Toward a High-performance System for Symbolic and Statistical Modeling

Neng-Fa Zhou
Department of Computer Science
Brooklyn College & Graduate Center
The City University of New York
zhou@sci.brooklyn.cuny.edu

Taisuke Sato
Department of Computer Science,
Tokyo Institute of Technology, CREST JST
sato@mi.cs.titech.ac.jp

Koiti Hasida
Cyber Assist Research Center
AIST Tokyo Waterfront
hasida.k@aist.go.jp

Abstract

We present in this paper a state-of-the-art implementation of PRISM, a language based on Prolog that supports statistical modeling and learning. We start with an interpreter of the language that incorporates a naive learning algorithm, and then turn to improve the interpreter. One of the improvements is to refine the learning algorithm such that it works on explanation graphs rather than flat explanations. Tabling is used to construct explanation graphs so that variant subgoals do not need to be considered redundantly. Another technique is compilation. PRISM programs are compiled into a form that facilitates searching for all solutions. The implemented system is, to our knowledge, the first of its kind that can support real-world applications. The implemented system, which will be available from *http://sato-www.cs.titech.ac.jp/prism/index.html*, is being applied to several problem domains ranging from statistical language processing, decision support, to game analysis.

1 Introduction

PRISM (PRogramming In Statistical Modeling) [Sato, 1995; Sato and Kameya, 2001] is a new language that integrates probability theory and Prolog, and is suitable for the description of computations in which randomness or uncertainty is involved. PRISM provides built-ins for describing experiments [1]. A PRISM program can be executed in three different modes, namely *sample execution*, *probability calculation*, and *learning*. In sample execution mode, a goal may give different results depending on the outcomes of the experiments. For example, it is possible for a goal to succeed if a coin shows the head after being tossed and to fail if the coin shows the tail. The probability calculation mode gives the probability of a goal to succeed. In the learning mode, the system estimates the probabilities of the outcomes of the experiments from given observed data. The PRISM system adopts the EM (Expectation and Maximization) algorithm [Dempster *et al.*, 1976] in probability estimation.

PRISM, as a symbolic statistical modeling language, subsumes several specific statistical tools such as HMM (Hidden Markov Models) [Rabiner, 1989], PCFG (Probabilistic Context Free Grammars) [Wetherell, 1980] and descrete Bayesian networks [Castillo *et al.*, 1997; Pearl, 1987]. Compared with numeric models where mathematical formulas are used, PRISM offers incomparable flexibility by allowing the use of arbitrary logic programs to describe probability distributions. PRISM can be used in many areas such as language processing, decision making, bio-informatics, and game theory where randomness or uncertainty is essential.

This project aims at implementing an efficient system for PRISM in B-Prolog. For most applications, learning is time-consuming especially when the amount of observed data is large. The EM learning algorithm estimates the probabilities of outcomes through two phases: the first phase searches for all explanations for the observed facts, and the second phase estimates the probabilities. The first phase is the neck of the learning algorithm. We have made several efforts to speed-up this phase. One is to tabulate partial explanations for subgoals such that explanations for variant subgoals are searched only once. With tabling, this phase gives an *explanation graph* that facilitates the estimation of probabilities. The tabling mecha-

[1] An experiment is defined by a sample space and a probability distribution for the outcomes in the sample space. For example, tossing a coin is an experiment where the sample space is {`head`, `tail`} and the probability distribution is uniform (this means that the events `head` and `tail` have the same likelihood to occur) if the coin is fair.

nism of B-Prolog is improved such that copy of data between the heap and the tabling area is reduced significantly. This improved version demonstrates a big speed-up when complex goals with structured data need to be tabulated. Another technique used in the system is compilation. PRISM programs are compiled into a form that facilitates searching for all solutions.

The main part of this paper is devoted to the implementation techniques. To make the paper self-contained, we start with an interpreter of PRISM in the next section. The description of the operational semantics is informal and is based on examples. The reader is referred to [Sato and Kameya, 2001] for a formal description of the semantics and the EM learning algorithm adopted in PRISM.

