PRISM User's Manual

(Version 1.10)

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

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) [2, 14], probabilistic context free grammars (PCFGs) [2] 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 [29]. 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 [23]. For an implementational view, please refer to [30].

This document describes the PRISM language, its programming system, and several program examples, targeting version 1.10. It is divided into three parts: the first part (Chapters 1 and 2) describes the language, the second one (Chapters 3 and 4) lists all functionality of the system, and the rest (Chapter 5) gives several sample program examples of PRISM.

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]` should be replaced with `@`.

Acknowledgements

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

The project was started as an ICOT Free Software Project, and then has been supported in part by “Discovery Science” Project, JST Basic Research Programs CREST “Advanced Media Technology for Everyday Living,” and the 21st Century COE Program “Framework for Systematization and Application of Large-scale Knowledge Resources” at Tokyo Institute of Technology.

Contents

Chapter 1

Overview of PRISM

PRISM is a probabilistic extension of Prolog. Syntactically, PRISM is just a 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 5 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 OR (`;`), the cut symbol (`!`) and if-then (`->`) statements as far as they work as expected according to the execution mechanism of the programming system. 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. If the installation is successful, we can invoke the programming system just running the command '`prism`':

```
% prism
 :
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.

Now 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).
```

Some may notice here that the file suffix '`.psm`' can be omitted. 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 runs with `sample/1` will show a different probabilistic behavior from that was made before:

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

Finally, we can quit the programming system as follows:

```
?- halt.
```

1.2 Basic probabilistic inference and 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 the father and mother.[1] For example,

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

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 [6]: $P_A = p_a^2 + 2p_ap_o$, $P_B = p_b^2 + 2p_bp_o$, $P_O = p_o^2$, and $P_{AB} = 2p_ap_b$. To represent a distribution of phenotypes instead of these mathematical formulas, we may write the following PRISM program:

```
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, b, o` denotes a random pick-up of gene `X` from the pool, and becomes true with probability p_a, p_b and p_o, respectively. Then, with a careful reading of this program, we can say that one of `bloodtype(P)` with `P = a, b, o, 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.[2]

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 can give appropriate parameters, a variety of probabilistic inferences are available. For example, as described in the previous section, 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*:

```
?- 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

[2] 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.

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 deterministically 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 [8], and the programming system 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:[3]

```
?- 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 [19]).

1.3 Utility programs and advanced probabilistic inferences

The last example in this chapter is a PRISM version of a hidden Markov model (HMM) [2, 14]. 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:

[3] 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.

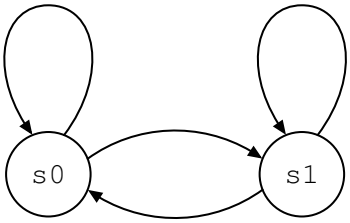

Figure 1.1: State transition diagram of a 2-state hidden Markov model.

