Probabilistic Inference in SWI-Prolog

Fabrizio Riguzzi  Jan Wielemaker  Riccardo Zese

MCS – DE – University of Ferrara, Centrum Wiskunde & Informatica
[fabrizio.riguzzi,riccarzo.zese]@unife.it, j.wielemaker@cs.vu.nl
Outline

- Tabling in SWI-Prolog
- Answer Subsumption
- PITA
- PITA for SWI-Prolog
Distribution Semantics [Sato ICLP95]

- A probabilistic logic program defines a probability distribution over normal logic programs (called instances or possible worlds or simply worlds)
- The distribution is extended to a joint distribution over worlds and interpretations (or queries)
- The probability of a query is obtained from this distribution
Probabilistic Logic Programming (PLP) Languages under the Distribution Semantics

- Probabilistic Logic Programs [Dantsin RCLP91]
- Probabilistic Horn Abduction [Poole NGC93], Independent Choice Logic (ICL) [Poole AI97]
- PRISM [Sato ICLP95]
- Logic Programs with Annotated Disjunctions (LPADs) [Vennekens et al ICLP04]
- ProbLog [De Raedt et al IJCAI07]
- They differ in the way they define the distribution over logic programs
http://cplint.eu/e/sneezing.pl

\[ C_1 = strong\_sneezing(X) : 0.3 ; moderate\_sneezing(X) : 0.5 \leftarrow flu(X). \]

\[ C_2 = strong\_sneezing(X) : 0.2 ; moderate\_sneezing(X) : 0.6 \leftarrow hay\_fever(X). \]

\[ C_3 = flu(bob). \]

\[ C_4 = hay\_fever(bob). \]

- Distributions over the head of rules
- Worlds obtained by selecting one atom from the head of every grounding of each clause
Tabling

- A logic programming technique for saving time and ensuring termination for programs without function symbols
- 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 re-computing them
- Implemented in XSB, YAP, SWI-Prolog, B-Prolog, Ciao
Implemented in SWI-Prolog using delimited control [Desouter et al TPLP15]

Two operators, reset and shift

reset(Goal,Cont,Term1) executes Goal and unifies the other two arguments on the basis of the results of calls to shift/1

If Goal calls shift(Term2)
- 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
- Term2 is unified with Term1
- The execution restarts from the code just after the call to reset/3
Example of Delimited Continuation

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

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

- 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.
Example of Delimited Continuation

- If we replace `writeln(Cont)` with `call(Cont)`
  
  ```prolog
  ?- p.
  before shift
  after shift
  end
  ```

- The continuation is called and the goal `writeln('after shift')` is executed
Predicates are declared as tabled using the `table/1` directive.

Tabled predicates are transformed.

`table/2` retrieves the `table` data structure containing the answers to the tabled predicate.

\[
\text{:- table } p/2.
\]

\[
p(X,Y) \leftarrow p(X,Z), e(Z,Y).
p(X,Y) \leftarrow e(X,Y).
\]

\[
\rightarrow p(X,Y) \leftarrow \text{table}(p(X,Y), 'p tabled'(X,Y)).
\]

\[
'p tabled'(X,Y) \leftarrow p(X,Z), e(Z,Y).
'p tabled'(X,Y) \leftarrow e(X,Y).
\]
When a tabled predicate is called, the execution enters in a reset phase for delimited answer computation.

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 continuations, to find answers for the tabled predicate in the reset phase.
A leader is a call to a tabled predicate that has only non-tabled ancestors in the dynamic call graph.

Other calls to tabled predicates are followers.

Every follower has a leader as its ancestor.

The leader and its followers make up a scheduling component.

Multiple scheduling components can occur during program execution.

Completion performed on one component at a time.
For each component:
- **Global worklist**: a queue of tables, each tables maps a subgoal for a tabled predicate (*call variant*) to a trie containing its answers and to a **local worklist**, a dequeue containing answers and dependencies
- **dependency** = (*source*, *continuation*, *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
- 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*
Tabling in SWI-Prolog

- **completion phase:** tables from the global worklist are extracted one at a time.
- The local worklist of the 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 $p$.
- During 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.
- The **answer** in the pair is an answer for the **source** predicate.
(answer, dependency) 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 are swapped, moving the answer to the right of the dependency

Then, answer and dependency from the pair are combined using values in answer to instantiate variables in source, continuation and target

The predicate in continuation is called to find new answers for the 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
Mode-Directed Tabling and Answer Subsumption

- Answer subsumption, also called mode-directed tabling [Swift, Warren TPLP 12, Vandenbroucke et al TPLP 16]
- A subset of the predicate arguments defines the call variant while answers for the remaining arguments are aggregated
- When a new answer is found, it is aggregated with an existing answer in the table
- Classical aggregation: minimum
- SWI-Prolog’s original tabling implementation was extended with mode-directed tabling
- Specification inherited from XSB, B-Prolog, YAP,
Answer Subsumption Example

:- table connection(_,_,min).

connection(X, Y,1) :-
    connection(X, Y).

connection(X, Y,N) :-
    connection(X, Z,N1),
    connection(Z, Y),
    N is N1+1.

connection('Amsterdam', 'Schiphol').
connection('Amsterdam', 'Haarlem').
connection('Schiphol', 'Leiden').
connection('Haarlem', 'Leiden').
connection('Amsterdam', 'Leiden').