2 PRISM: The Language and its Implementation

PRISM is an extension of Prolog that provides built-ins for statistical modeling and learning.

2.1 Built-ins

The built-in `msw(I,V)` describes a trial of an *experiment*, where `I` is the identifier of an experiment, and `V` is the outcome of the trial[2]. The identifier `I` can be any complex term, but `I` must be ground when the trial is conducted. In the sample-execution mode, the built-in `msw(I,V)` succeeds if the trial of the experiment `I` gives the outcome `V`. If `V` is a variable, then the built-in always succeeds, binding `V` to the outcome of the experiment.

For each experiment, the user must specify the *sample space* by defining the predicate `values(I,Space)`, where `I` is the identifier and `Space` is a list of possible outcomes of the experiment. A *probability distribution* of an experiment tells the probabilities of the outcomes in the sample space. The sum of the probabilities of the outcomes in any experiment must be 1.0. Probability distributions are either given by the programmer or obtained through learning from given *sample data*. The predicate `set_sw(I,Probs)` sets the probabilities of the outcomes in the experiment `I`, where `Probs` is a list of probabilities (floating-point numbers). The length of `Probs` must be the same as the number of outcomes in the sample space and the sum of the probabilities must be equal to 1.0.

The following shows an illustrative example:

```
direction(D):-
    msw(coin,Face),
    (Face==head->D=left;D=right).

values(coin,[head,tail]).
```

[2]The name *msw* is an abbreviation for *multi-outcome switch*. In the version presented in [Sato and Kameya, 2001], the built-in takes another argument called *trial number*. The same trial of the same experiment must give the same outcome. In the new version, all trials are considered independent by default. If the outcome of a trial needs to be reused, the programmer must have it passed as an argument or have it saved in the global database.

The predicate `direction(D)` determines the direction to go by tossing a coin; `D` is bound to `left` if the head is shown, and to `right` if the tail is shown. To set uniform distribution, we use `set_sw(coin,[0.5,0.5])` to set the probabilities to the two outcomes. Notice that the following gives a different definition of `direction`:

```
direction(left):-
    msw(coin,head).
direction(right):-
    msw(coin,tail).
```

While for the original definition, the query `direction(D)` always succeeds, binding `D` to either `left` or `right`. The same query may fail for the new definition since `msw(coin,head)` and `msw(coin,tail)` are two separate trials. If the first trial gives `tail` and the second trial gives `head`, then the query `direction(D)` fails.

In addition to `msw/2`, PRISM provides several other built-ins, including `prob(Goal,Prob)` for computing the probability of a goal, `sample(Goal)` for sample executing a goal, and `learn(Facts)` for estimating the probabilities of the switches in the program from the observed facts. These built-ins will be explained in the subsequent subsections.

A predicate is said to be *probabilistic* if it is defined in terms of `msw` or predicates that are probabilistic. Predicates that do not use (either directly or indirectly) `msw` in its definition are said to be *non-probabilistic*. This terminology is extended naturally to goals. A goal is said to be *probabilistic* if its predicate is probabilistic.

2.2 Sample execution

The subgoal `sample(Goal)` starts executing the program with respect to `Goal` in the sample execution mode. If `Goal` is the built-in `msw(I,V)`, then `sample(Goal)` succeeds if the trial of the experiment `I` gives the outcome `V`. The outcome of an experiment is chosen randomly, but the probability distribution is respected such that those outcomes that have the highest probabilities have the most chances to be chosen. Trials of experiments are independent regardless of whether or not the experiments are the same.

If `Goal` is non-probabilistic, then `sample(Goal)` behaves in the same way as `call(Goal)`. Otherwise, if `Goal` is probabilistic, then a clause `H:-Body` is selected from its predicate such that `H` unifies `Goal`, and `sample(Goal)` is reduced to `sample(Body)`.