```
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
```

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 routine 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`,[4] `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 [14], which computes the most likely sequence of (hidden) state 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 likely 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 likely sequence is: $\mathtt{s0} \to \mathtt{s1} \to \cdots \to \mathtt{s1} \to \mathtt{s0}$ (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 [22]. So we can handle moderately large datasets with PRISM. The efficiency comes from *linear tabling* [29], 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 [11, 22, 23].

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

[4] `get_samples/3` is provided since version 1.9. `get_samples(`*N*`,`*G*`,`*Goals*`)` generates *N* samples as *Goals* by invoking `sample(`*G*`)` *N* times.

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* [23], 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 [24, 25, 26]. This algorithm is based both on Cussen's FAM (failure-adjusted maximization) algorithm [7] and FOC (First Order Compiler) [17]. With this new learning framework, we are able to introduce some *constraints* (which causes some failures) to generative models.

1.5 Organization of this manual

It is hard to list all functionalities with full details in this chapter, and so please refer to the following chapters for the detailed description of the functionality you wish to use. The rest of this manual is organized as follows:

- Chapter 2 describes the detail of the PRISM language.
- Chapter 3 explains how to use the PRISM programming system.
- Chapter 4 gives the detailed descriptions of the built-in predicates provided by the programming system.
- Chapter 5 shows several program examples with detailed illustrations.

To learn PRISM, it would be helpful to see typical usages of PRISM illustrated in this chapter and Chapter 5 first, and then to run the example programs in the released package. The authors consider that the sections whose titles are marked with * have a little advanced contents, so the busy users can skip these sections for the first time. Chapter 2 may also be skipped until the examples have been explored, but to understand the program's behavior precisely, the descriptions in this chapter (especially §2.2, §2.3 and §2.4) are essential though they look complicated. Chapter 3 and 4 are expected to work as a (rough) reference manual of the programming system.

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 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:

- *The 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.
- *The utility part*: the remaining definitions of non-probabilistic predicates. This part is a usual Prolog program that utilizes the model, and that can often 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 [18, 23], 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 1.10, to make an efficient implementation of tabling, we use a different specification from the original one [21, 23] 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,`v_1`)`, ..., `msw(i,`v_k`)` 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 position 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`, `b`, `o` (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. $\theta_{\texttt{gene,a}} = p_a$, and so on. 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`, `o` and `Y` = `a`, `b`, `o` becomes exclusive, and hence the different instances of the 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`, `b`, `o`, `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

[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 *distribution semantics*), please consult [18, 23].

`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 make a programming in a top-down (consequently, a generative) manner. On the other hand, sometimes it is also crucial to inspect the probabilistic meaning in a bottom-up manner as above.

2.3 Probabilistic inferences

Before stepping into the further detail of the PRISM language, it would be worth listing what we can do with this language. In the current version of the PRISM programming system, the following five types of probabilistic inferences are available, where the first one works with *sampling execution*, and the rest utilize *explanation search*:

Sampling (§4.2):
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 (§4.3):
Given a goal G of a probabilistic predicate, compute $P_\theta(G)$.

Viterbi computation (§4.5):
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 switches.

Hindsight computation (§4.6):
Given a goal G of a probabilistic predicate, compute $P_\theta(G')$ or $P_\theta(G'|G)$ for each subgoal G' of G.

Parameter learning (§4.7):
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 $\prod_t P_\theta(G_t)$.

where $P_\theta(\cdot)$ stands for the probability distribution denoted by the program, under the parameters θ of switches buried in the program. For more details, please visit the corresponding sections. These sections will also provide the variations for each inference. The sections §2.4.1 and §2.4.2 respectively describe sampling execution and explanation search, the underlying execution mechanisms for the probabilistic inferences listed above.

2.4 Modeling part

We have seen a couple of examples of the modeling part (sections in Chapter 1 and §2.2). One of the interesting features 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.

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,

[3] For the theoretical details, please see [23].

and hence it is required to write carefully programs to satisfy the conditions (otherwise some unexpected behavior arises).

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

2.4.1 Sampling execution

Sampling execution is the underlying execution style for a sampling task (§2.3, §4.2). In the literature of Bayesian networks, this style is sometimes called *forward sampling*. In the recent versions including 1.10, 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-value 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)).
```

[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 they 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 program.

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.

2.4.2 Explanation search

Explanation search works as an underlying subroutine of built-in predicates for probabilistic inference such as probability calculation (§4.3), Viterbi computation (§4.5), hindsight computation (§4.6) and parameter learning (§4.7).[7] To simulate only explanation search, we can use the built-ins `probf/1-2` (§4.4). 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 genes A 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 consider that an explanation search finds all possible explanations for a given goal by a *failure-driven loop* [28]. 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 that $G \Leftrightarrow E_1 \vee E_2 \vee \ldots \vee E_k$.

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 exclusive condition, described in §2.4.5), 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* [29], 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

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

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.[8]

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[9] for G = `hmm([a,b,b])`:

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

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 seen as a modified form[10] of an instance of Clark's completion [5] 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

[8] From a viewpoint of knowledge representation, explanation graphs can be seen as AND-OR graphs consisting of ground (i.e. propositional) 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.

[9] 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.

[10] The instances of non-probabilistic predicate, which are entailed from the program, are omitted.

is hereafter called the *acyclic 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` (§4.4) 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:[11]

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

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. Also 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. Declarations except the multi-valued switch declarations are made with respect to the modeling part for explanation search.

2.4.3 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 *non-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.[12] Instead of this, we exclude the probability mass which is lost by failures, and try to maximize $\prod_t P_\theta(G_t \mid \mathit{succ})$, the product of conditional probabilities of observed data being generated under the case that no failure arises (indicated by *succ*).

