PRISM User's Manual

(Version 1.11.3)

Taisuke Sato*, Neng-Fa Zhou**, Yoshitaka Kameya* and Yusuke Izumi*
* Tokyo Institute of Technology
** CUNY Brooklyn College
Copyright © 2008 Taisuke Sato, Neng-Fa Zhou,
Yoshitaka Kameya and Yusuke Izumi

Preface

The past few years have witnessed a tremendous interest in logic-based probabilistic learning as testified by the number of formalisms and systems and their applications. Logic-based probabilistic learning is a multidisciplinary research area that integrates relational or logic formalisms, probabilistic reasoning mechanisms, and machine learning and data mining principles. Logic-based probabilistic learning has found its way into many application areas including bioinformatics, diagnosis and troubleshooting, stochastic language processing, information retrieval, linkage analysis and discovery, robot control, and probabilistic constraint solving.

PRISM (PRogramming In Statistical Modeling) is a logic-based language that integrates logic programming and probabilistic reasoning including parameter learning. It allows for the description of independent probabilistic choices and their consequences in general logic programs. PRISM supports parameter learning, i.e. for a given set of possibly incomplete observed data, PRISM can estimate the probability distributions to best explain the data. This power is suitable for applications such as learning parameters of stochastic grammars, training stochastic models for gene sequence analysis, game record analysis, user modeling, and obtaining probabilistic information for tuning systems performance. PRISM offers incomparable flexibility compared with specific statistical tools such as hidden Markov models (HMMs) [4, 26], probabilistic context free grammars (PCFGs) [4] and discrete Bayesian networks.

PRISM employs a proof-theoretic approach to learning. It conducts learning in two phases: the first phase searches for all the explanations for the observed data, and the second phase estimates the probability distributions by using the EM algorithm. Learning from flat explanations can be exponential in both space and time. To speed up learning, the authors proposed learning from explanation graphs and using tabling to reduce redundancy in the construction of explanation graphs. The PRISM programming system is implemented on top of B-Prolog (`http://www.probp.com/`), a constraint logic programming system that provides an efficient tabling system called *linear tabling* [42]. Tabling shares the same idea as dynamic programming in that both approaches make full use of intermediate results of computations. Using tabling in constructing explanation graphs resembles using dynamic programming in the Baum-Welch algorithm for HMMs and the Inside-Outside algorithm for PCFGs. Thanks to the good efficiency of the tabling system and the EM learner adopted in PRISM, PRISM is comparable in performance to specific statistical tools on relatively large amounts of data. The theoretical side of PRISM is comprehensively described in [35]. For an implementational view, please refer to [43].

The user is assumed to be familiar with logic programming, the basics of probability theory, and some of popular probabilistic models mentioned above. The programming system is an extension of the B-Prolog system, and only PRISM-specific built-ins are elaborated in this document. Please refer to the B-Prolog user's manual for details about Prolog built-ins.

Contact information

The latest information and resources on PRISM are available at the website below.

```
http://sato-www.cs.titech.ac.jp/prism/
```

For any questions, requests and bug-reports, please send an E-mail to:

```
prism-query[AT]mi.cs.titech.ac.jp
```

where `[AT]` is replaced with `@`.

Acknowledgments

The development team would like to thank all users of this software, and greatly appreciate those who gave valuable questions, comments and suggestions.

This software gratefully uses several free software packages, including public-domain modules used in B-Prolog, Mersenne Twister (`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`) and SPECFUN (`http://www.netlib.org/specfun/`, we implement C version of the digamma and the log-gamma functions based on the code of SPECFUN).

The project was started as an ICOT Free Software Project, and has been supported in part by the "Discovery Science" project, JST Basic Research Programs CREST "Advanced Media Technology for Everyday Living," the 21st Century COE Program "Framework for Systematization and Application of Large-scale Knowledge Resources" at Tokyo Institute of Technology, and Grant-in-Aid for Scientific Research (No. 17300043 and No. 17700140) from Ministry of Education, Culture, Sports, Science and Technology of Japan.

Organization of this manual

This document is organized as follows:

- Chapter 1 gives an overview of the PRISM language and the PRISM programming system.
- Chapter 2 describes the detail of the language.
- Chapter 3 explains how to use the programming system.
- Chapter 4 gives the detailed descriptions of the basic built-in predicates provided by the programming system.
- Chapter 5 explains how to use the utility for variational Bayesian learning, which is incorporated in version 1.11, with some introductory description.
- Chapter 6 explains how to use the utility for parallel EM learning using MPI (Message-Passing Interface), also introduced in version 1.11.
- Chapter 7 shows several program examples with detailed explanations.

To learn PRISM, it is better to see typical usages of PRISM illustrated in Chapter 1 and 7 first, and then to run the example programs in the released package. The chapters/sections whose titles are marked with * are considered as advanced, so you can skip these sections for the first time. Chapter 2 may also be skipped until the examples have been explored, but the content of this chapter (especially §2.2, §2.3 and §2.4) is essential to understanding the examples. Chapter 3 and 4 are expected to work as a (rough) reference manual. Chapters 5 and 6 have the facilities newly introduced in version 1.11, and the authors expect these chapters to be referred to (only) by the users who are interested in these extended facilities. Note that '1.11' is also referred to as a generic number of the versions numbered as 1.11.*x*, so if there is no proviso, all descriptions about version 1.11 apply to versions 1.11.*x*.

Contents

Chapter 1

Overview of PRISM

PRISM is a probabilistic extension of Prolog. Syntactically, PRISM is just Prolog augmented with a probabilistic built-in predicate and declarations. There is no restriction on the use of function symbols, predicate symbols or recursion, and PRISM programs are executed in a top-down left-to-right manner just like Prolog. In this chapter, we pick up three illustrative examples to overview the major features of PRISM. These examples will also be used in the following chapters, but for brevity of descriptions, only a part is shown here. For full descriptions of these examples, please refer to Chapter 7 or the comments in the example programs included in the released package.

1.1 Building a probabilistic model with random switches

The most characteristic feature of PRISM is that it provides random switches to make probabilistic choices. A random switch has a name, a space of possible outcomes, and a probability distribution. The first example is a simple program that uses just one random switch:

```
target(direction/1).
values(coin,[head,tail]).

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

The predicate `direction(D)` indicates that a person decides the direction to go as `D`. The decision is made by tossing a coin: `D` is bound to `left` if the head is shown, and to `right` if the tail is shown. In this sense, we can say the predicate `direction/1` is *probabilistic*. It is allowed to use disjunctions (`;`), the cut symbols (`!`) and if-then (`->`) statements as far as they work as expected according to the execution mechanism of the programming system.[1] By combining probabilistic predicates, the user can build a probabilistic model for the task at hand.

Besides the definitions of probabilistic predicates, we need to make some *declarations*. The clause `values(coin,[head,tail])` declares the outcome space of a switch named `coin`, and the call `msw(coin,Face)` makes a probabilistic choice (`Face` will be bound to the result), just like a coin-tossing. On the other hand, the clause `target(direction/1)` declares that the observable event is represented by the predicate `direction/1`. This means that we can observe the direction he/she goes.

Now let us use this program. After installation, we can invoke the programming system just running the command '`prism`':

[1] For detailed descriptions on the execution mechanism of the programming system, please visit §2.4.1 and §2.4.2.

```
% prism
PRISM 1.11.3, Sato Lab, TITECH, All Rights Reserved. Oct 2008
B-Prolog Version 7.0, All rights reserved, (C) Afany Software 1994-2008.

Type 'prism_help' for usage.
| ?-
```

where '`%`' is the prompt symbol of some shell (on Linux) or the command prompt (on Windows). In the following, removing the vertical bar, we use '`?-`' as the prompt symbol for PRISM.

Let us assume that the program above is contained in the file named '`direction.psm`'. Then, we can load the program using a built-in `prism/1` as follows:

```
?- prism(direction).
```

After loading the program, we can run the program using built-in predicates. For example, we can make a sampling by the built-in `sample/1`:

```
| ?- sample(direction(D)).
D = left ?
```

The probability distributions of switches are maintained by the programming system, so they are not buried directly in the definitions of probabilistic predicates. Since version 1.9, the switches have uniform distributions by default. So the results obtained by the multiple runs of the query above should not be biased.

On the other hand, the built-in predicate `set_sw/2` and its variations are available for setting probability distributions manually. For example, to make the coin biased, we may call

```
?- set_sw(coin,[0.7,0.3]).
```

which sets the probability of the head being shown to be 0.7. The status of random switches can be confirmed by:

```
?- show_sw.
Switch coin: unfixed: head (0.7) tail (0.3)
```

At this point, the run with `sample/1` will show a different probabilistic behavior from that was made before:

```
?- sample(direction(D)).
```

1.2 Basic probabilistic inference and parameter learning

Let us pick up another example that models the inheritance mechanism of human's ABO blood type. As is well-known, a human's blood type (phenotype) is determined by his/her genotype, which is a pair of two genes (A, B or O) inherited from his/her father and mother.[2] For example, when one's genotype is AA or AO (OA), his/her phenotype will be type A. In a probabilistic context, on the other hand, we consider a pool of genes, and let p_a, p_b and p_o denote the frequencies of gene A, B and O in the pool, respectively ($p_a + p_b + p_o = 1$). When random mating is assumed, the frequencies of phenotypes, namely, P_A, P_B, P_O and P_{AB}, are computed by Hardy-Weinberg's law [11]: $P_A = p_a^2 + 2p_ap_o$, $P_B = p_b^2 + 2p_bp_o$, $P_O = p_o^2$, and $P_{AB} = 2p_ap_b$. To represent a distribution of phenotypes instead of these mathematical formulas, we may write the following PRISM program:

[2] In this example, we take a view of classical population genetics, where a gene is considered as an abstract genetic factor proposed by Mendel.

```
target(bloodtype/1).
values(gene,[a,b,o]).

bloodtype(P) :-
    genotype(X,Y),
    ( X=Y -> P=X
    ; X=o -> P=Y
    ; Y=o -> P=X
    ; P=ab
    ).

genotype(X,Y) :- msw(gene,X),msw(gene,Y).
```

In this program, we let a switch `msw(gene,X)` instantiated with `X = a`, `X = b` and `X = o` denote a random pick-up of gene `X` from the pool, and becomes true with probability p_a, p_b and p_o, respectively. Then, from the definition of `bloodtype/1`, we can say that one of `bloodtype(P)` with `P = a`, `P = b`, `P = o` and `P = ab` becomes exclusively true with probability P_A, P_B, P_O and P_{AB}, respectively (see §2.2 for details). This implies the logical variable `P` in `bloodtype(P)` behaves as a random variable that follows the distribution of phenotypes.[3]

Here, just like the distribution $\{P_A, P_B, P_O, P_{AB}\}$ is computed from the basic one $\{p_a, p_b, p_o\}$, the probability distributions of switches form a basic distribution from which we can construct the probability distribution represented by the PRISM program. Then we consider each $\theta_{i,v}$, the probability of a *switch instance* `msw(`i`,`v`)` being true (i and v are ground terms), as a *parameter* of the program's distribution. If we give appropriate parameters, a variety of probabilistic inferences are available. For example, sampling is done with the built-in predicate `sample/1`:

```
?- sample(bloodtype(X)).
```

In the above query, the answer `X = b` will be returned with probability P_B, the frequency of blood type B. Also it is possible to compute the probability of a *probabilistic goal* (or simply, a goal):

```
?- prob(bloodtype(a)).
Probability of bloodtype(a) is: 0.360507016168634
```

Instead of being set manually, the parameters can be estimated from the observed data. We call this task *parameter learning* or more specifically, *maximum likelihood estimation* (ML estimation or MLE) — given some *observed data*, a bag of *observed goals*, find the parameters that maximize the probability of the observed data being occurred. In this case, the observed data should be a bag of instances of `bloodtype(X)`, which correspond to phenotypes of (randomly sampled) humans. This is declared in the program by the clause `target(bloodtype/1)`. Also it should be noted here that we are in a *partially observing situation*, that is, we cannot know which switch instances are true (i.e. which genes are inherited) for some given instances of `bloodtype(X)` (i.e. some phenotypes). For example, if we observed a person of blood type A, we do not know whether he has inherited two genes A from both parents, or he inherits gene A from one parent and gene O from the other. For MLE in such a situation, one solution is to use the EM (expectation-maximization) algorithm [13],[4] and the programming system

[3] From a similar discussion, in the previous example, we can see `D` in `direction(D)` as a random variable in a probabilistic context. In many cases, it is useful to define a program so that some logical variables behave as random variables, but it is also worth noting that there is no need to make all logical variables in the program behave as random variables.

[4] A more detailed description for this example (the problem of gene frequency estimation for blood types) can be found in Section 2.4 of [24].

has a built-in routine of the EM algorithm. By adding a couple of declarations and preparing some data, we can estimate the parameters from the data.

For example, let us consider that we have observed 40 persons of blood type A, 20 persons of B, 30 persons of O, and 10 persons of AB. To estimate the parameters from these observed data, we then invoke the learning command as follows:[5]

```
?- learn([count(bloodtype(a),40),count(bloodtype(b),20),
          count(bloodtype(o),30),count(bloodtype(ab),10)]).
```

After parameter learning, we may confirm the estimated parameters:

```
?- show_sw.
Switch gene: unfixed: a (0.292329558535712) b (0.163020241540856)
o (0.544650199923432)
```

It can be seen from above and the original meaning given to the program that the frequencies of genes are estimated as: $p_a = 0.292$, $p_b = 0.163$, $p_o = 0.545$. Thus in the context of population genetics, we can say that, inversely with Hardy-Weinberg's law, the hidden frequencies of genes can be estimated from the observed frequencies of phenotypes.

The inheritance model described in this section is considerably simple since we have assumed random mates. However with the expressive power of PRISM, the case of non-random mates can also be written (for example, as done in [31]).

1.3 Utility programs and advanced probabilistic inferences

Furthermore, let us consider a PRISM version of a hidden Markov model (HMM) [4, 26]. HMMs not only dominate in speech recognition but are also well-known as suited for many tasks such as part-of-speech tagging in natural language processing or biological sequence analysis. An HMM is a probabilistic finite automaton where the state transitions and the symbol emissions are all probabilistic.

Let us consider a two-state HMM in Figure 1.1. The HMM has the states `s0` and `s1`, and it emits a symbol `a` or `b` at each state. Each of state transitions and symbol emissions is probabilistic, and conditioned only on the current state. It is assumed in HMMs that we can only observe a string (i.e. a sequence of emitted symbols), not the sequence of state transitions. The program is described as follows:

```
target(hmm/1).           % hmm(L) is observable
values(init,[s0,s1]).    % Switch for state initialization
values(out(_),[a,b]).    %            symbol emission
values(tr(_),[s0,s1]).   %            state transition

hmm(L):-                 % To observe a string L:
   str_length(N),        %   Get the string length as N
   msw(init,S),          %   Choose an initial state randomly
   hmm(1,N,S,L).         %   Start stochastic transition (loop)

hmm(T,N,_,[]):- T>N,!.   % Stop the loop
```

[5] Actually in PRISM, at the query prompt, we cannot make a new line until reaching the end of the query. For readability, in this manual's illustrations, the text typed by the user or displayed by the system is sometimes beautified by the authors.

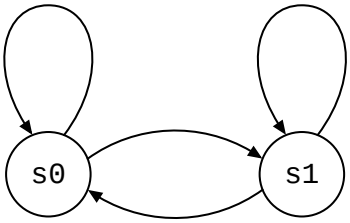

Figure 1.1: State transition diagram of a 2-state hidden Markov model.

```
hmm(T,N,S,[Ob|Y]) :-     % Loop: the state is S at time T
   msw(out(S),Ob),        %   Output Ob at the state S
   msw(tr(S),Next),       %   Transit from S to Next.
   T1 is T+1,             %   Count up time
   hmm(T1,N,Next,Y).      %   Go next (recursion)

str_length(10).           % String length is 10
```

Please note the comments in the program, each states a procedural reading of the corresponding predicate call. Then we may find that a top-down execution from `hmm(L)`, which represents the distribution for a string `L`, simulates a generation process that yields `L`, or in other words, that we observe `L` after a chain of probabilistic choices by switches. In this sense, it is possible to say that the program forms a *generative model*. Besides, it may be noticed that we are also in a partially observing situation for HMMs, since the information about state is hidden from the string `L` in `hmm(L)`.

In this manual, the code shown above is called the *modeling part* of the program, and on the other hand, we can also write non-probabilistic clauses (i.e. usual Prolog clauses) as the *utility part*. For example, we define the two predicates `hmm_learn/1` and `set_params/0`, where the former is a batch predicate for learning, and the latter is the former's subroutine that sets some particular values to parameters at once.

```
hmm_learn(N):-
   set_params,!,               % Set parameters manually
   get_samples(N,hmm(_),Gs),!, % Get N samples
   learn(Gs).                  % learn with these samples

set_params :-
   set_sw(init,   [0.9,0.1]),
   set_sw(tr(s0), [0.2,0.8]),
   set_sw(tr(s1), [0.8,0.2]),
   set_sw(out(s0),[0.5,0.5]),
   set_sw(out(s1),[0.6,0.4]).
```

`get_samples/3`,[6] `learn/1` and `set_sw/2` are the built-ins provided by the system, which run the predicates in the modeling part (at meta-level), or change the status of the system including parameter values. The built-ins except `msw/2` are non-probabilistic, and hence all predicates in the utility part above are also non-probabilistic. Programming with built-ins in the utility part allows users to take a variety of ways of experiments according to the application. For example, in the HMM program, we may add clauses to carry out tasks such as aligning and scoring sequences.

In the literature of applications with HMMs, several efficient algorithms are well-known. One of these algorithms is the Viterbi algorithm [26], which computes the most probable sequence of (hidden) state

[6] `get_samples(`*N*`,`*G*`,`*Goals*`)` generates *N* samples as *Goals* by invoking `sample(`*G*`)` for *N* times.

transitions given a string. This is done by *dynamic programming*, and the computation time is known to be linear in the length of the given string. The programming system provides a built-in for the Viterbi algorithm, which is a generalization of the one for HMMs. For example, `viterbif/1` writes the most probable sequence to the output:

```
?- viterbif(hmm([a,a,a,a,a,b,b,b,b,b])).

hmm([a,a,a,a,a,b,b,b,b,b])
     <= hmm(1,10,s0,[a,a,a,a,a,b,b,b,b,b]) & msw(init,s0)
hmm(1,10,s0,[a,a,a,a,a,b,b,b,b,b])
     <= hmm(2,10,s1,[a,a,a,a,b,b,b,b,b]) & msw(out(s0),a) & msw(tr(s0),s1)
hmm(2,10,s1,[a,a,a,a,b,b,b,b,b])
     <= hmm(3,10,s0,[a,a,a,b,b,b,b,b]) & msw(out(s1),a) & msw(tr(s1),s0)

  ...omitted...

hmm(10,10,s1,[b])
     <= hmm(11,10,s0,[]) & msw(out(s1),b) & msw(tr(s1),s0)
hmm(11,10,s0,[])

Viterbi_P = 0.000117528
```

We then read from here that the most probable sequence is: $\texttt{s0} \to \texttt{s1} \to \cdots \to \texttt{s1} \to \texttt{s0}$ (though the last transition may be redundant).

It is shown that the algorithm implemented as the system's built-in works as efficient as the one specialized for HMMs [34]. So we can handle moderately large datasets with PRISM. The efficiency comes from linear tabling [42], a tabling mechanism provided by B-Prolog, and an EM algorithm called the *graphical EM algorithm*. A similar mechanism is adopted for learning and probability computation mentioned above, which is also a generalization of the Baum-Welch algorithm (also known as the forward-backward algorithm) and the backward probability computation for HMMs respectively [18, 34, 35].

1.4 Handling failures in the generation process*

To realize efficient computation described in the previous section, we need to write PRISM programs which obey some restrictions. The first major one is the *exclusiveness condition*, in which all disjunctive paths in a proof tree are required to be probabilistically exclusive. The second one is the *uniqueness condition*, in which all observable goal patterns are probabilistically exclusive to each other and the sum of their probabilities needs to be unity. For parameter learning, this condition can be relaxed by assuming the *missing-at-random (MAR) condition* [35], and with the MAR condition, there is a case that we can handle the PRISM programs in which the sum of probabilities of observable patterns can exceed unity. On the other hand, the lack of probability mass with failure in the generation process (in which the sum of probabilities becomes less than one) is more serious. The uniqueness condition implies that *for every observable pattern, its generation process never fails*, and could be a strong restriction in our modeling. Recently, for a remedy of this, the programming system introduced a new graphical EM algorithm that takes such failures into account [36, 37, 38]. This algorithm is based both on Cussens's FAM (failure-adjusted maximization) algorithm [12] and FOC (First Order Compiler) [29]. With this new learning framework, we are able to introduce some *constraints* (which causes some failures) to generative models.

1.5 Bayesian approaches in PRISM*

When the observed data is not so large compared to the complexity of the model (i.e. the number of parameters), there should be a risk to rely on the parameters estimated from such data. For example, let us consider that we just have a data set on blood types of 10 persons, in which only the persons of blood type B and O are recorded. Even in such a situation, it seems inappropriate to conclude that gene A does not exist at all. In such a situation, we may take a Bayesian approach to combine our prior knowledge (bias) with the statistics from the data in a principled way.

In Bayesian approaches, we first consider a *prior distribution* $P(\theta)$ on parameters θ. In PRISM, a *Dirichlet distribution* is used as the built-in prior distribution. Since version 1.11, for each switch instance `msw(i,v)`, the *hyperparameter* $\alpha_{i,v}$ of the Dirichlet distribution can be specified through a value called the *pseudo count* $\delta_{i,v} = (\alpha_{i,v} - 1)$. Then, the programming system provides two types of facilities for Bayesian learning. One is for *MAP (maximum a posteriori) estimation*, and the other for *variational Bayesian learning*.

In MAP estimation, pseudo counts work as the statistics on what we have not actually observed. That is, in estimating a parameter $\theta_{i,v}$, the probability of a switch instance `msw(i,v)` being true, we perform $\hat{\theta}_{i,v} = (C_{i,v} + \delta_{i,v})/(\sum_{v' \in V_i}(C_{i,v'} + \delta_{i,v'}))$, where $C_{i,v}$ is the (expected) occurrences of the switch instance `msw(i,v)` in the data, and V_i is the set of possible outcomes of the switch named i. When the pseudo count $\delta_{i,v} = 0$, this procedure is just that for ML estimation (i.e. $\hat{\theta}_{i,v} = C_{i,v} / \sum_{v' \in V_i} C_{i,v'}$). When configuring $\delta_{i,v}$ to be positive, on the other hand, we can avoid the estimated parameter $\hat{\theta}_{i,v}$ being zero, and hence can relieve the problem of data sparseness to some extent. In the above example, we can assign a positive probability to the chance that gene A exists. Generally speaking, MAP estimation is a procedure to obtain the parameters that maximizes a posteriori probability $P(\theta \mid D, M) \propto P(D \mid M, \theta)P(\theta)$, where D is the observed data, i.e. a multiset of observed goals $G_1, G_2, \ldots, G_T$, and M is the model written as a PRISM program.

It is often said, on the other hand, that variational Bayesian (VB) learning has high robustness against data sparseness in model selection and prediction (Viterbi computation). This is because VB learning gives us an a posteriori distribution $P^*(\theta \mid D)$ and we can make inferences based on some averaged quantities with respect to $P^*(\theta \mid D)$, instead of particular point-estimated parameters.

Now let us run the blood type program with the facilities above. To set pseudo counts (hyperparameters), we may add the query below to the program:

```
:- set_prism_flag(default_sw_h,1.0).
```

The programming system provides dozens of *execution flags* to allow the users to change the behaviors of the built-in predicates. The query above will set a value `1.0` to the flag named '`default_sw_h`'. Under this setting, when the system tries to register a new switch (`gene`, in this case) to the internal database, its hyperparameters will be all set to 1.0. The suffix '`_h`' of the flag name means 'for hyperparameters.' Then, let us learn the parameters from the data in which 4 persons of blood type B and 6 persons of blood type O are recorded:

```
?- prism(bloodABO).
  :
?- learn([count(bloodtype(b),4),count(bloodtype(o),6)]).

#goals: 0(2)
Exporting switch information to the EM routine ...
#em-iterations: 0(4) (Converged: -6.730116931)
Statistics on learning:
        Graph size: 12
        Number of switches: 1
        Number of switch instances: 3
```

```
        Number of iterations: 4
        Final log likelihood: -6.730116931
        Total learning time: 0.012 seconds
        Explanation search time: 0.000 seconds
        Total table space used: 2232 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
```

After learning, we can confirm that a positive probability is assigned to the parameter of `msw(gene,a)`, and that the common pseudo count 1.0 are surely set to each switch:

```
?- show_sw_b.

Switch gene: unfixed_p,unfixed_h: a (p: 0.000000000, c: 1.000000000)
b (p: 0.225475582, c: 1.000000000) o (p: 0.774524418, c: 1.000000000)

yes
```

The suffix '`_b`' of the built-in predicate `show_sw_b/0` means 'for both parameters and pseudo counts (hyperparameters).' On the other hand, we can assign the pseudo counts by manual:

```
?- set_sw_h(gene,[0.5,1.0,1.0]).
  :
?- show_sw_b.

Switch gene: unfixed_p,unfixed_h: a (p: 0.000000000, c: 0.500000000)
b (p: 0.225475582, c: 1.000000000) o (p: 0.774524418, c: 1.000000000)

yes
```

VB learning is easily conducted by setting '`hparams`' to the execution flag named '`learn_mode`' and then invoking the usual learning command (note that there is *no* need to modify the modeling part):

```
?- set_prism_flag(learn_mode,hparams).
  :
?- learn([count(bloodtype(b),4),count(bloodtype(o),6)]).

#goals: 0(2)
Exporting switch information to the EM routine ...
#vbem-iterations: 0(4) (Converged: -10.758982897)
Statistics on learning:
        Graph size: 12
        Number of switches: 1
        Number of switch instances: 3
        Number of iterations: 4
        Final variational free energy: -10.758982897
        Total learning time: 0.016 seconds
        Explanation search time: 0.004 seconds
        Total table space used: 2232 bytes
Type show_sw_h or show_sw_b to show the probability distributions.

yes
```

We can see that the pseudo counts have been adjusted based on the given data. This implies that now we have the a posteriori distribution $P^*(\theta \mid D)$.

```
?- show_sw_b.

Switch gene: unfixed_p,unfixed_h: a (p: 0.000000000, c: 0.526767219)
b (p: 0.225475582, c: 5.645958140) o (p: 0.774524418, c: 16.425913692)
```

Similarly to parameter learning, Viterbi computation based on the a posteriori distribution $P^*(\theta \mid D)$ can be invoked with a setting for the execution flag '`viterbi_mode`'. For the HMM program, we may run the following after VB learning:

```
?- set_prism_flag(viterbi_mode,hparams).
  :
?- viterbif(hmm([a,a,a,a,a,b,b,b,b,b])).
```

1.6 Parallel EM learning*

Since version 1.9, a system command named `upprism` is provided for *batch execution* (or non-interactive execution) of PRISM programs. For a batch execution, we first write what we would like to execute in the clause body of `prism_main/0-1`. In the HMM program, for example, we may run `hmm_learn(100)`, which means to conduct EM learning with 100 observed goals (§1.3), in a batch execution:

```
prism_main:- hmm_learn(100).
```

Then, the batch execution can be started by running `upprism` (recall that the file name of the HMM program is '`hmm.psm`'):

```
% upprism hmm
  :
#goals: 0.........(95)
Exporting switch information to the EM routine ...
#em-iterations: 0.........100.........200.........300.........400...
......500.........600......(662) (Converged: -689.817528678)
Statistics on learning:
        Graph size: 5744
        Number of switches: 5
        Number of switch instances: 10
        Number of iterations: 662
        Final log likelihood: -689.817528678
        Total learning time: 0.132 seconds
        Explanation search time: 0.012 seconds
        Total table space used: 378896 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
```

Furthermore, since version 1.11, a utility for parallel EM learning is available. Namely, a command named `mpprism` (multi-process PRISM) is used instead of `upprism` (uni-process PRISM). Under some additional settings for a parallel computing environment (§6.2), we can run `mpprism` similarly to `upprism`. For example, we learn the HMM program from 100 observed goals in a data-parallel fashion by four processes:

```
% mpprism hmm
  :
loading::hmm.psm.out
```

```
loading::hmm.psm.out
loading::hmm.psm.out
loading::hmm.psm.out
#goals: 0.........(94)
Exporting switch information to the EM routine ...
Exporting switch information to the EM routine ...
Exporting switch information to the EM routine ...
Exporting switch information to the EM routine ...
#em-iterations: 0.........100.........200.........300.........400......
...500.........600.........700.........800.........900.........1000....
.....1100.........1200....(1250) (Converged: -680.941522532)
Statistics on learning:
        Graph size: 7288
        Number of switches: 5
        Number of switch instances: 10
        Number of iterations: 1250
        Final log likelihood: -680.941522532
        Total learning time: 0.449 seconds
        Explanation search time: 0.008 seconds
Type show_sw or show_sw_b to show the probability distributions.

yes
```

Chapter 2

PRISM Programs

Generally speaking, a probabilistic model represents some probability distribution which the probabilistic phenomena in the application domain are assumed to follow, and PRISM is a logic-based representation language for such probabilistic models. In this chapter, we describe the detail of the PRISM language, and the basic mechanism of the related algorithms provided as built-in predicates.

2.1 Overall organization

Let us first define that a *probabilistic predicate* is a predicate which eventually calls (at non-meta level) the built-in probabilistic predicate `msw/2`, i.e. random switches. Then we roughly classify the clauses in a PRISM program into the following three parts:

- *Modeling part*: the definitions of all probabilistic predicates, and of some non-probabilistic predicates which are called from probabilistic predicates. This part corresponds to the definition of the model.
- *Utility part*: the remaining definitions of non-probabilistic predicates. This part is a usual Prolog program that utilizes the model, and often that can be seen as a *meta program* of the modeling part.
- *Declarations*: the clauses of some particular built-in predicates which contain additional information on the model (of course, they are non-probabilistic).

In the rest of this chapter, we first describe the basic semantics of PRISM programs and the currently available probabilistic inferences. Then we proceed to describe the details of each part.

2.2 Basic semantics

PRISM is designed based on the distribution semantics [30, 35], a probabilistic extension of the least model semantics. In the distribution semantics, all ground atoms are considered as random variables taking on 1 (true) or 0 (false). With this semantics and the predefined probabilistic property of random switches, we can give a declarative semantics to programs. However, in the recent versions including 1.11, to make an efficient implementation of tabling, we use a different specification from the original one [33, 35] of random switches, in which some procedural notion is required. Here we describe `msw/2` as follows:

1. For each ground term i in `msw(`i`,`v`)` which is possible to appear in the program, a set of ground terms V_i should be given by the user with multi-valued switch declaration, and also $v \in V_i$ should

hold. Such an `msw(i,v)` is hereafter called a *switch instance*, where i is the *switch name*, v the *outcome* or the *value*, and V_i the *outcome space* of i. A collection of `msw(i,·)` forms *switch* i.

2. For a switch i, whose outcome space is $V_i = \{v_1, \ldots, v_k\}$ ($k \geq 1$), one of the ground atoms `msw(i,v1)`, ..., `msw(i,vk)` is exclusively true at the same position of a proof tree, and $\sum_{v \in V_i} \theta_{i,v} = 1$ holds, where $\theta_{i,v}$ is the probability of `msw(i,v)` being true and is called a *parameter* of the program. Intuitively, a logical variable `V` in a predicate call of `msw(i,V)` behaves as a random variable which takes a value v from V_i with the probability $\theta_{i,v}$.

3. The truth-values of switch instances at the different positions of a proof tree are independently assigned. This means that the predicate calls of `msw/2` behave independently of each other.

Hereafter, for understanding the third condition, it would be a help to introduce IDs which identify positions in the proof tree,[1] and then to associate each occurrence of switch instance with the ID of the corresponding position. Then the switches at different positions will be syntactically different. The third condition is referred to as the *independence condition*.

The probabilistic meaning of the modeling part can be understood in a bottom-up manner.[2] Now, for illustration, let us pick up again the blood type program:

```
bloodtype(P) :-
    genotype(X,Y),
    ( X=Y -> P=X
    ; X=o -> P=Y
    ; Y=o -> P=X
    ; P=ab
    ).

genotype(X,Y) :- msw(gene,X),msw(gene,Y).