The following shows a simplified version of the interpreter for sample execution:

```
sample((A,B)):-!,
    sample(A),
    sample(B).
sample(msw(I,V)):-!,
    R is random(0.0,1.0),
    % R is a random number in the range of 0.0..1.0
    prob_distribution(I,Values,Probs),
    % probability distribution assigned to the experiment
    choose_outcome(R,Values,Probs,V).
sample(Goal):-prob_predicate(Goal),!,
    clause(Goal,Body),
    sample(Body).
```

```
sample(Goal):- % non-probablistic
   call(Goal).

choose_outcome(R,Values,Probs,V):-
   choose_outcome(R,0.0,Values,Probs,V).
choose_outcome(R,Sum,[V|Values],[P|Probs],V):-
   Sum1 is Sum+P,
   R=<Sum1,!.
choose_outcome(R,Sum,[_|Values],[P|Probs],V):-
   Sum1 is Sum+P,
   choose_outcome(R,Sum1,Values,Probs,V).
```

For an experiment whose sample space is `[V1,V2,...,Vn]` and whose probability distribution is `[P1,P2,...,Pn]`, the call

```
choose_outcome(R,[V1,V2,...,Vn],[P1,P2,...,Pn],V)
```

selects the outcome `Vk` such that $\sum_{i=1}^{k} Pi \geq R$ and $\sum_{i=1}^{k-1} Pi < R$.

The real interpreter handles other constructs including negation, disjunction, if-then-else, and the cut operator in addition to conjunction.

2.3 Calculating the probabilities of goals

In statistical modeling, it is often necessary to calculate the probability of events. In PRISM, the built-in `prob(Goal,Prob)` calculates the probability `Prob` with which `Goal` becomes true. It is assumed that all probabilistic ground atoms in the Herbrand base of a program are probabilistically *independent* and *exclusive*. With these assumptions, the probability of the conjunction `(A,B)` is computed as the product of the probabilities of `A` and `B` (*independent*), and the probability of the disjunction `(A;B)` is computed as the sum of the probabilities of `A` and `B` (*exclusive*). For a switch `msw(I,V)`, the probability is 1.0 if `V` is a variable, and the probability assigned to the outcome `V` if `V` an element is the sample space.

For example, recall the illustrative example `direct`. Assume the distribution of the `coin` experiment is uniform. The probability of `direction(left)` is `0.5` since the probability of `msw(coin,head)` is `0.5`. The probability of `direction(D)` is 1.0 sine the sum of the probabilities of `msw(coin,head)` and `msw(coin,tail)` is 1.0.

The programmer must bear the above assumptions in mind when writing programs. Programs that violates this assumption will give wrong results. For example, the conjunction `(A,A)`, which makes sense logically, is not allowed probabilistically since the conjuncts are not independent. Likewise the disjunction `(A;A)` is not allowed. If the disjuncts are not independent, the probability of a goal may exceed 1.0.

One question arises: if events are assumed to be independent, then how to represent conditional events in PRISM? Let B and C be two experiments. Assume C has the possible outcomes $\{c_1, ..., c_n\}$. The conditional event $(B|C)$ can be represented by using n switches: $msw(b(c_i), V_i)$ (i=1,...,n). Consider, for example, the following problem taken from [Stirzaker, 1994], which is a typical example of Bayesian reasoning.

> You have a blood test for some rare disease which occurs by chance in 1 in every 100,000 people. The test is fairly reliable; if you have the disease it will correctly say so with probability 0.95; if you do not have the disease, the test will wrongly say you do with probability 0.005. If the test says you do have the disease, what is the probability that this is a correct diagnosis?

Let D be the event that you have the disease, D' the event that you do not have the disease, and T the event that the test says you do. Then the probability $P(D|T)$ is calculated as follows based on the Bayes' Theorem:

$$\begin{aligned} P(D|T) &= \frac{P(T|D)P(D)}{P(T)} \\ &= \frac{P(T|D)P(D)}{P(T|D)P(D)+P(T|D')P(D')} \\ &= \frac{0.95\times 0.00001}{0.95\times 0.00001+0.005\times 0.99999} \\ &= 0.1896 \end{aligned}$$

The Bayesian network for this problem consists of two nodes, called `disease` and `test`. The outcomes of both nodes are $\{$`yes, no`$\}$. The node `test` is dependent on the node `disease`. The following clause represents the network:

```
disease_test(D,T):-
   msw(disease,D),
   msw(test(D),T).
```