To be more specific, let us consider a program which considers the agreement in coin flipping.[13] The modeling part is written as follows:

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

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

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

[13] This program comes from [26].

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

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

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 non-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. With 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* [7] 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).[14] We cannot directly simulate this negation, and hence it is eliminated by *First Order Compiler* [17] when the program is loaded.[15] 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 §4.9, and a running example can be found in §5.4.

2.4.4 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 [15]. 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. $\mathcal{G} \subseteq \mathcal{G}^+$).

[14] 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.

[15] 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}

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^+. Then we further assume the MAR condition that comprises the following two sub-conditions:[16]

- 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.[17]

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[18] 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

[16] 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^+ [9].

[17] 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.

[18] It should be noted that the original definition of the MAR condition [15] 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 [8]. 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 ([9], for example).

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 test. For example, recent work by Jaeger tackles with this problem [10].

2.4.5 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.
- *Finiteness condition*: for any observable goal 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.
  - *Non-failure condition*: for any observable goal G, the generation process for G (i.e. a sampling execution of G) never fails.
- *Acyclic 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 5 that popular generative models including hidden Markov model or Bayesian networks are written in this fashion. If the program violates the non-failure condition, one possible solution is to utilize the system's facility described in §2.4.3.

Theoretically speaking, it is sometimes misunderstood and hence is desired to note that the distribution semantics [18, 23] itself assumes none of the conditions above. We can say PRISM's semantics is just a restricted version of the distribution semantics, that 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*`).`

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 sample data must be given as an argument of `learn/1` (See §4.7). The format of the data file is described in §4.7.3.

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 in a switch declaration *I* 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]).
```

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 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 unit clause which can work in the other part of the program (i.e. both the modeling part and the utility part). For example, we can run '`?- values(coin,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`; §4.1.2), 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`' (§4.12), 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:

[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 assert/retract predicates).

```
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/2-3` will be translated to `values/2` with the corresponding expanded values.[21] To be more 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).
```

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.[22] For the detail of handling switches, please visit §4.1. 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.

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.
```

[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/2-3` directly.
[22] For the declarations of switches with non-ground names, the parameters can neither be set nor fixed.

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.

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.

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 `prism110_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

In version 1.10, a single united package `prism110_linux.tar.gz` is provided for x86-based Linux systems. We have build and tested the package on glibc-2.3 systems. Typical steps for installation are as follows:

1. Download the package `prism110_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, then you should change the path `$HOME/prism/bin` accordingly.

Internally, the package contains three sorts of binaries: for 32-bit systems with 2.4.x kernels, for 32-bit with 2.6.x kernels, and for 64-bit with 2.4.x/2.6.x kernels. The start-up commands (`prism` and `upprism`) automatically choose the binary suitable for your environment.

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 simple 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 binary code program '`coin.psm.out`' and loads the program 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 have access to 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.[1]
- `v`. Monitor the learning process.
- `nv`. Do not monitor the learning process (default).

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

In addition, we can specify the values of execution flags (§4.12) 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 major built-ins in the programming system (Figure 3.1). The details of these built-ins are described in Chapter 4.

As mentioned above, 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*`)` (§4.2). In sample 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 numeric computations in learning (with `learn/0` or `learn/1`; §4.7), probability calculation (with `prob/2` and so on; §4.3), Viterbi computation (with `viterbif/3` and so on; §4.5), and hindsight computation (with `hindsight/3` and so on; §4.6). `probf/2` or its variation (§4.4) 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; §4.1) or the flags for various settings of the system (`set_prism_flags/2` and `get_prism_flags/2`; §4.12).