values(gene,[a,b,o]).
```

First, one of `msw(gene,X)` instantiated with `X = a`, `X = b` or `X = o` (i.e. a random pick-up of a gene `X` from the pool) becomes exclusively true, according to the probabilistic property of switches described above. Then we associate the parameters of switches with gene frequencies, i.e. $\theta_{\texttt{gene,a}} = p_a$, $\theta_{\texttt{gene,b}} = p_b$ and $\theta_{\texttt{gene,o}} = p_o$. Also in view of the independence of switches at different occurrences, the definition of `genotype/2` satisfies the random-mate assumption on genotypes, hence the probability of each is a product of two gene frequencies. In the body of `bloodtype/1`'s definition, one of `genotype(X,Y)` with `X = a,b` and `o`, and `Y = a,b` and `o` becomes exclusive, and hence the different instances of the clause body become exclusively true. We can also see the second conjunct makes a correct many-to-one mapping from genotypes to phenotypes. Therefore we can say that one of `bloodtype(P)` with `P = a`, `P = b`, `P = o` and `P = ab` becomes exclusively true with probability P_A, P_B, P_O, and P_{AB}, respectively. In addition, from the exclusiveness discussed above, each of logical variables `X` and `Y` in `genotype(X,Y)` behaves just like a random variable that takes a gene as its value, whereas `P` in `bloodtype(P)` behaves like a random variable that takes a phenotype.

In PRISM, it would be easier, and so is recommended, to write a program in a top-down (consequently, a generative) manner. On the other hand, sometimes it is also crucial to inspect the program's probabilistic meaning in a bottom-up manner, as shown above.

[1] In old SICStus Prolog versions, PRISM uses `msw(i,n,v)` where the users need to explicitly specify n, the ID of an independent choice by the switch. This definition is important to give a declarative semantics to programs, and hence the theoretical papers on PRISM still use `msw/3`.

[2] The discussion in this section should be considerably rough. For the readers interested in the formal semantics of PRISM (called the *distribution semantics*), please consult [30, 35].

2.3 Probabilistic inferences

Before proceeding to the further details of the PRISM language, it would be worth listing what we can do with this language. First let $P_\theta(\cdot)$ be the probability distribution specified by the program, under the parameters θ of switches buried in the program. Then, in the PRISM programming system, the following five types of probabilistic inferences are available:

Sampling (§4.3):
Given a goal G of a probabilistic predicate, return the answer substitution σ with the probability $P_\theta(G\sigma)$, or fail with the probability that $\exists G$ is false.

Probability calculation (§4.4):
Given a goal G of a probabilistic predicate, compute $P_\theta(G)$.

Viterbi computation (§4.6):
Given a goal G of a probabilistic predicate, find $E^* = \mathrm{argmax}_{E \in \{E_1,\ldots,E_K\}} P_\theta(E)$, where $E_1, \ldots, E_K$ are the explanations for G such that $G \Leftrightarrow E_1 \vee \cdots \vee E_K$ and each E_k is a conjunction of switch instances.

Hindsight computation (§4.7):
Given a goal G of a probabilistic predicate, compute $P_\theta(G')$ or $P_\theta(G' \mid G)$ for each subgoal G' of G.

Parameter learning (§4.8):
Given a bag of observed goals $\{G_1, G_2, \ldots, G_T\}$ of probabilistic predicates (i.e. training data), get the parameters θ of switches which maximizes the likelihood $\prod_t P_\theta(G_t)$.

The first inference task works with an execution style called the *sampling execution* (§2.4.1), and the rest utilize the *explanation search* (§2.4.2). For HMMs, the former execution style simulates the behavior of an HMM as a string generator (i.e. data sampler), and the latter simulates the behavior as an acceptor. For more details including their variations, please visit the corresponding sections.

2.4 Modeling part

We have seen a couple of examples of the modeling part (sections in Chapter 1 and §2.2). One interesting feature of PRISM is that we can (or we should) write models as *executable*. For various probabilistic inferences, there are two underlying execution styles called *sampling execution* and *explanation search*. So it is expected for users to write the modeling part so that it can work in these two execution styles. Besides, as far as we understand these two execution styles, it is allowed to write disjunctions (‘`;`’), the cut symbols (‘`!`’), or the if-then (‘`->`’) statements in a clause body.

In addition, for efficient execution of models, the system assumes that the model follows several conditions.[3] However, it is often difficult for the system to check these conditions, and hence it is required to write carefully programs to satisfy the conditions (otherwise some unexpected behavior arises).

In the rest of this section, we first explore two underlying execution styles for these inferences, and then make some advanced discussions concerning to parameter learning. Finally we summarize the conditions on the modeling part to be satisfied.

[3] For the theoretical details, please see [35].

2.4.1 Sampling execution

Sampling execution is the underlying execution style for a sampling task (§2.3, §4.3). In the literature of Bayesian networks, this style is sometimes called *forward sampling*. In the recent versions including 1.11, sampling execution becomes easier to understand. That is, the system only makes a top-down execution like Prolog, and determines the value v of `msw(`i`,`v`)` on the fly according to the parameters $\{\theta_{i,v}\}$. A sampling execution of probabilistic goal[4] G is invoked by:[5]

```
?- sample(G).
```

Internally, `msw/2` for sampling execution is essentially defined as follows:[6]

```
msw(I,V):-
    values(I,Values),!,
    $get_probs(I,Probs),
    $choose(Values,Probs,V).
```

In the definition above, `values(`*I*`,`*Values*`)` is declared as a multi-valued switch declaration by the user, and *I* should be a *ground* term. Then *Values*, a list of *ground* terms, will be returned based on the declaration. On the other hand, `$get_probs(`*I*`,`*Probs*`)` returns *Probs* which is a list of switch *I*'s parameters, and `$choose(`*Values*`,`*Probs*`,`*V*`)` returns *V* randomly from *Values* according to the probabilities *Probs*. Also note that `$get_probs/2` and `$choose/3` are not backtrackable.

One typical trap in sampling execution is the independence among switches. In the previous papers, the authors often use a blood type program similar to the one below, instead of the one illustrated in this manual:

```
bloodtype(a) :- (genotype(a,a) ; genotype(a,o) ; genotype(o,a)).
bloodtype(b) :- (genotype(b,b) ; genotype(b,o) ; genotype(o,b)).
bloodtype(o) :- genotype(o,o).
bloodtype(ab):- (genotype(a,b) ; genotype(b,a)).

genotype(X,Y) :- msw(gene,X),msw(gene,Y).

values(gene,[a,b,o]).
```

With this program, the following query for sampling execution sometimes fails:

```
?- sample(bloodtype(X)).
```

This is because there is a case that all predicate calls `genotype(a,a)`, `genotype(a,o)`, ..., and `genotype(b,a)` in the `bloodtype/1`'s definition independently fail, without sharing the results of sampling `msw/2`. The difference between the program above and the blood type programs in the previous papers is the use of `msw/3`, which can share the sampling results by referring to their second arguments. For sampling execution with `msw/2`, we need to write a program in a purely generative manner: once we get a result of a switch sampling, the result should be passed through the predicate arguments to the predicate which requires it as input.

---

[4] A probabilistic goal is a goal whose predicate is probabilistic.

[5] For ease of programming, it is also allowed to run *G* directly just like Prolog:

```
?- G.
```

[6] Note that the predicates in the clause body are introduced for illustration — in the actual implementation, they are more complicatedly defined with different predicate names. On the other hand, as described in §2.6.3, `values/2` is just treated as a unit clause which can work in the other part of the user program.

2.4.2 Explanation search

Explanation search works as an underlying subroutine of built-in predicates for probabilistic inference such as probability calculation (§4.4), Viterbi computation (§4.6), hindsight computation (§4.7) and parameter learning (§4.8).[7] To simulate only explanation search, we can use the built-ins `probf/1-2` (§4.5). In this section, we describe the explanation search by defining several terminologies.

First, in PRISM, an *explanation* for probabilistic goal G is a conjunction E of the ground switch instances, which occurs in a derivation path of a sampling execution for G. In the blood type program, for example, one possible explanation of goal `bloodtype(a)` is:

$$\texttt{msw(gene,a)} \wedge \texttt{msw(gene,a)}.$$

(if we know a person's blood type is A, one possibility is that he inherits two A genes from both parents.) This corresponds to a phenomenon that we will get `bloodtype(a)` as a solution of a sampling execution of `bloodtype(X)` by having `msw(gene,a)` twice. Each of two `msw(gene,a)`s above indicates an individual gene inheritance from one of the parents, so they should not be suppressed (see the discussion in §2.2).

Basically we can write the modeling part with keeping in mind that an explanation search finds all possible explanations for a given goal by a *failure-driven loop* [40]. For `bloodtype(a)`, we have three explanations:

$$\begin{aligned}&\texttt{msw(gene,a)} \wedge \texttt{msw(gene,a)},\\ &\texttt{msw(gene,a)} \wedge \texttt{msw(gene,o)},\\ &\texttt{msw(gene,o)} \wedge \texttt{msw(gene,a)}.\end{aligned}$$

Also please note here that the last two explanations correspond to different derivation paths, and so should not be suppressed. To be more specific, as mentioned in §2.2, this would be understood that, by associating switches with IDs of the positions in the proof tree, they are probabilistically exclusive. In PRISM, for the explanations $E_1, E_2, \ldots, E_k$ for a goal G, we assume that k is finite (the *finiteness condition*), and $G \Leftrightarrow E_1 \vee E_2 \vee \ldots \vee E_k$.

In a probabilistic context, an explanation E is a conjunction of independent switch instances, and hence the probability of E is the product of the probabilities of switch instances in E. Also, if we assume that possible explanations for any goal are all exclusive (i.e. the program satisfies the *exclusiveness condition*), the probability of a probabilistic goal G is the sum of probabilities of the explanations for G. For some probabilistic inference or learning given a goal G, the system makes an explanation search for G in advance of numeric computations.

Unfortunately, it is easily seen that in general, the number of explanations for a goal can be *exponential* depending on the complexity of the model or the given goal (input). To compress these explanations and make them manageable, the system adopts *tabling*, or more specifically *linear tabling* [42], for explanation search. In tabling, every solution of a predicate call is stored in the *solution table*, and once we have all solutions for the predicate call, the stored solutions are used for the later calls. After the explanation search by tabling, the stored solutions are converted to a data structure called *explanation graphs*, and then the system performs probabilistic computation on these graphs. Furthermore, explanation graphs can be seen as AND/OR graphs consisting of propositional (i.e. ground or existentially quantified) formulas, and tabling itself can be understood as a kind of *propositionalization* procedure in that it receives first-order expressions (i.e. a PRISM program) and observed goals as input, and generates as output propositional AND/OR graphs that explain observed goals.

For example, let us consider the HMM program in §1.3, with the string length being changed to 3. In this program, we have the following 16 explanations[8] for G = `hmm([a,b,b])`:

[7] The summary of these inferences is given in §2.3

[8] Our HMM program can be said as redundant since we distinguish the explanations by the last state transition which do not contribute to the final output. A more optimized one should have only 8 (= 2^3) explanations.

$$
\begin{aligned}
E_1 &= \texttt{msw(init,s0)} \wedge \texttt{msw(out(s0),a)} \wedge \texttt{msw(tr(s0),s0)} \wedge \\
&\quad \texttt{msw(out(s0),b)} \wedge \texttt{msw(tr(s0),s0)} \wedge \texttt{msw(out(s0),b)} \wedge \texttt{msw(tr(s0),s0)}, \\
E_2 &= \texttt{msw(init,s0)} \wedge \texttt{msw(out(s0),a)} \wedge \texttt{msw(tr(s0),s0)} \wedge \\
&\quad \texttt{msw(out(s0),b)} \wedge \texttt{msw(tr(s0),s0)} \wedge \texttt{msw(out(s0),b)} \wedge \texttt{msw(tr(s0),s1)}, \\
&\vdots \\
E_{16} &= \texttt{msw(init,s1)} \wedge \texttt{msw(out(s1),a)} \wedge \texttt{msw(tr(s1),s1)} \wedge \\
&\quad \texttt{msw(out(s1),b)} \wedge \texttt{msw(tr(s1),s1)} \wedge \texttt{msw(out(s1),b)} \wedge \texttt{msw(tr(s1),s1)}.
\end{aligned}
$$

Then we have $G \Leftrightarrow E_1 \vee E_2 \vee \cdots \vee E_{16}$, and this iff-formula can be converted to a conjunction of iff-formulas below, which can be derived from Clark's completion [7] constructed from the definitions of probabilistic predicates.

$$
\begin{aligned}
\texttt{hmm([a,b,b])} &\Leftrightarrow (\texttt{msw(init,s0)} \wedge \texttt{hmm(1,3,s0,[a,b,b])}) \\
&\quad \vee\ (\texttt{msw(init,s1)} \wedge \texttt{hmm(1,3,s1,[a,b,b])}) \\
\texttt{hmm(1,3,s0,[a,b,b])} &\Leftrightarrow (\texttt{msw(out(s0),a)} \wedge \texttt{msw(tr(s0),s0)} \wedge \texttt{hmm(2,3,s0,[b,b])}) \\
&\quad \vee\ (\texttt{msw(tr(s0),s1)} \wedge \texttt{msw(out(s0),a)} \wedge \texttt{hmm(2,3,s1,[b,b])}) \\
\texttt{hmm(1,3,s1,[a,b,b])} &\Leftrightarrow (\texttt{msw(out(s1),a)} \wedge \texttt{msw(tr(s1),s0)} \wedge \texttt{hmm(2,3,s0,[b,b])}) \\
&\quad \vee\ (\texttt{msw(out(s1),a)} \wedge \texttt{msw(tr(s1),s1)} \wedge \texttt{hmm(2,3,s1,[b,b])}) \\
\texttt{hmm(2,3,s0,[b,b])} &\Leftrightarrow (\texttt{msw(tr(s0),s0)} \wedge \texttt{msw(out(s0),b)} \wedge \texttt{hmm(3,3,s0,[b])}) \\
&\quad \vee\ (\texttt{msw(out(s0),b)} \wedge \texttt{msw(tr(s0),s1)} \wedge \texttt{hmm(3,3,s1,[b])}) \\
&\vdots \\
\texttt{hmm(3,3,s1,[b])} &\Leftrightarrow (\texttt{msw(out(s1),b)} \wedge \texttt{msw(tr(s1),s0)}) \\
&\quad \vee\ (\texttt{msw(out(s1),b)} \wedge \texttt{msw(tr(s1),s1)})
\end{aligned}
$$

In this converted iff-formula, the ground atoms appearing on the left hand side are called *subgoals*. Each conjunction on the right hand side of each iff-formula whose left hand side is G' is called a *sub-explanation* for G'. It is easy to see that a sub-explanation includes subgoals as well as switch instances, and that G' depends on the subgoals appearing in the sub-explanations for G'. It should be noticed that, to make an exact probability computation by dynamic programming possible, the system assumes that these dependencies cannot form a cycle. This condition is hereafter called the *acyclicity condition*. Assuming this condition, we treat the converted iff-formulas as *ordered*.

As mentioned above, in explanation search, the system tries to find all possible explanations. With tabling, each subgoal solved in the search process is stored into a table, together with its sub-explanation, and after the search terminates, the explanation graphs are constructed from the stored information. Finally the routines for probabilistic inference including learning works on the explanation graphs. The structure of explanation graphs are isomorphic to the ordered iff-formula described above. Some may notice that a subgoal `hmm(2,3,s0,[b,b])` is found in both sub-explanations for `hmm(1,3,s0,[a,b,b])` and `hmm(1,3,s1,[a,b,b])`. In this data structure, a substructure can be shared by the upper substructures to avoid redundant computations. In other words, we can enjoy the efficiency which comes from *dynamic programming*. The programming system provides the built-in `probf/2` (§4.5) to get an explanation graph as a Prolog term.

Besides, at a more detailed level, we have a different definition of `msw/2` for explanation search:[9]

```
msw(I,V):- values(I,Values),!,member(V,Values).
```

Note again that it is assumed that, in a predicate call of `values(I,Values)`, `I` is a ground term. One may find that there are no probabilistic predicates in the body that work at random. This is because the explanation search only aims to enumerate all possibilities that a given goal holds, and it requires no probabilistic consideration.

2.4.3 Additional notes on writing the modeling part

◇ Writing the modeling part in two styles

It is crucial to notice that the blood type program shown in §2.4.1 (not the one shown in §1.2) can work for explanation search, while it does not for sampling execution. It would be fine for the modeling part to work both for sampling execution and explanation search, but if it is difficult or inefficient, we need to write the modeling part in two styles — one is for sampling execution, and the other for explanation search. The declarations except the multi-valued switch declarations are made with respect to the modeling part for explanation search.

◇ Representing dependent choices by independent random switches

In §2.2, it is mentioned that the random switches appearing at different positions in the proof tree behave independently of each other. On the other hand, some may wonder how we can make the next choice conditioned on the previous choice(s). To consider about this question, let us consider again the HMM program picked up in §1.3:

```
target(hmm/1).           % hmm(L) is observable
values(init,[s0,s1]).    % Switch for state initialization
values(out(_),[a,b]).    %            symbol emission
values(tr(_),[s0,s1]).   %            state transition

hmm(L):-                 % To observe a string L:
   str_length(N),        %   Get the string length as N
   msw(init,S),          %   Choose an initial state randomly
   hmm(1,N,S,L).         %   Start stochastic transition (loop)

hmm(T,N,_,[]):- T>N,!.   % Stop the loop
hmm(T,N,S,[Ob|Y]) :-     % Loop: the state is S at time T
   msw(out(S),Ob),       %   Output Ob at the state S
   msw(tr(S),Next),      %   Transit from S to Next.
   T1 is T+1,            %   Count up time
   hmm(T1,N,Next,Y).     %   Go next (recursion)

str_length(10).          % String length is 10
```

Then, we get a trace of sampling execution (§2.4.1) of `hmm(L)` as shown in Figure 2.1 (see §3.6 for the usage of the trace mode). From this trace and the definition of `hmm/4`, it can be seen that, in the first recursive call of `hmm/4`, we use random switches `out(`S`)` and `tr(`S`)` where the current state S is chosen by the switch `init`. Also in the T-th recursive call ($T > 2$), random switches `out(`S`)` and `tr(`S`)` are used, where S is chosen by the switch `tr(`S'`)` used in the $(T-1)$-th recursive call. That is, one may notice that, in the first recursive call of `hmm/4` (beginning from Line 14 in Figure 2.1), we

[9] Note that the predicate name of `msw/2` is different from the one in the actual implementation.

```
1  ?- prism([consult],hmm).
2    :
3  ?- trace.
4    :
5  {Trace mode}
6  ?- sample(hmm(L)).
7
8  Call:  (0) sample(hmm(_c60)) ?
9  Call:  (2) hmm(_c60) ?
10   Call:  (3) str_length(_d20) ?
11   Exit:  (3) str_length(10) ?
12   Call:  (4) msw(init,_d3c):_e34 ?
13   Exit:  (4) msw(init,s1):0.5 ?        ··· switch init takes a value s1
14   Call:  (7) hmm(1,10,s1,_c60) ?       ··· first recursive call of hmm/4
15     Call:  (8) 1>10 ?
16     Fail:  (8) 1>10 ?
17     Call:  (9) msw(out(s1),_f24):_1060 ?
18     Exit:  (9) msw(out(s1),a):0.5 ?      ··· switch out(s1) takes a value a
19     Call:  (12) msw(tr(s1),_f6c):_11bc ?
20     Exit:  (12) msw(tr(s1),s0):0.5 ?     ··· switch tr(s1) takes a value s0
21     Call:  (15) _f88 is 1+1 ?
22     Exit:  (15) 2 is 1+1 ?
23     Call:  (16) hmm(2,10,s0,_f28) ?      ··· second recursive call of hmm/4
24       Call:  (17) 2>10 ?
25       Fail:  (17) 2>10 ?
26       Call:  (18) msw(out(s0),_12cc):_1408 ?
27       Exit:  (18) msw(out(s0),b):0.5 ?     ··· switch out(s0) takes a value b
28       Call:  (21) msw(tr(s0),_1314):_1574 ?
29       Exit:  (21) msw(tr(s0),s0):0.5 ?     ··· switch tr(s0) takes a value s0
30       Call:  (24) _1330 is 2+1 ?
31       Exit:  (24) 3 is 2+1 ?
32       Call:  (25) hmm(3,10,s0,_12d0) ?     ··· third recursive call of hmm/4
33         Call:  (26) 3>10 ?
34         Fail:  (26) 3>10 ?
35         Call:  (27) msw(out(s0),_1684):_17c0 ?
36         Exit:  (27) msw(out(s0),a):0.5 ?     ··· switch out(s0) takes a value b
37         Call:  (30) msw(tr(s0),_16cc):_191c ?
38         Exit:  (30) msw(tr(s0),s1):0.5 ?     ··· switch tr(s0) takes a value s0
39         Call:  (33) _16e8 is 3+1 ?
40         Exit:  (33) 4 is 3+1 ?
41         Call:  (34) hmm(4,10,s1,_1688) ?     ··· fourth recursive call of hmm/4
42           Call:  (35) 4>10 ?
43           :
```

Figure 2.1: Trace of a sampling execution of `hmm(L)`.

obtain `s0` as a sampled value of the switch `tr(s1)` (Lines 19–20). Then, in the second recursive call, letting the current state S = `s0`, we use switches `out(s0)` and `tr(s0)`, and get the value `b` and `s0`, respectively (Lines 26–27 and Lines 28–29).

We can say from the above example that, to make a choice C depending on the results R_1, R_2, ..., R_K of previous choices, it is sufficent to use a switch named c`(`r_1`,`r_2`,...,`r_K`)`, where c is a functor name that refers to the choice C and r_k is a ground term that refers to the results R_k ($1 \leq k \leq K$). Of course, the switch name can be an arbitrary ground term, e.g. `choose(`c`,[`r_1`,`r_2`,...,`r_K`])`, as long as it uniquely refers to the choice C that depends on R_1, R_2, ..., R_K. To summarize, in PRISM, it is only allowed to use independent random switches, but we can represent dependent choices by using different random switches according to the context, i.e. the results of some of previous choices.

With keeping this discussion in mind, we can write a Mealy-type HMM,[10] in which each output probability depends on the state transion (i.e. both the current state and the next state), by modifying only

[10] On the other hand, the original HMM program picked up in §1.3 defines a Moore-type HMM, in which each output probability depends only on the current status.

a few clauses:

```
target(hmm/1).
values(init,[s0,s1]).
values(out(_,_),[a,b]).   % modified
values(tr(_),[s0,s1]).

hmm(L):-
   str_length(N),
   msw(init,S),
   hmm(1,N,S,L).

hmm(T,N,_,[]):- T>N,!.
hmm(T,N,S,[Ob|Y]) :-
   msw(tr(S),Next),       % modified
   msw(out(S,Next),Ob),   % modified
   T1 is T+1,
   hmm(T1,N,Next,Y).

str_length(10).
```

Note here that, in the recursive clause of `hmm/4`, the switch `out(S,Next)` should be called after `Next` is determined as a ground term `s0` or `s1` by the switch `tr(S)`. The Bayesian network programs shown in §7.3 are also a typical example.

2.4.4 Handling failures*

As previously mentioned, a PRISM program basically describes a probabilistic generation process of the data at hand. On the other hand, there could be a case where failures may be caused in the process by some constraints. In a probabilistic context, this implies that some probability mass is lost, and hence we cannot directly apply a traditional learning algorithm which assumes the *no-failure condition*, i.e. there is no failure in the generation process. However it is sometimes difficult to write a program without failures. In such a case, the difficulty could be resolved by using a special learning routine.

In usual maximum likelihood (ML) estimation, we try to find the parameters θ that maximize the likelihood $\prod_t P_\theta(G_t)$, the product of probabilities of observed data G_t being generated.[11] Instead of this, we exclude the probability mass which is lost by failures, and try to maximize $\prod_t P_\theta(G_t \mid succ)$, the product of conditional probabilities of observed data being generated under the condition that no failure arises (indicated by *succ*).

To be more specific, let us consider a program which considers the agreement in coin flipping.[12] The modeling part is written as follows:

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

failure :- not(success).
success :- agree(_).

agree(A):-
    msw(coin(a),A),
    msw(coin(b),B),
    A=B.
```

[11] We assume here that the propositional random variables corresponding to the data are independent and identically distributed (i.i.d.).

[12] This program comes from [38].

The predicate `agree(A)` means that two outcomes of flipping two coins meet as `A`, and that we fail to observe any result when they differ. So this program violates the no-failure condition. On the other hand, the predicate `success/0` denotes the event *succ* above since it is equivalent to ∃`X agree(X)`, i.e. we have some observation. PRISM assumes that all possibilities in which a failure arises are denoted by a predefined predicate `failure/0`. In this program, and probably in many cases, `failure/0` is defined as a negation of `success/0`. But in other cases, it is necessary to define `failure/0` explicitly. Under this setting, the target of maximization for the system is rewritten as $\prod_t P_\theta(G_t \mid succ) = \prod_t \{P_\theta(G_t)/P_\theta(succ)\} = \prod_t \{P_\theta(G_t)/(1 - P_\theta(fail))\}$, where *fail* is the event represented by `failure/0`, i.e. some failure arises. The *failure-adjusted maximization (FAM) algorithm* [12] is an EM algorithm that solves this maximization, by considering the number of failures as hidden information.

It is important to notice that `not/1` in the `failure/0`'s definition does not mean *negation as failure* (NAF).[13] We cannot directly simulate this negation, and hence it is eliminated by *First Order Compiler* [29] when the program is loaded.[14] The program above, excluding the declarations by `values/2`, will be compiled as:

```
failure:- closure_success0(f0).
closure_success0(A):- closure_agree0(A).
closure_agree0(_):-
  msw(coin(a),A),
  msw(coin(b),B),
  \+ A=B.
```

where `\+/1` means negation as failure. To enable such a compilation, we use the predicate `prismn/1`, not the usual one (i.e. `prism/1`). Then it is also required to invoke the learning command with adding a special symbol `failure` to the list of observed goals. A detailed description for the usage is given in §4.11, and a program example can be found in §7.5.

2.4.5 Learning from goals with logical variables*

In parameter learning, the system accepts observed goals with (existentially quantified) logical variables. However, we need to be aware that it is justified under the condition called the *missing-at-random (MAR) condition*, which is firstly addressed by Rubin [27]. The discussion made in this section can be generalized to some cases where the sum of probabilities of observable goal patterns exceeds unity, but as a typical case, we will concentrate on the case of observed goals with logical variables.

First, let $\mathcal{G}$ be a set of observable ground atoms, and $\mathcal{G}^+$ be a set of atoms in $\mathcal{G}$ or atoms with existentially quantified logical variables, whose ground instances are in $\mathcal{G}$ (i.e. $\mathcal{G} \subseteq \mathcal{G}^+$). Also let us consider that the uniqueness condition holds with $\mathcal{G}$ (i.e. $\sum_{G \in \mathcal{G}} P_\theta(G) = 1$ for any θ). Furthermore, for explanatory simplicity, we assume here that every atom in $\mathcal{G}$ has a positive probability. For example, in the HMM program with the string length being 2, `hmm([a,b])` is in $\mathcal{G}$, and `hmm([a,X])` in $\mathcal{G}^+$. Here, it is easily seen that there is a many-to-many mapping on ground instantiation from $\mathcal{G}$ to $\mathcal{G}^+$, and hence the sum of probabilities of goals in $\mathcal{G}^+$ can exceed unity.

For such a case, logical variables can be seen as a kind of missing values, and sometimes we assume that there is a *missing-data mechanism* that lurks in our observation process where some part of data turns to be missing. To be more specific, the missing-data mechanism is modeled as $P_\phi(G^+|G)$, a conditional distribution of final observations $G^+ \in \mathcal{G}^+$ on events $G \in \mathcal{G}$, which are fully informative but hidden from us (ϕ are the distribution parameters). Trivially, $P_\phi(G^+|G) = 0$ holds where G is not the instance of G^+.

[13] Please do not confuse it with `not/1` provided by B-Prolog, which simulates negation as failure. From the theoretical view, it is important to notice that PRISM allows *general clauses*, i.e. clauses that may contain negated atoms in the body.

[14] More generally, First Order Compiler eliminates universally quantified implications, i.e. goals of the form $\forall y(p(x, y) \rightarrow q(y, z))$)

Table 2.1: The conditional probability table $P_\phi(G^+|G)$ for the HMM program which satisfies the MAR condition. The predicate name `hmm` is simply abbreviated to `h`. All logical variables are existentially quantified.

	$G^+ \in \mathcal{G}^+$									
$G \in \mathcal{G}$	`h([X,Y])`	`h([X,X])`	`h([a,X])`	`h([b,X])`	`h([X,a])`	`h([X,b])`	`h([a,a])`	`h([a,b])`	`h([b,a])`	`h([b,b])`
`h([a,a])`	p_1	p_2	p_3	0	p_5	0	p_7	0	0	0
`h([a,b])`	p_1	0	p_3	0	0	p_6	0	p_8	0	0
`h([b,a])`	p_1	0	0	p_4	p_5	0	0	0	p_9	0
`h([b,b])`	p_1	p_2	0	p_4	0	p_6	0	0	0	p_{10}

Then we further assume the MAR condition and the *parameter distinctness condition*, respectively, as follows:[15]

- For an actual observation $G^+ \in \mathcal{G}^+$ and some ϕ, $P_\phi(G^+|G_1) = P_\phi(G^+|G_2)$ holds for any ground instances G_1, G_2 of $\mathcal{G}$.
- ϕ is distinct from θ.[16]

For the HMM program, the conditional probability table $P_\phi(G^+|G)$ under the MAR condition is shown in Table 2.1, where $p_1, p_2, \ldots, p_{10}$ (which form ϕ) need to be assigned so that $\sum_{G^+} P_\phi(G^+|G) = 1$ holds for each $G \in \mathcal{G}$. For example, we may have: $p_1 = 1/2$, $p_2 = 0$, $p_3 = p_4 = \cdots = p_{10} = 1/6$.

As we have mentioned, in this situation, the logical variables can be seen as the missing part, and one may find from Table 2.1 that the probability of $G^+ \in \mathcal{G}^+$ only depends on the observed part, not on the missing part[17] in the case with $\mathcal{G}^+$. For example, we have a constant probability p_3 for the different instantiations of `X` in `hmm([a,X])`.

If the MAR condition holds, it is shown that the missing-data mechanism is *ignorable* in making inferences for the model parameters θ (i.e. learning θ). The programming system blindly ignores the missing-data mechanism, but under the MAR condition, learning θ based on the goals from $\mathcal{G}^+$ (goals with logical variables) is justified. Otherwise, the missing-data mechanism is said to be *non-ignorable*, and we may need to consider an explicit model of the observation process. One difficulty with the MAR condition is its testability. For example, a recent work by Jaeger tackles with this problem [17].

2.4.6 Summary: modeling assumptions

For all efficient probability computations offered by the system to be realized, we have pointed out several assumptions on the modeling part. In this section, let us summarize them as follows:

- *Independence condition*: the sampling results of the different switches are probabilistically independent, and the sampling results of a switch with different trials (i.e. at different positions in a proof tree) are also probabilistically independent.

[15] The first sub-condition implies that $P_\phi(G^+|G) = P_\phi(G^+)/\sum_{G'\text{: } G' \text{ is an instance of } G^+} P_\theta(G')$ for any ground instance G of G^+ [16].

[16] ϕ is said to be distinct from θ if the joint parameter space of θ and ϕ is the product of θ's parameter space and ϕ's parameter space.

[17] It should be noted that the original definition of the MAR condition [27] is made on a data matrix which has missing-data cells. We can make a correspondence between our setting (the many-to-many mapping from $\mathcal{G}$ to $\mathcal{G}^+$) and such a data matrix, by an encoding method briefly described in Section 4.1.1 of [13]. The MAR condition roughly defined in this section should rather be called the *coarsened-at-random (CAR) condition*, a generalization of the MAR condition. There are several formal definitions on the MAR/CAR condition, so it would be useful for the interested users to consult the papers in the literature ([16], for example).

- *Finiteness condition*: for any observable goal[18] G, both the size of any explanation for G and the number of explanations for G are finite.
- *Exclusiveness condition*: with any parameter settings, for any observable goal G, the explanations for G are probabilistically exclusive to each other, and the sub-explanations for each subgoal of G are also probabilistically exclusive to each other.
- *Uniqueness condition*: with any parameter settings, all observable goals are exclusive to each other, and the sum of probabilities of all observable goals is equal to unity. For parameter learning, the following two conditions form a relaxation of the uniqueness condition:
  - *Missing-at-random (MAR) condition*: in the observation process for the data of interest, there is a missing-data mechanism in which the probability of the data being generated does not depend on its missing part.
  - *No-failure condition*: for any observable goal G, the generation process for G (i.e. a sampling execution of G) never fails.
- *Acyclicity condition*: for any observable goal G, there is no cyclic dependency with respect to the calling relationship among the subgoals, which are found in a generation process for G.

It may look difficult to satisfy all the conditions above. But if we keep in mind to write terminating programs in a generative fashion with care for the exclusiveness among disjunctive paths, these conditions are likely to be satisfied. It can be seen in Chapter 7 that popular generative models including hidden Markov models, probabilistic context-free grammars or Bayesian networks are written in this fashion. If the program violates the no-failure condition, one possible solution is to utilize the system's facility described in §2.4.4.

Theoretically speaking, it is sometimes misunderstood and hence is desired to note that the distribution semantics [30, 35] itself assumes none of the conditions above. We can say PRISM's semantics is just a restricted version of the distribution semantics, which is conscious of efficient probability computation.

2.5 Utility part

As compared to the modeling part, the utility part is quite simple — it is just a usual Prolog program with the system's built-ins. It is also possible to write queries, each of which takes the form "`:-` Q`.`" The queries are executed after the program is completely loaded.

2.6 Declarations

Declarations are made with several predefined predicates to give additional information to the system — observable probabilistic predicates (*target declarations*), outcome spaces of switches (*multi-valued switch declarations*), the source of observed data (*data file declarations*), tabled and non-tabled predicates (*table declarations*), and some other program files to be included (*inclusion declarations*).

2.6.1 Target declarations

A target declaration takes the following form:

`target(`*Pred*`,`*Arity*`).`

[18] Observable goals are the goals which can all potentially arise in the data. We can of course consider a countably infinite number of observable goals.

or

`target(`*Pred*`/`*Arity*`).`

A target declaration specifies a *target predicate*, i.e. a predicate that is observable. Training data used in learning must be atoms of observable predicate. The target predicate must be probabilistic and each program must contain at least one target declaration.

2.6.2 Data file declaration

A data file declaration takes the form:

`data(`*Filename*`).`

where *Filename* is the filename of observed data. As in Prolog, a filename must be an atomic symbol. If the filename contains a special symbol such as dot (“`.`”), it should be quoted by “`'`”. For example,

```
data('bloodtype.dat').
```

Data file declarations are optional. If no data declaration is given, then observed data must be given as an argument of `learn/1` (See §4.8). The format of the data file is described in §4.8.3.

2.6.3 Multi-valued switch declarations

◇ Basic form

A multi-valued switch declaration takes the following form:

`values(`*I*`,`*Values*`).`

where *I* denotes a switch identifier and *Values* is the list of ground terms indicating possible outcomes (or outcome space) of *I*. For example,

```
values(color,[red,yellow,blue]).
```

declares that switch `color` has three possible outcomes: `red`, `yellow` and `blue`.

The first argument *I* in a switch declaration can be an arbitrary Prolog term. All switches that have matching identifiers will have a declaration list of outcomes. If there are multiple declarations for a switch, the first matching declaration is used. For instance, consider the declarations:

```
values(f(a,a),[1,2,3]).
values(f(X,X),[a,b]).
values(f(_,_),[x,y,z]).
```

Then, switch `f(a,a)` has the outcomes `1`, `2` and `3`, switch `f(b,b)` has the outcomes `a` and `b`, and switch `f(a,b)` has the outcomes `x`, `y` and `z`.

◇ On-demand specification of the outcome space

A value declaration can have a body that dynamically generates a list of outcomes (a list of *ground* terms) for the corresponding switch. For instance, in the following declaration,

```
values(s,Vals):-
    findall([X,Y],(member(X,[1,2,3]),member(Y,[a,b])),Vals).
```

switch `s` has as outcomes the pairs of terms in which one from {`1`, `2`, `3`} and another from {`a`, `b`}. From a view point of efficiency, however, please remember that the body of a value declaration is evaluated at each time the corresponding `msw/2` is called.[19] Furthermore, `values/2` is just treated as a non-probabilistic clause which can work in the other part of the program (i.e. both the modeling part and the utility part). In the example above, we can run '`?- values(s,X).`' directly.