The sample spaces of all the experiments are `[yes,no]`. The switch `msw(disease,yes)` says that you have the disease, and the switch `msw(disease,no)` says no. The switch `msw(test(D),T)`, which depends on the outcome of the node `disease`, says that the diagnostic result is `T` if the outcome of `disease` is `D`. For the problem, the given probabilities are set as follows:

```
set_sw(disease,[0.00001,0.99999]),
   % P(D)=0.00001
set_sw(test(yes),[0.95,0.05]),
   % P(T|D)=0.95
set_sw(test(no),[0.005,0.995])
   % P(T|D')=0.005
```

If the test says you do have the disease, then the probability that this is a correct diagnosis is calculated by the query:

```
prob(disease_test(yes,yes),P1),
prob(disease_test(_,yes),P2),
P is P1/P2.
```

The goal `prob(disease_test(yes,yes),P1)` gives the probability of the event that you have the disease and is also diagnosed so, and the goal `prob(disease_test(_,yes),P2)` gives the probability of the event that you are diagnosed of the disease regardless whether or not you have the disease. The query gives the same result 0.1896 as the one obtained by using Bayes' Theorem directly.

Since new switches can be created when needed, it is possible to represent in PRISM any Bayesian networks and perform Bayesian reasoning on them.

2.4 Learning

The built-in `learn(Facts)` takes `Facts`, a list of observed facts, and estimates the probabilities of the switches that explain `Facts`. While `sample(Goal)` and `prob(Goal,Prob)` are deductive, using the current distributions of switches to deduct `Goal`, `learn(Facts)` is abductive, which finds the explanations for `Facts` and use the explanations to estimate the distributions of the switches.

PRISM adopts the EM learning algorithm to learn distributions. It first finds all the explanations for the observed facts. Then it repeatedly estimates and maximizes the likelihood of the observed facts until the estimation is stable.

An *explanation* for an observed fact is a set of switches that occur in a path of the execution of the fact. The following is an interpreter that searches for explanations for a goal:

```
expls(G,Exs):- %Exs is a list of explanations for G
    findall(Ex,expl(G,Ex,[]),Exs).
expl((G1,G2),Ex,ExR):-!,
    expl(G1,Ex,Ex1),
    expl(G2,Ex1,ExR).
expl(msw(I,V),[mse(I,V)|ExR],ExR):-!,
    values(I,Values), % sample space is Values
    member(V,Values).
expl(G,Ex,ExR):-
    prob_predicate(G),!, %G is a probabilistic
    clause(G,B),
    expl(B,Ex,ExR).
expl(G,Ex,Ex):-
    call(G).
```

Recall our illustrative example `direction`. For the fact `direction(left)`, the interpreter finds `[msw(coin,head)]`, and for the fact `direction(right)` it finds `[msw(coin,tail)]` as the explanations. In general, there may exist multiple execution paths for an observed fact and each execution path may contain multiple switches.

After all the explanations are found, the EM algorithm turns to estimate the probabilities of the switches in the explanations. Let I be the set of switches, and V_i be the sample space of switch i. For each switch $msw(i,v)$, $\theta_{i,v}$ denotes the probability of the outcome v. The following assertion must hold

$$\forall_{i \in I} \Sigma_{v \in V i} \theta_{i,v} = 1.0.$$

Let F be a set of observed facts. For each fact $f \in F$, E_f denotes the set of explanations. Let $e \in E_f$ be an explanation. The probability of e is the product of the probabilities of all the switches in the explanation:

$$\theta_e = \prod_{msw(i,v) \in e} (\theta_{i,v})$$

The probability of fact f is the sum of the probabilities of all its explanations:

$$\theta_f = \sum_{e \in E_f} (\theta_e)$$

The *log likelihood* of fact f is defined as $\ln(\theta_f)$. For each explanation $e \in E_f$, let $\delta_{i,v}(e)$ denote the number of occurrences of the switch $msw(i,v)$ in e. Figure 1 shows the EM algorithm. The algorithm repeats the estimation until the likelihood of the observed facts becomes stable.