3.6 Debugging PRISM programs

Programs can be executed in the debugging mode. The command

```
trace
```

```
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,Dist)       -- Set the probability distribution 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`.

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

```
notrace
```

switches the execution mode back to the usual mode. In debugging 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 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, 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, in the following path

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

the subgoal `direction(left)` is explained by the outcome '`head`' of the switch '`coin`'.

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:[2]

[2] 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`".

- `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 §4.9). This is a replacement for the command `upprismn`, which is 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.

3.8 Error handling

In the current implementation, when the system met 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).

Chapter 4

PRISM Built-in Utilities

4.1 Random switches

4.1.1 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 previously mentioned, switches have different behaviors for sampling execution (§2.4.1) and explanation search (§2.4.2). In the former case, the probability distribution must also have been set by using `set_sw/2` (§4.1.2) or by parameter learning (§4.7).

4.1.2 Setting parameters of switches

The built-in `set_sw(`I`,`*Probs*`)` sets the parameters of outcomes of a switch I to *Probs* where *Probs* is a list of numbers `[`p_1`,`p_2`,...,`p_K`]` (recommended) or a term of the form p_1+p_2+$\cdots$+p_K that sums 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,

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

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 §4.12.

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

We need to add an explanation 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 §4.12 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 from the beginning. 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 are used so far, or specified in `values/2` with ground names.

- `set_sw_all(`*Patt*`)` sets a default distribution to all switches matching with *Patt*
- `set_sw_all(`*Patt*`,`*D*`)` sets a distribution *D* to all switches matching with *Patt*
- `set_sw_all` (with no args) is the same as `set_sw_all(_)`.

4.1.3 Fixing parameters 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.

The built-in predicate `fix_sw(`*I*`)` fixes all switches whose names unify with *I*. The parameters of fixed switches cannot be updated and will be kept unchanged during learning. Also `fix_sw(`*I*`,`*Params*`)` sets 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. On the other hand, `unfix_sw(`*I*`)` is used to make the parameters of all switches whose names unify with *I* changeable.

4.1.4 Displaying switch information

The built-in `show_sw/0` displays information associated with all switches used so far.[2] For example, in the direction program,

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

The built-in `show_sw(`*I*`)` displays information about switches whose names match with *I*. For example:

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

[2]This does not mean that all potential switches will be shown — in many programs, the system does not know all ground instances of `msw/2` in advance.

4.1.5 Getting switch information

The built-in `get_sw(`*I*`,`*Info*`)` binds *Info* to a term in the form `[`*Status*`,`*Values*`,`*Params*`]` that contains information about a switch *I*, where *Status* is either `fixed` or `unfixed`, *Values* is a list of possible outcomes of the switch, and *Params* is a list of the parameters of the outcomes. For example:

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

The built-in predicate `get_sw(`*SwInfo*`)` binds *SwInfo* to a term in the form of `switch(`*Id*`,` *Status*`,`*Values*`,`*Params*`)` where *Id* is the identifier, *Status* is either `fixed` or `unfixed`, *Values* is a list of possible outcomes, and *Params* is a list of the parameters. On backtracking, *SwInfo* is bound to the next switch. `get_sw(`*I*`,`*Status*`,` *Values*`,`*Params*`)` is the same as `get_sw(`*I*`,[`*Status*`,`*Values*`,`*Params*`]).`

Since version 1.10, `get_sw(`*Id*`,`*Status*`,`*Values*`,`*Params*`,`*Cs*`)` is available after learning. This built-in additionally returns the expected counts *Cs* of a switch named *Id*, which are computed in learning (§4.7). These expected counts are used in computing Cheeseman-Stutz score (§4.8), and might be used to judge whether we need to apply so-called *backoff smoothing*.[3]

The following is a note for the users who also used version 1.9: In version 1.9, if some default distribution (e.g. `uniform`) is specified, a new switch's distribution will be dynamically assigned as a side-effect when calling `get_sw/2`. For example, we have:

```
?- get_sw(foobar,Sw).
Sw = [unfixed,[a,b],[0.5,0.5]] ?
```

where `foobar` is a new switch. Since version 1.10, on the other hand, `get_sw/2` is changed so that it only tries to get switch information, with no side-effect. So for a new switch `foobar`, we will see:

```
?- get_sw(foobar,Sw).
no
```

Please note that a similar change is done for `show_sw/1`.

4.1.6 Saving switch information

The built-in `save_sw(`*File*`)` saves all switch information into the file *File*. *File* can be omitted (i.e. `save_sw/0` is available besides `save_sw/1`), in which case the information will be stored into a file named '`Saved_SW`'. On the other hand, the saved information can be restored by `restore_sw(`*File*`)` from the file *File*. If *File* is omitted, the programming system will try to restore the information from the file named '`Saved_SW`'.

4.2 Sampling

An execution with `sample(`*Goal*`)` (or a direct execution of *Goal*) simulates a sampling execution. A more detailed 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 ?
```

[3] If the observed data is complete (§4.7.1), *Cs* is just a list of numbers of occurrences of `msw(`*Id*`,·)` in the data.

