a$ are handled by first collecting the BDD representing all the explanations for a by means of answer subsumption. The final BDD is then negated to find the BDD for b.

7 Conclusion

In this paper we presented an extension of the tabling system of SWI-Prolog for including sound answer subsumption. Moreover, we presented an extension of the PITA transformation, which takes an LPAD program and translates it into a normal program using the tabling implementation of SWI-Prolog. Possible future directions for improving the tabling implementation are sharing tables between threads, incremental tabling, handling negation, improving space and time performance. We will also extend PITA to handle other reasoning types, such as inference and learning for probabilistic abductive logic programs, extending [12]. In addition, we plan to make a comparison with XSB in terms of performance.

References

1. Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., Zese, R.: `cplint` on SWISH: Probabilistic logical inference with a web browser. Intell. Artif. 11(1), 47–64 (2017)
2. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: Adorni, G., Cagnoni, S., Gori, M., Maratea, M. (eds.) AI*IA 2016. LNCS, vol. 10037, pp. 351–363. Springer International Publishing (2016)
3. Bellodi, E., Riguzzi, F.: Experimentation of an expectation maximization algorithm for probabilistic logic programs. Intell. Artif. 8(1), 3–18 (2012)
4. Bellodi, E., Riguzzi, F.: Expectation maximization over binary decision diagrams for probabilistic logic programs. Intell. Data Anal. 17(2), 343–363 (2013)
5. Bellodi, E., Riguzzi, F.: Structure learning of probabilistic logic programs by searching the clause space. Theor. Pract. Log. Prog. 15(2), 169–212 (2015)
6. Danvy, O., Filinski, A.: Abstracting control. In: LISP and Functional Programming. pp. 151–160 (1990), `http://doi.acm.org/10.1145/91556.91622`
7. 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)
8. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming, LNCS, vol. 4911. Springer (2008)
9. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007. vol. 7, pp. 2462–2467. AAAI Press/IJCAI (2007)

10. Desouter, B., van Dooren, M., Schrijvers, T.: Tabling as a library with delimited control. Theor. Pract. Log. Prog. 15(4-5), 419–433 (2015)
11. Felleisen, M.: The theory and practice of first-class prompts. In: Ferrante, J., Mager, P. (eds.) Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988. pp. 180–190. ACM Press (1988), `http://doi.acm.org/10.1145/73560.73576`
12. Kakas, A.C., Riguzzi, F.: Abductive concept learning. New Generat. Comput. 18(3), 243–294 (2000)
13. Pfeffer, A.: Practical Probabilistic Programming. Manning Publications (2016)
14. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. 94, 7–56 (1997)
15. Riguzzi, F.: The distribution semantics for normal programs with function symbols. Int. J. Approx. Reason. 77, 1 – 19 (October 2016)
16. Riguzzi, F.: Foundations of Probabilistic Logic Programming. River Publishers (2018), to appear
17. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Softw.-Pract. Exper. 46(10), 1381–1396 (10 2016)
18. Riguzzi, F., Swift, T.: Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In: ICLP TC 2010. LIPIcs, vol. 7, pp. 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
19. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theor. Pract. Log. Prog. 11(4–5), 433–449 (2011)
20. Riguzzi, F., Swift, T.: Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. Theor. Pract. Log. Prog. 13(2), 279–302 (2013)
21. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog system. Theor. Pract. Log. Prog. 12(1-2), 5–34 (2012)
22. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995)
23. Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. University of Colorado (2015), `http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf`
24. Swift, T., Warren, D.S.: XSB: Extending prolog with tabled logic programming. Theor. Pract. Log. Prog. 12(1-2), 157–187 (2012)
25. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)
26. Vandenbroucke, A., Piróg, M., Desouter, B., Schrijvers, T.: Tabling with sound answer subsumption. Theor. Pract. Log. Prog. 16(5-6), 933–949 (2016)
27. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer (2004)
28. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3131, pp. 431–445. Springer (2004)
29. Wielemaker, J., Lager, T., Riguzzi, F.: SWISH: SWI-Prolog for sharing. In: Ellmauthaler, S., Schulz, C. (eds.) International Workshop on User-Oriented Logic Programming (IULP 2015) (2015)
30. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theor. Pract. Log. Prog. 12(1-2), 67–96 (2012)