There is a case where some switches have outcome spaces that *dynamically change*. Let us consider a part of a program as follows:

```
:- dynamic s2_vals/1.

values(s2,Vs):- s2_vals(Vs).  % Value declaration

s2_vals([a,b,c]).

change_values(Vs):- retract(s2_vals(_)), assert(s2_vals(Vs)).
```

In this program fragment, the outcome space of a switch `s2` is specified by `s2_vals/1`, a user-defined non-probabilistic predicate. Also it is easy to see that the outcome space of `s2` are (indirectly) modified by calling `change_values(`*Vs*`)`, where *Vs* is a list of new outcomes. For such a case, the probability distributions (or parameters) of `s2` maintained by the programming system can be inconsistent, and should be problematic in many cases. By default, when some modification in the outcome space of a switch is detected, the system automatically sets the default distribution to the switch (by `set_sw/1`; §4.2.3), before invoking the routines that refer to the distributions of switches (e.g. sampling, probability computations, `get_sw/2` and so on). If you wish to disable such automatic configuration, set the `dynamic_default_sw` flag to '`off`' (§4.14), and if necessary, call suitable `set_sw` predicates before calling the routines that refer to the switch distributions.

◇ Extended form

Since version 1.9, `values_x/2-3` are introduced as a syntactic sugar for `values/2`. With `values_x/2`, we can rewrite the value declaration above as:

```
values_x(s,[1-10]).
```

We can specify two or more ranges in a list, and it is also possible to specify the skip number *N* in the form `@`*N* suffixed to the range element. For example,

```
values_x(foo,[3,8,0-3@2,7-20@5]).
```

is the same as `values(foo,[3,8,0,2,7,12,17])`.[20] Internally, `values_x/2` will be translated to `values/2` with the corresponding expanded values.[21] To be specific, the clauses "`values_x(`*Sw*`,`*Values*`)`" and "`values_x(`*Sw*`,`*Values*`):-` *Body*" will be translated respectively to:

```
values(Sw,Values1):- expand_values(Values,Values1).
values(Sw,Values1):- Body,expand_values(Values,Values1).
```

[19] If you wish to avoid the repetitive evaluation of the body, one way is to specify `values/2` as a tabled predicate (see B-Prolog's manual for details):

```
:- table values/2.
```

However it should be noted that this declaration could lead to a trouble when the evaluation result dynamically changes (e.g. by some randomness, or a dynamic modification of the program with the assert/retract predicates).

[20] Currently, the system does not consider sorting or deletion of duplicate values on the expanded values.

[21] This also implies that we cannot execute `values_x/2` directly.

The built-in `expand_values/2` will make an expansion of values like above. Thus we can have *parameterized* value declarations:

```
num_class(20).
values_x(class,[1-X]):- num_class(X).
```

In addition, using `values_x/3`, we can set/fix parameters of switches with ground names after loading the program. Please note however that, for the declarations of switches with non-ground names, the parameters can neither be set nor fixed. Similarly to `values_x/2`, `values_x/3` will be also translated to `values/2` with the corresponding expanded values. For the detailed descriptions on setting and fixing switch parameters, please visit §4.2.3 and §4.2.4, respectively. Now let us consider the examples:

```
values_x(foo(0),[1,2,3],fix@[0.2,0.7,0.1]).
values_x(bar,[1,2,3],set@[0.2,0.7,0.1]).
values_x(baz(a,b),[1,2,3],[0.2,0.7,0.1]).
values_x(u_sw,[1,2,3],uniform).
```

In the first case, we declare a switch `foo(0)` whose values are `1`, `2`, and `3` and whose parameters are fixed to 0.2, 0.7, and 0.1 respectively. In the second case, we declare a switch `bar`, only setting parameters, not fixing parameters. In the third case in which `set@` or `fix@` prefixes are omitted, the parameters will not be fixed (i.e. the default is `set@`). As in the last case, we can set/fix the parameters in a distribution form.

Inside the system, to set/fix parameters, `set_sw/2` or `fix_sw/2` will be invoked after loading without evaluating the body of `values_x/3`. So no parameters will be set for the declarations with `values_x/3` whose third argument includes logical variables. Also it should be noted that, for each of the declarations with `values_x/3`, `set_sw/2` or `fix_sw/2` is called *only once* after loading — not every time the specified switch is called. So for the switches whose outcome spaces are dynamically changed, `values_x/3` may not work as expected.

Since version 1.11, we can configure the pseudo counts of switches as well as the parameters. For the switches specified with `set_h@` (resp. `fix_h@`), the programming system will call `set_sw_h/2` (resp. `fix_sw_h/2`) while loading the program. `h@` can be used as an abbreviation of `set_h@`. For example, we may declare:

```
values_x(foo(0),[1,2,3],fix_h@[1.0,2.0,0.5]).
values_x(bar,[1,2,3],set_h@[1.0,2.0,0.5]).
values_x(baz(a,b),[1,2,3],h@[1.0,2.0,0.5]).
values_x(u_sw,[1,2,3],h@0.5).
```

2.6.4 Table declarations

In PRISM, all probabilistic predicates are tabled by default. On the other hand, the user can declare what predicates are to be tabled. The statement,

```
:- p_table p/n.
```

declares that the probabilistic predicate p/n is tabled, where p is the predicate name and n is the arity. In this case, please note here that all other probabilistic predicates that are not declared will not be tabled.

The user can also declare predicates that need not be tabled by using the statement:

```
:- p_not_table p/n.
```

The declaration `p_table` and `p_not_table` cannot co-exist in a program. Once a program contains a `p_not_table` declaration, all the probabilistic predicates that do not occur in any `p_not_table` declaration are assumed to be tabled. `p_not_table` seems useful in the following cases:

- It is obviously inefficient (especially in space) to store the solutions for probabilistic but deterministic predicates (i.e. the predicates which only call probabilistic predicates deterministically). So it is recommended to use the `p_not_table` declarations for such predicates, as long as they are not referred to as subgoals.[22]
- When tracing the program (by `trace/0`, §3.6), it is also recommended to disable tabling for all probabilistic predicates.
- The solutions for tabled predicates will appear as subgoals in the explanation graphs, and we can handle such explanation graphs in various ways (by `probf/2` or `viterbif/3`, for example). If we wish to make such explanation graphs simple and readable, it might be useful to use `p_not_table` for the predicates which are not important to understand the explanation graphs. Of course there is a trade-off between the readability of such explanation graphs and the efficiency in computation.

For non-probabilistic predicates, B-Prolog's table declaration is available (see B-Prolog's manual for details):

```
:- table p/n.
```

2.6.5 Inclusion declarations

If probabilistic predicates are stored in several files, then all these files must be included by using the directive `:- include(`*File*`)` in the main file. The filename of a PRISM program should be enclosed by the single quotation mark like `:- include('foo.psm').`

2.6.6 Mode declarations

Mode declarations supported by B-Prolog can also be used in PRISM. For a detailed description, please consult the user's manual of B-Prolog.

[22] In hindsight computation (§4.7) or after extracting explanations graphs (§4.5), we often need to refer to some particular subgoals explicitly. In such cases, we cannot apply `p_not_table` to the predicates of these subgoals.

Chapter 3

PRISM Programming System

3.1 Installing PRISM

PRISM is implemented on top of B-Prolog. The release package contains all standard functionalities of B-Prolog, and therefore it is unnecessary to install B-Prolog separately.

3.1.1 Windows

To install PRISM on Windows, you need to make the following steps:

1. Download the package `prism1112_win.zip`.
2. Unzip the downloaded package under `C:\`.
3. Append `C:\prism\bin` to the environment variable `PATH` so PRISM can be started at every working folder.

Note that if PRISM is installed in a folder other than `C:\`, then you have to change the batch file `prism.bat` in the `bin` folder and the path `C:\prism\bin` accordingly.

3.1.2 Linux

A single united package `prism1112_linux.tar.gz` is provided for x86-based Linux systems. The binaries are expected to work on the systems with glibc 2.3 or higher.[1] Typical steps for installation are as follows:

1. Download the package `prism1112_linux.tar.gz` into your home directory.
2. Unpack the downloaded package using the `tar` command.
3. Append `$HOME/prism/bin` to the environment variable `PATH` so PRISM can be started at every working directory.

Note that if PRISM is installed in a directory other than your home directory, please change the path `$HOME/prism/bin` accordingly. Internally, the package contains both binaries for 32-bit and 64-bit systems. The start-up commands (`prism`, `upprism` and `mpprism`) automatically choose a binary suitable for your environment.

[1] Note that the utility of parallel EM learning has more requirements on the environments; see §6.2 for details.

3.1.3 Mac OS X

Since version 1.11, a binary package is also provided for Mac OS X (currently this package is considered to be experimental). The package `prism1112_macx.tar.gz` contains a universal binary for PowerPC and Intel processors. To install the package, please follow the steps for Linux (§3.1.2). Please note that we have not tested the Mac OS X package well, since our test environment for Mac OS X is rather limited.

3.2 Entering and quitting PRISM

You need to open a command terminal first before entering PRISM. To do so on Windows, select [Start] → [Run] and then run `cmd`, or select

[Start] → [Programs] → [Accessories] → [Command Prompt].

To enter PRISM, type

```
prism
```

at the command prompt. Once the system is started, it responds with the prompt '`| ?-`' (in this manual, we simply write '`?-`' instead) and is ready to accept Prolog queries.

To quit the system, use the query:

```
?- halt.
```

or simply enter `^D` (Control-D) when the cursor is located at an empty line.

3.3 Loading PRISM programs

The command `prism(`*File*`)` compiles the program in *File* and loads the binary code into the system. For example, suppose '`coin.psm`' stores a PRISM program, then the command

```
?- prism(coin).
```

compiles the program into a byte code program '`coin.psm.out`' and loads '`coin.psm.out`' into the system.

A program may be stored in multiple files, but only the main file may be loaded. In the main file, all the files in the program that contain probabilistic predicates must be included by using the directive '`:- include(`*FileName*`)`' (§2.6.5). In this way, the system's compiler will access all the probabilistic predicates when the program is loaded. Standard Prolog program files that do not contain probabilistic predicates can be compiled and loaded separately by using `compile/1` and `load/1` commands of B-Prolog.

The command `prism(`*Options*`,`*File*`)` loads the PRISM program stored in *File* into the system under the control of the options given in a list *Options*. If the file has the extension name '`.psm`', then only the main file name needs to be given. The following options are allowed:

- `compile` — Load the program after it is compiled (default).
- `consult` — Load the program without compilation. This option must be specified if the program is to be debugged.

- `load` — Load the (compiled) binary code program with the suffix `.psm.out` . This option allows us to save the compilation time. To load a program containing probabilistic predicates, it is highly recommended to use this option rather than direct use of `load/1` (B-Prolog's built-in), though it was described in the manuals of the previous versions.[2]
- `v` — Monitor the learning process.
- `nv` — Do not monitor the learning process (default).

For example, by `?- prism([consult],foo)`, we can load the program without compilation.

In addition, we can specify the values of execution flags (§4.14) as options, each takes the form '*Flagname=Value*'. For example, if we want to set a value `on` to the `log_viterbi` flag, add `log_viterbi=on` to *Options*. The above options `v` and `nv` can also be specified by '`verb=on`' and '`verb=off`', respectively. The command `prism(`*File*`)` described above is the same as `prism([],`*File*`)`, which means that the program is loaded with the default options and the default flag values.

3.4 Configuring the sizes of memory areas*

B-Prolog, the fundamental system of the PRISM programming system, has four memory areas: program area, control stack + heap, trail stack and table area. Since version 1.10, these areas are *automatically* expanded on demand, so there is no need to specify the sizes of memory areas by manual.

If you already know the memory sizes used by your program, as did in version 1.9 or earlier, you can specify the sizes of *initial* memory areas by modifying the corresponding values in the start-up commands `prism` (a shell script on Linux) and `prism.bat` (a batch file on Windows), or by specifying command line options `-s` (control stack + heap), `-b` (trail stack), `-t` (table area) and `-p` (program area). For example,

```
prism -s 8000000
```

starts the programming system with 8 megawords (32 megabytes on 32-bit environments, 64 megabytes on 64-bit environments) allocated to the control stack + heap. B-Prolog's built-in `statistics/0` will show the allocated sizes of these memory areas.

3.5 Running PRISM programs

The command `prism_help/0` displays the usage of the basic built-ins in the programming system (Figure 3.1). The details of these built-ins are described in Chapter 4.

As mentioned before, the modeling part of a PRISM program can be executed in two different styles, namely *sampling execution* (§2.4.1) and *explanation search* (§2.4.2). The system is in sampling execution if it is given a probabilistic goal or `sample(`*Goal*`)` (§4.3). In sampling execution, a goal may give different results depending on the outcomes of the switches. On the other hand, an explanation search will be invoked in advance of numerical computations in learning (with `learn/0` or `learn/1`; §4.8), probability calculation (with `prob/2` and so on; §4.4), Viterbi computation (with `viterbif/3` and so on; §4.6), and hindsight computation (with `hindsight/3` and so on; §4.7). `probf/2` or its variation (§4.5) only makes an explanation search and outputs explanation graphs, the intermediate data structure used in the numeric computations above.

In addition, there are miscellaneous built-in predicates which handle switch parameters (`set_sw/2` and so on; §4.2) or the flags for various settings of the system (`set_prism_flags/2` and `get_prism_flags/2`; §4.14).

[2] Despite that, we can load the compiled binary code of a usual (i.e. non-probabilistic) Prolog program by `load/1`.

```
prism(File)          -- Load a program in File.
prism(Opts,File)     -- Load a program in File under control of Opts.

msw(I,V)             -- Switch I randomly outputs the value V.

learn(Facts)         -- Learn the parameters of the switches using Facts.
learn                -- Learn the parameters of the switches using
                          facts stored in the file declared by data(File).
sample(Goal)         -- The same as call(Goal) but Goal must be probabilistic.
prob(Goal,P)         -- P is the probability of Goal.
probf(Goal,F)        -- F is the explanation graph of Goal.
viterbi(Goal,P)      -- P is the Viterbi probability of Goal.
viterbif(Goal,P,F)   -- F is the Viterbi explanation of Goal, and P is
                          the probability of F (the Viterbi probability of Goal).
hindsight(G,G1,Ps)   -- Ps are the hindsight probs of G's subgoals matching
                          with G1.

set_sw(S,Params)     -- Set the parameters Params of the switch S.
get_sw(S,Info)       -- Info contains the information about the switch S.
set_prism_flag(F,V)  -- Set the value V to the execution flag F.
get_prism_flag(F,V)  -- Get the current value V of the execution flag F.
```

Figure 3.1: The output of `prism_help/0`.

3.6 Debugging PRISM programs

As described above, probabilistic inferences with some given goal G are made on the explanations for G. So `probf/1-2` (§4.5) should be the first choice as a debugging tool at symbolic level since they output all explanations for G.

Furthermore, programs can be executed in the trace mode. The command

```
trace
```

switches the execution mode to the trace mode, and the command

```
notrace
```

switches the execution mode back to the usual mode. In the trace mode, the execution steps of programs loaded with the option `consult` (§3.3) can be traced. To trace part of the execution of a program, use `spy` to set spy points:

`spy` (*Atom*`/`*Arity*) `.`

The spy points can be removed by:

```
nospy.
```

To remove only one spy point, use:

`nospy` (*Atom*`/`*Arity*) `.`

In (forward) sampling, the trace of a program looks the same as that of a normal Prolog program except that for the built-in `msw(I,V)` the probability of the outcome V is shown. For example, the following trace steps show that the outcome of the trial of the switch is 'head', which has probability 0.5.

```
Call: (7) msw(coin,_580ebc):_580ff8 ?
Exit: (7) msw(coin,head):0.5 ?
```

In explanation search (§2.4.2), a trace displays the steps that lead to the findings of explanation paths. Each explanation path consists of a subgoal to be explained, a list of explaining subgoals and a list of switch instances. For instance, we may see the following path:

```
Add: (12) path(direction(left),[],[msw(coin,head)])
```

From this we can see that a subgoal `direction(left)` is explained by the outcome 'head' of the switch 'coin'. Unfortunately, however, it should be difficult to trace the process of explanation search with tabling. Please turn off tabling for all probabilistic predicates by the `p_not_table` declarations (§2.6.4).

In our experience, it is also difficult to identify the subgoal which causes an unexpected failure. One ad-hoc way should be so-called "printf debugging." Or we may rewrite the clause

```
p(X):- q(X,Y).
```

into

```
p(X):- ( q(X,Y) ; format("Failed ~w !!", [q(X,Y)]), fail ).
```

where `q(X,Y)` is a suspicious subgoal, and check the call pattern of `q/2` that leads to failure. To provide a debugging facility for unexpected failures in explanation search is future work.

3.7 Batch execution*

Since version 1.9, the package provides additional commands for batch execution. To enable batch execution, we need the following two steps:

- Add a query we attempt to run as a batch execution to the program.
- Run the command `upprism` at the shell prompt (Linux) or the command prompt (Windows), instead of `prism`.

The query for batch execution is specified in the body of `prism_main/0-1`. For example, for a simple learning session, we may add the following definition of `prism_main/0` to the program `foo.psm`:

```
prism_main:-
  set_seed(5893421),
  get_data_from_somewhere(Gs),  % user-defined predicate
  learn(Gs).
```

Then we run `upprism` specifying the program name:

```
upprism foo
```

at the shell prompt (Linux) or the command prompt (Windows). If we want to pass arguments to `upprism`, it is needed to define `prism_main/1` instead of `prism_main/0`. For example, let us introduce two arguments, where the first is a seed for random numbers and the second is the data size. The corresponding batch clause could be as follows:

```
prism_main([Arg1,Arg2]):-
  parse_atom(Arg1,Seed),  % parse_atom/2 is provided by B-Prolog
  parse_atom(Arg2,N),
  set_seed(Seed),
  get_data_from_somewhere(N,Gs),  % assume that we'll get N data
  learn(Gs).                      %   as Gs here
```

The command arguments will be passed to `prism_main/1` as a list of atoms. Hence it is important to note that to pass integers, we need to parse the corresponding atoms in advance, that is, we need to get an integer `5893421` from an atom `'5893421'`. The parsing is done by `parse_atom/2`, a built-in provided by B-Prolog. After this setting, we can conduct a batch execution as follows:

```
upprism foo 5893421 1000
```

If both `prism_main/0` and `prism_main/1` co-exist in one program, `upprism` will try to run *only* `prism_main/1`. For such a program, if we invoke `upprism` with no command-line arguments, `prism_main([])` will be called, and so an unexpected behavior is likely to be caused.

Furthermore, `upprism` provides some variations in the file specification:[3]

- `upprism prism:`*foo*
  This is the same as "`upprism` *foo*", that is, the system will read a usual program file *foo*`.psm` (which has no definition of the predicate `failure/0`).
- `upprism prismn:`*foo*
  The system will read a failure program file *foo*`.psm` (which has a definition of `failure/0`; see §4.11). This is a replacement for the command `upprismn`, which was introduced in version 1.9.
- `upprism load:`*foo*
  The system will read a (compiled) binary code file *foo*`.psm.out` . By this, we would save the compilation time.

In version 1.1, `mpprism` is newly introduced as a command for batch execution of parallel learning. Please consult Chapter 6 for the detailed usage.

3.8 Error handling

In the current implementation, when the system encounters an error, the current query is immediately halted by `abort/0` (B-Prolog's built-in). In such a case, to avoid being affected by the remaining side-effects, it is recommended to quit the system by `halt/0` and then to start the system again. If the error message you meet includes "`internal error`", the problem should not have been caused by the user program, but the system. In such a case, please make a contact to the development team (see page i).

[3] Some users may want to use '`-g`' option introduced since B-Prolog 6.9. That is, we can run "`prism foo.psm.out -g 'go'`" to load the binary code '`foo.psm.out`' and then to execute a query "`go`".

Chapter 4

PRISM Built-in Utilities

4.1 Program information

After a program loaded, we can get the basic information about the program by the following built-ins:

- `show_values/0` displays the value declarations.
- `show_prob_preds/0` displays the list of probabilistic predicates.
- `show_tabled_preds/0` displays the list of tabled predicates.

4.2 Random switches

4.2.1 Making probabilistic choices

The built-in `msw(I,V)` succeeds if a trial of a random switch I gives an outcome V. To use a switch I, there must be a multi-valued switch declaration (§2.6.3) for I in the program. As described in §2.2, the probabilistic behavior of random switches are specified by their parameters. That is, a random switch I gives an outcome V with probability $\theta_{I,V}$, and we consider $\theta_{I,V}$ as a parameter for the switch I. Also, as previously mentioned, switches have different behaviors for sampling execution (§2.4.1) and explanation search (§2.4.2). These parameters can be set by using `set_sw/2` (§4.2.3) or by parameter learning (§4.8).

Furthermore, in Bayesian approaches, we consider that the parameters θ follow the prior distribution (a Dirichlet distribution) which has hyperparameters $\alpha_{I,V}$, each corresponding to a parameter $\theta_{I,V}$. For maximum a posteriori (MAP) estimation (§4.8.2) or variational Bayesian (VB) learning (§5.1), we need to handle these hyperparameters. It should be noted however that, in the programming system, each hyperparameter $\alpha_{I,V}$ can be accessed only through the corresponding pseudo count $\delta_{I,V} = (\alpha_{I,V} - 1)$. Also in the current implementation, $\delta_{I,V}$ should be non-negative. These pseudo counts can be set manually by the built-ins such as `set_sw_h/2`, as described below.

4.2.2 Registration of switches

Let us consider a program which contains no query statements (that begin with '`:-`'). Just after the program loaded, the programming system will not have recognized any random switches at all. This is because the switch names in the program are not always ground, and the system does not know at that moment what switches will be used later (please recall that each switch is identified by a ground

term). Random switches are registered to the programming system's internal database only after their parameters or pseudo counts are set explicitly by manual (i.e. with the built-ins in §4.2.3, §4.2.4, and so on) or by parameter learning (§4.8).

4.2.3 Setting the parameters/hyperparameters of switches

The built-in `set_sw(`*I*`,` *Params*`)` sets the parameters of outcomes of a switch *I* to *Params* where *Params* is a list `[`p_1`,` p_2`, . . . ,` p_K`]` (recommended) or a term of the form $p_1+p_2+\cdots+p_K$ where the numbers p_1, p_2, ..., p_K sum up to unity (i.e. $\sum_k p_k = 1$). Please note that the switch name *I* must be ground. For example, to make a biased coin, we may run:

```
?- set_sw(coin,[0.8,0.2]).
```

That is, this will set 0.8 to the parameter of the first value of switch `coin`, and set 0.2 to the parameter of the second value, where the order of values follows the multi-valued switch declaration (§2.6.3).

Since version 1.9, it is also allowed to set parameters in a distribution form:[1]

- `set_sw(`*I*`)` is the same as `set_sw(`*I*`,default)`.
- `set_sw(`*I*`,default)` sets a distribution specified by the `default_sw` flag.
- `set_sw(`*I*`,uniform)` sets a uniform distribution.
- `set_sw(`*I*`,f_geometric)` is the same as `set_sw(`*I*`,f_geometric(2,desc))`.
- `set_sw(`*I*`,f_geometric(`*Base*`))` is the same as `set_sw(`*I*`,f_geometric(`*Base*`,desc))`.
- `set_sw(`*I*`,f_geometric(`*Base*`,`*Type*`))` sets a finite geometric distribution, where *Base* is its base (an integer greater than 1) and *Type* is `asc` or `desc`. For finite geometric distributions, see the description on the `default_sw` flag in §4.14.

We need to add descriptions for the first two cases. In the versions earlier than 1.9, parameters should be set explicitly by manual if we do not have learning data. On the other hand, since 1.9, we can specify the default parameters in a distribution form. For example,

```
?- set_prism_flag(default_sw,uniform).
```

makes the default parameters to be uniform (see §4.14 for handling execution flags). Then, if we attempt a sampling, or a probability computation, the parameters of switches that has not been used yet will be set to be uniform on the fly.

Since the default value of the `default_sw` flag is '`uniform`', we can use switches which follow a uniform distribution just after invoking the system. The other available values for the flag are '`none`', '`f_geometric(`*Base*`)`' (*Base* is the base, an integer greater than 1), and so on. The first one means that we have no default parameters, as in the previous versions. The second one stands for a finite geometric distribution.

Also, the following predicates set the parameters to one or more switches that have been registered to the internal database at that time (see §4.2.2):

- `set_sw_all(`*Patt*`)` sets a default distribution to all switches matching with *Patt* (i.e. all switches whose names unify with *Patt*).
- `set_sw_all(`*Patt*`,` *Dist*`)` sets a distribution *Dist* to all switches matching with *Patt*.

[1] The introduction of finite geometric distributions is inspired by [1].

- `set_sw_all` (with no args) is the same as `set_sw_all(_)`.

Similarly to the above, pseudo counts of random switches can be set by `set_sw_h/1-2` and `set_sw_all_h/0-2`:

- `set_sw_h(`*I*`)` is the same as `set_sw_h(`*I*`,default)`.
- `set_sw_h(`*I*`,[`δ_1`,`δ_2`,...,`δ_K`])` sets the pseudo counts $\delta_1, \delta_2, \ldots, \delta_K$ to switch *I*, where *K* is the number of possible values of switch *I*, and each δ_k ($1 \leq k \leq K$) is a non-negative floating point number.
- `set_sw_h(`*I*`,`δ`)` is the same as `set_sw_h(`*I*`,[`δ`,`δ`,...,`δ`])`, where δ is a non-negative floating-point number.
- `set_sw_h(`*I*`,uniform(`δ`))` is the same as `set_sw_h(`*I*`,[`δ/K`,`δ/K`,...,`δ/K`])`, where δ is a non-negative floating-point number, *K* is the number of possible values of switch *I*.
- `set_sw_h(`*I*`,uniform)` is the same as `set_sw_h(`*I*`,uniform(1.0))`.[2]
- `set_sw_h(`*I*`,default)` sets the default pseudo counts specified by the `default_sw_h` flag.
- `set_sw_all_h(`*Patt*`,`*PseudoCs*`)` sets the pseudo counts *PseudoCs* to all switches matching with *Patt*, where *PseudoCs* is a Prolog term allowed to the second argument of `set_sw_h/2`.
- `set_sw_all_h(`*Patt*`)` is the same as `set_sw_all_h(`*Patt*`,default)`.
- `set_sw_all_h` (with no args) is the same as `set_sw_all_h(_)`.

4.2.4 Fixing the parameters/hyperparameters of switches

Sometimes we need constant parameters which are not updated during learning. For example, letting *g* be a gene of interest, we may want the probability of *g* being selected from one parent to be constant at 1/2. To handle with such situations, the following built-in predicates are provided:

- `fix_sw(`*I*`)` fixes the parameters of all switches matching with *I* (i.e. all switches whose names unify with *I*). Then the parameters of these switches cannot be updated and will be kept unchanged during learning. These switches are said to be *fixed*.
- `fix_sw(`*I*`,`*Params*`)` sets the parameters *Params* to a switch *I*, as done in `set_sw/2`, and then fixes the parameters. Please note that *I* in `fix_sw(`*I*`,`*Params*`)` should be ground, while *I* in `fix_sw(`*I*`)` does not need to be ground.
- `unfix_sw(`*I*`)` makes changeable the parameters of all switches matching with *I*.
- `fix_sw_h(`*I*`)` fixes the pseudo counts (or equivalently, the hyperparameters) of all switches matching with *I*. Then the pseudo counts of these switches cannot be updated and will be kept unchanged during VB learning (§5.2.1).
- `fix_sw_h(`*I*`,`*PseudoCs*`)` sets the pseudo counts *PseudoCs* to a switch *I*, as done in `set_sw_h/2`, and then fixes the pseudo counts. Similarly to `fix_sw/2`, *I* should be ground here.
- `unfix_sw_h(`*I*`)` makes changeable the pseudo counts of all switches matching with *I*.

[2] This setting is the same as that in AutoClass, a well-known probabilistic clustering tool [5].

4.2.5 Displaying the switch information

The programming system provides the built-in predicates for checking the current status of switches (we hereafter call this the *switch information*):

- `show_sw` (with no args) displays information about the parameters of all switches used so far (i.e. all switches that have been registered into the internal database at that time; see §4.2.2). For example, in the direction program, we may run:

```
?- show_sw.
Switch coin: head (0.8) tail (0.2)
```

- `show_sw(`*I*`)` displays information about the parameters of the switches whose names match with *I*. For example:

```
?- show_sw(coin).
Switch coin: head (0.8) tail (0.2)
```

- `show_sw_h` (with no args) displays information about the pseudo counts of all switches.
- `show_sw_h(`*I*`)` displays information about the pseudo counts of the switches whose names match with *I*.
- `show_sw_b` (with no args) displays information about both the parameters and the pseudo counts of all switches.
- `show_sw_b(`*I*`)` displays information about both the parameters and the pseudo counts of the switches whose names match with *I*.

4.2.6 Getting the switch information

The switch information can be obtained as Prolog terms:

- `get_sw(`*I*`,`*Info*`)` binds *Info* to a Prolog term in the form `[`*Status*`,`*Vals*`,`*Params*`]` that contains information about switch *I*:
  - *Status* is either `fixed` or `unfixed`. The former (resp. the latter) indicates that the parameters of switch *I* is fixed (resp. unfixed).
  - *Vals* is a list of possible outcomes of switch *I*.
  - *Params* is a list of the parameters of switch *I*.

  For example, we may run:

```
?- get_sw(coin,Info)
Info = [unfixed,[head,tail],[0.8,0.2]]
```

- `get_sw(`*Info*`)` binds *Info* to a Prolog term in the form `switch(`*I*`,`*Status*`,`*Vals*`,`*Params*`)` where *I* is the identifier, *Status* is either `fixed` or `unfixed`, *Vals* is a list of possible outcomes, and *Params* is a list of the parameters. On backtracking, *Info* is bound to the one about the next switch.
- `get_sw(`*I*`,`*Status*`,`*Vals*`,`*Params*`)` is the same as `get_sw(`*I*`, [`*Status*`,`*Vals*`,`*Params*`]).`

- `get_sw(`*I*`,`*Status*`,`*Vals*`,`*Params*`,`*Cs*`)` additionally returns the expected occurrences *Cs* of switch *I*, which are computed in EM learning (this built-in is of course available after learning; see §4.8).[3]
- `get_sw_h(`*I*`,`*Info*`)` binds *Info* to a Prolog term in the form `[`*Status*`,`*Vals*`,`*PseudoCs*`]` that contains information about switch *I*:
  - *Status* is either `fixed_h` or `unfixed_h`. The former (resp. the latter) indicates that the pseudo counts of switch *I* is fixed (resp. unfixed).
  - *Vals* is a list of possible outcomes of switch *I*.
  - *PseudoCs* is a list of the pseudo counts of switch *I*.
- `get_sw_h(`*Info*`)` binds *Info* to a Prolog term in the form `switch(`*I*`,`*Status*`,`*Vals*`,`*PseudoCs*`)` where *I* is the identifier, *Status* is either `fixed_h` or `unfixed_h`, *Vals* is a list of possible outcomes, and *PseudoCs* is a list of the pseudo counts. On backtracking, *Info* is bound to the one about the next switch.
- `get_sw_h(`*I*`,`*Status*`,`*Vals*`,`*PseudoCs*`)` is the same as `get_sw_h(`*I*`,[`*Status*`,`*Vals*`,`*PseudoCs*`])`.
- `get_sw_b(`*I*`,`*Info*`)` binds *Info* to a Prolog term in the form `[[`*StatusP*`,`*StatusH*`],`*Vals*`,`*Params*`,` *PseudoCs*`]` that contains information about switch *I*, that is:
  - *StatusP* is either `fixed` or `unfixed`. The former (resp. the latter) indicates that the parameters of switch *I* is fixed (resp. unfixed).
  - *StatusH* is either `fixed_h` or `unfixed_h`. The former (resp. the latter) indicates that the pseudo counts of switch *I* is fixed (resp. unfixed).
  - *Vals* is a list of possible outcomes of switch *I*.
  - *Params* is a list of the parameters of switch *I*.
  - *PseudoCs* is a list of the pseudo counts of switch *I*.
- `get_sw_b(`*Info*`)` binds *Info* to a Prolog term in the form `switch(`*I*`,[`*StatusP*`,`*StatusH*`],`*Vals*`,` *Params*`,`*PseudoCs*`)` where *I* is the identifier, *StatusP* is either `fixed` or `unfixed`, *StatusH* is either `fixed_h` or `unfixed_h`, *Vals* is a list of possible outcomes, *Params* is a list of the parameters, and *PseudoCs* is a list of the pseudo counts. On backtracking, *Info* is bound to the one about the next switch.
- `get_sw_b(`*I*`,[`*StatP*`,`*StatH*`],`*Vals*`,`*Ps*`,`*PseudoCs*`)` is the same as `get_sw_b(`*I*`,[[`*StatP*`,` *StatH*`],`*Vals*`,`*Ps*`,`*PseudoCs*`])`.
- `get_sw_b(`*I*`,[`*StatP*`,`*StatH*`],`*Vals*`,`*Ps*`,`*Cs*`,`*PseudoCs*`)` additionally returns the expected occurrences *Cs* of switch *I*, which are computed in EM learning.

4.2.7 Saving the switch information

By using the following built-ins, all switch information can be saved into, or restored from, a file:

- `save_sw(`*File*`)` saves all switch information about the parameters into the file *File*.
- `save_sw` (with no args) is the same as `save_sw('Saved_SW')`.

[3] These expected occurrences are used in computing Cheeseman-Stutz score (§4.10), and might be used to judge whether we need to apply so-called *backoff smoothing*. If the observed data is complete (§4.8.1), *Cs* is just a list of numbers of occurrences of `msw(`*I*`,·)` in the data.

- `restore_sw(`*File*`)` restores all switch information about the parameters from the file *File*.
- `restore_sw` (with no args) is the same as `restore_sw('Saved_SW')`.
- `save_sw_h(`*File*`)` saves all switch information about the pseudo counts (or equivalently, the hyperparameters) into the file *File*.
- `save_sw_h` (with no args) is the same as `save_sw_h('Saved_SW_H')`.
- `restore_sw_h(`*File*`)` restores all switch information about the pseudo counts (or equivalently, the hyperparameters) from the file *File*.
- `restore_sw_h` (with no args) is the same as `restore_sw_h('Saved_SW_H')`.

4.3 Sampling

An execution with `sample(`*Goal*`)` (or a direct execution of *Goal*) simulates a sampling execution. A more detail description of sampling execution is found in §2.4.1. For example, for the program in §1.1, we may have a result of sampling execution such as:

```
?- sample(direction(D)).
D = left ?
```

Of course, the result is at random, and follows the distribution specified by the program.

Besides, there are some built-ins to get two or more samples. `get_samples(`*N*`,` *G*`,`*Gs*`)` returns a list *Gs* which contains the results of sampling *G* for *N* times. For example:

```
?- get_samples(10,direction(D),Gs).
Gs = [direction(right),direction(left),direction(right),
      direction(left),direction(right),direction(right),
      direction(right),direction(right),direction(left),
      direction(right)] ?
```

Inside the system, on each trial of sampling, a copy G' of the target goal G is created and called by `sample(`G'`)`. Please note that if one of *N* trials ends in failure, this predicate totally fails.