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 predicate `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 of `[`*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 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 `pcfg_s/1` for sampling, and use `pcfg/1` for explanation search. To get training data for `pcfg/1` by sampling `pcfg_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,[pcfg_s(Ws),pcfg(Ws)],true,Gs).
```

4.3 Probability calculation

The built-in `prob(`*Goal*`,`*Prob*`)` calculates the probability *Prob* with which *Goal* becomes true. Under the independence and exclusiveness conditions (see §2.4.5), 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.

4.4 Explanation graphs

The built-in `probf(`*Goal*`,`*EGraph*`)` returns an 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. 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*.

4.5 Viterbi computation

By the *Viterbi computation*, we mean to get the most likely 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 of G.
- `viterbif(`G`,`P`,`*Expl*`)` returns the Viterbi probability of G to P, and a Prolog-term representation of the Viterbi explanation E^* of G to *Expl*.
- `viterbig(`G`)` is the same as `viterbi(`G`)` except that G is unified with its instantiation found in the most likely 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 likely 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 likely 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 following 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. According to the purpose, we may extract the list of subgoals $G'_1,\ldots,G'_n$, or the list of switches (by concatenating $SL_1,\ldots,SL_n$) from this Viterbi explanation. Also this Prolog term can be printed in a human-readable form by using `print_graph/1-2` (see §4.4).

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 §4.12 for details), and so the problem of underflow will be cleared.

4.6 Hindsight computation*

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]] ?
```

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).

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

[4] The name of 'hindsight' comes from an inference task with temporal models such as dynamic Bayesian networks [16].

`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:

```
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 itself
   T1 is T+1,
   hmm(T1,N,Next1,Next2,Y).

str_length(4).
```

Each state variable takes on 3 states (`s0`, `s1` and `s2`), and 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,b,a])).

hindsight probabilities:
  hmm(1,4,s0,s0,[a,b,b,a]): 0.003117125538065
  hmm(1,4,s0,s1,[a,b,b,a]): 0.000119071852861
  hmm(1,4,s0,s2,[a,b,b,a]): 0.002529688812405
      :
  hmm(5,4,s2,s0,[]): 0.000594140868831
  hmm(5,4,s2,s1,[]): 0.002737517626889
  hmm(5,4,s2,s2,[]): 0.001108525263899
```

Now let us suppose that we want to marginalize the second state variable (i.e. the 4th argument). It is achieved by running `hindsight_agg/2` as follows:

```
?- hindsight_agg(hmm([a,b,b,a]),hmm(integer,_,query,_,_)).

hindsight probabilities:
  hmm(1,*,s0,*,*): 0.005765886203332
  hmm(1,*,s1,*,*): 0.063400618136553
  hmm(1,*,s2,*,*): 0.011350757707280
  hmm(2,*,s0,*,*): 0.059382025259221
  hmm(2,*,s1,*,*): 0.004143958471003
      :
```

```
  hmm(5,*,s0,*,*): 0.033166672736384
  hmm(5,*,s1,*,*): 0.042910405551161
  hmm(5,*,s2,*,*): 0.004440183759620
```

In the above, `hmm(integer,_,query,_,_)` is a control statement that means "group subgoals according to the 1st (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 3rd 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 5th argument, instead of the 1st 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): 0.022812689075136
  hmm(*,*,s1,*,L-0): 0.020949331948366
  hmm(*,*,s2,*,L-0): 0.014598811876160
      :
  hmm(*,*,s1,*,L-4): 0.028716685449848
  hmm(*,*,s2,*,L-4): 0.012200549420048
```

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,b,a]),hmm(3,_,query,_,_)).