procedure em(F) **begin**
  initialize ϵ to a small positive number;
  foreach $i \in I$, $v \in V_i$ initialize $\theta_{i,v}$;
  $\lambda^1 = \sum_{f \in F} (\ln(\theta_f))$; /* initial likelihood */
  repeat
    $\lambda^0 = \lambda^1$;
    foreach $i \in I$, $v \in V_i$
      $\eta_{i,v} = \sum_{f \in F} (\frac{\sum_{e \in E_f} (\theta_e \times \delta_{i,v}(e))}{\theta_f})$
      /* expected count of $msw(i,v)$ */
    foreach $i \in I$, $v \in V_i$
      $\theta_{i,v} = \frac{\eta_{i,v}}{\sum_{v' \in V i} (\eta_{i,v'})}$
    $\lambda^1 = \sum_{f \in F} (\ln(\theta_f))$;
  until $\lambda^1 - \lambda^0 < \epsilon$
end

Figure 1: The EM algorithm

The use of the term $\eta_{i,v}$, which estimates the number of occurrences of the switch $msw(i,v)$ that contribute to the observed facts, is essential in the algorithm. The probability of $msw(i,v)$ is estimated as the ratio of its count to the count of all the outcomes of the switch.

$$\theta_{i,v} = \frac{\eta_{i,v}}{\sum_{v' \in V i} (\eta_{i,v'})}$$

For our illustrative example, the algorithm converges in a few iterations. If only `direction(left)` is observed, then the estimated probability of `head` is close to 1.0 and that of `tail` is close to 0.0; if `direction(left)` and `direction(right)` each occupy half of the observed facts, then the estimated distribution is close to uniform. For more complicated programs, more iterative steps are required to obtain a stable estimation.

3 Improvements of the Implementation

The interpreters and the EM learning algorithm presented in the previous section are naive and inefficient. The number of explanations for a set of observed facts may be exponential. Therefore, it is expensive to find explanations and it is also expensive to go though the explanations to estimate the probabilities of the switches in the explanations. In this section, we propose several techniques for improving the implementation, especially the learning algorithm.

3.1 Explanation Graphs

It is not hard to notice that explanations differ from each other by only a small number of switches. Just as it is important to factor out common sub-expressions in evaluating expressions, it is important to factor out common switches among explanations. Actually, a logic program provides a natural structure for factoring out common switches. Instead of considering explanations as lists of switches, we consider explanations as a graph.

An *explanation path* for a fact H is defined as $(H \rightarrow B_g \;\&\; B_s)$ where B_g is a set of facts and B_s is a set of switches. H is called the *root* of the path. An explanation path corresponds to an instance of a clause where B_g is the set of probabilistic subgoals, and B_s the set of switches in the body. An *explanation tree* for a fact consists of a set of explanation paths that have the fact as the root. The root of the paths is also called the root of the tree. An explanation tree corresponds to an instance of a predicate. An *explanation graph* consists of a set of explanation trees whose roots are all distinct.

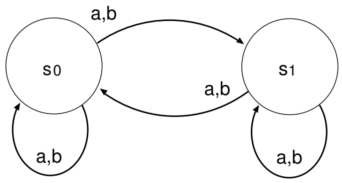

Figure 2: An example HMM.

Consider, for example, the following program that represents the two-state HMM[3] in Figure 2,

```
hmm(L,N) :-
    msw(init,Si),
    hmm(1,N,Si,L).

% Current state is S, current position is I.
hmm(I,N,S,[]) :- I>N,!.
hmm(I,N,S,[C|L]) :-
    msw(out(S),C),
    msw(tr(S),NextS),
    I1 is I+1,
    hmm(I1,N,NextS,L).

values(init,[s0,s1]).
values(out(_),[a,b]).
values(tr(_),[s0,s1]).
```

The predicate `hmm(L,N)` analyses or generates a string `L` of length `N`. The explanation graph for `hmm([a,b,a],3)` is shown in Figure 3.