On the other hand, `get_samples_c(`*N*`,`*G*`,`*C*`,`*Gs*`)` tries to make sampling *G* under the constraint *C* for *N* times, and returns a list *Gs* which only contains the successful results of sampling. Note here that this predicate never fails by sampling, and if some trial ends in failure, nothing is added to *Gs* (thus the size of *Gs* can be less than *N*). Internally, this predicate first creates a copy `[`G'`,`C'`]` of `[`*G*`,`*C*`]`, and then executes `sample(`G'`)` and `call(`C'`)` in this order. In addition, `get_samples_c/4` writes the numbers of successful and failed trials to the current output stream. For example,

```
?- get_samples_c(10,pcfg(Ws),(length(Ws,L),L<5),Gs).
```

will return to `Gs` a list of sampled `pcfg(Ws)` where the length of `Ws` is less than 5. Besides, the last two of the following queries show the same behavior, but the first query may fail due to the failure at some trial of sampling:

```
?- get_samples(100,hmm([a|_]),Gs).
?- get_samples_c(100,hmm([a|_]),true,Gs).
?- get_samples_c(100,hmm(Xs),Xs=[a|_],Gs).
```

The built-in `get_samples_c(N,G,C,Gs,[SN,FN])` behaves similarly to `get_samples_c(N,G,C,Gs)`, except returning the numbers of successful and failed trials to *SN* and *FN*, respectively.

Since version 1.10, the programming system additionally provides a couple of variations on arguments for `get_samples_c/4-5`. If we specify the first argument in the form `[N,M]`, the predicates will try to make sampling for *N* times at maximum to get *M* samples. If we specify `[inf,M]`, then the system tries to get *M* samples with no limit on the number of trials. For example, we can always get 100 samples with the following query:

```
?- get_samples_c([inf,100],pcfg(Ws),(length(Ws,L),L<5),Gs).
```

However it should be noticed here that there is a risk of entering an almost infinite loop in the use of '`inf`' if the goal *G* (or *G* under the constraint *C*) is unlikely to succeed.

As discussed in §2.4.1 and §2.4.2, sometimes we need to write models in two different styles for sampling and explanation search with different sets of predicates. For example, we may use a predicate `foo_s/1` for sampling, and use `foo/1` for explanation search. To get training data for `foo/1` by sampling `foo_s/1` in an artificial experiment, we may replace the predicate name of sampled goals by modifying the second argument as follows:

```
?- get_samples_c(100,[foo_s(Ws),foo(Ws)],true,Gs).
```

4.4 Probability calculation

The built-in `prob(`*Goal*`,`*Prob*`)` calculates the probability *Prob* with which *Goal* becomes true. Under the independence and exclusiveness conditions (see §2.4.6), the probability of a conjunction `(`*A*`,`*B*`)` is computed as the product of the probabilities of *A* and *B* (because they are assumed to be independent), and the probability of a disjunction `(`*A*`;`*B*`)` is computed as the sum of the probabilities of *A* and *B* (because they are assumed to be exclusive). For a switch instance `msw(`*I*`,`*V*`)`, the probability is 1.0 if *V* is a variable, and the probability assigned to the outcome *V* if *V* is one of outcomes of switch *I*. For example, for the program in §1.1, we have:

```
?- prob(direction(left),P).
P = 0.5
```

The built-in `prob(`*Goal*`)` is the same as `prob(`*Goal*`,`*Prob*`)` except that the computed probability *Prob* is sent to the current output stream. Note here that, when enabling the methods for avoiding underflow (§4.12), `prob/1-2` returns the log of probabilities. `log_prob(`*G*`)` and `log_prob(`*G*`,`*P*`)` are the same as `prob(`*G*`)` and `prob(`*G*`,`*P*`)`, respectively, except that they always return the log-valued probability of the goal *G*.

4.5 Explanation graphs

The built-in `probf(`*Goal*`,`*EGraph*`)` returns the explanation graph *EGraph* for *Goal* as a Prolog term, where *Goal* must be a subgoal of the target predicate. An explanation graph is represented as a list of nodes, each corresponds to one of the ordered iff-formulas in §2.4.2. Each node takes the form `node(`G'`,`*Paths*`)` where G' is a subgoal of *G* and *Paths* is a list of paths that explain G'. With the terminology in §2.4.2, one of these paths corresponds to a sub-explanation E' for G'. Each path takes the form `path(`*Nodes*`,`*Switches*`)` where *Nodes* is a list of subgoals found in E', and *Switches* is a list of switch instances also found in E'. If we have subgoals which include logical variables, all of these variables will be treated as the distinct ones, for implementational reasons.

For example, in the HMM program with string length being 2, the explanation graph for `hmm([a,b])` is obtained as follows:

```
?- probf(hmm([a,b]),EGraph).

EGraph =
  [node(hmm([a,b]),
        [path([hmm(1,2,s0,[a,b])],[msw(init,s0)]),
         path([hmm(1,2,s1,[a,b])],[msw(init,s1)])]),
   node(hmm(1,2,s0,[a,b]),
        [path([hmm(2,2,s,[b])],[msw(out(s0),a),msw(tr(s0),s0)]),
         path([hmm(2,2,s1,[b])],[msw(out(s0),a),msw(tr(s0),s1)])]),
   node(hmm(1,2,s1,[a,b]),
        [path([hmm(2,2,s0,[b])],[msw(out(s1),a),msw(tr(s1),s0)]),
         path([hmm(2,2,s1,[b])],[msw(out(s1),a),msw(tr(s1),s1)])]),
   node(hmm(2,2,s0,[b]),
        [path([hmm(3,2,s0,[])],[msw(out(s0),b),msw(tr(s0),s0)]),
         path([hmm(3,2,s1,[])],[msw(out(s0),b),msw(tr(s0),s1)])]),
   node(hmm(2,2,s1,[b]),
        [path([hmm(3,2,s0,[])],[msw(out(s1),b),msw(tr(s1),s0)]),
         path([hmm(3,2,s1,[])],[msw(out(s1),b),msw(tr(s1),s1)])]),
   node(hmm(3,2,s0,[]),[]),
   node(hmm(3,2,s1,[]),[])] ?
```

Be warned that the result is manually beautified by the authors for making the data structure clear. Usually, the results by `probf/2` are appropriate to be handled by the program, but too complicated for humans to understand. For post-processing such Prolog-term representation of an explanation graph, since version 1.11, we can use `strip_switches(`*EGraph*`,`*EGraph′*`)`, which drops all switch instances from *EGraph* and then returns the resultant graph as *EGraph′*. Furthermore, the built-in `probf(`*Goal*`)` finds and displays the explanation graph for *Goal* in a human-readable form. For the same goal as above, we have:

```
?- probf(hmm([a,b])).

hmm([a,b])
   <=> hmm(1,2,s0,[a,b]) & msw(init,s0)
     v hmm(1,2,s1,[a,b]) & msw(init,s1)
hmm(1,2,s0,[a,b])
   <=> hmm(2,2,s0,[b]) & msw(out(s0),a) & msw(tr(s0),s0)
     v hmm(2,2,s1,[b]) & msw(out(s0),a) & msw(tr(s0),s1)
hmm(1,2,s1,[a,b])
   <=> hmm(2,2,s0,[b]) & msw(out(s1),a) & msw(tr(s1),s0)
     v hmm(2,2,s1,[b]) & msw(out(s1),a) & msw(tr(s1),s1)
hmm(2,2,s0,[b])
   <=> hmm(3,2,s0,[]) & msw(out(s0),b) & msw(tr(s0),s0)
     v hmm(3,2,s1,[]) & msw(out(s0),b) & msw(tr(s0),s1)
hmm(2,2,s1,[b])
   <=> hmm(3,2,s0,[]) & msw(out(s1),b) & msw(tr(s1),s0)
     v hmm(3,2,s1,[]) & msw(out(s1),b) & msw(tr(s1),s1)
hmm(3,2,s0,[])
hmm(3,2,s1,[])
```

We may notice that this output corresponds to the ordered iff-formula described in §2.4.2. The last two formulas say that subgoals `hmm(3,2,s0,[])` and `hmm(3,2,s1,[])` are always true.

The built-in predicate `probef(`*Goal*`)` is the same as `probf(`*Goal*`)` except that all subgoals and switches in explanations are encoded. Also `probef(`*Goal*`,`*EGraph*`)` is the same as `probf(`*Goal*`,` *EGraph*`)` except that all the subgoals and switches in the graph are encoded. In these predicates, each

subgoal has a unique number and so does each switch instance (i.e. they are *encoded*) . The subgoal table stores the relation between subgoals and their numbers, and the switch table stores the relation between switch instances and their numbers. The following built-ins are provided to get the tables:

- `get_subgoal_hashtable(Table)` gets the subgoal hashtable which can be used to decode encoded subgoals in explanation graphs.
- `get_switch_hashtable(Table)` gets the switch hashtable which can be used to decode encoded switches in explanation graphs.

Some pretty-printing routines used internally in `probf/1` are also available as built-ins. `print_graph(`*Graph*`)` prints an explanation graph *Graph* (as a Prolog term with functors `node` and `path`, as illustrated above) to the current output stream. `print_graph(`*Graph*`,` *Options*`)` is the same as `print_graph(`*Graph*`)` except it replaces connectives with the ones specified in *Options*. *Options* can contain `and(`C_1`)`, `or(`C_2`)` and `lr(`C_3`)`, which indicates the AND connectives will be replaced with C_1, the OR connectives with C_2, and the primary connectives with C_3, respectively. For example, we can have:

```
?- probf(hmm([a,b]),EGraph),print_graph(EGraph,[lr('iff')]).

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

`print_graph(`*Stream*`,` *Graph*`,` *Options*`)` is the same as `print_graph(`*Graph*`,` *Options*`)` except the output is set to *Stream*.

4.6 Viterbi computation

4.6.1 Basic usage

By the *Viterbi computation*, we mean to get the most probable explanation E^* for a given goal G, that is, $E^* = \arg\max_{E \in \psi(G)} P(E)$, where $\psi(G)$ is a set of explanations for G. Also the probability of E^* can be obtained. Here we call them respectively the *Viterbi explanation* and the *Viterbi probability* of G.

- `viterbi(`G`)` displays the Viterbi probability of G.
- `viterbi(`G`,`P`)` returns the Viterbi probability of G to P.
- `viterbif(`G`)` displays the Viterbi probability and the Viterbi explanation for G.

- `viterbif(`G`,`P`,`$Expl$`)` returns the Viterbi probability of G to P, and a Prolog-term representation of the Viterbi explanation E^* for G to $Expl$.
- `viterbig(`G`)` is the same as `viterbi(`G`)` except that G is unified with its instantiation found in the most probable path when G is non-ground.
- `viterbig(`G`,`P`)` is the same as `viterbi(`G`,`P`)` except that G is unified with its instantiation found in the most probable path when G is non-ground.
- `viterbig(`G`,`P`,`$Expl$`)` is the same as `viterbif(`G`,`P`,`$Expl$`)` except that G is unified with its instantiation found in the most probable path when G is non-ground.

If there is no explanation for G, the call of the predicates above will fail. A Prolog-term representation of an explanation takes the same form as an explanation graph except that a node has exactly one path. That is, it takes the form:

`[node(`G'_1`,[path(`GL_1`,`SL_1`)]),...,node(`G'_n`,[path(`GL_n`,`SL_n`)])]`,

where G'_i is a subgoal in the explanation path for G, and G'_i is directly explained by subgoals GL_i and switches SL_i. This Prolog term can be printed in a human-readable form by using `print_graph/1-2` (see §4.5).

In a practical situation, we often suffer from the problem of underflow for a very long Viterbi explanation. Setting '`on`' to the `log_viterbi` flag enables log-valued Viterbi computation in which all probabilities are contained as log-valued (see §4.14 for details), and so the problem of underflow will be cleared.

4.6.2 Top-N Viterbi computation

Furthermore, in version 1.11, built-in predicates for computing *top-N Viterbi explanations* or *top-N Viterbi probabilities* are available. That is, we can obtain N explanations with the highest probabilities, where the number N can be specified in the query. This procedure is sometimes called *top-N Viterbi computation* or *N-Viterbi computation* in short. The following is a list of built-ins for top-N Viterbi computation where the specifications of the last two predicates were a bit changed since version 1.11.2:

- `n_viterbi(`N`,`G`)` displays the top-N Viterbi probabilities of the goal G.
- `n_viterbi(`N`,`G`,`Ps`)` returns the top-N Viterbi probabilities of the goal G as a list Ps.
- `n_viterbif(`N`,`G`)` diplays the top-N Viterbi explanations for the goal G.
- `n_viterbif(`N`,`G`,`$VPathL$`)` returns Prolog-term representations of the top-N Viterbi explanations for the goal G as a list $VPathL$. Each element in $VPathL$ takes the form `v_expl(`K`,`P`,`$Expl$`)`, where $Expl$ is the K-th ranked explanation and P is its generative probability.
- `n_viterbig(`N`,`G`,`P`,`$Expl$`)` unifies G with its instantiation found in the most probable path when G is non-ground. This built-in also returns the corresponding Viterbi probability and the corresponding Viterbi explanation to P and $Expl$, respectively. On backtracking, this built-in returns the answers w.r.t. the second most probable path, the third most probable path, and so on, in turn.
- `n_viterbig(`N`,`G`)` is the same as `n_viterbig(`N`,`G`,_,_)` when G is non-ground, and is the same as `n_viterbi(`N`,`G`)` when G is ground, except that the Viterbi probability will be displayed.

- `n_viterbig(N,G,P)` is the same as `n_viterbig(N,G,P,_)` when the goal G is non-ground, or returns top-N Viterbi probabilities of G to P one by one on backtracking when G is ground.

Since the implemenatation of these N-Viterbi routines is different from (and is more complicated than) that of the basic Viterbi routines such as `viterbif/1` (§4.6.1), the efficiency (both time and space) of the N-Viterbi routines seems inferior to that of the basic ones. So it is recommended to use the basic ones if you only need to the most probable explanation (i.e. $N = 1$). Besides, for the same reason, the results from `n_viterbif(1,G)` and `viterbif(G)` can be different if there are more than one Viterbi explanation of G with the same generative probability.

4.6.3 Post-processing

In version 1.11, two post-processing built-ins for Viterbi computation are newly introduced:

- `viterbi_subgoals(`*Expl*`,`*Goals*`)` extracts the subgoals $G'_1,\ldots,G'_n$ in the explanation *Expl*, and returns them as a list *Goals*.
- `viterbi_switches(`*Expl*`,`*Sws*`)` extracts the switch instances in the explanation *Expl*, and returns them as a list *Sws* (i.e. returns the concatenation of $SL_1,\ldots,SL_n$).

4.7 Hindsight computation*

4.7.1 Basic usage

A *hindsight probability* is $P_\theta(G')$, the probability of a subgoal G' for a given top-goal G.[4] Inside the system, the hindsight probability of a subgoal G' is computed as a product of the inside probability and the outside probability of G'. For illustration, let us consider the HMM program (§1.3) with string length being 4. In an HMM given some sequence, we may want to compute the probability distribution on states for every time step. The programming system computes such a probability distribution as hindsight probabilities. That is, we get the distribution at time step 2 as follows:

```
?- hindsight(hmm([a,b,a,b]),hmm(2,_,_,_)).
hindsight probabilities:
  hmm(2,4,s0,[b,a,b]): 0.013880247702822
  hmm(2,4,s1,[b,a,b]): 0.054497179729564
```

We read from above that, given a string `[a,b,a,b]`, the probability of the hidden state being `s0` at time step 2 is about 0.0139, whereas the probability of the hidden state being `s1` is about 0.0545. Generally speaking, `hindsight(`*G*`,`*GPatt*`)` writes the hindsight probabilities of *G*'s subgoals that match with *GPatt* to the current output. In a similar way, `hindsight(`*G*`,`*GPatt*`,`*Ps*`)` returns the list of pairs of subgoal and its hindsight probability to *Ps*:

```
?- hindsight(hmm([a,b,a,b]),hmm(2,_,_,_),Ps).

Ps = [[hmm(2,4,s0,[b,a,b]),0.013880247702822],
      [hmm(2,4,s1,[b,a,b]),0.054497179729564]] ?
```

[4] The name of 'hindsight' comes from an inference task with temporal models such as dynamic Bayesian networks [28].

When omitting the matching pattern *GPatt*, `hindsight(G)` writes the hindsight probabilities for all subgoals of *G* to the current output.

```
?- hindsight(hmm([a,b,a,b])).
hindsight probabilities:
  hmm(1,4,s0,[a,b,a,b]): 0.058058181772934
  hmm(1,4,s1,[a,b,a,b]): 0.010319245659452
  hmm(2,4,s0,[b,a,b]): 0.013880247702822
  hmm(2,4,s1,[b,a,b]): 0.054497179729564
  hmm(3,4,s0,[a,b]): 0.062748214275926
  hmm(3,4,s1,[a,b]): 0.005629213156460
  hmm(4,4,s0,[b]): 0.015964697775827
  hmm(4,4,s1,[b]): 0.052412729656559
  hmm(5,4,s0,[]): 0.047234593867704
  hmm(5,4,s1,[]): 0.021142833564682
```

It should be noted that, if you want the list of all pairs of subgoal and its hindsight probability, we need to run `hindsight(G,_,Ps)` (not `hindsight(G,Ps)`, in which *Ps* will be interpreted as the matching pattern).

4.7.2 Summing up hindsight probabilities

Furthermore, sometimes it is required to compute the sum of hindsight probabilities of several particular subgoals. Although this procedure may be implemented by the user with `hindsight/1-3` and additional Prolog routines, for ease of programming, the system provides a built-in utility of such summation (marginalization).

To illustrate this utility, let us consider another example that describes an extended hidden Markov model, in which there are two state variables, only one depends on another:

```
target(hmm/1).

values(init,[s0,s1,s2]).
values(out(_),[a,b]).
values(tr(_),[s0,s1,s2]).
values(tr(_,_),[s0,s1,s2]).

hmm(L):-
    str_length(N),
    msw(init,S1),
    msw(init,S2),
    hmm(1,N,S1,S2,L).

hmm(T,N,S1,S2,[]) :-T>N,!.
hmm(T,N,S1,S2,[Ob|Y]) :-
    msw(out(S2),Ob),
    msw(tr(S1),Next1),    % Transition in S1 depends on S1 itself
    msw(tr(S1,S2),Next2), % Transition in S2 depends both on S1 and S2
    T1 is T+1,
    hmm(T1,N,Next1,Next2,Y).

str_length(4).
```

Each state variable takes on three values (`s0`, `s1` and `s2`), and the state of the HMM itself is determined as a combination of the values of the two variables (hence we can say that the number of possible states is $(3 \times 3 =)$ 9). Under some parameter configuration (e.g. after learning), we can compute the hindsight probabilities for all subgoals.

```
?- hindsight(hmm([a,b,a,b])).
hindsight probabilities:
  hmm(1,4,s0,s0,[a,b,a,b]): 0.129277300817752
  hmm(1,4,s0,s1,[a,b,a,b]): 0.000547187686019
  hmm(1,4,s0,s2,[a,b,a,b]): 0.001995647575806
      :
  hmm(5,4,s2,s0,[]): 0.038066015885796
  hmm(5,4,s2,s1,[]): 0.030640117459401
  hmm(5,4,s2,s2,[]): 0.013513864959245
```

Now let us suppose that we want to marginalize out the second state variable (i.e. the fourth argument). It is achieved by running `hindsight_agg/2` as follows:

```
?- hindsight_agg(hmm([a,b,a,b]),hmm(integer,_,query,_,_)).
hindsight probabilities:
  hmm(1,*,s0,*,*): 0.131820136079577
  hmm(1,*,s1,*,*): 0.012972174566148
  hmm(1,*,s2,*,*): 0.050479679093070
  hmm(2,*,s0,*,*): 0.031258649883958
  hmm(2,*,s1,*,*): 0.116570845419607
  hmm(2,*,s2,*,*): 0.047442494435231
      :
  hmm(5,*,s0,*,*): 0.041483563280137
  hmm(5,*,s1,*,*): 0.071568428154217
  hmm(5,*,s2,*,*): 0.082219998304441
```

In the above, `hmm(integer,_,query,_,_)` is a control statement that means "group subgoals according to the first (integer) argument, and then, within each group, sum up the hindsight probabilities among the subgoals that has the same pattern in the argument specified by `query` (i.e. the third argument). In general, `query` is a reserved constant symbol that specifies an argument of interest, and the arguments specified by unbound variables are ineffective in grouping and then bundled up in summation.

For the control of grouping, 6 reserved constant symbols are defined: `integer`, `atom`, `compound`, `length`, `d_length`, `depth`. The first 3 symbols just mean grouping by exact matching[5] for the integer argument, the argument with an atoms, and the argument with a compound term, respectively. On the other hand, `length` will make groups according to the length of a list in the corresponding argument. Similarly, `d_length` considers the length of a difference list (which is assumed to take the form D_0−D_1), and `depth` considers the term depth. The last 3 symbols would be useful if we have no appropriate argument for exact matching. For example, we can make grouping by the list length in the fifth argument, instead of the first argument (`L`−n means that the length is n):

```
?- hindsight_agg(hmm([a,b,a,b]),hmm(_,_,query,_,length)).
hindsight probabilities:
  hmm(*,*,s0,*,L-0): 0.041483563280137
  hmm(*,*,s1,*,L-0): 0.071568428154217
  hmm(*,*,s2,*,L-0): 0.082219998304441
```

[5] The matching is done by `==/2`, where the variables in the distinct subgoals are considered as different and thus do not match with each other.

```
    :
  hmm(*,*,s0,*,L-4): 0.131820136079577
  hmm(*,*,s1,*,L-4): 0.012972174566148
  hmm(*,*,s2,*,L-4): 0.050479679093070
```

The arguments in the control statement, which are neither variable nor reserved constant symbols, will be used for filtering, that is, they are considered as matching patterns, just as in `hindsight/1-3`. For example, to get the distribution at time step 3, we run:

```
?- hindsight_agg(hmm([a,b,a,b]),hmm(2,_,query,_,_)).
hindsight probabilities:
  hmm(2,*,s0,*,*): 0.031258649883958
  hmm(2,*,s1,*,*): 0.116570845419607
  hmm(2,*,s2,*,*): 0.047442494435231
```

Besides, `hindsight_agg(`*G*`,`*GPatt*`,`*Ps*`)` will return to *Ps* a Prolog term representing the above computed results, where '`*`' can be handled just as a Prolog's constant symbol.

By default, each group in the computed result is sorted in the Prolog's standard order with respect to the subgoals. When setting '`by_prob`' to the `sort_hindsight` flag (§4.14), the group will be sorted by the magnitude of the hindsight probabilities.

Furthermore, `chindsight/1-3` and `chindsight_agg/2-3` compute the conditional hindsight probabilities $P_\theta(G'|G) = P_\theta(G')/P_\theta(G)$ instead of $P_\theta(G')$, where G is a given top-goal and G' is its subgoal.[6] The usages for them are respectively the same as those for the `hindsight` or the `hindsight_agg` predicates with the same arity. Conditional hindsight probabilities can be seen as a restricted version of conditional probabilities. For instance, in the example program which represents a Bayesian network (§7.3), we compute conditional probabilities on the network by using conditional hindsight probabilities.

4.7.3 Computing goal probabilities all at once

One interesting use of the `hindsight` predicates is to compute the probabilities of several goals all at once. For example, in the HMM program, let us compute the conditional distrubution on the strings that have a prefix '`ab`'. To do this, we compute the hindsight probabilities of subgoals of `hmm([a,b,_,_])`, which take the form `hmm(_)`:

```
?- chindsight(hmm([a,b,_,_]),hmm(_)).
conditional hindsight probabilities:
  hmm([a,b,a,a]): 0.150882383997529
  hmm([a,b,a,b]): 0.375321053537642
  hmm([a,b,b,a]): 0.162375115518536
  hmm([a,b,b,b]): 0.311421446946293
```

On the other hand, in the blood type program, we may compute the distribution on blood types:

```
?- hindsight(bloodtype(_),bloodtype(_)).
hindsight probabilities:
```

[6] Generally speaking, we need to say that what is computed by the `chindsight` predicates is *not* a probability but $E_\theta[G'|G]$, the expected occurrences of G' given G, which can exceed unity. This is because, in a general case, some subgoal G' can appear more than once in G's proof tree. On the other hand, in typical programs of HMMs, PCFGs (with neither ε-rule nor chain of unit productions) or Bayesian networks, each of subgoals should appear just once, hence $E_\theta[G'|G]$ can be considered as a conditional probability, say $P_\theta(G'|G)$. The discussion in this footnote also holds for the `hindsight` predicates.

```
  bloodtype(a): 0.403912166491685
  bloodtype(ab): 0.095321638418523
  bloodtype(b): 0.204152312431112
  bloodtype(o): 0.296613882658681
```

Furthermore, by giving 'by_prob' to the sort_hindsight flag (§4.14), we can list goals in descending order of their probabilities:

```
?- set_prism_flag(sort_hindsight,by_prob).
  :
?- hindsight(bloodtype(_),bloodtype(_)).
hindsight probabilities:
  bloodtype(a): 0.403912166491685
  bloodtype(o): 0.296613882658681
  bloodtype(b): 0.204152312431112
  bloodtype(ab): 0.095321638418523
```

Of course it is important to note that, since we use a top goal which contains logical variables, the computational cost (especially the size of memory consumption) can be very large for some programs.

4.8 Parameter learning

4.8.1 Maximum likelihood estimation and EM learning

The programming system supports parameter learning called *maximum likelihood estimation* (ML estimation). That is, we can learn the parameters θ of switches buried in a program from data. More concretely, in ML estimation, the system tries to find the parameters θ that maximize the likelihood $\prod_t P_\theta(G_t)$, the product of probabilities of given observed goals (i.e. *training data*).[7]

If we know that there is just one way to yield each observation G_t, ML estimation of the parameters θ is quite easy. In such a case, G_t has only one explanation E_t (a conjunction of switch instances which used to generate G_t; see §2.4.2 for illustrated details of explanations), and hence it is only required to count up $C_{i,v}$, the number of occurrences of msw(i,v) among all E_t, and then to get the estimate $\hat{\theta}_{i,v} = C_{i,v} / \sum_{v'} C_{i,v'}$ of the parameters of the switch.

The situation above is frequently seen in *supervised learning* where we say each observation G_t is a *complete data*. In partially observing situation such as *unsupervised* or *semi-supervised* learning, on the other hand, we can consider two or more ways to yield G_t (i.e. G_t has two or more explanations). To deal with such partially observed goals (*incomplete data*) as observations, the programming system provides the utility of *EM learning* .

In the system, EM learning is conducted in two phases: the first phase searches for all explanations for observed data G_t (i.e. make an explanation search for G_t; see §2.4.2), and the second phase finds an ML estimate of θ by using the EM algorithm. The EM algorithm is an iterative algorithm:

Initialization step:
Initialize the parameters as $\theta^{(0)}$, and then iterate the next two steps until the likelihood converges.

Expectation step:
For each msw(i,v), compute $\hat{C}_{i,v}$, the expected occurrences of msw(i,v) under the parameters $\theta^{(m)}$.

[7] It should be noted here that each goal G_t is assumed to be observed independently.

Maximization step:

Using the expected occurrences, update each parameter by $\hat{\theta}_{i,v}^{(m+1)} = \hat{C}_{i,v} / \sum_{v'} \hat{C}_{i,v'}$ and then increment m by one.

When the likelihood converges, the system stores the estimated parameters to its internal database, and then we can make further probabilistic inferences based on these parameters. The threshold ε is used for judging convergence, that is, if the difference between the likelihood under the updated parameters and one under the original parameters is less than ε (i.e. sufficiently small), we can think that the likelihood converges. The value of ε can be configured by the `epsilon` flag (see §4.14; the default is 10^{-4}).

4.8.2 Maximum a posteriori estimation

As mentioned in §1.5, the programming system also supports *maximum a posteriori estimation* (MAP estimation) for parameter learning, which tries to find parameters θ that maximize, $P(\theta \mid G_1, \ldots, G_T) \propto P(\theta) \prod_t P_\theta(G_t)$, the a posteriori probability of the parameters given training data from a Bayesian point of view.[8] In MAP estimation, the system assumes the prior distribution $P(\theta)$ follows a Dirichlet distribution $P(\theta) = \frac{1}{Z} \prod_{i,v} \theta_{i,v}^{\alpha_{i,v}-1}$, where Z is a normalizing constant and each $\alpha_{i,v}$ is a hyperparameter of the Dirichlet distribution, which corresponds to `msw(i,v)`. Then in estimating parameters, it introduces $\delta_{i,v} = (\alpha_{i,v} - 1)$, as a *pseudo count* for each `msw(i,v)`.[9]

This term comes from the fact that, in the complete-data case, each parameter is estimated by $\hat{\theta}_{i,v} = (C_{i,v} + \delta_{i,v}) / (\sum_{v'} (C_{i,v'} + \delta_{i,v'}))$. Similarly, in the incomplete-data case, each parameter is updated by the EM algorithm with $\hat{\theta}_{i,v} = (\hat{C}_{i,v} + \delta) / (\sum_{v'} (\hat{C}_{i,v'} + \delta_{i,v'}))$, until the a posteriori probability converges. Practically speaking, even for small training data (compared to the number of parameters to be estimated), this pseudo count guarantees all estimated parameters to be positive, and hence we can escape from the problem of so-called data sparseness or zero frequency. If all pseudo count are zero, the MAP estimation is just an ML estimation, and it is sometimes called *Laplace smoothing* when all pseudo counts are set to be unity. We can configure these pseudo counts individually via the built-ins for handling switches (§4.2).

4.8.3 Running learning commands

The built-in `learn(`*Goals*`)` takes *Goals*, a list of observed goals, and estimates the parameters of the switches to maximize the likelihood of the goals. For example, in the direction program (§1.1), we make the program learn with three observed goals:

```
?- learn([direction(left),direction(right),direction(left)]).
```

Then we may receive messages like:

```
#goals: 0(2)
Exporting switch information to the EM routine ...
#em-iterations: 0(2) (Converged: -1.909542505)
Statistics on learning:
        Graph size: 2
        Number of switches: 1
```

[8] In this view, the parameterized probability distribution $P_\theta(G)$ which we used so far should be considered as $P(G|\theta)$, a conditional probability given the parameters.

[9] We use the term 'pseudo counts' in the sense of ones used in the MAP estimator, and for various compatibilities, it is designed that the users are expected to configure the hyperparameters $\alpha_{i,v}$ through the corresponding pseudo counts $\delta_{i,v}$ (even in VB learning). In the Bayesian estimator, on the other hand, hyperparameters $\alpha_{i,v}$ themselves can be considered as pseudo counts. Another confusing issue is that $\delta_{i,v}$ is not allowed to be negative in version 1.11. Of course this restriction is reasonable for MAP estimation, but theoretically it should be noted that the prior distribution $P(\theta)$ itself is defined for $\alpha_{i,v} \geq 0$ (equivalently, $\delta_{i,v} \geq -1$).

```
        Number of switch instances: 2
        Number of iterations: 2
        Final log likelihood: -1.909542505
        Total learning time: 0.004 seconds
        Explanation search time: 0.004 seconds
        Total table space used: 1088 bytes
Type show_sw or show_sw_b to show the probability distributions.
```

The line beginning with `#goals` shows the number of *distinct* goals whose explanation searches have been done. The line beginning with `#iterations` show the number of EM iterations. Since each of `direction(left)` and `direction(right)` has just one explanation `msw(coin,head)` and `msw(coin,tail)` respectively (i.e. they are complete data), EM learning finishes with only two iterations. After learning, the statistics on learning are displayed. These statistics can also be obtained as Prolog terms (see §4.9). We may confirm the estimated parameters by `show_sw/0` (§4.2.5):

```
?- show_sw.
Switch coin: unfixed: head (0.666666666666667) tail (0.333333333333333)
```

This result indicates that the estimated parameters are $\hat{\theta}_{\texttt{coin,head}} = 2/3$ and $\hat{\theta}_{\texttt{coin,tail}} = 1/3$. It is easily seen that this is because, for the whole training data, we have the explanation `msw(coin,head)` for two goals, and `msw(coin,tail)` for one goal.

The built-in `learn/0` can be used only when the program gives the data file declaration (§2.6.2) which specifies the file containing observed goals. The built-in `learn` (with no arguments) is the same as `learn(Goals)` except that the observed goals are read from the file specified by the data file declaration (§2.6.2). For example, assume the file '`direction.dat`' contains the following two unit clauses:

```
direction(left).
direction(right).
```

and the program contains the declaration:

```
data('direction.dat').
```

Then running the command `learn/0` is equivalent to:

```
?- learn([direction(left),direction(right)]).
```

Furthermore, we can specify the data by goal-count pairs by using `count/2`. That is, the data

```
count(direction(left),3).
count(direction(right),2).
```

are equally treated as below:

```
direction(left).
direction(left).
direction(left).
direction(right).
direction(right).
```

Such goal-count pairs can also be given to `learn/1`:

```
?- learn([count(direction(left),3),count(direction(right),2)]).
```

In the programming system, the default learning method is ML estimation (§4.8.1). On the other hand, as mentioned above, we can enable MAP estimation (§4.8.2) by setting the pseudo count $\delta_{I,V}$, which is greater than zero, for each switch instance `msw(I,V)`. For example, let us set all pseudo counts as 0.5. There are two typical cases:

- No random switches have been registered into the internal database yet (§4.2.2). In such a case, we set the default pseudo counts as follows:

  ```
  ?- set_prism_flag(default_sw_h,0.5).
  ```

  With this setting, the pseudo counts of the switches found (and registered) in the next learning will be all set to 0.5.

- The switches whose parameters are the target of learning have already been registered. In such a case, we use `set_sw_all_h/2` to change the pseudo counts of these switches as follows:

  ```
  ?- set_sw_all_h(Patt,0.5).
  ```

  In the query above, *Patt* is the matching pattern of the target switches. See §4.2.3 for the detailed usage of `set_sw_all_h/2` and other built-ins for setting the pseudo counts of switches.

Note that the settings above can co-exist. Finally, the learning command is invoked in the same way as that of ML estimation:

```
?- learn([direction(left),direction(right),direction(left)]).

#goals: 0(2)
Exporting switch information to the EM routine ...
#em-iterations: 0(2) (Converged: -2.646252953)
Statistics on learning:
        Graph size: 2
        Number of switches: 1
        Number of switch instances: 2
        Number of iterations: 2
        Final log of a posteriori prob: -2.646252953
        Total learning time: 0.004 seconds
        Explanation search time: 0.000 seconds
        Total table space used: 1088 bytes
Type show_sw or show_sw_b to show the probability distributions.
```

It may be confusing that '`log of a posteriori prob`' in the messages above is indeed the log of *unnormalized* a posteriori probability of the observed goals (i.e. the sum of the log-likelihood and the log-valued prior probability[10]), which is the substantial target of maximization. Finally we find the estimated parameters are $\hat{\theta}_{\texttt{coin,head}} = (2+0.5)/(3+2*0.5) = 0.625$ and $\hat{\theta}_{\texttt{coin,tail}} = (1+0.5)/(3+2*0.5) = 0.375$.