hindsight probabilities:
  hmm(3,*,s0,*,*): 0.006164189835510
  hmm(3,*,s1,*,*): 0.071139567166696
  hmm(3,*,s2,*,*): 0.003213505044959
```

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 (§4.12), 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

[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.

G' is its subgoal.[6] The usage for them is respectively the same as that of the corresponding `hindsight` or `hindsight_agg` predicate 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 (§5.2), we compute conditional probabilities on the network by using conditional hindsight probabilities.

4.7 Learning

4.7.1 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 guessed counts of `msw(`i`,`v`)` under the parameters $\theta^{(m)}$.

Maximization step:
: Using the guessed counts, 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

[6] Generally speaking, we need to say 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.

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

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 §4.12; the default is 10^{-4}).

4.7.2 Maximum a posteriori estimation

The programming system also supports *maximum a posteriori estimation* (MAP estimation) for parameter learning, which tries to find parameters θ that maximize, $P(\theta|G_1,\ldots,G_T) \propto P(\theta)\prod_t P_\theta(G_t)$, 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, and then in estimating parameters, it introduces δ, a single *pseudo count*. That is, in the complete-data case, each parameter is estimated by $\hat{\theta}_{i,v} = (C_{i,v}+\delta)/(\sum_{v'} C_{i,v'} + |V_i|\delta)$, where $|V_i|$ is the number of switch i's possible outcomes. Similarly in the incomplete-data case, each parameter is updated by the EM algorithm with $\hat{\theta}_{i,v} = (\hat{C}_{i,v}+\delta)/(\sum_{v'} \hat{C}_{i,v'} + |V_i|\delta)$, until 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 the pseudo count is zero, the MAP estimation is just an ML estimation, and it is sometimes called *Laplace smoothing* when the pseudo count set to be unity. We can set/get this pseudo count via the '`smooth`' flag (§4.12).

4.7.3 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)
#graphs: 0(2)
#iterations: 0(Converged: -1.909542505)
Finished learning
        Number of tabled subgoals: 2
        Number of switches: 1
        Number of switch values: 2
        Number of iterations: 2
        Final log likelihood: -1.909543
        Total learning time: 0.010 seconds
        All solution search time: 0.010 seconds
        Total table space used: 604 bytes
Type show_sw to show the probability distributions.
```

The line beginning with `#goals` (resp. `#graphs`) shows the number of *distinct* goals whose explanation searches have been done (resp. whose explanation graphs have been constructed).

[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.

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 §4.7.5). We may confirm the estimated parameters by `show_sw/0` (§4.1.4):

```
?- 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. 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)]).
```

It should be noticed that the default learning method is ML estimation (§4.7.1). On the other hand, as mentioned above, we can enable MAP estimation (§4.7.2) by setting the pseudo count δ, which is greater than zero, via the '`smooth`' flag (§4.12). For example, let us set the pseudo count as 0.5:

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

The learning command is invoked in the same way as that of ML estimation:

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

#goals: 0(2)
#graphs: 0(2)
#iterations: 0(Converged: -2.646252953)
Finished learning
        Number of tabled subgoals: 2
        Number of switches: 1
        Number of switch values: 2
        Number of iterations: 2
        Final log of a posteriori prob: -2.646253
        Total learning time: 0.010 seconds
        All solution search time: 0.010 seconds
        Total table space used: 604 bytes
Type show_sw 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[9]), 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: head (0.625) tail (0.375)
```

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.

4.7.4 Avoiding bad 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 bad local maxima. In the current version, the system provides a quite simple solution. That is, we can try multiple running of the EM algorithm by restarting with different initializations of 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 §4.12).

4.7.5 Getting statistics on learning

After learning (both ML and MAP), the built-in `get_log_likelihood(`*LL*`)` returns the log-likelihood of the given observed goals. In MAP estimation (§4.7.2), i.e. when some positive value is given to the '`smooth`' flag, we can also get the log of unnormalized a posteriori probability (§4.7.3) of the observed goals by `get_log_post(`*LPost*`)`, which is the target of maximization in MAP estimation. `get_lambda(`*L*`)` returns the log-likelihood after ML estimation,

[9] 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 $P(\theta|D) = P(\theta)P(D|\theta)/P(D)$, where $P(\theta)$ is a prior probability of θ, and $P(D|\theta)$ is the likelihood of D under θ. As stated in §4.7.2, $P(\theta)$ is assumed to follow a Dirichlet distribution, and the 'unnormalized' a posteriori probability is just $P(\theta|D)$ 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 running of the EM algorithm (§4.7.4).

or returns the log of unnormalized a posteriori probability after MAP estimation. Combining these statistics with the facilities for saving/restoring switch information (§4.1.6), we can write the extensions of the routine for multiple running of the EM algorithm (§4.7.4).

We can also get the number of occurred switches, occurred switch instances, and free parameters with respect to the last learning[10] via the built-in predicates `get_num_switches/1`, `get_num_switch_values/1`, and `get_num_parameters/1`. 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]] ?
```