It is assumed that explanation graphs are acyclic, i.e., a fact cannot be used to explain the fact itself. This assumption, however, does not rule out left recursion. Consider, for example, the following CFG rule,

```
s(I,J):-s(I,I1),a(I1,J).
```

Although `s(I,J)` and `s(I,I1)` are variants as subgoals, they are instantiated to different instances and thus no fact is used to explain the fact itself.

3.2 Constructing Explanation Graphs Using Tabling

If goals were treated independently in constructing explanation graphs, the computation would still be exponential in general. Recall the explanation graph in Figure 3. The size of the graph is $O(N \times S)$ where N is the length of the string and S is the size of the largest sample space. If shared goals in different paths, such as the two underlined ones, are considered only once, then it takes only linear time to construct the explanation graph.

```
hmm([a,b,a],3)
   → hmm(1,3,s0,[a,b,a]) & msw(init,s0)
   → hmm(1,3,s1,[a,b,a]) & msw(init,s1)
hmm(1,3,s0,[a,b,a])
   → hmm(2,3,s0,[b,a]) & msw(tr(s0),s0), msw(out(s0),a)
   → hmm(2,3,s1,[b,a]) & msw(tr(s0),s1), msw(out(s0),a)
hmm(1,3,s1,[a,b,a])
   → hmm(2,3,s0,[b,a]) & msw(tr(s1),s0), msw(out(s1),a)
   → hmm(2,3,s1,[b,a]) & msw(tr(s1),s1), msw(out(s1),a)
hmm(2,3,s0,[b,a])
   → hmm(3,3,s0,[a]) & msw(tr(s0),s0), msw(out(s0),b)
   → hmm(3,3,s1,[a]) & msw(tr(s0),s1), msw(out(s0),b)
hmm(2,3,s1,[b,a])
   → hmm(3,3,s0,[a]) & msw(tr(s1),s0), msw(out(s1),b)
   → hmm(3,3,s1,[a]) & msw(tr(s1),s1), msw(out(s1),b)
hmm(3,3,s0,[a])
   → hmm(4,3,s0,[]) & msw(tr(s0),s0), msw(out(s0),a)
   → hmm(4,3,s1,[]) & msw(tr(s0),s1), msw(out(s0),a)
hmm(3,3,s1,[a])
   → msw(tr(s1),s0), msw(out(s1),a)
   → msw(tr(s1),s1), msw(out(s1),a)
```

Figure 3: The explanation graph for `hmm([a,b,a],3)`.

Tabling or memoization [Tamaki and Sato, 1986; Warren, 1992; Zhou *et al.*, 2000] can used to avoid redundant computations. The idea of tabling is to memorize the answers to subgoals and use the answers to resolve subsequent variant subgoals. The table area is global and answers stored in it can survive over backtracking. Therefore, variant subgoals can share answers regardless where they occur in execution. They can occur in the same execution path or different paths.

The following gives an interpreter for constructing the explanation graph for a goal.

```
expls(G):-
    expl(G,_,[],_,[]),fail.
        %backtrack to find all paths
expls(G).

expl((G1,G2),Bg,BgR,Bs,BsR):-!,
    expl(G1,Bg,Bg1,Bs,Bs1),
    expl(G2,Bg1,BgR,Bs1,BsR).
expl(msw(I,V),Bg,Bg,[mse(I,V)|Bs],Bs):-!,
    values(I,Values), % sample space is Values
    member(V,Values).
expl(G,[G|Bg],Bg,Bs,Bs):-
    prob_predicate(G),!,
        %G is a probabilistic predicate
    expl_prob_goal(G).
expl(G,Bg,Bg,Bs,Bs):-
    call(G).

:-table expl_prob_goal/1.
expl_prob_goal(G):-
    clause(G,Body),
    expl(Body,Bg,[],Bs,[]),
    add_to_database(path(G,Bg,Bs)).
```

[3] An HMM is a probabilistic automaton in which the selections of the initial state, output symbols, and transitions on the symbols are all probabilistic.