?- connection('Amsterdam','Leiden',N).
N=1
Most generic aggregation function: \texttt{lattice}, a user defined predicate determines the subsumer for the aggregated answer so far and a new answer:

\begin{verbatim}
:- table pred(_,_,lattice(join)).
\end{verbatim}

The answer table assigns each answer in the trie an aggregated value.
Mode-Directed Tabling and Answer Subsumption

- 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.
- If one mode-directed tabled goal is the follower of another we may get incorrect results.
In the initial implementation \( p(A) \) succeeded with answer \( A = 1-2-(1-2) \) instead of the desired \( A = (1-2) \)
[Vandenbroucke et al TPLP 16] showed that many implementations of mode-directed tabling produce unsound results.

Formal semantics for mode-directed tabling that allows the evaluation of the soundness of implementations.

Aggregation is a post-processing step.

Real systems aggregate intermediate results during resolution for efficiency and to avoid loops.
In SWI-Prolog: create 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.

If in a subcomponent we encounter a variant of a tabled goal that was started before the subcomponent but has not yet been completed, failure.
PITA [Riguzzi, Swift ICLP10, ICLP11, TPLP13] applies a program transformation to an LPAD to create a normal program that contains calls for manipulating BDDs

Library:

- `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(+BDD1, -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`. 
PITA transformation

- Auxiliary predicate `get_var_n/4` used to wrap `add_var/3` and avoid adding a new variable when one already exists for an instantiation.
- Atom \( a \): \( PITA(a, D) \), is \( a \) with the variable \( D \) added as the last argument.
- Negative literal \( b = \text{not } a \): \[
(PITA(a, DN) \rightarrow \text{not}(DN, D); \text{one}(D))
\]
- Conjunction of literals \( b_1, \ldots, b_m \): \[
PITA(b_1, \ldots, b_m, D) = \text{one}(DD_0), \\
PITA(b_1, D_1), \text{and}(DD_0, D_1, DD_1), \ldots, \\
PITA(b_m, D_m), \text{and}(DD_{m-1}, D_m, D).
\]
PITA transformation

- Disjunctive clause
  \[ C_r = h_1 : \Pi_1 \lor \ldots \lor h_n : \Pi_n \leftarrow b_1, \ldots, b_m. \]

\[
PITA(C_r, i) = PITA(h_i, D) \leftarrow PITA(b_1, \ldots, b_m, DD_m),
\]

\[
get\_var\_n(r, S, [\Pi_1, \ldots, \Pi_n], Var), equality(Var, i, DD),
\]

\[
and(DD_m, DD, D).
\]

for \( i = 1, \ldots, n \), where \( S \) is a list containing all the variables appearing in \( r \)
Clause $C_1$ from the example LPAD is translated to

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

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

Clause $C_3$:

$$\text{flu}(\text{david}, \text{BDD}) \leftarrow \text{one}(\text{BDD}).$$
PITA transformation

- Predicates tabled as

  \[-\text{table } p(_, \ldots, \text{lattice}(or/3)),\]

- \text{prob}(\text{Goal}, P) to answer queries:

  \[
  \text{prob}(\text{Goal}, P) \leftarrow \text{init}, \text{retractall}(\text{var}(\_, \_, \_)), \\
  \text{add\_bdd\_arg}(\text{Goal}, \text{BDD}, \text{GoalBDD}), \\
  (\text{call}(\text{GoalBDD}) \rightarrow \text{ret\_prob}(\text{BDD}, P); P = 0.0), \\
  \text{end}.
  \]
Extension of PITA for SWI-Prolog

- Extra 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.

- Transformation for a negative literal $b = \text{not } a$, $PITA(b, DN)$:
  \[
PITA(a, D), \text{not}(D, DN)
  \]

- Conjunction of literals $b_1, \ldots, b_m$:
  \[
PITA(b_1, \ldots, b_m, D) = \text{one}(DD_0),
  
PITA(b_1, D_1), \text{and}_\text{check}(DD_0, D_1, DD_1), \ldots,
  
PITA(b_m, D_m), \text{and}_\text{check}(DD_{m-1}, D_m, D).
  \]
For each predicate \( p/n \), an extra clause (zero clauses) of the form

\[
p(X_1, \ldots, X_n, D) \leftarrow \text{nonvar}(X_1), \ldots, \text{nonvar}(X_n), \text{zero}(D).
\]

If the goal fails, the only BDD returned is the one representing the 0 constant, negated we get the 1 constant.

In conjunctions, failure of \texttt{and_check/3}.

In disjunctions, the zero BDD is disjoint with other BDDs, keeping unchanged their truth value.
Conclusions & Future Work

**Conclusions**
- Tabling in SWI-Prolog
- Answer Subsumption
- PITA
- PITA for SWI-Prolog

**Future work**
- Sharing tables between threads, incremental tabling, handling negation, improving space and time performance
- Extending PITA for probabilistic abductive logic programs
- Comparison with XSB in terms of performance
THANKS FOR LISTENING AND ANY QUESTIONS?