`get_search_time(`*Time*`)` and `get_learn_time(`*Time*`)` can be used to get the time (in seconds) consumed for explanation search and for the entire learning procedure, respectively.

Since version 1.10, `learn_statistics/2` gives a unified way to get these learning statistics. That is, `learn_statistics(`*Name*`,`*Stat*`)` returns as *Stat* the statistic named *Name*. Basically, `learn_statistics(`*Name*`,`*Stat*`)` behaves the same as the built-in `get_`*Name*`(`*Stat*`)`, described in this section. On the other hand, when calling `learn_statistics(`*Name*`,`*Stat*`)` with *Name* being unbound, we can get all available statistics one after another by backtracking. The available statistics are shown in Table 4.1.

4.8 Model scoring*

In many 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 machine learning community, this problem should have been one of the most intensively explored topics in the last decade. In PRISM, the programming system just provides two simple Bayesian scores based on ML (§4.7.1) or MAP (§4.7.2) estimation, called Bayesian Information Criterion (BIC) [27] and Cheeseman-Stutz (CS) score [3]. Generally speaking, these Bayesian scores are known to be approximations of $\log P(D \mid M) = \log \int_\Theta P(D \mid \theta, M)P(\theta \mid M)d\theta$, 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). See [4] for more detailed descriptions about BIC and CS scores.

In PRISM, the model M is of course defined in the modeling part (§2.4), and after ML or MAP learning with some observed goals D (§4.7.3), `get_bic(`*Score*`)` returns *Score* as the BIC score of the model M given D. Also after MAP learning with D, `get_cs(`*Score*`)` returns *Score* as the CS score of M given D. Please note here that `get_cs/1` is available only after MAP

[10] 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 values subtracted by the number of occurred switches.

Table 4.1: Statistics referred to by `learn_statistics(`*Name*`, `*Stat*`)`.

Name	*Stat*
`log_likelihood`	Log likelihood
`log_post` †	Log of unnormalized a posteriori probability
`lambda`	Same as `log_likelihood` (in ML case) or `log_post` (in MAP case)
`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 parameters 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 score (see §4.8)
`cs` †	Cheeseman-Stutz score (see §4.8)
`search_time` †	Time consumed for the solution search (in seconds)
`learn_time` †	Time consumed for the entire learning procedure (in seconds)

†Only available after MAP estimation.

learning where the '`smooth`' flag (§4.12) is set as positive. Instead of using `get_bic(`*Score*`)` or `get_cs(`*Score*`)`, we can use `learn_statistics(bic,`*Score*`)` or `learn_statistics(cs,` *Score*`)`, respectively.

4.9 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 §2.4.3, 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 §2.4.3 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 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 [17]) 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 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 batch execution (§3.7) of the program that deals with failure, we need to run `upprismn` (also note that '`n`' is added), instead of `upprism`.

`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.

4.10 Avoiding underflow*

4.10.1 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 §4.5), 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, scaling is one way to deal with quite small numeric values which often lead to underflow. In the context of probabilistic modeling, for instance, hidden Markov models (HMMs) or other temporal models could have a very small probability for a long sequence, and so standard HMM-related systems employ some model-specific scaling methods. Another solution is to compute log-valued probabilities (as done in log-valued Viterbi computation), which is done by alternately calling the logarithmic function and the exponential function.

For the probabilistic inferences other than Viterbi computation, the programming system supports two scaling methods below as well as the method for computing log-valued probabilities.

- *Constant scaling*:

  In this scaling, 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 the appropriate constant number as the scaling factor.

- *Layered scaling*:

  In this scaling, promising scaling factors are automatically determined (thus the users need not to specify it), but the applicable programs are limited. That is, we can apply the method only to the program in which tabled subgoals can be partitioned into several groups called *layers*, which follow the conditions:

  1. The layers are acyclic with respect to the calling relationship.
  2. For any layer L, the number of L's occurrences is constant in every (unfolded) explanation.