The `expl(G,Bg,BgR,Bs,BsR)` is true if `Bg-BgR` is the list of probabilistic subgoals and `Bs-BsR` is the list of switches in `G`. For each probabilistic subgoal `G`, the `expl_prob_goal(G)` finds the explanation paths for `G`. The predicate `expl_prob_goal/1` is tabled. So variant probabilistic subgoals share explanation paths. The `add_to_database(path(G,Bg,Bs))` adds the path to the database if the path is not there yet.

The naive EM learning algorithm is reformulated such that it works on explanation graphs. Since explanation graphs are acyclic, it is possible to sort the trees in an explanation graph based on the calling relationship in the program. The refined algorithm is able to exploit the hierarchical structure to propagate probabilities over sorted explanation graphs efficiently.

3.3 Compilation

The interpreter presented above is inefficient since it introduces an extra level of interpretation. The interpreter version of the PRISM system is used in debugging programs. For learning from a large amount of data, it is recommended that the compiler version be used. The PRISM compiler translates a program into a form that facilitates the construction of explanation graphs.

Let `p(X1,...,Xn):-B` be a clause in a probabilistic predicate. The compiler translates it into:

```
expl_p(X1,...,Xn):-
   B',
   add_to_database(path(p(X1,...,Xn),Bg,Bs)).
```

where `B'` is the translation of `B`, `Bg` is the list of probabilistic subgoals in `B'`, and `Bs` is the list of switches in `B`. For each subgoal `G` in `B`, if `G` is `msw(I,V)`, then it is translated into `values(I,Values),member(V,Values)`. Otherwise, it is copied to `B'`, renaming each predicate `p` to `expl_p`. The translated predicate is declared as a tabled predicate, so explanation trees need to be constructed only once for variant subgoals.

For example, the predicate

```
hmm(I,N,S,[]) :- I>N,!.
hmm(I,N,S,[C|L]) :-
   msw(out(S),C),
   msw(tr(S),NextS),
   I1 is I+1,
   hmm(I1,N,NextS,L).
```

is translated into:

```
:-table expl_hmm/4.
expl_hmm(I,N,S,[]) :- I>N,!.
expl_hmm(I,N,S,[C|L]) :-
   values(out(S),Values1), % msw(out(S),C),
   member(C,Values1),
   values(tr(S),Values2), % msw(tr(S),NextS)
   member(NextS,Values2),
   I1 is I+1,
   expl_hmm(I1,N,NextS,L),
   add_to_database(path(hmm(I,N,S,[C|L]),
         [hmm(I1,N,NextS,L)],
         [msw(out(S),C),
        msw(tr(S),NextS)])).
```

Table 1: Learning times for a corpus (seconds).

# sentences	all-solution-search	EM learning
1000	268	2022
1500	445	3938
2000	855	5542

Notice that no path is added to the database for the first clause since the body does not contain switches nor probabilistic subgoals.

4 Experience

The PRISM system has been fully implemented in B-Prolog, a CLP system that supports tabling. The tabling system in B-Prolog was first implemented in 1999 [Zhou *et al.*, 2000] and was recently re-implemented to meet the requirements of PRISM. The new implementation inherits the linear tabling idea, and incorporates new strategies and optimization techniques for fast computation of fixpoints [Zhou and Sato, 2003]. As a tabling system, B-Prolog is twice as fast as and consumes an order of magnitude less stack space than XSB, a Prolog system developed at SUNY Stony Brook.

The current version of PRISM is, to our knowledge, the first of its kind that can support real-world applications. Several application projects are going on at the moment [Sato and Zhou, 2003]. One of the projects is to use PRISM to learn probabilities of the Japanese grammar rules from corpora. Table 1 shows the times spent in learning from various numbers of sentences on Windows XP (1.7G CPU, 760M RAM). The first phase of learning, i.e., finding explanations has improved significantly thanks to the adoption of the new tabling system in B-Prolog. The EM learning phase dominates the learning time now. In the current version, explanation graphs are represented as Prolog terms. The EM learning phase can be improved if better data structures are used.