```
?- show_sw.
Switch coin: unfixed_p: head (p: 0.625000000) tail (p: 0.375000000)
```

[10] To be precise, suppose we have some predefined probabilistic model and let D be the data at hand. Then, from a Bayesian point of view, a posteriori probability of parameter θ given D is computed by $P(\theta \mid D) = P(\theta)P(D \mid \theta)/P(D)$, where $P(\theta)$ is a prior probability of θ, and $P(D \mid \theta)$ is the likelihood of D under θ. As stated in §4.8.2, $P(\theta)$ is assumed to follow a Dirichlet distribution, and the 'unnormalized' a posteriori probability is just $P(\theta \mid D)$ ignoring the constant factors with respect to θ (i.e. the constant factors in the Dirichlet distribution and $P(D)$). Of course, such an unnormalized version can be used only for relative comparison such as a judgment of the EM algorithm's convergence, or selecting the 'best' parameters in multiple runs of the EM algorithm (§4.8.4).

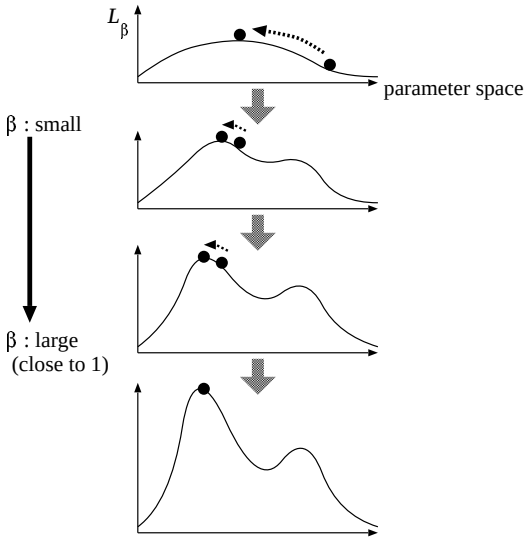

Figure 4.1: Image of the deterministic annealing EM algorithm.

Let us recall that the above example is a program with complete data. When EM learning is conducted with incomplete data, the procedure is the same as above, but the larger number of iterations may be required for complex models or large data. If some parameters are fixed (§4.2.4), they will not be updated in the process of learning. Please note however that it is not allowed to fix any parameters at zero in MAP estimation.

4.8.4 Avoiding undesirable local maxima

It is only guaranteed by the EM algorithm that each iteration monotonically increases the likelihood (or a posteriori probability), and hence we often face the problem of being trapped in undesirable local maxima. In the current version, the system provides two solutions. The first one is quite simple. That is, we try multiple runs of the EM algorithm by restarting with different initial parameters. The final estimates are the ones with the highest likelihood (or a posteriori probability) among all trials. The number of such trials can be specified by the `restart` flag (see §4.14). For example, if you wish to restart for 10 times, just type:

```
?- set_prism_flag(restart,10).
```

Another solution is to use the deterministic annealing EM (DAEM) algorithm [41]. It is easy to see that, in the usual EM algorithm, the final estimate of the parameters depends on the choice of initial parameters. On the other hand, the DAEM algorithm is designed to reduce an undesirable influence from the initial parameters in the early stage of EM iterations. In the rest of this section, we briefly describe the DAEM algorithm.

Let us consider first that we have the observed data (a multiset of observed goals) $D = \{G_1, G_2, \ldots, G_T\}$, and $\psi(G_t)$ is the set of explanations for the t-th observed goal. Then, from analogy to statistical mechanics, the free energy is introduced as:

$$\mathcal{F}_\beta = -\frac{1}{\beta}\sum_{t=1}^{T}\log\sum_{E\in\psi(G_t)} P_\theta(E)^\beta, \tag{4.1}$$

where β is the *inverse temperature* which controls the influence from the initial parameters. The DAEM algorithm is derived so that it tries to minimize the free energy $\mathcal{F}_\beta$ at each temperature $1/\beta$. Fig. 4.1 shows an expected behavior of the DAEM algorithm, where L_β is introduced as $-\mathcal{F}_\beta$ (then we will try to maximize L_β). In the DAEM algorithm, we start from the small β, under which L_β is expected to have a smooth shape, and hopefully has only one local maximum (i.e. the global maximum). So under the smaller β, we may be able to find the global maximum or good local maxima. When β increases, on the other hand, the shape of L_β changes (becomes sharper), and hence we should continue to update the parameters by EM iterations. Please note that the starting point of these EM iterations is expected to be more promising than the initial parameters. Finally we perform EM iterations at $\beta = 1$, which is equivalent to the usual EM iterations.

For an effective use of the DAEM algorithm, the annealing schedule is important. In PRISM, following [41], we start from $\beta_0 = \beta_{\text{init}}$ and then update β by the update rule $\beta_{t+1} \leftarrow \beta_t \cdot \beta_{\text{rate}}$, where β_{init} and β_{rate} are given by the user (the default values are 0.1 and 1.5, respectively). In our experience, the appropriate annealing schedule seems to vary depending on the model and the observed data.

The DAEM algorithm will be enabled when the `daem` flag is set as '`on`', and controlled by the `itemp_init` and the `itemp_rate` flags which correspond to β_{init} (the initial value) and β_{rate} (the increasing rate), respectively. For example, the followings will enable the DAEM algorithm with $\beta_{\text{init}} = 0.3$ and $\beta_{\text{rate}} = 1.2$.

```
?- set_prism_flag(daem,on).
?- set_prism_flag(itemp_init,0.3).
?- set_prism_flag(itemp_rate,1.2).
```

While the DAEM algorithm running, the programming system displays the inverse temperature at the moment it is updated. Namely, '$<\beta_t>$' ($t = 0, 1, \ldots$) will be displayed in the line beginning with '`#em-iterations`'. For example, in the HMM program, we will see the messages as follows:

```
?- prism(hmm).
  :
?- set_prism_flag(daem,on).
  :
?- set_prism_flag(itemp_init,0.3).
  :
?- set_prism_flag(itemp_rate,1.2).
  :
?- hmm_learn(100).

#goals: 0.........(96)
Exporting switch information to the EM routine ...
#em-iterations: <0.300>0<0.360><0.432><0.518><0.622>.<0.746><0.896>
<1.000>(14) (Converged: -692.178867976)
Statistics on learning:
        Graph size: 5832
        Number of switches: 5
        Number of switch instances: 10
        Number of iterations: 14
        Final log likelihood: -692.178867976
        Total learning time: 0.024 seconds
        Explanation search time: 0.004 seconds
        Total table space used: 769232 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
```

Table 4.1: Available statistics on the explanation graphs, on learning, and on the probabilistic inference other than learning

graph_statistics(*Name*,*Stat*)	
Name	*Stat*
num_subgraphs	Number of subgraphs in the explanation graphs
num_nodes	Total number of nodes in the explanation graphs (the sum of num_goal_nodes and num_switch_nodes)
num_goal_nodes	Number of subgoal nodes
num_switch_nodes	Number of switch nodes
avg_shared	Average number of nodes which shares a particular node (note: this average value can be misleading if there is a node which is shared by extremely many nodes)
learn_statistics(*Name*,*Stat*)	
Name	*Stat*
log_likelihood	Log likelihood (only available in ML/MAP)
log_post	Log of unnormalized a posteriori probability (in MAP)
log_prior	Log of a priori probability (in MAP)
lambda	Same as log_likelihood (in ML) or log_post (in MAP)
num_switches	Number of occurred switches in the last learning
num_switch_values	Number of occurred switch values in the last learning
num_parameters	Number of free parameters in the last learning
num_iterations	Number of EM iterations in the last learning
goals	List of goals used in the last learning
goal_counts	List of goal-count pairs used in the last learning
bic	Bayesian Information Criterion (in ML/MAP, see §4.10)
cs	Cheeseman-Stutz score (in MAP, see §4.10)
free_energy	Variational free energy (in VB, see §5.1)
learn_time	Total time consumed by the built-in (in seconds, including miscellaneous jobs)
learn_search_time	Time consumed by the explanation search (in seconds)
em_time	Time consumed by the EM algorithm (in seconds)
infer_statistics(*Name*,*Stat*)	
Name	*Stat*
infer_time	Total time consumed by the built-in (in seconds, including miscellaneous jobs)
infer_search_time	Time consumed by the explanation search (in seconds)
infer_calc_time	Time consumed by the numerical calculation (in seconds)

4.9 Getting statistics on probabilistic inferences

In version 1.11, the routines for accessing the statistics on probabilistic inferences in PRISM were entirely reorganized.[11] The built-ins graph_statistics/0, learn_statistics/0 and infer_statistics/0 display the statistics on the explanation graphs, on learning, and on the probabilistic inferences other than learning. prism_statistics/0 displays all statistics displayed by the above three built-ins.

To get an individual statistic, we can respectively use graph_statistics(*Name*,*Stat*), learn_statistics(*Name*,*Stat*), infer_statistics(*Name*,*Stat*) and prism_statistics(*Name*,*Stat*), where *Name* is the name of a statistic and *Stat* is the value of the statistic. For example, to get the time consumed by learning, we may run:

```
?- prism_statistics(learn_time,T).
```

[11] The built-ins such as get_log_likelihood/1 have been removed.

When calling `prism_statistics(`*Name*`,` *Stat*`)` with *Name* being unbound, we can get all available statistics one after another by backtracking (this behavior also applies to `graph_statistics/2`, `learn_statistics/2` and `infer_statistics/2`). The available statistics are shown in Table 4.1.[12] Combining these statistics with the facilities for saving/restoring switch information (§4.2.7), it is possible to a customized routine for multiple runs of the EM algorithm (§4.8.4).

In addition, the observed goals (with their counts and frequencies) used in the last learning is displayed by `show_goals`, and can be obtained as Prolog terms by `get_goals/1` and `get_goal_counts/1`:

```
?- show_goals.
Goal direction(right) (count=1, freq=33.333%)
Goal direction(left) (count=2, freq=66.667%)
Total_count=3

?- get_goals(Gs).
Gs = [direction(left),direction(right)] ?

?- get_goal_counts(GCs).
GCs = [[direction(left),2,66.666666666666657],
       [direction(right),1,33.333333333333329]] ?
```

4.10 Model scoring*

In practical applications, we often face a problem of *model selection* — that is, we need to select the model that fits best the data at hand, from possible candidates. In PRISM, the programming system just provides three Bayesian scores called *Bayesian Information Criterion* (BIC) [39], the *Cheeseman-Stutz (CS) score* [5] and *variational (negative) free energy*. The first two are used after ML (§4.8.1) or MAP (§4.8.2) estimation, whereas the last one is used after variational Bayesian learning (Chapter 5). Generally speaking, these Bayesian scores are known to be 'deterministic' approximations of $\log P(D \mid M)$, log of the *marginal likelihood* of the observed data D under the model M, and so in model selection with some Bayesian score (BIC, for example), we compare the model candidates according to the score (i.e. the model with the larger score is considered to be better).

To be more concrete, let us consider first that the joint distribution $p(D, M, \theta)$ of the observed data D, a probabilistic model M, and its parameters θ. In PRISM, D is a multiset of observed goals $G_1, G_2, \ldots, G_T$, and M corresponds to the modeling part of a PRISM program. $p(D, M, \theta)$ is then factored as $p(D \mid M, \theta)p(\theta \mid M)p(M)$ by the chain rule, where $p(M)$ is the *prior distribution* of the model M, $p(\theta \mid M)$ is the *a posteriori distribution* of the parameters θ of the model M, and $p(D \mid M, \theta)$ is the *likelihood* of the data D based on the model M with the parameters θ. Then, in model selection, our goal is to find the most probable model M^* based on the data D at hand, that is, we attempt to find M^* such that:

$$M^* = \mathrm{argmax}_M\ p(M \mid D) = \mathrm{argmax}_M\ \frac{p(D \mid M)p(M)}{p(D)} = \mathrm{argmax}_M\ p(D \mid M),$$

where we assume $p(M)$ to be uniform for simplicity. Now the goal is reduced to finding M $(= M^*)$ that maximizes $p(D \mid M)$. $p(D \mid M)$ is commonly called the *marginal likelihood* of D given M, and is used as a Bayesian score for model selection. The marginal likelihood can be interpreted as the expectation (or

[12] The number of occurred switch instances is just the sum of the numbers of possible outcomes of switches occurred in all explanations for all observed goals. This means that the switch instances not occurring in any of these explanations are also taken into account there. The number of free parameters is just computed as the number of occurred switch instances subtracted by the number of occurred switches.

the average) of the likelihood $p(D \mid M, \theta)$ with respect to the prior distribution $p(\theta \mid M)$:

$$p(D \mid M) = \int_\Theta p(D, \theta \mid M) d\theta = \int_\Theta p(D \mid M, \theta) p(\theta \mid M) d\theta = \langle p(D \mid M, \theta) \rangle_{p(\theta|M)} .$$

If the observed data were complete data D_c, where each $d \in D_c$ is a pair (G_t, E_t) of the t-th goal G_t and its *unique* explanation E_t, then $p(D_c \mid M)$ is obtained in closed form (see [9] for the case with a Bayesian network). On the other hand, when the data is incomplete, as in the case of probabilistic clustering, the integral in the above equation is difficult to compute. As mentioned above, BIC and the CS score are the approximations of log of the marginal likelihood, which are defined as:

$$\begin{aligned} Score_{\mathrm{BIC}}(M) &\stackrel{\mathrm{def}}{=} p(D \mid M, \hat{\theta}_{\mathrm{MAP}}) - \frac{|\theta|}{2} \log N \\ Score_{\mathrm{CS}}(M) &\stackrel{\mathrm{def}}{=} p(\tilde{D}_c \mid M) - p(\tilde{D}_c \mid M, \hat{\theta}_{\mathrm{MAP}}) + p(D \mid M, \hat{\theta}_{\mathrm{MAP}}), \end{aligned}$$

where N is the total size of dataset, $|\theta|$ denotes the number of free parameters, $\hat{\theta}_{\mathrm{MAP}}$ is the MAP estimate of the parameters, and $\tilde{D}_c$ is pseudo complete data whose sufficient statistics are the expected occurrences of random switches obtained by the EM algorithm. See [6] for more detailed descriptions about BIC and the CS score. The definition of the variational free energy will be shown in Chapter 5. In the programming system, `learn_statistics(bic,`*Score*`)` or `learn_statistics(cs,`*Score*`)` (§4.9) will provide us BIC and the CS score after ML or MAP learning (§4.8.3) with some observed goals D.

4.11 Handling failures*

The programming system provides a facility of dealing with failure in generative models. The background and general descriptions are given in §1.4 and §2.4.4, and so in this section, we will concentrate on the usage of this facility.

For example, let us consider again the program which takes into account the agreement in the results of coin-tossings, and suppose that the program is contained in the file named '`agree.psm`':

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

failure :- not(success).
success :- agree(_).

agree(A):-
    msw(coin(a),A),
    msw(coin(b),B),
    A=B.
```

See §2.4.4 for a detailed reading of this program. Like the program above, for the model that may cause failures, we need to define the predicate `failure/0` which describes all generation processes leading to failure. In a probabilistic context, the sum of probabilities of successful generation processes and the probability that `failure/0` holds (called *failure probability*) should always sum to unity. Of course it is possible to define `failure/0` in a usual manner of PRISM programming, but the definition should be much simpler if we can appropriately use the negation `not/1` as above.

When some negation `not/1` occurs in a program, the system first attempts to eliminate it from the program by applying a certain type of program transformation, called First Order Compiler (FOC) [29], to produce an ordinary PRISM program. If this transformation is successful, PRISM then loads the transformed program into memory. `prismn(`*File*`)` carries out this two-staged process automatically (please note that '`n`' is added to the last). *File* must include a definition of the `failure/0` predicate described above.

By default, the transformed program is stored into the file 'temp' in the current working directory. If you prefer another file, say *TempFile*, prismn(*File*,*TempFile*) should be used instead. For example, for the agreement program above,

```
?- prismn(agree).
```

loads 'agree.psm' into memory. The user can check the result of the transformation by FOC, looking at 'temp'. To estimate the parameters of switches for this program, include a special symbol failure as data:

```
?-learn([failure,agree(heads),agree(heads),agree(tails)]).
```

For a batch execution (§3.7) of the program that deals with failures, we need to run a command 'upprism prismn:*foo*' instead of 'upprism *foo*'.

foc/2 is the built-in predicate internally invoked by prismn/1-2. That is, foc(*File*,*TempFile*) eliminates negation (or more generally universally quantified implications) and generates executable code into *TempFile*. For example, we can find the program 'max' in the 'foc' directory obtained by extracting the package. With the following query, we transform 'max' into 'temp', and load the translated program:

```
?- foc(max,temp),[temp].
```

Allowing negation in the clause body is equivalent to allowing arbitrary first-order formulas as goals which are obviously impossible to solve in general. So foc/2 may fail depending on the source program. Users are advised to look into the examples of foc/2 usage in the 'foc' directory.

It is unfortunate that the deterministic annealing EM (DAEM) algorithm (§4.8.4) does not work with the failure-adjusted maximization (FAM) algorithm. This is because, under $\beta < 1$ (β is the inverse temperature used in the DAEM algorithm), the failure probability can exceed unity, whereas the FAM algorithm is derived from the property of a negative binomial distribution under the condition that the failure probability is less than unity [12].

4.12 Avoiding underflow*

4.12.1 Background

For large data, such as very long sequential data, we often suffer from the problem that the probability of some explanation goes into underflow. For Viterbi computation (§2.3 or §4.6), since no summations of probabilities arise in the computation, we have an easy solution — keeping probabilities as log-valued.

For the probabilistic inferences other than Viterbi computation, on the other hand, the programming system supports two methods — *constant scaling* and *log-valued probability computation*.[13] In the former, each time we multiply a parameter of msw/2 to the probability of some explanation, we also multiply a constant number (greater than one) to avoid underflow. Hereafter this number is called a *scaling factor* . It is assumed that the users can give some appropriate constant number as the scaling factor.

4.12.2 Using methods for avoiding underflow

For Viterbi computation, setting 'on' to the log_viterbi flag enables the log-valued Viterbi computation. See §4.14 for handling execution flags. The returned probability is log-valued.

For the other probabilistic inferences, the methods described in the previous section (§4.12.1) are specified by the scaling flag. This flag takes on none, const and log_exp. The value none

[13] To make the system's internal architecture simple, the method called layered-scaling was removed in version 1.11.

(default) means that we do not care about underflow. `const` means doing the constant scaling. By specifying `log_exp`, we perform the log-valued probability computations. For example, the following query enables the constant scaling:

```
:- set_prism_flag(scaling,const).
```

Keep in mind that, for either the constant scaling or the log-valued probability computation, the probabilities returned by built-ins that computes probabilities (§4.4 and §4.7) will be log-valued. For the constant scaling, we need to tell the scaling factor to the system, by specifying the `scaling_factor` flag. For example, the following specifies it to be 2.0 as follows (the default is 8.0):

```
:- set_prism_flag(scaling_factor,2.0).
```

4.12.3 Efficiency

It is desired to understand that the methods for avoiding underflow bring loss of computation time. For the probabilistic inferences other than Viterbi computation, the constant scaling (specified by '`const`') runs faster than the log-valued probability computation (specified by '`log_exp`') since we only need to multiply a constant number for each occurrence of switch instances. On the other hand, the log-valued probability computation requires additional computation time to call the logarithmic and the exponential functions.

4.13 Keeping the solution table*

Since version 1.10, when the `clean_table` flag is set as '`off`' (see §4.14), the programming system will come *not* to clean up the solution table. On the other hand, if this flag is set as '`on`', which is the default, the programming system will automatically clean up all past results of explanation search (say, solutions) in the solution table[14] when invoking a routine that performs explanation search (i.e. learning (§4.8) and other probabilistic inferences (§4.4, §4.6, §4.7)). Keeping and reusing the past solutions can be significantly useful when we only attempt to repeatedly compute the probabilities of some specific goal repeatedly with different parameter settings. Of course, the efficiency is gained at the price of memory space, so we need to care about the size of memory (i.e. the table area).

4.14 Execution flags

4.14.1 Handling execution flags

Since version 1.9, the programming system provides dozens of execution flags to change its behavior. The below is the usage of these execution flags:

- *Setting flags*:

  Flags are set by the command `set_prism_flag(`*FlagName*`,`*Value*`).` When writing the query "`:- set_prism_flag(`*FlagName*`,`*Value*`).`" in a program, the flag will be set when the program is loaded. Also, flags can be specified by the command `prism/2` (§3.3), that is, by running:

  ```
  ?- prism([FlagName=Value],Filename).
  ```

[14] Internally, the system calls both `initialize_table/0` (B-Prolog's built-in) and the routine that erases the ID tables of PRISM's own. So it is not guaranteed for the system to work when you call only `initialize_table/0` at an arbitrary timing.

- *Printing flags*:

  `show_flags/0` will print the current values of flags.

- *Getting flag values*:

  By `get_prism_flag(`*FlagName*`,X)`, you can get the value of *FlagName* as `X`. If we call this with *FlagName* being unbound, all available flags and their values are retrieved one after another by backtracking.

- *Running built-ins based on flags*:

  For example, to enable the log-valued version of Viterbi routine (§4.12), we need to run `set_prism_flag(log_viterbi,on)` beforehand. Also we may run as a query `set_prism_flag(epsilon,`*E*`)` in advance to conduct EM learning with the threshold of convergence being *E* (§4.8.1).

4.14.2 Available execution flags

Here we list the available execution flags in the alphabetical order. Please note that this list also includes ones for the functions described in later chapters.

- `clean_table` (possible values: `on` and `off`; default: `on`) — the flag for automatic cleaning of the solution table (see §4.13 for details). If this flag is set as '`on`', the programming system will automatically clean up all past solutions in the solution table when invoking any routine that executes the explanation search. On the other hand, with this flag turned '`off`', we can keep the past solutions.

- `daem` (possible values: `on` and `off`; default: `off`) — the flag for enabling the deterministic annealing EM (DAEM) algorithm (see §4.8.4). If this flag is set as '`on`', the programming system will invoke the DAEM algorithm while EM learning. On the other hand, with this flag turned '`off`', it will be disabled.

- `default_sw` (possible values: `none`, `uniform`, `f_geometric`, `f_geometric(`*Base*`)`, `f_geometric(`*Base*`,`*Type*`)`; default: `uniform`) — the default distribution for parameters. If `none` is set, we have no default distribution for parameters, and hence as in the versions earlier than 1.9, we cannot make sampling or probability computation without an explicit parameter setting (via `set_sw/2`, and so on) or learning. `uniform` means that the default distribution for each switch is a uniform distribution. `f_geometric(`*Base*`,`*Type*`)` means the default distribution for each switch is a finite geometric distribution where *Base* is its base (an integer greater than 1) and *Type* is `asc` (ascending order) or `desc` (descending order). For example, when the flag is set as `f_geometric(2,asc)`, the parameters of some 3-valued switch are set to $0.142\cdots$ $(= 2^0/(2^0+2^1+2^2))$, $0.285\cdots$ $(= 2^1/(2^0+2^1+2^2))$, and $0.574\cdots$ $(= 2^2/(2^0+2^1+2^2))$, according to the order of values specified in the corresponding value declaration. `f_geometric(`*Base*`)` is the same as `f_geometric(`*Base*`,desc)`, and `f_geometric` is the same as `f_geometric(2,desc)`.

- `default_sw_h` (possible values: `none`, `uniform`, `uniform(`δ`)`, δ (δ is a non-negative float); default: 0.0) — the default value for pseudo counts. If `none` is set, we have no default distribution for pseudo counts, and hence we cannot perform probabilistic inferences unless giving the pseudo counts, by `set_sw_h/2` or variational Bayesian learning (§5.2.1). `uniform` (resp. `uniform(`δ`)`) means that each pseudo count will be set as $1/K$ (resp. δ/K) by default, where K is the number of possible values of the corresponding switch. If a non-negative floating-point number δ is set to this flag, the system use δ as the default value of pseudo counts. Since version 1.11, the execution flag named `smooth` is deprecated and so please use this flag instead.

- `dynamic_default_sw` (possible values: `on` and `off`; default: `on`) — the flag on the mode on automatic setting of the default distributions to the switches whose outcome spaces are dynamically changed (see §2.6.3 for a typical case). If this flag is set as '`on`', the programming system automatically sets the default distribution to such switches before invoking the routines that refers to the switch distributions (e.g. sampling, probability computations, `get_sw/2`, and so on). The default distribution is given by the `default_sw` flag.

- `dynamic_default_sw_h` (possible values: `on` and `off`; default: `on`) — the flag on the mode on automatic setting of the default pseudo counts to the switches whose outcome spaces are dynamically changed (see §2.6.3 for a typical case). If this flag is set as '`on`', the programming system automatically sets the default pseudo counts to such switches before invoking the routines that refers to the pseudo counts (e.g. VB learning). The default pseudo counts are given by the `default_sw_h` flag.

- `em_progress` (possible value: non-negative integer; default: `10`) — the frequency of printing the progress message (i.e. the dot symbol) in the EM algorithm (§4.8.1). If this flag is set as `0`, the message is suppressed.

- `epsilon` (possible value: non-negative float; default: `1.0e-4`) — the threshold ε for convergence in the EM algorithm (see §4.8.1).

- `error_on_cycle` (possible values: `on` and `off`; default: `on`) — the flag for checking cycles in the calling relationship. By default or when this flag is set as '`on`', the programming system checks the existence of a cycle in the calling relationship, and if any cycle exists, the system will stop immediately. When this flag is set as '`off`', the system does *not* check such acyclicity and we may be able to obtain an explanation graph that violates the acyclicity condition. Of course this flag is very experimental and seems not to be used in usual cases.

- `fix_init_order` (possible values: `on` and `off`; default: `on`) — the flag for fixing the order of parameter initialization among switches. For an implementational reason, in the EM algorithm (§4.8.1), the order of parameter initialization among switches can vary according to the platform, and hence we may have different learning results among the various platforms. Turning this flag '`on`' fixes the initialization order in some manner, and will yield the same learning result.

- `init` (possible values: `none`, `random` and `noisy_u`; default: `random`) — the initialization method in the EM algorithm (§4.8.1). `none` means no initialization, `random` means that the parameters are initialized considerably at random, and `noisy_u` means that the parameters are initialized to be uniform with (small) Gaussian noises. The variance of Gaussian noises can be changed by the `std_ratio` flag.

- `itemp_init` (possible value: float b such that $0 < b \leq 1$; default: `0.1`) — the initial value β_{init} of the inverse temperature β used in the deterministic annealing EM (DAEM) algorithm (§4.8.4).

- `itemp_rate` (possible value: float b such that $b > 1$; default: `1.5`) — the increasing rate β_{rate} of the inverse temperature β used in the DAEM algorithm (§4.8.4).

- `learn_mode` (possible values: `params`, `hparams` and `both`; default: `params`) — the underlying statistical framework for parameter learning. If this flag is set as '`params`', the system will conduct the EM algorithm for ML/MAP estimation (§4.8), by which we can get the point-estimated parameters of random switches. If this flag is set as '`hparams`', the system will conduct the EM algorithm for VB learning (§5.2.1), by which we can get the adjusted pseudo counts (or equivalently, the hyperparameters) of switches. With '`both`', we can get both the point-estimated parameters and the adjusted hyperparameters.

- `log_viterbi` (possible values: `on` and `off`; default: `off`) — the flag for enabling/disabling the log-valued version of Viterbi computation (§4.12). For large data, we often suffer from the problem that the probability of some explanation goes into underflow. Specifically to the Viterbi computation however, we can avoid this problem by changing the multiplication of probabilities to summation of log-valued probabilities. Please note that the value of this flag does not make any influence on the scaling methods (§4.12). If you wish to use some scaling method, use the `scaling` flag.

- `max_iterate` (possible value: positive integer, `default` and `inf`; default: `default`) — the maximum number of EM iterations to be performed. In the EM algorithm (§4.8.1), sometimes we need a large number of iterations until convergence. For such a case, we can stop the EM algorithm before convergence by this flag. '`default`' means that the maximum number of iterations is the system's default value (10000, in version 1.11). With '`inf`', the system do not put any limit on the number of iterations.

- `params_after_vbem` (possible values: `none`, `mean` and `max`; default: `mean`) — the method for obtaining the new point-estimated parameters after VB learning (§5.2.1). '`none`' means that the programming system does not newly produce the parameters. If '`mean`' is set, the system will compute the mean values of the parameters $\bar{\theta}_{i,v} = \alpha^*_{i,v} / \sum_{v'} \alpha^*_{i,v'}$ as the new parameters. With '`max`', the system will further conduct the ML/MAP-EM algorithm to obtain the new parameters.

- `reduce_copy` (possible values: `on` and `off`; default: `off`) — the flag for automatic copying of the Prolog terms returned by several built-ins (`probf/2`, `viterbif/3`, and so on; See §4.18). If this flag is set as '`off`', the programming system will automatically make a copy of the Prolog term returned by these built-ins. On the other hand, with this flag turned '`on`', such a copying will be skipped.

- `rerank` (possible value: positive integer; default: `5`) — the number of intermediate candidates in reranking for the Viterbi computation based on the hyperparameters (§5.2.2).

- `reset_hparams` (possible values: `on` and `off`; default: `off`) — the flag on resetting of the pseudo counts (hyperparameters) in the repeated runs of VB learning (§5.2.1). In the default settings, it can be observed that the pseudo counts monotonically increases as we repeatedly run VB learning. If this flag is set as '`on`', on the other hand, the programming system will reset the pseudo counts with the default values (that is, it calls `set_sw_all_h/0`; §4.2.3) in advance of VB learning.

- `restart` (possible value: positive integer; default: `1`) — the number of restarts (§4.8.4). Generally speaking, the EM algorithm (§4.8.1) only finds a local ML/MAP estimate, so we often restart the EM algorithm for several times with different initial parameters, and get the best parameters (i.e. with the highest log-likelihood or log of a posteriori probability) among these restarts.

- `scaling` (possible values: `none`, `const` and `log_exp`; default: `none`) — the scaling methods. `none` means no scaling, `const` means doing the constant scaling, and `log_exp` means performing log-valued computation of probabilities. `log_exp` is the most general and applicable to any programs, but is preferred to be used with MAP estimation (§4.8.2) in parameter learning (this is because all relevant parameters should be non-zero to use `log_exp`). See §4.12 for a general description on these scaling methods. If any value other than `none` is specified, the computed probabilities are obtained as log-valued. Also note that the value of this flag does not make any influence on the use of the log-valued version of Viterbi computation (§4.6). If you wish to enable/disable the log-valued Viterbi computation, use the `log_viterbi` flag.

- `scaling_factor` (possible value: float (> 1); default: `8.0`) — the scaling factor for constant scaling.
- `search_progress` (possible value: non-negative integer; default: `10`) — the frequency of printing the progress message (i.e. the dot symbol) in explanation search and in constructing explanation graphs. If this flag is set as `0`, the message is suppressed.
- `smooth` (possible value: non-negative float; default: `0`) — this has become an alias of '`default_sw_h`' since version 1.11. This flag is only for backward compatibility and it is recommended to use '`default_sw_h`' instead, combining with `set_sw_all_h/0-2` (see the descriptions for the case of MAP estimation in §4.8.3).
- `sort_hindsight` (possible values: `by_goal` and `by_prob`; default: `by_goal`) — the flag for the mode on sorting the results of hindsight computation (§4.7). With `by_goal`, the result will be sorted in the Prolog's standard order with respect to the subgoals. With `by_prob`, the result will be ordered by the magnitude of the hindsight probability.
- `std_ratio` (possible value: non-negative float; default: `0.2`) — control parameter for the variance of Gaussian noises used in initialization of switch parameters in the EM algorithm (§4.8.1; see also the description on the `init` flag). When we initialize parameters with a k-valued switch according to a uniform distribution with Gaussian noises from $N(1/k, (\texttt{std\_ratio}*(1/k))^2)$. The parameters will be normalized at the end of initialization.
- `verb` (possible values: `none`, `graph`, `em` and `full`; default: `none`) — the flag for extra messages in EM learning (§4.8.1). '`none`' means that no extra message will be displayed. If this flag is set as '`graph`', the explanation graphs will be displayed after the explanation search. By '`em`', we can get the more detailed information about the EM algorithm. If '`full`' is set, we will see both the explanation graphs and the information about EM.
- `viterbi_mode` (possible values: `params` and `hparams`; default: `params`) — the underlying statistical framework for Viterbi computation. If this flag is set as '`params`', the system will conduct the Viterbi computation based on the current parameter values (§4.6). If '`hparams`' is set, on the other hand, the system will conduct the Viterbi computation for VB learning based on the adjusted hyperparameters (§5.2.2), which utilizes reranking.
- `warn` (possible values: `on` and `off`; default: `off`) — the flag for enabling/disabling warning messages.

4.15 Random number generator

The following built-ins are provided to set information or retrieve information of the random number generator. As a random number generator, the programming system uses *Mersenne Twister* (`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`). For sampling utilities based on a finite set of discrete values, see §4.16.

- `random_float(`*Max*`,`*R*`)` : Generates a random floating-point number R from the range $0 \leq R \leq Max$ ($Max > 0$).
- `random_int(`*Max*`,`*I*`)` : Generates a random integer I from the range `0...`*Max* ($Max > 0$).
- `set_seed(`*Seed*`)` : *Seed* is set to be the new seed used in the random number generator.
- `set_seed_time`: The current time is set to be the seed used in the random number generator.

- `set_seed_time(T)` : The current time is set to both *T* and the seed used in the random number generator.

4.16 Sampling on temporary distributions

By sampling, random switches (`msw/2`) can generate random outputs, but sometimes it is tedious to assign their parameters in advance of sampling. `dice/2-3` are sampling utilities that work independently of the model, based on the probabilities *temporarily* assigned. These built-ins are implemented on the random number generator described in §4.15.

`dice(`*Values*`,`*Probs*`,`*V*`)` chooses *V* randomly from *Values* according to the distribution *Probs*, and `dice(`*Values*`,`*V*`)` chooses *V* randomly from *Values* according to the uniform distribution. For example, we may sample the phenotypes of blood type according to the distribution $P_A = 0.4$, $P_B = 0.2$, $P_O = 0.3$, $P_{AB} = 0.1$:

```
?- dice([a,b,o,ab],[0.4,0.2,0.3,0.1],X).
X = a ?

?- dice([a,b,o,ab],[0.4,0.2,0.3,0.1],X).
X = o ?

?- dice([a,b,o,ab],[0.4,0.2,0.3,0.1],X).
X = b ?
```

These runs would be useful for generating synthetic samples without specifying a distribution of genes.

Moreover, we can specify some extended form of a set of integer values. Namely, each element of the list *Values* can take the form 'N_{min}-N_{max}@N_{skip}' or 'N_{min}-N_{max}', where N_{min} (resp. N_{max}) is the minimum (resp. the maximum) value of some range, and N_{skip} is the skip number. For example, the following choose a value from `[1,3,5,10,15,20]`.

```
?- dice([1-5@2,10-20@5],X).
```

At the implementation level, the conversion from such an extended form to the basic one is done by `expand_values/2`, which is also used internally for `values_x/2-3`, the extended multi-valued switch declarations (see §2.6.3).

4.17 File IO

Basically, all B-Prolog's built-ins for file IO are also available for PRISM. In addition, the programming system provides utilities for loading/saving Prolog clauses:

- `load_clauses(`*File*`,`*Clauses*`)` reads all clauses as a list *Clauses* from a file *File*,
- `load_clauses(`*File*`,`*Clauses*`,`*M*`,`*N*`)` reads *N* clauses in *File* as *Clauses*, starting at the *M*-th line, where the lines are numbered from zero. This built-in is deprecated, and it is recommended to use `load_clauses/3` instead.
- `load_clauses(`*File*`,`*Clauses*`,`*Options*`)` reads clauses in *File* as *Clauses*, with the options *Options*, which is a list of the following Prolog terms:
  - `from(`*K*`)` — read from the *K*-th clause (*K* is a zero-based index). If this option is omitted, *K* will be set as zero.

  - `size(`*N*`)` — read *N* clauses. If this option is omitted or *N* is '`max`', the built-in will read clauses until reaching at the end of file.

- `save_clauses(`*File*`,` *Clauses*`)` writes each element in *Clauses* as a clause into *File*.
- `save_clauses(`*File*`,` *Clauses*`,` *M*`,` *N*`)` writes *N* clauses in *Clauses* into *File*, starting at the *M*-th element, where the elements are numbered from zero. This built-in is deprecated, and it is recommended to use `save_clauses/3` instead.
- `save_clauses(`*File*`,` *Clauses*`,` *Options*`)` writes clauses *Clauses* into *File*, with the options *Options*, which is a list of the following Prolog terms:
  - `from(`*K*`)` — write from the *K*-th element in *Clauses* (*K* is a zero-based index). If this option is omitted, *K* will be set as zero.
  - `size(`*N*`)` — write *N* elements. If this option is omitted or *N* is '`max`', the built-in will write elements until reaching at the end of *Clauses*.

Besides, we can load the data in the CSV format by the following built-ins:

- `load_csv(`*File*`,` *Rows*`)` reads the contents of a CSV file *File* as *Rows*.
- `load_csv(`*File*`,` *Rows*`,` *Options*`)` reads the contents of a CSV file *File* as *Rows*, with the options *Options*, which is a list of the following Prolog terms:
  - ◇ Options on the range of rows to be read:
    - `row_from(`*K*`)` or `row_skip(`*K*`)` — read from the *K*-th row (*K* is a zero-based index). If this option is omitted, *K* will be set as zero.
    - `row_size(`*N*`)` — read *N* rows. If this option is omitted or *N* is '`max`', the built-in will read rows until reaching at the end of file.
    - `column_from(`*K*`)` or `column_skip(`*K*`)` — read from the *K*-th column (*K* is a zero-based index). If this option is omitted, *K* will be set as zero.
    - `column_size(`*N*`)` — read *N* columns. If this option is omitted or *N* is '`max`', the built-in will read columns until reaching at the end of line.
  - ◇ Options on the format of a row:
    - `pred([])` — read each row in the form `[`Col_1`,` Col_2`, ...]`, where Col_1, Col_2, ... are the values separated by commas.
    - `pred(`*p*`/1)` or `pred(`*p*`)` — read each row in the form *p*`(`[Col_1`,` Col_2`, ...]`), where *p* is an arbitrary predicate name.
    - `pred(`*p*`/n)` — read each row in the form *p*`(`Col_1`,` Col_2`, ...)`, where *p* is an arbitrary predicate name.
  - ◇ Other options:
    - `comment(`*C*`)` — regard as comments the rows beginning with the character *C*.
    - `comment` — regard as comments the rows beginning with the character `'#'` (this is the same as `comment('#')`).
    - `double_quote(`*X*`)` — enable (with *X* = `yes`) or disable (with *X* = `no`) to process the double-quoted columns following RFC 4180 (by default, *X* = `yes`).
    - `parse_number(`*X*`)` — enable (with *X* = `yes`) or disable (with *X* = `no`) to parse numeric strings in the input file (by default, *X* = `yes`). For example, by default or if we specify `parse_number(yes)`, a value "`123456`" in the input file will be converted into `123456`, which can be evaluated as a number. Otherwise, we obtain `'123456'`, which is just an atom.

For example, let us consider a CSV file named `foo.csv` which includes three rows:

```
bill,14
jeff,15
peter,18
```

Then we can read these three rows by using `load_csv/2-3` as follows:

```
?- load_csv('foo.csv',Rs).
Rs = [csvrow([bill,14]),csvrow([jeff,15]),csvrow([peter,18])] ?

?- load_csv('foo.csv',Rs,[pred(age/n)]).
Rs = [age(bill,14),age(jeff,15),age(peter,18)] ?
```

4.18 Accessing Prolog terms returned from the built-ins*

(This section is targeted at the users who are already familiar with PRISM.)

There are several built-in predicates that return Prolog terms consisting of subgoals or switch instances: `probf/2`, `viterbif/3`, `viterbig/1-2`, `hindsight/3`, `hindsight_agg/3`, `chindsight/3`, and `chindsight_agg/3`. Now let us consider a situation where we are setting the `clean_table` flag to '`on`' (i.e. the system cleans up the solution table at each call of the built-ins), and where a predicate p, one from the built-ins above, is called repeatedly in a query. Then, after a call of p has finished, the references to the Prolog terms returned by the previous calls of p would be lost, and thus it is possible that a memory fault is arisen if we try to follow these references. It would cause no problem if we can finish the task before the next call of p, but to make things safer, the predicates above are implemented to return copies by default. One drawback of this implementation, on the other hand, is that the term copying requires memory in the heap area, and could lead to running out of memory when we deal with quite large Prolog terms.

To adapt to various situations, we introduce another flag named '`reduce_copy`', as a temporary treatment. If the `reduce_copy` flag is '`on`' (resp. '`off`'), the term copying described above will be disabled (resp. enabled). Three typical cases can be considered in the possible flag settings:

- `clean_table` = `on` and `reduce_copy` = `off`:
  This is the default. The memory is consumed by copying but the solution table is always cleaned up.
- `clean_table` = `on` and `reduce_copy` = `on`:
  This case is least memory consuming but has a risk of the memory fault as described above. Fortunately, it would be safe if we are able to finish accessing the terms before the next call of p.
- `clean_table` = `off` with any value for `reduce_copy`:
  In this case, the solution table will not be cleaned up, so it should be always safe except a risk of memory exhaustion.

In typical programs, there seems to be no need to care about the issue described in this section since the default setting is safe, and sufficiently efficient in most cases. Also as mentioned above, the mechanism introduced here is considered as a temporary treatment, and could be changed in the future version.

Chapter 5

Variational Bayesian learning*

5.1 Background

5.1.1 VB-EM learning

As mentioned in §1.5, variational Bayesian (VB) learning has high robustness against data sparseness in model selection and prediction (Viterbi computation). For model selection, an introductory description in Bayesian approaches is given in §4.10. To choose the best model (the best PRISM program) M^* that fits best the data D at hand, we consider $M = M^*$ is the model that maximizes the marginal likelihood $p(D \mid M)$. It has been also known that if D is complete data D_c, $p(D \mid M)$ can be obtained in closed form. However, when D is incomplete, i.e. there is some hidden data z such that $D_c = (D, z)$ (in PRISM, z corresponds to the explanations for the observed goals), some approximation is required. In the followings, we briefly describe the approximation via the VB approach.

First, let us consider log of the marginal likelihood $L(D) \stackrel{\text{def}}{=} \log p(D \mid M)$, and then we have:

$$\begin{aligned} L(D) &= \log \sum_z \int_\Theta p(D, z, \theta \mid M) d\theta \\ &= \log \sum_z \int_\Theta q(z, \theta \mid D, M) \frac{p(D, z, \theta \mid M)}{q(z, \theta \mid D, M)} d\theta \\ &\geq \sum_z \int_\Theta q(z, \theta \mid D, M) \log \frac{p(D, z, \theta \mid M)}{q(z, \theta \mid D, M)} d\theta. \qquad \text{(from Jensen's inequality)} \end{aligned}$$

For the space limitation, we fix the model M for the moment, and simply write $p(\cdot \mid M) = p(\cdot)$ and $q(\cdot \mid D, M) = q(\cdot \mid D)$, and then obtain:

$$L(D) \geq F[q] \stackrel{\text{def}}{=} \sum_z \int_\Theta q(z, \theta \mid D) \log \frac{p(D, z, \theta)}{q(z, \theta \mid D)} d\theta$$

where $F[q]$ can be seen as a lower limit of $L(D)$, and is called the *variational free energy*. So to get a good approximation of $L(D)$, we attempt to find a distribution function $q = q^*$ that maximizes a functional $F[q]$. In model selection, we use the variational free energy $F[q]$ as a model score. Besides, to get another

view, we have the following by considering $L(D) = \sum_z \int_\Theta q(z, \theta \mid D) \log p(D) d\theta$:

$$\begin{aligned} L(D) - F[q] &= \sum_z \int_\Theta q(z, \theta \mid D) \log \left\{ p(D) \cdot \frac{q(z, \theta \mid D)}{p(D, z, \theta)} \right\} d\theta \\ &= \sum_z \int_\Theta q(z, \theta \mid D) \log \frac{q(z, \theta \mid D)}{p(z, \theta \mid D)} d\theta \;=\; \mathrm{KL}(q(z, \theta \mid D) \parallel p(z, \theta \mid D)). \end{aligned}$$

From the above, maximizing $F[q]$ implies minimizing the Kullback-Liebler divergence between $q(z, \theta \mid D)$ and $p(z, \theta \mid D)$. So finding q^* is to make a good approximation of $p(z, \theta \mid D)$, the conditional distribution of hidden variables and parameters.

In VB learning, we further assume $q(z, \theta \mid D) \approx q(z \mid D) q(\theta \mid D)$, and obtain a generic form of *variational Bayesian EM (VB-EM) algorithm* as an iterative procedure consisting of the following two updating rules:

$$\begin{aligned} q(z \mid D) &\propto \exp\left(\int_\Theta q(\theta \mid D) \log p(D, z \mid \theta) d\theta \right), \\ q(\theta \mid D) &\propto p(\theta) \exp\left(\sum_z q(z \mid D) \log p(D, z \mid \theta) \right). \end{aligned}$$

Please recall that, in PRISM, D is a multiset of the observed goals $G_1, G_2, \ldots, G_T$, and that z corresponds to (hidden) explanations for the goals. The VB-EM algorithm for PRISM is then derived from the above generic procedure as follows:

Initialization step:
: Initialize the hyperparameters of random switches as $\alpha_{i,v}^{(0)} = \alpha_{i,v} + \xi_{i,v}$ where $\alpha_{i,v}$ are the hyperparameters configured by the user and $\xi_{i,v}$ are small random noises, and then iterate the next two steps until the variational free energy converges.

Expectation step:
: For each `msw(i,v)`, compute $\tilde{C}_{i,v}$, the sufficient statistics corresponding to the expected occurrences of `msw(i,v)` under the hyperparameters $\alpha_{i,v}^{(m)}$.

Maximization step:
: Using the expected occurrences, update each hyperparameter by $\alpha_{i,v}^{(m+1)} = \alpha^{(0)} + \tilde{C}_{i,v}$ and then increment m by one.

After VB-EM learning, we finally obtain the adjusted hyperparameters $\alpha_{i,v}^*$ of random switches instead of the parameters, and the converged variational free energy which is considered as an approximation of log of the marginal likelihood. $\alpha_{i,v}$ need to be configured in advance by the user through the corresponding pseudo counts $\delta_{i,v} = (\alpha_{i,v} - 1)$ via the built-ins for handling switches (§4.2). By default, the system considers that $P(\theta)$ is uninformative, that is, $\alpha_{i,v} = 1$ (or equivalently $\delta_{i,v} = 0$). Besides, as long as the user program satisfies the modeling conditions listed in §2.4.6, it is still possible to compute $\tilde{C}_{i,v}$ in the expectation step in a dynamic programming fashion. So at least in algorithmic level, we can perform VB learning as fast as in the case of ML/MAP estimation.[1]

[1] In this sense, the derived VB-EM algorithm can be seen as a generalization of dynamic programming based VB-EM algorithm for hidden Markov models [23], probabilistic context-free grammars [21], and directed graphical models (Bayesian networks) [3].

5.1.2 Viterbi computation

Now let $P^*(\theta)$ be the a posteriori distribution given the observed data, which includes the adjusted hyper-parameters $\alpha^*_{i,v}$. Then we can perform the Viterbi computation based on the a posteriori distribution:

$$\begin{aligned} E^* &= \operatorname*{argmax}_{E \in \psi(G)} P(E \mid G) = \operatorname*{argmax}_{E \in \psi(G)} \frac{P(E,G)}{P(G)} = \operatorname*{argmax}_{E \in \psi(G)} P(E) \\ &= \operatorname*{argmax}_{E \in \psi(G)} \int_\Theta P^*(\theta) P(E \mid \theta) d\theta. \end{aligned}$$

The inference based on $\int_\Theta P^*(\theta)P(E \mid \theta)d\theta$ seems more robust than that based on $P(E \mid \hat{\theta})$, since the former relies on the averaged quantity with respect to the a posteriori distribution, not on any particular point-estimated parameters.

However, there still remains a computational problem. Although $\int_\Theta P^*(\theta)P(E \mid \theta)d\theta$ can be computed efficiently in closed form for each $E \in \psi(G)$, the number of explanations for an observed goal G (i.e. $|\psi(G)|$) can exponentially grow. In addition, the integral over θ prevents us from introducing a simple dynamic programming based computation.

As a remedy for this difficulty, we take a *reranking* approach [8], which is popular for the predictive tasks (part-of-speech tagging, parsing, and so on) in statistical natural language processing. To be specific, for a given goal G, we follow the two-staged procedure below:

1. Run top-K Viterbi computation in a dynamic programming fashion based on the point-estimated parameters. These parameters obtained by ML/MAP estimation or the mean values of the parameters $\bar{\theta}_{i,v}$ obtained by $\bar{\theta}_{i,v} = \alpha^*_{i,v} / \sum_{v'} \alpha^*_{i,v'}$.

2. Return $E = \tilde{E}^*$ which comes with the highest $\int_\Theta P^*(\theta)P(E \mid \theta)d\theta$ among K explanations obtained in the first step.

The point-estimated parameters used in the first step seems reliable to some extent, so if K is sufficiently large, the true Viterbi explanation E^* based on the a posteriori distribution (i.e. $E^* = \operatorname{argmax}_E \int_\Theta P^*(\theta)P(E \mid \theta)d\theta$) will be found in K explanations obtained in the first step. So we can expect $\tilde{E}^*$ to be E^* in most cases.

It is obvious from above that reranking requires extra computational effort. On the other hand, we need not use reranking if every random switch i (i.e. an atom of the form `msw(i,·)`) only appears at most once in any explanation for any observed goal, or in other words, if we do not use any random switch twice or more in any generation process of any observed goal. Instead, for such a case, the first step above with $\bar{\theta}_{i,v}$ and $K = 1$ will return the exact E^*. To be specific, it is easy to see that the following Bayesian network program (see §7.3 for detailed descriptions) does not use any random switch twice or more to yield an observation represented by `world/2`:

```
world(Sm,Re) :- world(_,_,_,Sm,_,Re).

world(Fi,Ta,Al,Sm,Le,Re) :-
   msw(fi,Fi),
   msw(ta,Ta),
   msw(sm(Fi),Sm),
   msw(al(Fi,Ta),Al),
   msw(le(Al),Le),
   msw(re(Le),Re).
```

On contrary, the HMM program (§1.3) may use repeatedly a particular switch such as `msw(tr(s0),·)`. This fact implies that we need not use reranking for the Bayesian network program above, while reranking is indispensable for the HMM program.

5.1.3 Other probabilistic inferences

For the probabilistic inferences other than Viterbi computation, it is also required to compute quantities based on the a posteriori distribution $P^*(\theta)$. For example, the marginal (averaged) probability of goal G will be computed as:

$$P(G) = \int_\Theta P^*(\theta)P(G \mid \theta)d\theta = \int_\Theta P^*(\theta)\left(\sum_{E\in\psi(G)} P(E \mid \theta)\right)d\theta.$$

In VB, it also seems difficult to perform dynamic programming based computation for these probabilistic inferences. This is because, as explained in [2], the independencies among subgoals, which are fully exploited in dynamic programming, are lost due to the integral over θ.

In the programming system, we may utilize the routines for inferences used in ML/MAP with considering the parameters θ to be the mean values of the parameters $\bar{\theta}_{i,v} = \alpha^*_{i,v} / \sum_{v'} \alpha^*_{i,v'}$ [2, 23], on the assumption that these mean values are a representative of the entire a posteriori distribution. Another workaround provided by the programming system is to run the MAP-EM algorithm under $\alpha^*_{i,v}$.

5.1.4 Deterministic annealing EM for VB learning

The deterministic annealing EM (DAEM) algorithm (§4.8.4) is also supported for VB learning. To be specific, following [20], let us transform the variational free energy as follows:

$$F[q] = \sum_z \int_\Theta q(z,\theta \mid D)\log p(D,z,\theta)d\theta - \sum_z \int_\Theta q(z,\theta \mid D)\log q(z,\theta \mid D)d\theta$$

Again, from an analogy to statistical mechanics, we correspond $F[q]$ with $-\mathcal{F}$ ($\mathcal{F}$: the free energy), the first term in the above equation with $-\mathcal{U}$ ($\mathcal{U}$: the internal energy) and the second term with $\mathcal{S}$ ($\mathcal{S}$: the entropy). Then we newly introduce the variational free energy that takes into account the inverse temperature β:

$$F_\beta[q] \stackrel{\text{def}}{=} \sum_z \int_\Theta q(z,\theta \mid D)\log p(D,z,\theta)d\theta - \frac{1}{\beta}\sum_z \int_\Theta q(z,\theta \mid D)\log q(z,\theta \mid D)d\theta.$$

The VB-EM algorithm that tries to maximize $F_\beta[q]$ (i.e. the deterministic annealing version of the VB-EM algorithm) has a similar procedure to that of the DAEM algorithm (§4.8.4) for ML/MAP estimation.

5.2 Built-in utilities for variational Bayesian learning

5.2.1 VB-EM learning

On contrary to the long descriptions above on VB learning, the usages of the built-in predicates are considerably simple. That is, in the programming system, we can switch between ML/MAP-EM learning and VB-EM learning only by configuring the execution flag '`learn_mode`'. To enable VB-EM learning, we give a value '`hparams`' (which indicates that we wish to get the adjusted hyperparameters by VB-EM) to the `learn_mode` flag, and then run the usual learning command (`learn/0-1`) as follows:

```
?- set_prism_flag(learn_mode,hparams).
?- Goals=[hmm([a,b,a,a,a]),hmm([b,b,b,a,b])],learn(Goals).
```

While learning, we will see the messages similar to those in the case of ML/MAP-EM learning. Another way is to call `learn_h/0-1` directly (the suffix '`_h`' indicates that the target of learning is hyperparameters):

```
?- Goals=[hmm([a,b,a,a,a]),hmm([b,b,b,a,b])],learn_h(Goals).
```

On the other hand, to disable VB-EM, please give 'params' to the learn_mode flag (the default value of the learn_mode flag is 'params'). This indicates that we wish to get the point-estimated parameters of the model, and indeed the next call of learn/0-1 will start ML/MAP-EM learning:

```
?- set_prism_flag(learn_mode,params).
?- Goals=[hmm([a,b,a,a,a]),hmm([b,b,b,a,b])],learn(Goals).
```

It is also possible to run ML/MAP-EM learning by invoking learn_p/0-1 directly:

```
?- Goals=[hmm([a,b,a,a,a]),hmm([b,b,b,a,b])],learn_p(Goals).
```

Furthermore, as described above, we sometimes need the point-estimated parameters as well as hyperparameters for the later probabilistic inferences. To get such point-estimated parameters, we give 'both' (i.e. we wish to get *both* the adjusted hyperparameters and the point-estimated parameters) to the flag 'learn_mode'.

```
?- set_prism_flag(learn_mode,both).
?- Goals=[hmm([a,b,a,a,a]),hmm([b,b,b,a,b])],learn(Goals).
```

learn_b/0-1 is also available for conducting VB-EM learning directly. By default or by giving 'mean' to the params_after_vbem flag, we will obtain the mean values of the parameters $\bar{\theta}_{i,v} = \alpha^*_{i,v} / \sum_{v'} \alpha^*_{i,v'}$ as the point-estimated parameters. On the other hand, with the params_after_vbem flag set as 'max', the programming system will run the MAP-EM algorithm after the VB-EM algorithm to get the MAP estimate of parameters:

```
?- set_prism_flag(learn_mode,both).
?- set_prism_flag(params_after_vbem,max).
?- Goals=[hmm([a,b,a,a,a]),hmm([b,b,b,a,b])],learn(Goals).
```

After the point-estimated parameters obtained, we can run as usual the routines for the probabilistic inferences other than Viterbi computation (see §5.2.2 for the case of Viterbi computation). The DAEM algorithm can be used in the same way as that in ML/MAP-EM learning, which is described in §4.8.4.

5.2.2 Viterbi computation

Similarly to the case of EM learning, by configuring the viterbi_mode flag, we can switch the underlying statistical framework for Viterbi computation. If we give a value 'hparams' to this flag, the programming system will invoke a routine for the Viterbi computation based on the adjusted hyperparameters (and the point-estimated parameters) using reranking (§5.1.2). On the other hand, if we give a value 'params' to the viterbi_mode flag, the system will invoke the usual Viterbi routines based only on the point-estimated parameters.

The built-ins shown in §4.6 also work within the framework of VB learning. In these built-ins, the number K of the intermediate candidates of the Viterbi explanation(s) in reranking can be specified by the rerank flag ($K = 5$ by default; see §4.14 for details). In addition, K can be specified as an argument of the built-ins. That is, for top-N Viterbi routines such as n_viterbif([N,K],G), we can give a pair [N,K] to the first argument, where K is the number of intermediate candidates in reranking. For example, n_viterbif([N,K],G) is the same as n_viterbif(N,G) which uses K intermediate candidates. If $N > K$, the built-ins return only top-K Viterbi explanations.

Instead of configuring the viterbi_mode flag, we can directly call the built-ins for Viterbi computation based on VB. To do this, we add a suffix '_h' to the predicate name of the built-in we would like to use. For example,

```
?- set_prism_flag(viterbi_mode,hparams).
?- viterbif(hmm([a,b,b,b,a]).
```

and

```
?- viterbif_h(hmm([a,b,b,b,a]).
```

yield the same result. On the other hand, we can directly run the ML/MAP-based Viterbi routines by adding '_p' to the predicate name of the corresponding built-in (e.g. `viterbif_p/1`).

Furthermore, as discribed in §5.1.2, if we are sure that every random switch i only appears at most once in any explanation for any observed goal, we need not take the reranking approach. Instead, in variational Bayesian learning, we first obtain the mean values of parameters as the point-estimated parameters (by giving '`mean`' to the `params_after_vbem` flag), and then run built-ins for usual (basic) Viterbi computations, such as `viterbif/2` (§4.6). It is also worth noting that, at the implementation level, the usual Viterbi built-ins work more efficiently (in both time and space) than ones for top-K Viterbi computation.

Chapter 6

Parallel EM learning*

6.1 Background

In these days, there are more and more opportunities for us to work with parallel computing environments such as computer grids. To benefit from those environments on large-scale EM learning, the programming system provides a parallel learning utility since version 1.11. This utility is characterized by the following features:

- *Data parallelism.* Since we assume that observed goals in training data are i.i.d. (independent and identically distributed), the major part of the learning procedure, the explanation search (§2.4.2) and large part of the EM algorithm (§4.8.1), can be conducted independently for each observed goal.
- *Master-slave model.* Our implementation is supposed to run with one master process and many (one or more) slave processes, which are allocated over processors. The master process controls the entire procedure, whereas the slave processes perform the substantial tasks of the explanation search and the expectation steps of the EM algorithm. The expected occurrences of random switches are accumulated among the processes before every maximization step, then the parameters are updated on each process.
- *Dynamic load balancing.* The computation time required for each observed goal G is linear in the size of the explanation graph for G, but in general the size is unknown before the explanation search. This makes it difficult to partition the entire observed data into the subsets which require an almost equal amount of efforts to complete. To cope with such difficulty, we take a work-pool approach (also known as a processor-farm approach), in which all observed goals are firstly put into a work pool, and then the master process picks up observed goals one by one and assigns each of them to a slave process that becomes available.
- *Distributed memory computing.* The algorithm used in this utility is primarily designed for parallel computer systems in which each processor has a local memory of its own. The communications among the processes are realized by message passing via MPI (message-passing interface) [14]. Thanks to this design, we would be able to collectively utilize memory resources which are distributed among computers.

The parallel learning algorithm implemented in this system is empirically shown in [15] to have an advantage in computation time and space for hidden Markov models (HMMs) and probabilistic context-free grammars (PCFGs).[1]

[1] Due to the removal of some redundant computations in version 1.11, the speed-up might not be so drastic as reported in [15].

6.2 Requirements

The parallel learning utility is provided as an experimental feature and only for Linux systems (32-bit and 64-bit) with the following runtime libraries installed:

- glibc version 2.3.4 or higher, and
- MPICH version 1.x with the `ch_p4` device.

MPICH is one of open-source MPI implementations and is available at its authors' website (`http://www-unix.mcs.anl.gov/mpi/mpich1/`). Many Linux distributions also provide official and/or unofficial packages for MPICH, and we believe most of these packages are suitable for running the utility. All binaries for parallel learning in the released package of PRISM were built with GCC 4.0.2 and MPICH 1.2.7 provided as part of openSUSE 10.0.

In addition to the above requirements, the programming system needs to be installed into a directory accessible from all computers used for parallel learning. The utility is expected to work well even on environments that consist of heterogeneous (but not so much different) computers, except that mixed use of 32-bit and 64-bit systems is not supported.

It is also possible to run the utility on a single computer with a multi-core processor (or multiple processors) in order to reduce the learning time (§6.3.3), as long as the required libraries are available in that computer. Note that, however, parallel learning requires more memory space than non-parallel learning (§6.4).

6.3 Usage

6.3.1 Running the utility

The parallel learning utility provides no interactive sessions. All programs therefore have to run via batch execution (§3.7). Also, the utility needs to be started on a directory shared among the computers, since all processes require access to byte-code files of compiled PRISM programs.[2]

The utility can be started by invoking `mpprism` instead of `prism` and `upprism`. Basically, its usage is the same as `upprism`. The user who is familiar with running MPI programs should note that `mpirun` is called inside `mpprism`. Here are a couple of example commands:

```
mpprism foo
mpprism foo 5893421 1000
mpprism load:foo
```

The utility runs with four processes in default. The number of processes can be changed by the variable `NPROCS`. For example, the command below starts the utility with twelve processes:

```
env NPROCS=12 mpprism foo
```

If you are familiar with how to use `mpirun`, and you have options you wish to pass, you can specify them in the variable `PRISM_MPIRUN_OPTS`. Note that the `-np` option (the number of processes) should not be included in this variable. Here is an example:

```
env NPROCS=8 PRISM_MPIRUN_OPTS="-machinefile machines" mpprism foo
```

[2] PRISM programs given to `mpprism` are firstly compiled on the master process, and then the resulting byte-code files are loaded by each process (master and slave).

6.3.2 Writing programs for parallel learning

Most PRISM programs are expected to run without changes, provided batch clauses (`prism_main/0-1`) are defined. Note that, however, only the parameter learning is conducted in parallel. The other computations are simply performed on a (single) master process and thus no performance improvement will be made. There are also some limitations in functionalities (§6.4).

6.3.3 Some remarks for effective use

Here are some remarks on use of the parallel learning utility:

- The parallel learning utility is not yet so reliable as the non-parallel one in many respects. It is highly recommended to make sure that your program works on `prism` or `upprism` before using `mpprism`.
- It is often a good idea to have a single processor (or computer) shared between a master process and one of slave processes, in particular if the number of processors is limited. The influence of the master process is considered to be small, since the master process is usually at a very low load throughout parameter learning. Moreover, the influence is mostly adjusted by dynamic load balancing (§6.1). This can be done by specifying $(n + 1)$ as the number of processes where n is the number of available processors. Accordingly, for learning on a single computer with a dual-core processor (or dual processors), you can gain the best time performance by running with three processes. In this setting, the first processor is expected to work for the master and one slave processes, and the second processor for the other slave process. Be warned sufficient memory space is needed on that computer (§6.4).
- If possible, order the goals (training data) so that larger ones precede shorter. Here, large goals mean ones which consume much time in the explanation search and the expectation steps of the EM algorithm. The work-pool approach works more effectively when heavy subtasks enqueued first in the work pool. In PCFG programs (§7.2), for instance, we can list training sentences in the decreasing order of their lengths.
- The relationship between speed-up and the number of processors depends on programs. For some programs, the learning time is reduced simply as the number of processors increases. For others, on the other hand, there are even cases in which learning with less processors is faster than with more processors. It is therefore not recommended to stick on as-many-as-possible strategies.
- The amount of memory consumed by each process is expected to be roughly proportional to the speed of processor on which it runs. Recall this property if you wish to make full use of memory resources distributed among multiple computers.
- The resulting parameters of parallel learning can be saved by calling `save_sw/0-1` (§4.2.7) in the batch clause (`prism_main/0-1`). Then they can be restored on interactive sessions (of the normal `prism` command) by `restore_sw/0-1` to be utilized on sampling, probability calculation, Viterbi computation, and hindsight computation. This also applies to the cases with pseudo counts (hyperparameters).

6.4 Limitations

The parallel learning utility has the following limitations (note that many of them have already been mentioned above):

- No computations other than parameter learning are parallelized.
- The utility has not been tested sufficiently yet.
- When the utility is aborted by some error, there occasionally remain defunct processes. This is due to difficulty in aborting MPI programs cleanly. In case you face this situation, please kill those processes manually.
- Parallel learning requires, in total, more memory resources than non-parallel learning. This might be critical when the utility is run on a single computer or shared-memory systems.
- The learning time might not be reduced as expected for some programs, in particular those with failure (§4.11).
- The statistics on the explanation graph (§4.9) can be different from those obtained on the non-parallel utility, and even can vary from execution to execution.[3]
- The explanation graph is not displayed even with the `verb` flag set to '`graph`' or '`full`'.
- The total table space used for learning is not displayed.
- The learning time is given by elapsed time, not by CPU time as on the non-parallel utility (this is not actually a limitation).

[3] The reason is as follows. Since version 1.11, in the constructed explanation graphs, there can be subgoals which are *shared* among distinct observed goals (this mechanism is called *inter-goal sharing* [19]). In parallel learning, however, such sharing will be made only within each slave process, and therefore the number of subgoals in the entire graph varies depending on how the observed goals are assigned to the slave processes.

Chapter 7

Examples

PRISM is suited for building complex systems that involve both symbolic and probabilistic elements such as discrete hidden Markov models , stochastic string/graph grammars, game analysis, data mining, performance tuning and bio-sequence analysis. In this chapter, we describe several program examples including the ones that can be found under the directories named 'exs' or 'exs_fail' in the released package.

7.1 Hidden Markov models

The HMM (hidden Markov model) program has been fragmentarily picked up throughout this manual. In this section, on the other hand, we attempt to collect the previous descriptions as a single session of an artificial experiment.

As described in §1.3, the HMM we consider has only two states 's0' and 's1', and two emission symbols 'a' and 'b'. In top-down writing such an HMM, we make several declarations first:

```
target(hmm,1).
data(user).

values(init,[s0,s1]).   % state initialization
values(out(_),[a,b]).   % symbol emission
values(tr(_),[s0,s1]).  % state transition
```

The first declaration means observed goals take the form `hmm(`L`)` where L is an output string, i.e. a list of emitted symbols. The last three declarations declare three types of switches: switch `init` chooses 's0' or 's1' as an initial state to start with, the symbol emission switches `out(·)` chooses 'a' or 'b' as an emitted symbol at each state, and the state transition switches `tr(·)` chooses the next state 's0' or 's1'.

We then proceed to the modeling part. The model part is described only with four clauses:

```
hmm(L):-                % To observe a string L:
   str_length(N),       %   Get the string length as N
   msw(init,S),         %   Choose an initial state randomly
   hmm(1,N,S,L).        %   Start stochastic transition (loop)

hmm(T,N,_,[]):- T>N,!. % Stop the loop
hmm(T,N,S,[Ob|Y]) :-    % Loop: The state is S at time T
```

```
    msw(out(S),Ob),      %   Output Ob at the state S
    msw(tr(S),Next),     %   Transit from S to Next.
    T1 is T+1,           %   Count up time
    hmm(T1,N,Next,Y).    %   Go next (recursion)

str_length(10).          % String length is 10
```

As described in the comments, the modeling part expresses a probabilistic generation process for an output string in the HMM. If possible, we recommend such a purely generative fashion in writing the modeling part. One of its benefits here is that the modeling part works both in sampling execution and explanation search.[1]

Optionally we can add the utility part. In the utility part, we can write an arbitrary Prolog program which may use built-ins of the programming system. Here, we conduct a simple and artificial learning experiment. That is, in this experiment, we first give some predefined parameters to the HMM, and generate 100 strings under the parameters. Then we learn the parameters from such sampled strings. Instead of running each step interactively, we write the following utility part that makes a batch execution of the learning procedure:

```
hmm_learn(N):-
    set_params,!,                % Set parameters manually
    get_samples(N,hmm(_),Gs),!, % Get N samples
    learn(Gs).                   % learn with the samples

set_params :-
    set_sw(init,   [0.9,0.1]),
    set_sw(tr(s0), [0.2,0.8]),
    set_sw(tr(s1), [0.8,0.2]),
    set_sw(out(s0),[0.5,0.5]),
    set_sw(out(s1),[0.6,0.4]).
```

`hmm_learn(N)` is a batch predicate for the experiment, where N is the number of samples used for learning. `set_params/0` specifies the parameters of each switch manually. Since `hmm/1` works in sampling execution, we can use a PRISM's built-in `get_samples/3` (§4.3) that calls `hmm/1` for N times.