5 Related Work

PRISM was first designed by Sato [Sato, 1995] who proposed a formal semantics, called *distribution semantics*, for logic programs with probabilistic built-ins, and derived an EM learning algorithm for the language from the semantics. The need for structural explanations was envisioned in [Sato and Kameya, 2001], but this paper presents the first serious implementation of the EM learning algorithm that works on explanation graphs.

Poole's abduction language [Poole, 1993] incorporates Prolog and Bayesian networks, in which probability distributions are given as *joint declarations*. Muggleton's stochastic logic language [Muggleton, 1996] is an extension of PCFG where clauses are annotated with probabilities. In both languages, probability distributions are specified by the users, and learning from sample data is not considered.

Non-logic based languages have also been designed to support statistical modeling (e.g., [Pfeffer *et al.*, 1999; Ramsey and Pfeffer, 2002]). The built-in function `choose` in the stochastic lambda calculus [Ramsey and Pfeffer, 2002] is similar to `msw` in PRISM, which returns a value from the sample space randomly. Non-logic languages do not support nondeterminism. Therefore, it would be difficult to devise an EM like learning algorithm for these languages.

Tabling shares the same idea as dynamic programming in that both approaches make full use of intermediate results of computation. Using tabling in constructing explanation graphs is analogous to using dynamic programming in the Baum-Welch algorithm for HMM [Rabiner, 1989] and the Inside-Outside algorithm for PCFG [Baker, 1979].

6 Concluding Remarks

This paper has presented an efficient implementation of PRISM, a language designed for statistical modeling and learning. The implementation is the first serious one of its kind that integrates logic programming and statistical reasoning/learning. The high performance is attributed to several techniques. One is to adopt explanation graphs rather than flat explanations in learning and use tabling to construct explanation graphs. Another technique is compilation. Programs are compiled into a form that facilitates searching for all solutions.

Acknowledgement

Thanks go to Shigeru Abe for his help in implementing the refined EM learning algorithm. Part of the work by Neng-Fa Zhou was conducted while he visited Tokyo Institute of Technology in the summer of 2002.

References

[Baker, 1979] J. K. Baker. Trainable grammars for speech recognition. In *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pages 547–550, 1979.

[Castillo *et al.*, 1997] Enrique Castillo, Jose Manuel Gutierrez, and Ali S. Hadi. *Expert Systems and Probabilistic Network Models*. Springer, New York, 1 edition, 1997.

[Dempster *et al.*, 1976] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Proceedings of the Royal Statistical Society*, pages 1–38, 1976.

[Muggleton, 1996] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.

[Pearl, 1987] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann Publishers, Inc., 1987.

[Pfeffer *et al.*, 1999] Avi Pfeffer, Daphne Koller, Brian Milch, and Ken T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In Kathryn B. Laskey and Henri Prade, editors, *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 541–550, S.F., Cal., July 30–August 1 1999. Morgan Kaufmann Publishers.

[Poole, 1993] David Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[Rabiner, 1989] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recoginition. *Proceedings of the IEEE*, 77:257–286, 1989.

[Ramsey and Pfeffer, 2002] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 154–165, Portland, Oregon, January 16–18, 2002.

[Sato and Kameya, 2001] A. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, pages 391–454, 1999.

[Sato and Zhou, 2003] T. Sato and N.F. Zhou. A new perspective of prism. *IJCAI Workshop on Learning Statistical Models from Relational Data*, submitted, 2003.

[Sato, 1995] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 715–730, Cambridge, June 13–18 1995. MIT Press.

[Stirzaker, 1994] D. Stirzaker. *Elementary Probability*. Cambridge University Press, 1 edition, 1994.

[Tamaki and Sato, 1986] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 84–98, London, 1986. Springer-Verlag.

[Warren, 1992] D. S. Warren. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming*, 35(3):93, March 1992.

[Wetherell, 1980] C. S. Wetherell. Probabilistic languages: A review and some open questions. *ACM Computing Surveys*, 12(4):361–379, December 1980.

[Zhou and Sato, 2003] N.F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. to be submitted, 2003.

[Zhou *et al.*, 2000] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. *Lecture Notes in Computer Science*, 1753:109–123, 2000.