Let us run the program. We first load the program:

```
% prism
  :
?- prism(hmm).

compiled in 4 milliseconds
```

[1] Since version 1.9, if we wish, we can confirm even at this point whether it is possible to run sampling or the explanation search. To be more concrete, let us include only the declarations and the modeling part to the file named '`hmm.psm`', and load the program:

```
% prism
  :
?- prism(hmm).
```

Then, for example, we may run the following to sample a goal with a string `X` and get the explanations for it:

```
?- sample(hmm(X)),probf(hmm(X)).
```

It should be noted that `sample/1` and `probf/1` simulate sampling execution and explanation search, respectively. Also one may notice that, since we have no specific parameter settings for switches here, the sampling is made under the (default) uniform parameters.

```
loading::hmm.psm.out

yes
```

Then we run the batch predicate to generate 100 samples and to learn the parameters from them:

```
?- hmm_learn(100).

#goals: 0.........(93)
Exporting switch information to the EM routine ...
#em-iterations: 0......(63) (Converged: -683.493898022)
Statistics on learning:
        Graph size: 5520
        Number of switches: 5
        Number of switch instances: 10
        Number of iterations: 63
        Final log likelihood: -683.493898022
        Total learning time: 0.020 seconds
        Explanation search time: 0.008 seconds
        Total table space used: 728832 bytes
Type show_sw or show_sw_b to show the probability distributions.
```

We can confirm the learned parameters by the built-in `show_sw/0` (§4.2.5):[2]

```
?- show_sw.

Switch init: unfixed_p: s0 (p: 0.722841424) s1 (p: 0.277158576)
Switch out(s0): unfixed_p: a (p: 0.623359863) b (p: 0.376640137)
Switch out(s1): unfixed_p: a (p: 0.497027993) b (p: 0.502972007)
Switch tr(s0): unfixed_p: s0 (p: 0.554684130) s1 (p: 0.445315870)
Switch tr(s1): unfixed_p: s0 (p: 0.550030827) s1 (p: 0.449969173)
```

Here we can make some probabilistic inferences based on the parameters estimated as above. To compute the most probable explanation (the Viterbi explanation) and its probability (the Viterbi probability) for a given observation, we can use the built-in `viterbif/1` (§4.6).

```
| ?- viterbif(hmm([a,a,a,a,a,b,b,b,b,b])).

hmm([a,a,a,a,a,b,b,b,b,b])
   <= hmm(1,10,s0,[a,a,a,a,a,b,b,b,b,b]) & msw(init,s0)
hmm(1,10,s0,[a,a,a,a,a,b,b,b,b,b])
   <= hmm(2,10,s0,[a,a,a,a,b,b,b,b,b]) & msw(out(s0),a) & msw(tr(s0),s0)
hmm(2,10,s0,[a,a,a,a,b,b,b,b,b])
   <= hmm(3,10,s0,[a,a,a,b,b,b,b,b]) & msw(out(s0),a) & msw(tr(s0),s0)
hmm(3,10,s0,[a,a,a,b,b,b,b,b])
   <= hmm(4,10,s0,[a,a,b,b,b,b,b]) & msw(out(s0),a) & msw(tr(s0),s0)
hmm(4,10,s0,[a,a,b,b,b,b,b])
   <= hmm(5,10,s0,[a,b,b,b,b,b]) & msw(out(s0),a) & msw(tr(s0),s0)

  ...omitted...

hmm(8,10,s1,[b,b,b])
   <= hmm(9,10,s1,[b,b]) & msw(out(s1),b) & msw(tr(s1),s1)
```

[2] At least there are many local maxima for ML estimation, so it is not guaranteed that we can restore the parameters that have been set by `set_params/0`.

```
hmm(9,10,s1,[b,b])
   <= hmm(10,10,s1,[b]) & msw(out(s1),b) & msw(tr(s1),s1)
hmm(10,10,s1,[b])
   <= hmm(11,10,s0,[]) & msw(out(s1),b) & msw(tr(s1),s0)
hmm(11,10,s0,[])

Viterbi_P = 0.000002081735251
```

On the other hand, to compute the hindsight probabilities (§4.7) of subgoals for a goal `hmm([a,a,a,a,a,b,b,b,b,b])`, we may run:

```
| ?- hindsight(hmm([a,a,a,a,a,b,b,b,b,b])).

hindsight probabilities:
  hmm(1,10,s0,[a,a,a,a,a,b,b,b,b,b]): 0.000710038386251
  hmm(1,10,s1,[a,a,a,a,a,b,b,b,b,b]): 0.000216848626541
  hmm(2,10,s0,[a,a,a,a,b,b,b,b,b]): 0.000564388970965
  hmm(2,10,s1,[a,a,a,a,b,b,b,b,b]): 0.000362498041827
  hmm(3,10,s0,[a,a,a,b,b,b,b,b]): 0.000563735498733
  hmm(3,10,s1,[a,a,a,b,b,b,b,b]): 0.000363151514060

  ...omitted...

  hmm(8,10,s0,[b,b,b]): 0.000444735040586
  hmm(8,10,s1,[b,b,b]): 0.000482151972207
  hmm(9,10,s0,[b,b]): 0.000444736503096
  hmm(9,10,s1,[b,b]): 0.000482150509696
  hmm(10,10,s0,[b]): 0.000445050456081
  hmm(10,10,s1,[b]): 0.000481836556711
  hmm(11,10,s0,[]): 0.000511887384988
  hmm(11,10,s1,[]): 0.000414999627805
```

According to the purpose, the queries above can be included to the batch predicate in the utility part.

By specifying the execution flags (§4.14), we can add some variations to learning or the other probabilistic inferences. For example, we may conduct an MAP estimation with the pseudo count being 0.5, and try 10 runs of the EM algorithm. To do this, we first set the flags for multiple rules of the EM algorithm as follows:

```
?- set_prism_flag(restart,10).
```

Next we set all pseudo counts to 0.5:

```
?- set_sw_all_h(_,0.5).
```

Now the batch predicate and the routines for later probabilistic inferences can be run in the same way as above:

```
?- hmm_learn(100).

#goals: 0.........(98)
Exporting switch information to the EM routine ...
[0] #em-iterations: 0.........100.(115) (Converged: -692.022272523)
[1] #em-iterations: 0.........100.(115) (Converged: -692.022846163)
[2] #em-iterations: 0.........100..(130) (Converged: -692.028058623)
[3] #em-iterations: 0.........100.........200...(240) (Converged: -692.0
24704657)
```

```
[4] #em-iterations: 0.......(79) (Converged: -692.022673972)
[5] #em-iterations: 0......(62) (Converged: -692.024814351)
[6] #em-iterations: 0.........100.........(192) (Converged: -692.0231354
79)
[7] #em-iterations: 0.........100.(111) (Converged: -692.020478776)
[8] #em-iterations: 0.........100.........200..(228) (Converged: -692.03
1937456)
[9] #em-iterations: 0(2) (Converged: -692.010584638)
Statistics on learning:
        Graph size: 5840
        Number of switches: 5
        Number of switch instances: 10
        Number of iterations: 2
        Final log of a posteriori prob: -692.010584638
        Total learning time: 0.148 seconds
        Explanation search time: 0.008 seconds
        Total table space used: 770832 bytes
Type show_sw or show_sw_b to show the probability distributions.
```

If we always use the above flag values, it should be useful to include the following queries into the utility part:

```
:- set_prism_flag(restart,10).
:- set_prism_flag(default_sw_h,0.5).
```

By the latter query we can give the default pseudo counts as 0.5, instead of setting the pseudo counts manually using `set_sw_all_h/2`.

Furthermore, let us conduct a batch execution of learning at the shell (or command prompt) level. As a preparation, we define a clause with `prism_main/1` (see §3.7) as follows:

```
prism_main([Arg]):-
  parse_atom(Arg,N),
  hmm_learn(N).
```

With this definition, the system receives one argument `Arg` from the shell an atomic symbol (for example, `'100'`) and then converts such a symbol to the data `N` which can be numerically handled (i.e. as an integer), and finally the batch predicate used above is invoked with the argument `N`. So if we run the command `upprism` at the shell prompt with specifying the filename of the program and the argument to be passed to `prism_main/1` above:

```
% upprism hmm 50
```

then a learning with 50 samples will be conducted:

```
% upprism hmm 50
  :
#goals: 0....(49)
Exporting switch information to the EM routine ...
[0] #em-iterations: 0.........100......(163) (Converged: -347.326727176)
[1] #em-iterations: 0.........100.....(151) (Converged: -347.326798056)
[2] #em-iterations: 0.........100.........200........(289) (Converged: -347
.330719096)
[3] #em-iterations: 0.........100.........(194) (Converged: -347.326873331)
[4] #em-iterations: 0.........100.........200.........(293) (Converged: -34
```

```
7.330935748)
[5] #em-iterations: 0.........100.........200........(287) (Converged: -347
.330848992)
[6] #em-iterations: 0.........100........(185) (Converged: -347.327995530)
[7] #em-iterations: 0.........100.......(180) (Converged: -347.327563031)
[8] #em-iterations: 0.........100........(189) (Converged: -347.327339025)
[9] #em-iterations: 0.........100......(163) (Converged: -347.327150784)
Statistics on learning:
        Graph size: 3400
        Number of switches: 5
        Number of switch instances: 10
        Number of iterations: 163
        Final log of a posteriori prob: -347.326727176
        Total learning time: 0.124 seconds
        Explanation search time: 0.004 seconds
        Total table space used: 447392 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
%
```

It is worth noting that the control is returned back to the shell after the execution, so we can make more flexible experiments by combining this batch execution with the other facilities in a shell script.

7.2 Probabilistic context-free grammars

Probabilistic context-free grammars (PCFGs) are another well-known model class that can handle sequences of symbols. A PCFG is a context-free grammar whose production rules are annotated probabilities. Starting from the start symbol and applying production rules one by one, with a probability annotated to the rule, we can generate a sequence of terminal symbols (i.e. a sentence). Figure 7.1 shows an example of a PCFG introduced in [4], where 's' is the start symbol.

Now let us write a PRISM program that represents the PCFG in Figure 7.1. We first show the declarations:

```
target(pcfg/1).

values(s,[[np,vp],[vp]]).
values(np,[[noun],[noun,pp],[noun,np]]).
values(vp,[[verb],[verb,np],[verb,pp],[verb,np,pp]]).
values(pp,[[prep,np]]).
values(verb,[[swat],[flies],[like]]).
values(noun,[[swat],[flies],[ants]]).
values(prep,[[like]]).

:- p_not_table proj/2.
```

By `target/1`, we declare that the goals of the form `pcfg(`*Words*`)` will be observed, where *Words* is a sentence to be generated. It is seen from the `values` declarations that we use random switches whose instance takes the form `msw(`A`,[`B_1`,`B_2`,...,`B_n`])`, which represents a probabilistic event "a production rule $A \to B_1 B_2 \cdots B_n$ is chosen." Then, the parameter of a switch instance `msw(`A`,[`B_1`,`B_2`,...,`B_n`])` corresponds to the rule probability of $A \to B_1 B_2 \cdots B_n$. In this example, we will not table the probabilistic predicates `proj/2` (this is just for making the inference results simple and readable; see §2.6.4). We may write the modeling part as follows:

s	→	np vp	(0.8)	pp	→	prep np	(1.0)
s	→	vp	(0.2)				
np	→	noun	(0.4)	verb	→	swat	(0.2)
np	→	noun pp	(0.4)	verb	→	flies	(0.4)
np	→	noun np	(0.2)	verb	→	like	(0.4)
vp	→	verb	(0.3)	noun	→	swat	(0.05)
vp	→	verb np	(0.3)	noun	→	flies	(0.45)
vp	→	verb pp	(0.2)	noun	→	ants	(0.5)
vp	→	verb np pp	(0.2)	prep	→	like	(1.0)

Figure 7.1: Example of a probabilistic context-free grammar from [4].

```
pcfg(L):- pcfg(s,L-[]).

pcfg(LHS,L0-L1):-
  ( nonterminal(LHS) -> msw(LHS,RHS),proj(RHS,L0-L1)
  ; L0 = [LHS|L1]
  ).

proj([],L-L).
proj([X|Xs],L0-L1):-
  pcfg(X,L0-L2),proj(Xs,L2-L1).

nonterminal(s).
nonterminal(np).
nonterminal(vp).
nonterminal(pp).
nonterminal(verb).
nonterminal(noun).
nonterminal(prep).
```

`pcfg/1-2` and `proj/2` are generic in the sense that these predicates can be applied to any underlying context-free grammar which does not include ε-rules.[3] Also, as is usually done for definite clause grammars, we use difference lists to represent the substrings. The if-then statement `nonterminal(LHS) -> ...` in the body of `pcfg/2` is used to check if `LHS` is a non-terminal symbol. Lastly, in the utility part, we assign the rule probabilities by using query statements:

```
:- set_sw(s,[0.8,0.2]).
:- set_sw(np,[0.4,0.4,0.2]).
:- set_sw(vp,[0.3,0.3,0.2,0.2]).
:- set_sw(pp,[1.0]).
:- set_sw(verb,[0.2,0.4,0.4]).
:- set_sw(noun,[0.05,0.45,0.5]).
:- set_sw(prep,[1.0]).
```

Let us run the program. First, we compute the generative probability of a sentence "swat flies like ants." `prob/1` can be utilized for this purpose:

```
?- prob(pcfg([swat,flies,like,ants])).

Probability of pcfg([swat,flies,like,ants]) is: 0.001010560000000
```

[3] We also assume that the underlying grammar does not produce a unit chain $A \stackrel{*}{\Rightarrow} A$.

We can also get the most probable parse tree for "swat flies like ants." This is nothing but probabilistic parsing using a PCFG model. Looking into the result of `viterbif/1` shown below,[4] it can be found that the most probable parse tree is $[[\text{swat}_{\text{verb}}[\text{flies}_{\text{noun}}[\text{like}_{\text{prep}}\ [\text{ants}_{\text{noun}}]_{\text{np}}]_{\text{pp}}]_{\text{np}}]_{\text{vp}}]_{\text{s}}$, and its generative probability is 0.000432.

```
?- viterbif(pcfg([swat,flies,like,ants])).

pcfg([swat,flies,like,ants])
   <= pcfg(s,[swat,flies,like,ants]-[])
pcfg(s,[swat,flies,like,ants]-[])
   <= pcfg(vp,[swat,flies,like,ants]-[]) & msw(s,[vp])
pcfg(vp,[swat,flies,like,ants]-[])
   <= pcfg(verb,[swat,flies,like,ants]-[flies,like,ants])
        & pcfg(np,[flies,like,ants]-[]) & msw(vp,[verb,np])
pcfg(verb,[swat,flies,like,ants]-[flies,like,ants])
   <= pcfg(swat,[swat,flies,like,ants]-[flies,like,ants]) & msw(verb,[swat])
pcfg(swat,[swat,flies,like,ants]-[flies,like,ants])
pcfg(np,[flies,like,ants]-[])
   <= pcfg(noun,[flies,like,ants]-[like,ants])
      & pcfg(pp,[like,ants]-[]) & msw(np,[noun,pp])
pcfg(noun,[flies,like,ants]-[like,ants])
   <= pcfg(flies,[flies,like,ants]-[like,ants]) & msw(noun,[flies])
pcfg(flies,[flies,like,ants]-[like,ants])
pcfg(pp,[like,ants]-[])
   <= pcfg(prep,[like,ants]-[ants]) & pcfg(np,[ants]-[]) & msw(pp,[prep,np])
pcfg(prep,[like,ants]-[ants])
   <= pcfg(like,[like,ants]-[ants]) & msw(prep,[like])
pcfg(like,[like,ants]-[ants])
pcfg(np,[ants]-[])
   <= pcfg(noun,[ants]-[]) & msw(np,[noun])
pcfg(noun,[ants]-[])
   <= pcfg(ants,[ants]-[]) & msw(noun,[ants])
pcfg(ants,[ants]-[])

Viterbi_P = 0.000432
```

Furthermore, using `n_viterbif/2`, we can get the three most probable parse trees for "swat flies like ants" as follows:

```
?- n_viterbif(3,pcfg([swat,flies,like,ants])).
```

7.3 Discrete Bayesian networks

7.3.1 Representing Bayesian networks

Bayesian networks have become a popular representation for encoding and reasoning about uncertainty in various applications. A Bayesian network is a directed acyclic graph whose nodes are considered as random variables and whose directed edges indicate conditional independencies among such variables. Conditional probability tables (CPTs) in a Bayesian network can be represented by switches with *complex* names in PRISM. To be more specific, let B and C be two random variables, and assume B (resp. C) has

[4] For the space limitation, we have inserted some line breaks and indentions.

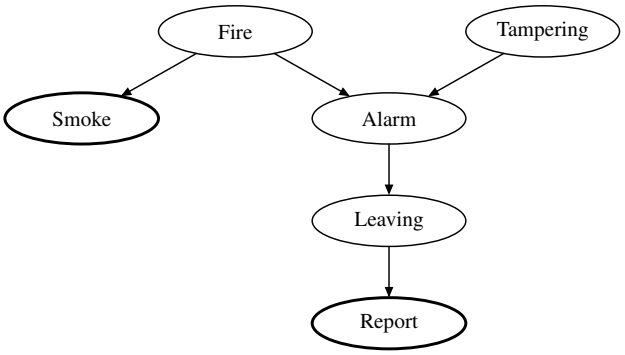

Figure 7.2: Example of a discrete Bayesian network.

the k (resp. n) possible values. Then a conditional distribution $P(B|C)$ can be represented by n switches: `msw(b(`c_i`),·)` $(i = 1, \ldots, n)$, each of which has k outcomes: $v_{i,j}$ $(j = 1, \ldots, k)$.[5] Then it is easily seen that each switch parameter corresponds to one entry of the CPT.

For illustration, let us consider an example from [25], shown in Figure 7.2. In this network, we assume that all random variables take on *yes* or *no* (i.e. they are binary), and also assume that only two nodes, *Smoke* and *Report*, are observable. This Bayesian network defines a joint distribution:

$$p(Fire, Tampering, Smoke, Alarm, Leaving, Report).$$

From the conditional independencies indicated by the graph structure, this joint distribution is reduced to a computationally feasible form:

$$\begin{aligned} & p(Fire, Tampering, Smoke, Alarm, Leaving, Report) \\ = \;& p(Fire)p(Tampering)p(Smoke \mid Fire) \cdot \\ & \quad p(Alarm \mid Fire, Tampering)p(Leaving \mid Alarm)p(Report \mid Leaving). \end{aligned} \tag{7.1}$$

The factored probabilities in the RHS will be stored in CPTs, where $P(Fire)$ and $P(Tampering)$ are seen as conditional probabilities with an empty condition. On the other hand, the observable distribution on *Smoke* and *Report* is computed by marginalizing the joint distribution:

$$\begin{aligned} & p(Smoke, Report) \\ = \;& \sum_{Fire,\ Tampering,\ Alarm,\ Leaving} p(Fire, Tampering, Smoke, Alarm, Leaving, Report). \end{aligned} \tag{7.2}$$

It is easy to notice that the marginalization above takes an exponential time with respect to the number of variable to marginalize. In the literature of research on Bayesian networks, efficient algorithms are known to compute such marginalization, but in this section, we concentrate on how we represent Bayesian networks in PRISM. Indeed, for a certain class called singly-connected Bayesian networks, it is shown in [35] that we can write a PRISM program that can simulate the Pearl's propagation algorithm.

Now we start to describe the Bayesian network in Figure 7.2. Also for this case, a generative way of thinking should be useful in writing the modeling part. For example, we first get the value of *Fire* by flipping a coin (i.e. sampling) according to $P(Fire)$. We then proceed to flip a coin for *Smoke* according to $P(Smoke \mid Fire)$, and so on. Here we represent such a coin flipping by `msw(I,V)`, and define the joint distribution (Eq. 7.1) with a predicate `world/6`:

[5] In other words, we have $(n \times k)$ switch instances: `msw(b(`c_i`),`$v_{i,j}$`)` $(i = 1, \ldots, n$ and $j = 1, \ldots, k)$.

```
world(Fi,Ta,Al,Sm,Le,Re) :-
   msw(fi,Fi),
   msw(ta,Ta),
   msw(sm(Fi),Sm),
   msw(al(Fi,Ta),Al),
   msw(le(Al),Le),
   msw(re(Le),Re).
```

This clause indicates that we flip the coins in the order of *Fire*, *Tampering*, *Smoke*, *Alarm*, *Leaving* and *Report*. As is declared later, the switches above are assumed here to output `yes` or `no`. The switch named `fi` corresponds to the coin flipping for *Fire*, and switch `sm(Fi)` corresponds to the coin flipping for *Smoke*, given the value of *Fire* as `Fi`. Recall here that each parameter of these switches corresponds to one entry of the CPTs in the target Bayesian network. For instance, the parameter $\theta_{\texttt{sm(yes)},\texttt{no}}$, the probability of a switch instance `msw(sm(yes),no)` being true corresponds to the conditional probability $P(Smoke = no \mid Fire = yes)$.

The observable distribution is defined by `world/2`:

```
world(Sm,Re) :- world(_,_,_,Sm,_,Re).
```

The probability of `world(yes,no)` corresponds to $P(Smoke = yes, Report = no)$. We can find that, for `world(yes,no)`, all instantiations of the body are probabilistically exclusive to each other, so we can compute the probability of `world(yes,no)` by summing up the probabilities of these instantiations. This fact correspond to Eq. 7.2, so we can say the program precisely express what we would like to model. The model part of our Bayesian network program consists of the two clauses above.

We add some declarations as follows:

```
target(world,2).
data(user).
values(_,[yes,no]).
```

The first clause means `world/2` is observable, and from the second clause, we can use the built-in `learn/1` for learning, by passing a list of observed goals to its arguments. The third clause specifies all switches have outcomes `yes` and `no`.

Now let us make a similar experiment to that with the HMM program (§7.1). Namely, we first generate goals by sampling as training data under some predefined parameters, and then learn the parameters from such training data. The difference is that we attempt to *fix* (or preserve) one parameter in learning. Such a parameter can be considered as a constant parameter in the model. The utility part may contain the following batch predicate for the experiment:

```
alarm_learn(N) :-
  unfix_sw(_),                    % Make all parameters changeable
  set_params,                     % Set parameters as you specified
  get_samples(N,world(_,_),Gs),   % Get N samples
  fix_sw(fi),                     % Preserve the parameter values
  learn(Gs).                      %   for {msw(fi,yes), msw(fi,no)}
```

The experimental steps are written as comments. In this predicate, `set_params/0` (which specifies the parameters of all switches; §4.2.3), `get_samples/3` (which generate training data; §4.3), and `learn/1` (§4.8.3) are used similarly to those in the batch routine for the experiments with HMMs (§7.1). `set_params/0` is a user-defined predicate:

```
set_params :-
  set_sw(fi,[0.1,0.9]),
```

```
  set_sw(ta,[0.15,0.85]),
  set_sw(sm(yes),[0.95,0.05]),
  set_sw(sm(no),[0.05,0.95]),
  set_sw(al(yes,yes),[0.50,0.50]),
  set_sw(al(yes,no),[0.90,0.10]),
  set_sw(al(no,yes),[0.85,0.15]),
  set_sw(al(no,no),[0.05,0.95]),
  set_sw(le(yes),[0.88,0.12]),
  set_sw(le(no),[0.01,0.99]),
  set_sw(re(yes),[0.75,0.25]),
  set_sw(re(no),[0.10,0.90]).
```

As described above, the additional functionality is that we do not learn (i.e. fix or preserve) the parameters for switch `fi`. This is done by using the built-ins `unfix_sw/1` and `fix_sw/1` (§4.2.4).

Now our PRISM program has been completed, and we are ready to run the program. Let us assume that the program is contained in the file '`alarm.psm`', then load the program by the command `prism(alarm)`:

```
?- prism(alarm).
```

We conduct learning with 500 samples by `alarm_learn/1` which is previously defined:

```
?- alarm_learn(500).

#goals: 0(4)
Exporting switch information to the EM routine ...
#em-iterations: 0(2) (Converged: -464.034430688)
Statistics on learning:
        Graph size: 448
        Number of switches: 12
        Number of switch instances: 24
        Number of iterations: 2
        Final log likelihood: -464.034430688
        Total learning time: 0.004 seconds
        Explanation search time: 0.000 seconds
        Total table space used: 47008 bytes
Type show_sw or show_sw_b to show the probability distributions.
```

We can confirm the learned parameters as follows:

```
?- show_sw.

Switch fi: fixed_p: yes (p: 0.100000000) no (p: 0.900000000)
Switch ta: unfixed_p: yes (p: 0.682231979) no (p: 0.317768021)
Switch le(no): unfixed_p: yes (p: 0.419688112) no (p: 0.580311888)
Switch le(yes): unfixed_p: yes (p: 0.476437741) no (p: 0.523562259)
Switch re(no): unfixed_p: yes (p: 0.283975504) no (p: 0.716024496)
Switch re(yes): unfixed_p: yes (p: 0.167325271) no (p: 0.832674729)
Switch sm(no): unfixed_p: yes (p: 0.130802678) no (p: 0.869197322)
Switch sm(yes): unfixed_p: yes (p: 0.122775877) no (p: 0.877224123)
Switch al(no,no): unfixed_p: yes (p: 0.480950708) no (p: 0.519049292)
Switch al(no,yes): unfixed_p: yes (p: 0.451939009) no (p: 0.548060991)
Switch al(yes,no): unfixed_p: yes (p: 0.472514062) no (p: 0.527485938)
Switch al(yes,yes): unfixed_p: yes (p: 0.380557386) no (p: 0.619442614)
```

It is also possible to get the frequencies of the sampled goals:

```
?- show_goals.

Goal world(yes,yes) (count=34, freq=6.800%)
Goal world(no,no) (count=353, freq=70.600%)
Goal world(yes,no) (count=31, freq=6.200%)
Goal world(no,yes) (count=82, freq=16.400%)
Total_count=500
```

7.3.2 Computing conditional probabilities

Furthermore, for the Bayesian network program described in this section, conditional probabilities can be computed as conditional hindsight probabilities (§4.7). Let us recall that a conditional hindsight probability is denoted as $P_\theta(G'|G) = P_\theta(G')/P_\theta(G)$, where G is a given top goal and G' is one of its subgoals. For instance, let us consider to compute the conditional probability $p(Alarm \mid Smoke = yes, Report = no)$ by using conditional hindsight probabilities. Since the target conditional probability $p(Alarm = x \mid Smoke = yes, Report = no)$ can be computed as $p(Alarm = x, Smoke = yes, Report = no)/p(Smoke = yes, Report = no)$, if we let $G =$ `world(_,_,_,yes,_,no)` and $G' =$ `world(_,_,x,yes,_,no)`, it can be seen that $P_\theta(G'|G)$ is equal to the target conditional probability. To get the conditional distribution on *Alarm*, we run `chindsight_agg/2` with specifying the third argument in `world/6` (which corresponds to *Alarm*) as a query argument:[6]

```
?- chindsight_agg(world(_,_,_,yes,_,no),world(_,_,query,yes,_,no)).
conditional hindsight probabilities:
  world(*,*,no,yes,*,no): 0.620773027495463
  world(*,*,yes,yes,*,no): 0.379226972504537
```

Of course, from the definition of `world/2`, we can make the same computation with `world/2`:

```
?- chindsight_agg(world(yes,no),world(_,_,query,yes,_,no)).
conditional hindsight probabilities:
  world(*,*,no,yes,*,no): 0.620773027495463
  world(*,*,yes,yes,*,no): 0.379226972504537
```

As mentioned before, the definition of `world/6` is computationally naive, so we need to write a different representation of Bayesian networks which takes into account the computational effort for conditional hindsight probabilities, as shown in the next section.

7.3.3 Bayesian networks in a junction-tree form

For probabilistic inferences on Bayesian networks, especially, on multiply-connected Bayesian networks (BNs), several sophisticated techniques have been proposed so far. As another example of a BN, let us consider a Bayesian network called the Asia network [22], which is illustrated in Figure 7.3. This network can be said to be a multiply-connected BN since there are two paths from S to D: $S \to L \to TL \to D$ and $S \to B \to D$. One of the most popular inference methods for such multiply-connected BNs is the junction-tree algorithm. In the junction-tree algorithm, we first convert the original network to an undirected tree-structured network called a junction tree, whose node corresponds to a set consisting of one or more original nodes. Figure 7.4 depicts a junction tree for the Asia network. For example, α_2 in Figure 7.4 corresponds to a set $\{S, L, B\}$ of the original nodes in Figure 7.3.

We can write a 'naive' version of the PRISM program that represents the Asia network as did in the previous section. Also in this program, all switches are supposed to be binary, i.e. they take values '`t`'

[6] In this computation, it is assumed that the parameters are set by `set_params/0` in advance.

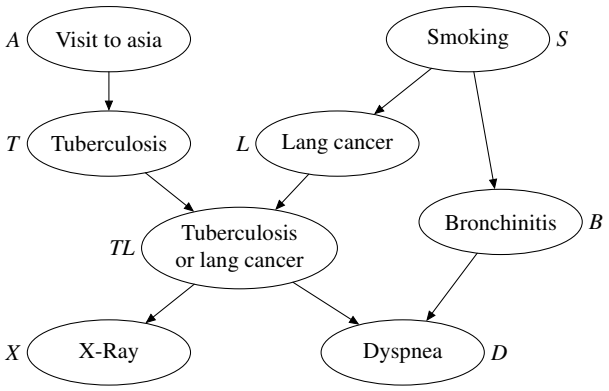

Figure 7.3: Example of a multiply-connected Bayesian network (known as the Asia network).

(true) and 'f' (false). `incl_or/3` represents the inclusive OR. We set the parameters given in [22] by `set_params/0`.

```
target(world/4).
values(bn(_,_),[t,f]).

world(A,S,X,D):- world(A,_,S,_,_,X,_,D).

world(A,T,S,L,TL,X,B,D) :-
   msw(bn(a,[]),A),msw(bn(t,[A]),T),
   msw(bn(s,[]),S),msw(bn(l,[S]),L),
   incl_or(T,L,TL),
   msw(bn(x,[TL]),X),msw(bn(b,[S]),B),
   msw(bn(d,[TL,B]),D).

incl_or(t,t,t).
incl_or(t,f,t).
incl_or(f,t,t).
incl_or(f,f,f).

:- set_params.

set_params:-
  set_sw(bn(a,[]),[0.01,0.99]),
  set_sw(bn(t,[t]),[0.05,0.95]),
  set_sw(bn(t,[f]),[0.01,0.99]),
  set_sw(bn(s,[]),[0.5,0.5]),
  set_sw(bn(l,[t]),[0.1,0.9]),
  set_sw(bn(l,[f]),[0.01,0.99]),
  set_sw(bn(x,[t]),[0.98,0.02]),
  set_sw(bn(x,[f]),[0.05,0.95]),
  set_sw(bn(b,[t]),[0.60,0.40]),
  set_sw(bn(b,[f]),[0.30,0.70]),
  set_sw(bn(d,[t,t]),[0.90,0.10]),
  set_sw(bn(d,[t,f]),[0.70,0.30]),
  set_sw(bn(d,[f,t]),[0.80,0.20]),
  set_sw(bn(d,[f,f]),[0.10,0.90]).
```

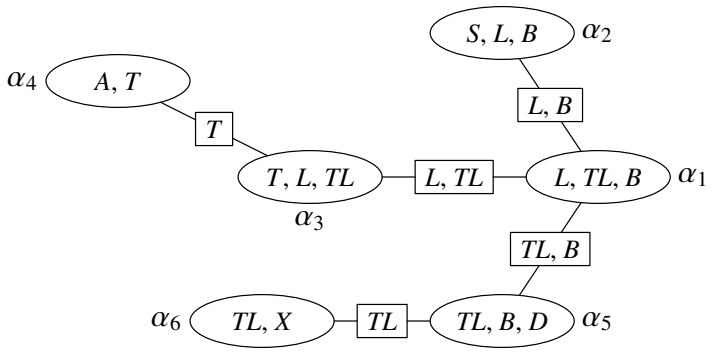

Figure 7.4: Junction tree for the Asia network.

After loading the program, for example, we can compute the conditional distribution $P(T = true \mid A = false, D = true) = 0.018$ and $P(T = false \mid A = false, D = true) = 0.982$ as follows:

```
?- chindsight_agg(world(f,_,_,t),world(_,query,_,_,_,_,_,_)).
conditional hindsight probabilities:
  world(*,f,*,*,*,*,*,*): 0.981873562361255
  world(*,t,*,*,*,*,*,*): 0.0181264376387 45
```

Surely this program returns the consistent results, but it is not so efficient. On the other hand, let us see another PRISM program that represents a junction tree and is expected to run faster than the naive version. For the readers who are interested in the formal discussion on such PRISM programs in a junction-tree form, please consult [32]. It is possible to implement a translator (including a junction-tree converter) from a network specification in some standard format (e.g. XMLBIF) to a PRISM program of the corresponding junction tree. For instance, the following is a junction-tree version of the PRISM program for the Asia network:

```
target(world/1).
values(bn(_,_),[t,f]).

world(E):- msg_1_0(E-[]).

msg_1_0(E0-E1)       :- node_1(L,TL,B,E0-E1).
msg_2_1(L,B,E0-E1)   :- node_2(S,L,B,E0-E1).
msg_3_1(L,TL,E0-E1)  :- node_3(T,L,TL,E0-E1).
msg_4_3(T,E0-E1)     :- node_4(A,T,E0-E1).
msg_5_1(TL,B,E0-E1)  :- node_5(TL,B,D,E0-E1).
msg_6_5(TL,E0-E1)    :- node_6(TL,X,E0-E1).

node_1(L,TL,B,E0-E1) :-
    msg_2_1(L,B,E0-E2),msg_3_1(L,TL,E2-E3),msg_5_1(TL,B,E3-E1).
node_2(S,L,B,E0-E1)  :-
    cpt(s,[],S,E0-E2),cpt(l,[S],L,E2-E3),cpt(b,[S],B,E3-E1).
node_3(T,L,TL,E0-E1) :- incl_or(L,T,TL),msg_4_3(T,E0-E1).
node_4(A,T,E0-E1)    :- cpt(a,[],A,E0-E2),cpt(t,[A],T,E2-E1).
node_5(TL,B,D,E0-E1) :- cpt(d,[TL,B],D,E0-E2),msg_6_5(TL,E2-E1).
node_6(TL,X,E0-E1)   :- cpt(x,[TL],X,E0-E1).

cpt(X,Par,V,E0-E1):- ( E0=[(X,V)|E1] -> true ; E0=E1 ),msw(bn(X,Par),V).
```

Table 7.1: CPT for *Alarm* constructed by the noisy-OR rule

Fire	*Tampering*	$P(alarm)$	$P(\neg alarm)$
true	*true*	$0.94 = 1 - 0.3 \times 0.2$	$0.06 = 0.3 \times 0.2$
true	*false*	$0.7 = 1 - 0.3$	0.3
false	*true*	$0.8 = 1 - 0.2$	0.2
false	*false*	0	1

```
incl_or(t,t,t).
incl_or(t,f,t).
incl_or(f,t,t).
incl_or(f,f,f).
```

In this program, we consider that α_1 in Figure 7.4 is the root node of the junction tree. The predicate whose name is `msg_i_j` corresponds to the edge between nodes i and j in the junction tree. We also define a predicate named `node_i` for each node i in the junction tree. One may find that the evidences will be kept as difference lists in the last arguments of the `msg_i_j` and the `node_i` predicates. We can input evidences through the argument of `world/1`, but for simplicity, the evidences are assumed here to be given in the same order as that of the appearances of `msw/2` in the top-down execution of `world/1`. `cpt/4` is a 'wrapper' predicate that can handle evidences. We omit here `set_params/0` which is also included in the naive version.

Using this program, let us compute the conditional distribution $P(T \mid A = false, D = true)$. To realize this, We attempt to compute the hindsight probabilities for the predicate `node_4/3` since α_4 includes the original node (i.e. the random variable) T, as shown in Figure 7.4.

```
?- chindsight_agg(world([(a,f),(d,t)]),node_4(_,query,_)).
conditional hindsight probabilities:
  node_4(*,f,*): 0.981873562361255
  node_4(*,t,*): 0.018126437638745
```

It is proved in [32] that this hindsight computation is equivalent to the belief propagtion procedure in a junction tree.

7.3.4 Using noisy OR

In modeling with Bayesian networks, we sometimes use *combination rules* to make the CPTs simpler, and *noisy OR* is one of the most well-known combination rules [28]. To be specific, let us consider the alarm network (Figure 7.2) again, and suppose that the *Alarm* node in the alarm network has a CPT defined with the noisy-OR rule. Also we suppose that the individual inhibition probabilities are given as follows:[7]

$$\begin{aligned} P(\neg alarm \mid fire, \neg tamparing) &= 0.3 \\ P(\neg alarm \mid \neg fire, tamparing) &= 0.2. \end{aligned}$$

Then we have a CPT for *Alarm* shown in Table 7.1. To write the alarm network program that deals with the noisy-OR rules, we modify the definitions of `world/6` and introduce the predicates named `cpt_x` for each variable named x. Then `world/6` calls such `cpt_x` predicates instead of directly calling random switches. The modeling part of the resulting program is as follows:

[7] We denote the propositions $Alarm = true$, $Alarm = false$, $Fire = true$, and so on by *alarm*, $\neg$*alarm*, *fire*, and so on, respectively.

```
world(Fi,Ta,Al,Sm,Le,Re) :-
   cpt_fi(Fi),
   cpt_ta(Ta),
   cpt_sm(Fi,Sm),
   cpt_al(Fi,Ta,Al),
   cpt_le(Al,Le),
   cpt_re(Le,Re).

cpt_fi(Fi):- msw(fi,Fi).
cpt_ta(Ta):- msw(ta,Ta).
cpt_sm(Fi,Sm):- msw(sm(Fi),Sm).
cpt_al(Fi,Ta,Al):-
   ( Fi = yes, Ta = yes ->
       msw(cause_al_fi,N_Al_Fi),
       msw(cause_al_ta,N_Al_Ta),
       ( N_Al_Fi = no, N_Al_Ta = no -> Al = no
       ; Al = yes
       )
   ; Fi = yes, Ta = no  -> msw(cause_al_fi,Al)
   ; Fi = no,  Ta = yes -> msw(cause_al_ta,Al)
   ; Fi = no,  Ta = no  -> Al = no
   ).
cpt_le(Al,Le):- msw(le(Al),Le).
cpt_re(Le,Re):- msw(re(Le),Re).
```

It can be seen that `cpt_al/3` is an implementation of the noisy-OR rule. The key step is to consider the generation process underlying the noisy-OR rule. For example, when *Fire = true* and *Tampering = true*, we make choices twice by random switches named `cause_al_fi` and `cause_al_ta` according to the corresponding inhibition probabilities. Then, if one of these choices returns `yes`, we consider that *Alarm* becomes true.

Let us further write a more generic version. We first write the network-specific part of the model by modifying the definition of `world/6` and by adding `noisy_or/3` for the specifications of noisy-OR nodes:

```
world(Sm,Re):- world(_,_,_,Sm,_,Re).

world(Fi,Ta,Al,Sm,Le,Re) :-
   cpt(fi,[],Fi),
   cpt(ta,[],Ta),
   cpt(sm,[Fi],Sm),
   cpt(al,[Fi,Ta],Al),
   cpt(le,[Al],Le),
   cpt(re,[Le],Re).

noisy_or(al,[fi,ta],[[0.7,0.3],[0.8,0.2]]).
```

In the above, `cpt/3` in the clause body of `world/6` is an abstract (or a wrapper) predicate that can deal with the noisy-OR rule, and its definition is included in the network-independent part of the model:

```
:- p_not_table choose_noisy_or/4, choose_noisy_or/6.

cpt(X,PaVs,V):-
   ( noisy_or(X,Pa,_) -> choose_noisy_or(X,Pa,PaVs,V)
   ; msw(bn(X,PaVs),V)
   ).
```

```
choose_noisy_or(X,Pa,PaVs,V):- choose_noisy_or(X,Pa,PaVs,no,no,V).

choose_noisy_or(_,[],[],yes,V,V).
choose_noisy_or(_,[],[],no,_,no).
choose_noisy_or(X,[Y|Pa],[PaV|PaVs],PaHasYes0,ValHasYes0,V):-
   ( PaV=yes ->
       msw(cause(X,Y),V0),
       PaHasYes=yes,
       ( ValHasYes0=no, V0=no -> ValHasYes=no
       ; ValHasYes=yes
       )
   ; PaHasYes=PaHasYes0,
     ValHasYes=ValHasYes0
   ),
   choose_noisy_or(X,Pa,PaVs,PaHasYes,ValHasYes,V).
```

`choose_noisy_or/4` is a generalization of `cpt_al/3` described above. Some might feel this network-independent part procedural, but conversely we can say that this exhibits the flexibility of the PRISM (and underlying Prolog) language. It is also possible to put the definition of `choose_noisy_or/4` into a separate library file loaded by the inclusion declaration (§2.6.5), and then the network-specific part (namely, the definitions of `world/2`, `world/6` and `noisy_or/3`) will be left more declarative. The PRISM language only provides a simple built-in probabilistic predicate implementing random switches, but as long as we deal with generative models, there seems to be ways to construct a more abstract formalism combining these random switches. The `p_not_table` declarations are added for making the inference results simple and readable.

The utility part should be modified accordingly. First, we add a couple of batch routines for setting parameters:

```
set_params:-
   set_sw(bn(fi,[]),[0.1,0.9]),
   set_sw(bn(ta,[]),[0.15,0.85]),
   set_sw(bn(sm,[yes]),[0.95,0.05]),
   set_sw(bn(sm,[no]),[0.05,0.95]),
   set_sw(bn(le,[yes]),[0.88,0.12]),
   set_sw(bn(le,[no]),[0.01,0.99]),
   set_sw(bn(re,[yes]),[0.75,0.25]),
   set_sw(bn(re,[no]),[0.10,0.90]).

set_nor_params:-
   ( noisy_or(X,Pa,DistList),
     set_nor_params(X,Pa,DistList),
     fail
   ; true
   ).

set_nor_params(_,[],[]).
set_nor_params(X,[Y|Pa],[Dist|DistList]):-
   set_sw(cause(X,Y),Dist),!,
   set_nor_params(X,Pa,DistList).

:- set_params.
:- set_nor_params.
```

In the above, `set_nor_params/0` sets the switch parameters according to the specifications of the noisy-OR nodes. To confirm whether the network-independent part of the model works well, let us introduce the following routines:

```
print_dist_al:-
   ( member(Fi,[yes,no]),
     member(Ta,[yes,no]),
     member(Al,[yes,no]),
     get_cpt_prob(al,[Fi,Ta],Al,P),
     format("P(al=~w | fi=~w, ta=~w):~t~6f~n",[Al,Fi,Ta,P]),
     fail
   ; true
   ).

print_expl_al:-
   ( member(Fi,[yes,no]),
     member(Ta,[yes,no]),
     member(Al,[yes,no]),
     get_cpt_probf(al,[Fi,Ta],Al),
     fail
   ; true
   ).

get_cpt_prob(X,PaVs,V,P):-
   ( prob(cpt(X,PaVs,V),P)
   ; P = 0.0
   ),!.

get_cpt_probf(X,PaVs,V):-
   ( probf(cpt(X,PaVs,V))
   ; format("cpt(~w,~w,~w): always false~n",[X,PaVs,V])
   ),!.
```

`print_dist_al/0` shows the distribution of the *Alarm* node for each instantiations of its parents by a failure-driven loop, and `print_expl_al/0` shows a logical expression of the probabilistic behavior of the *Alarm* node. `get_cpt_prob/4` and `get_cpt_probf/3` are just introduced for dealing with the cases that `prob/2` or `probf/1` fails. Finally, we can confirm that the generic version of the alarm network program with the noisy-OR rule works correctly:

```
?- print_dist_al.

P(al=yes | fi=yes, ta=yes):     0.940000
P(al=no | fi=yes, ta=yes):      0.060000
P(al=yes | fi=yes, ta=no):      0.700000
P(al=no | fi=yes, ta=no):       0.300000
P(al=yes | fi=no, ta=yes):      0.800000
P(al=no | fi=no, ta=yes):       0.200000
P(al=yes | fi=no, ta=no):       0.000000
P(al=no | fi=no, ta=no):        1.000000

?- print_expl_al.

cpt(al,[yes,yes],yes)
   <=> msw(cause(al,fi),yes) & msw(cause(al,ta),yes)
     v msw(cause(al,fi),yes) & msw(cause(al,ta),no)
```

```
     v msw(cause(al,fi),no) & msw(cause(al,ta),yes)
cpt(al,[yes,yes],no)
   <=> msw(cause(al,fi),no) & msw(cause(al,ta),no)
cpt(al,[yes,no],yes)
   <=> msw(cause(al,fi),yes)
cpt(al,[yes,no],no)
   <=> msw(cause(al,fi),no)
cpt(al,[no,yes],yes)
   <=> msw(cause(al,ta),yes)
cpt(al,[no,yes],no)
   <=> msw(cause(al,ta),no)
cpt(al,[no,no],yes): always false
cpt(al,[no,no],no)
```

7.4 Statistical analysis

PRISM is a suitable tool for analyzing statistical data. In this section, we present three examples. In the first example, we consider gene inheritance of human's blood type again, and show a typical way to answer the question of model selection. The second example attempts to find a probabilistic justification for a common practice seen in tennis games: players serve second services more conservatively than first services. We write a program to demonstrate that the percentage of points won would normally decline should a player serve second services as hard as first ones. The third example attempts to obtain statistics that can be used to tune the unification procedure.

7.4.1 Another hypothesis on blood type inheritance

The ABO gene model on the inheritance of ABO blood type, described in §1.2, was introduced in early 20th century [10]. Around that time, there was another hypothesis that we have two loci for ABO blood type with dominant alleles A/a and B/b. According to this hypothesis, genotypes aabb, A∗bb, aaB∗ and A∗B∗ correspond to the blood types (phenotypes) O, A, B and AB, respectively, where ∗ stands for a "don't care" symbol. In this section, let us call this hypothesis the AaBb gene model. The following is a PRISM program for the AaBb gene model:

```
%%%%  Declarations:

target(bloodtype,1).
data('bloodtype.dat').

values(locus1,['A',a]).
values(locus2,['B',b]).

%%%%  Modeling part:

bloodtype(P) :-
   genotype(locus1,X1,Y1),
   genotype(locus2,X2,Y2),
   ( X1=a, Y1=a, X2=b, Y2=b -> P=o
   ; ( X1='A' ; Y1='A' ), X2=b, Y2=b -> P=a
   ; X1=a, Y1=a, ( X2='B' ; Y2='B')  -> P=b
   ; P=ab
```

```
    ).

genotype(L,X,Y) :- msw(L,X),msw(L,Y).
```

In this program, we use two random switches each of which represents a random pick-up of a gene in the corresponding locus. The question here is which hypothesis from these two hypotheses on blood type inheritance (i.e. the ABO gene model and the AaBb gene model) is more plausible. To answer this question, we consider to use a Bayesian model score called BIC (Bayesian Information Criterion). One may notice that this is an example of a problem of model selection.

Suppose that `bloodABO.psm` and `bloodAaBb.psm` are the program files for the ABO gene model (given in §1.2) and for the AaBb gene model (given just above), respectively. We also assume that a data file named `bloodtype.dat` which contains 38 persons of blood type A, 22 persons of blood type B, 31 persons of blood type O and 9 persons of blood type AB. The ratio of frequencies of blood types in this data is almost the same as that in Japanese people. Lastly, for simplicity, we consider that either program has the following data file declaration:

```
data('bloodtype.dat').
```

Under these settings, we first load `bloodABO.psm`, and then call a built-in for EM learning. Finally we can get the BIC value as −132.667082:

```
?- prism(bloodABO).
  :
?- learn.
#goals: 0(4)
Exporting switch information to the EM routine ...
#em-iterations: 0(5) (Converged: -128.061911600)
Statistics on learning:
        Graph size: 27
        Number of switches: 1
        Number of switch instances: 3
        Number of iterations: 5
        Final log likelihood: -128.061911600
        Total learning time: 0.004 seconds
        Explanation search time: 0.000 seconds
        Total table space used: 5888 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
?- show_sw.
Switch gene: unfixed_p: a (p: 0.272288804) b (p: 0.169511387) o (p: 0.55
8199809)
  :
?- learn_statistics(bic,BIC).
BIC = -132.667081786147037 ?
```

On the other hand, we repeat the same procedure for `bloodAaBb.psm`, and get the BIC value as −135.649847:

```
?- prism(bloodAaBb).
  :
?- learn.
#goals: 0(4)
```

```
Exporting switch information to the EM routine ...
#em-iterations: 0(5) (Converged: -131.044676485)
Statistics on learning:
        Graph size: 48
        Number of switches: 2
        Number of switch instances: 4
        Number of iterations: 5
        Final log likelihood: -131.044676485
        Total learning time: 0.004 seconds
        Explanation search time: 0.000 seconds
        Total table space used: 7808 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
?- show_sw.
Switch locus1: unfixed_p: A (p: 0.272006612) a (p: 0.727993388)
Switch locus2: unfixed_p: B (p: 0.169341684) b (p: 0.830658316)
  :
?- learn_statistics(bic,BIC).
BIC = -135.649846671234258 ?
```

As a result, the ABO gene model has a larger BIC value, so we can conclude that the ABO gene model is more plausible than the AaBb gene model according to the data in `bloodtype.dat`.

7.4.2 Why not serving second services as hard in tennis?

In tennis games, we observe a common practice, namely, players normally serve second services much more conservatively than serving first services. Most people accept the practice without asking why. We write a program to model the statistical relationship between serving and winning in tennis games and use real statistics of Andy Roddick, one of top players, to answer the question.

In tennis, a player has at most two chances to serve in each point. If the first service is a fault, he has another chance to serve. If both services are faults, he loses the point. The following program models this process.

```
values(serve(_),[in,out]).    % switches serve(1) serve(2)
values(result(_),[win,loss]). % switches result(1) result(2)

target(play,1).

play(Res):-
   msw(serve(1),S1),
   (S1==in ->
      msw(result(1),Res);
      msw(serve(2),S2),
      (S2==in ->
         msw(result(2),Res);
         Res=loss)).
```

We use two switches, `serve(1)` and `serve(2)`, to represent the outcomes of services, and use another two switches, `result(1)` and `result(2)`, to represent the results: `result(1)` gives the result of the point when the first service is legal and `result(2)` the result of the point when the second service is legal. The result is loss if both services are faults.

The following sets the parameters of the switches based on Andy Roddick's statistics: his serving percentages are 61 and 95 at first and second services, respectively, and his percentages of points won at two services are 81 and 56, respectively.

```
roddick:-
   set_sw(serve(1),[0.61,0.39]),
   set_sw(serve(2),[0.95,0.05]),
   set_sw(result(1),[0.81,0.19]),
   set_sw(result(2),[0.56,0.44]).
```

From the program and the switch parameters, we know Andy Roddick's wining probability is 0.70158.

```
?- prob(play(win),Prob)
Prob = 0.70158
```

If Andy Roddick served second services like first services, the predicate `play` should be redefined as follows:

```
play(Res):-
   msw(serve(1),S1),
   (S1==in ->
      msw(result(1),Res);
      msw(serve(1),S2),
      (S2==in ->
         msw(result(1),Res);
         Res=loss)).
```

His winning probability would decline to 0.686799. This explains why serious tennis players serve second services much more conservatively than first services although the percentage of points won at first services is much higher than that at second services.

7.4.3 Tuning the unification procedure

Given two terms, the unification procedure determines if they are unifiable, and if so finds a substitution for the variables in the two terms to make them identical. A term is one of the following four types: *variable*, *atomic*, *list*, and *structure*. The unification procedure behaves as follows:

unify(t_1,t_2) {
  if (t_1 is variable) bind t_1 to t_2;
  else if (t_1 is atomic){
    if (t_2 is variable) bind t_2 to t_1;
    else return t_1==t_2;
  } else if (t_1 is a list){
    if (t_2 is variable) bind t_2 to t_1;
    else if (t_2 is a list)
      return unify(car(t_1),car(t_2)) && unify(cdr(t_1),cdr(t_2));
    else return false;
  } else if (t_1 is a structure){
    if (t_2 is variable) bind t_2 to t_1;
    else if (t_2 is a structure) {
      let t_1 be f(a_1,...,a_n) and t_2 be g(b_1,...,b_m);

```
            if (f != g || m != n) return false;
            return unify(a1,b1) && ... && unify(an,bn);
        } else return false;
    }
}
```

Since the order of tests affects the speed of the unification procedure, one question arises: how to tune the procedure such that it performs fewest tests on a set of sample data.

The following shows a PRISM program written for this purpose:

```
target(prob_unify/3).
values(s1,[var,atom,list,struct]).
values(s2(_),[var,atom,list,struct]). %switches: s2(var),s2(atom),...

data('unification.dat').

prob_unify(T1,T2,Res):-
   get_type(T1,Type1),
   msw(s1,Type1),
   get_type(T2,Type2),
   msw(s2(Type1),Type2),
   unify(T1,T2,Res).

unify(T1,T2,Res):-var(T1),!,T1=T2,Res=true.
unify(T1,T2,Res):-var(T2),!,T1=T2,Res=true.
unify(T1,T2,Res):-atomic(T1),!,(T1==T2->Res=true;Res=false).
unify([H1|T1],[H2|T2],Res):-!,
   prob_unify(H1,H2,Res1),
   (Res1=true->prob_unify(T1,T2,Res);Res=false).
unify(T1,T2,Res):-
   functor(T1,F1,N1),
   functor(T2,F2,N2),!,
   ((F1\=F2;N1\=N2)->Res=false;
   unify(T1,T2,1,N1,Res)).

unify(T1,T2,N0,N,Res):-N0>N,!,Res=true.
unify(T1,T2,N0,N,Res):-
   arg(N0,T1,A1),
   arg(N0,T2,A2),
   prob_unify(A1,A2,Res1),
   N1 is N0+1,
   (Res1=true->unify(T1,T2,N1,N,Res);Res=false).

get_type(T,var):-var(T),!.
get_type(T,atom):-atomic(T),!.
get_type(T,list):-nonvar(T),T=[_|_],!.
get_type(T,struct):-nonvar(T),functor(T,F,N),N>0.
```

In learning mode, this program basically counts the occurrences of each type encountered in execution. The switch `s1` gives the probability distribution of the types of the first argument, and for each type of the first argument `T` the switch `s2(T)` gives the probability distribution of the second argument.

For the following sample data stored in `'unification.dat'`

```
prob_unify(f(A,B,1,C),f(0,0,0,1),false).
```

```
prob_unify(A,def,true).
prob_unify(g(A,B),g(A,fin),true).
```

we can conduct learning and see the results of learning as follows:

```
?- learn.

#goals: 0(3)
Exporting switch information to the EM routine ...
#em-iterations: 0(2) (Converged: -9.704060528)
Statistics on learning:
        Graph size: 35
        Number of switches: 4
        Number of switch instances: 16
        Number of iterations: 2
        Final log likelihood: -9.704060528
        Total learning time: 0.000 seconds
        Explanation search time: 0.000 seconds
        Total table space used: 12688 bytes
Type show_sw or show_sw_b to show the probability distributions.

yes
?- show_sw.

Switch s1: unfixed_p: var (p: 0.625000000) atom (p: 0.125000000) list
(p: 0.000000000) struct (p: 0.250000000)
Switch s2(atom): unfixed_p: var (p: 0.000000000) atom (p: 1.000000000)
list (p: 0.000000000) struct (p: 0.000000000)
Switch s2(struct): unfixed_p: var (p: 0.000000000) atom (p: 0.00000000
0) list (p: 0.000000000) struct (p: 1.000000000)
Switch s2(var): unfixed_p: var (p: 0.200000000) atom (p: 0.800000000)
list (p: 0.000000000) struct (p: 0.000000000)
```

From this result, we know how to order the tests of types so that the unification procedure performs the best on the samples.

7.5 Dieting professor*

The last example is a program that deals with failures in the generation process. Let us consider a scenario as follows. There is a professor who takes a lunch everyday at one of two restaurants '`s0`' and '`s1`', and he changes the restaurant to visit probabilistically. Also as he is on a diet, he needs to satisfy a *constraint* that the total calories for lunch in a week are less than 4K calories. He probabilistically orders pizza (which is denoted by '`p`' and has 900 calories) or sandwich ('`s`'; 400 calories) at the restaurant '`s0`', and hamburger ('`h`'; 400 calories) or sandwich ('`s`'; 500 calories) at the restaurant '`s1`'. He records what he has eaten like `[p,s,s,p,h,s,h]` in a week and he preserves the record *if and only if* he succeeds in keeping the constraint. For example, we have a list of preserved records, and attempt to estimate the probability that he violates the constraint.

First of all, let us introduce a two-state hidden Markov model (HMM), shown in Figure 7.5, as a basic model that captures the professor's probabilistic behavior. We then try to write a PRISM program which represents this basic model with the additional constraint on the total calories. Hereafter we call the model a *constrained HMM*. Let us proceed to describe the program. From Figure 7.5, we can see that four switches are required as follows:

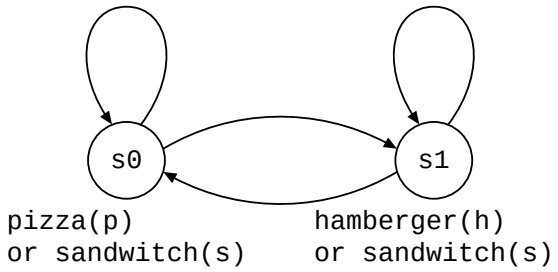

Figure 7.5: State transition diagram of the dieting professor.

```
values(tr(s0),[s0,s1]).
values(tr(s1),[s1,s0]).
values(lunch(s0),[p,s]).  % pizza:900,  sandwich:400
values(lunch(s1),[h,s]).  % hanburger:400, sandwich:500
```

where the switches named `tr(·)` choose the next restaurant, and those named `lunch(·)` select the menu of lunch at the chosen restaurant.

The central part of the model is `chmm/4`, which is defined as follows:

```
chmm(L,S,C,N):- N>0,
  msw(tr(S),S2),
  msw(lunch(S),D),
  ( S == s0,
     ( D = p, C2 is C+900
     ; D = s, C2 is C+400 )
  ; S == s1,
     ( D = h, C2 is C+400
     ; D = s, C2 is C+500 )
  ),
  L=[D|L2],
  N2 is N-1,
  chmm(L2,S2,C2,N2).
chmm([],_,C,0):- C < 4000.
```

This predicate behaves similarly to `hmm/3` (§7.1), a recursive routine, except that `chmm/4` has an additional argument that accumulates the total calories in a week. It is important to notice here that, when the recursion terminates, the total calories will be checked in the second clause, and if the total calories violate the constraint, the predicate `chmm/4` totally fails. This corresponds to the scenario that the professor only preserves the record if and only if he succeeds to keep the constraint.

To learn the parameters from his records, or to know the probability that he fails to keep the constraint, we need to make further settings. For example, we may define the four predicates as follows:

```
failure:- not(success).
success:- success(_).
success(L):- chmm(L,s0,0,7).
failure(L):- not(success(L)).
```

From the definition of `chmm/4`, `success(`L`)` says that the professor succeeds to keep the constraint with the menus L. So `success/0` indicates the fact that he succeeds to keep the constraint. `failure/0`

is the negation of success/0 and therefore means that he fails to satisfy the constraint. failure(*L*) is optional here but says that he fails to keep the constraint due to the menus *L*. Then we made the rest of declarations:

```
target(success,1).
target(failure,0).
data(user).
```

We consider the predicates success/1 and failure/0 as observable predicates, and we use learn/1 as a learning command.

The experiment we attempt is artificial, similarly to those with HMMs (§7.1) and discrete Bayesian networks (§7.3) — we first generate samples under the predefined parameters, and then learn the parameters from the generated samples. For this experiment, we define a predicate in the utility part, that specifies some predefined parameters:

```
set_params:-
  set_sw(tr(s0),[0.7,0.3]),
  set_sw(tr(s1),[0.7,0.3]),
  set_sw(lunch(s0),[0.4,0.6]),
  set_sw(lunch(s1),[0.5,0.5]).
```

Now we are in a position to start the experiment. We first load the program with the built-in prismn/1 (please note 'n' at the last of the predicate name):

```
?- prismn(chmm).

step1.
step2.
step3.
Compilation done by FOC

compiled in 12 milliseconds
loading::temp.out

yes
```

Let us recall that the definition clauses of failure/0 and failure/1 have negation not/1 in their bodies. This is not negation as failure (NAF), and we need a special treatment for such negation. prismn/1 calls an implementation of First Order Compiler (FOC) [29] to eliminate negation not/1. In the messages above, the messages from "step1" to "Compilation done by FOC" are produced by the FOC routine, and we may notice that the predicates whose names start with 'closure_' are newly created by the FOC routine and registered as table predicates (because they are probabilistic).

After loading, we set the parameters by set_params/0, and confirm the specified parameters:

```
?- set_params,show_sw.

Switch lunch(s0): unfixed_p: p (p: 0.400000000) s (p: 0.600000000)
Switch lunch(s1): unfixed_p: h (p: 0.500000000) s (p: 0.500000000)
Switch tr(s0): unfixed_p: s0 (p: 0.700000000) s1 (p: 0.300000000)
Switch tr(s1): unfixed_p: s1 (p: 0.700000000) s0 (p: 0.300000000)
```

We can compute the probability that the professor fails to keep the constraint under the parameters above:

```
?- prob(failure).
Probability of failure is: 0.348592596784000
```

From this, we can say that the professor skips preserving the record once in three weeks.

To make it sure that the program correctly represents our model (in particular, the definition of the `failure` predicate), we may give a couple of queries. For example, the following query confirms whether the sum of the probability that the professor satisfy the constraint and the probability that he does not becomes unity:[8]

```
?- prob(success,Ps),prob(failure,Pf),X is Ps+Pf.

Pf = 0.348592596784
Ps = 0.651407403215999
X = 0.999999999999998 ?
```

Or we have a similar query which is limited to some specific menu (obtained as `L` by sampling):

```
?- sample(success(L)),
   prob(success(L),Ps),prob(failure(L),Pf),
   X is Ps+Pf.

Pf = 0.9999321868
Ps = 0.0000678132
L = [s,p,h,s,h,p,h]
X = 1.0 ?
```

It is confirmed for each goal appearing in the queries above that the sum of probabilities of the goal and its negation is always unity, so we can proceed to a learning experiment. To conduct it, we use the built-in `get_samples_c/4` to generate 500 samples (note that we cannot simply use `get_samples/3` since a sampling of `success(L)` may fail), and invoke the learning command with the samples:

```
?- get_samples_c([inf,500],success(L),true,Gs),learn([failure|Gs]).

sampling -- #success = 500
sampling -- #failure = 249
#goals: 0.........100.........200......(266)
Exporting switch information to the EM routine ...
#em-iterations: 0........(83) (Converged: -2964.788301553)
Statistics on learning:
        Graph size: 9328
        Number of switches: 4
        Number of switch instances: 8
        Number of iterations: 83
        Final log likelihood: -2964.788301553
        Total learning time: 0.036 seconds
        Explanation search time: 0.016 seconds
        Total table space used: 1486208 bytes
Type show_sw or show_sw_b to show the probability distributions.
Gs = [success([s,s,s,h,s,h,h]),success([s,p,h,s,h,h,s]),
      ... omitted ...
      success([s,p,h,h,s,p,s]),success([p,s,s,s,h,s,s])] ?
yes
```

[8] Unfortunately, as shown here, the actual result of the sum will not always be unity for precision errors.

It should be noted that, if a special symbol `failure` is included to the goals in `learn/1`, the EM algorithm considering failure called the failure-adjusted maximization (FAM) algorithm will be invoked. After learning, we can confirm the learned parameters as usual:

```
?- show_sw.

Switch lunch(s0): unfixed_p: p (p: 0.380041828) s (p: 0.619958172)
Switch lunch(s1): unfixed_p: h (p: 0.537922906) s (p: 0.462077094)
Switch tr(s0): unfixed_p: s0 (p: 0.714988121) s1 (p: 0.285011879)
Switch tr(s1): unfixed_p: s1 (p: 0.677016948) s0 (p: 0.322983052)
```

Bibliography

[1] N. Angelopoulos. Extending the CLP engine for reasoning under uncertainty. In *14th International Symposium on Methodologies for Intelligent Systems (ISMIS-2003)*, pages 365–373, 2003.

[2] M. J. Beal. *Variational Algorithms for Approximate Bayesian Inference*. PhD thesis, University College London, 2003.

[3] M. J. Beal and Z. Ghahramani. Variational Bayesian learning of directed graphical models with hidden variables. *Bayesian Analysis*, 1(4):793–832, 2006.

[4] E. Charniak. *Statistical Language Learning*. The MIT Press, 1993.

[5] P. Cheeseman and J. Stutz. Bayesian classification (AutoClass): Theory and results. In U. Fayyad, G. Piatesky, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 153–180. The MIT Press, 1995.

[6] D. Chickering and D. Heckerman. Efficient approximation for the marginal likelihood of Bayesian networks with hidden variables. *Machine Learning*, 29:181–212, 1997.

[7] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[8] M. Collins. Discriminative reranking for natural language parsing. In *Proceedings of the 17th International Conference on Machine Learning (ICML-2000)*, pages 175–182, 2000.

[9] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[10] J. F. Crow. *Genetic Notes*. Burgess Publishing Company, 1983. Translated into Japanese.

[11] J. F. Crow. *Basic Concepts in Population Quantitative and Evolutionary Genetics*. W. H. Freeman and Company, 1986. Translated into Japanese.

[12] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44:245–271, 2001.

[13] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, B39:1–38, 1977.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 2nd edition, 1999.

[15] Y. Izumi, Y. Kameya, and T. Sato. Parallel EM learning for symbolic-statistical models. In *Proceedings of the International Workshop on Data-Mining and Statistical Science (DMSS-2006)*, pages 133–140, 2006.

[16] M. Jaeger. Ignorability for categorical data. *The Annals of Statistics*, 33(4):1964–1981, 2005.

[17] M. Jaeger. On testing the missing at random assumption. In *Proceedings of the 17th European Conference on Machine Learning (ECML-2006)*, pages 671–678, 2006.

[18] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. In *Proceedings of the 1st International Conference on Computational Logic (CL-2000)*, pages 269–294, 2000.

[19] Y. Kameya, T. Sato, and N.-F. Zhou. Yet more efficient EM learning for parameterized logic programs by inter-goal sharing. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, pages 490–494, 2004.

[20] K. Katahira, K. Watanabe, and M. Okada. Deterministic annealing in variational Bayesian algorithm. *Neurocomputing, IEICE Technical Report (NC2006-183)*, 106(589):177–182, 2007. In Japanese.

[21] K. Kurihara and T. Sato. An application of the variational Bayesian approach to probabilistic context-free grammars. In *Proceedings of the IJCNLP-04 Workshop: Beyond shallow analyses*, 2004.

[22] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of Royal Statistical Society*, B50(2):157–194, 1988.

[23] D. J. C. MacKay. Ensemble learning for hidden Markov models. Technical report, Cavendish Laboratory, University of Cambridge, 1997.

[24] G. J. McLachlan and T. Krishnan. *The EM algorithm and extensions*. Wiley Interscience, 1997.

[25] D. Poole. Probabilistic Horn abduction. *Artificial Intelligence*, 64(1):81–129, 1993.

[26] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of IEEE*, volume 77, pages 257–286, 1989.

[27] D. B. Rubin. Inference and missing data. *Biometrika*, 63:581–592, 1976.

[28] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[29] T. Sato. First Order Compiler: a deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8:605–627, 1989.

[30] T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP-95)*, pages 715–729, 1995.

[31] T. Sato. Modeling scientific theories as PRISM programs. In *Proceedings of ECAI-98 Workshop on Machine Discovery*, pages 37–45, 1998.

[32] T. Sato. Inside-outside probability computation for belief propagation. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007.

[33] T. Sato and Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1330–1335, 1997.

[34] T. Sato and Y. Kameya. A Viterbi-like algorithm and EM learning for statistical abduction. In *Proceedings of UAI-2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support*, 2000.

[35] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[36] T. Sato and Y. Kameya. A dynamic programming approach to parameter learning of generative models with failure. In *Proceedings of ICML Workshop on Statistical Relational Learning and its Connection to the Other Fields (SRL-04)*, 2004.

[37] T. Sato and Y. Kameya. Negation elimination for finite PCFGs. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation 2004 (LOPSTR-04)*, 2004.

[38] T. Sato, Y. Kameya, and N.-F. Zhou. Generative modeling with failure in PRISM. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 847–852, 2005.

[39] G. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, 6(2):461–464, 1978.

[40] L. Sterling and E. Sharpiro. *The Art of Prolog*. The MIT Press, 1986.

[41] N. Ueda and R. Nakano. Deterministic annealing EM algorithm. *Neural Networks*, 11(2):271–282, 1998.

[42] N.-F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. In *Proceedings of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP-03)*, pages 275–283, 2003.

[43] N.-F. Zhou, T. Sato, and K. Hashida. Toward a high-performance system for symbolic and statistical modeling. In *Proceedings of IJCAI-03 workshop on Learning Statistical Models from Relational Data (SRL-03)*, pages 153–159, 2003. The extend version is available as: Technical Report (Computer Science) TR-200212, City University of New York (2002).

Concept Index

Programming Index

Example Index