  Fortunately however, it is confirmed that temporal models including HMMs, dynamic Bayesian networks (DBNs), and some specific case of probabilistic context-free grammars in Chomsky normal form satisfy the conditions above. Indeed, this scaling method is just a generalization of the one developed for HMMs. For example, let us consider the HMM program with the string length being 3. Then, for an observed goal `hmm([a,b,a])`, we can consider the layers from U_1 to U_5:

  $$\begin{aligned}
  U_1 &= \{\,\texttt{hmm([a,b,a])}\,\},\\
  U_2 &= \{\,\texttt{hmm(3,3,s0,[a,b,a])},\texttt{hmm(3,3,s1,[a,b,a])}\,\},\\
  U_3 &= \{\,\texttt{hmm(2,3,s0,[b,a])},\texttt{hmm(2,3,s1,[b,a])}\,\},\\
  U_4 &= \{\,\texttt{hmm(1,3,s0,[a])},\texttt{hmm(1,3,s1,[a])}\,\},\\
  U_5 &= \{\,\texttt{hmm(0,3,s0,[])},\texttt{hmm(0,3,s1,[])}\,\}
  \end{aligned}$$

  In layered scaling, an individual scaling factor is automatically determined for each layer. Only the users need to do is confirming whether, for each layer, the number of the layer's occurrences is constant among all search paths for the observed goal. In the above example, U_i ($i = 1,\ldots,5$) (to be more exact, one subgoal from U_i) appears just once in every search path for `hmm([a,b,a])`, and so we can say the layered scaling is applicable to this program.

4.10.2 Using methods for avoiding underflow

For Viterbi computation, setting '`on`' to the '`log_viterbi`' flag enables the log-valued Viterbi computation. See §4.12 for handling execution flags. The returned probability is log-valued.

For the other probabilistic inferences, the methods described in the previous section (§4.10.1) are specified by the '`scaling`' flag. This flag takes on `none`, `const`, `layer`, and `log_exp`. The value `none` (default) means we perform no scaling. `const` and `layer` mean doing the constant scaling and the layered scaling, respectively. By specifying `log_exp`, we make probability computations based on log-valued probabilities. For example, the following query enables constant scaling:

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

Keep in mind that, when using any scaling method, the probabilities returned by built-ins that computes probabilities (§4.3 and §4.6) will be log-valued.

To enable a scaling method *except* `log_exp`, we need to make extra settings:

- *Constant scaling*:

  We need to tell the scaling factor to the system, by setting 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).
  ```

- *Layered scaling*:

  To specify the layers, we introduce the predicate `layered/4`. For example, in the HMM program above, the following declarations will group the subgoals into the layers $U_1 \ldots U_5$:

  ```
  layered(hmm,4,1,integer).
  ```

  The meaning of this declaration is that, among the subgoals of `hmm/3`, ones that have the same integer value in the first argument will belong to the same layer. Moreover, for the case that we have non-ground observed goals, we need to add the following declaration:

  ```
  layered(hmm,1,1,length).
  ```

  This declaration means that among the subgoals of `hmm/1`, ones that have a list of the same length in the first argument will belong to the same layer. After adding this declaration, we can consider a non-ground goal `hmm([a,b,X])` and the following layers:

  U'_1 = {`hmm([a,b,X])`},
  U'_2 = {`hmm([a,b,a])`, `hmm([a,b,b])`},
  U'_3 = {`hmm(3,3,s0,[a,b,a])`, `hmm(3,3,s1,[a,b,a])`,
  `hmm(3,3,s0,[a,b,b])`, `hmm(3,3,s1,[a,b,b])`},
  U'_4 = {`hmm(2,3,s0,[b,a])`, `hmm(2,3,s1,[b,a])`,
  `hmm(2,3,s0,[b,b])`, `hmm(2,3,s1,[b,b])`},
  U'_5 = {`hmm(1,3,s0,[a])`, `hmm(1,3,s1,[a])`, `hmm(1,3,s0,[b])`, `hmm(1,3,s1,[b])`},
  U'_6 = {`hmm(0,3,s0,[])`, `hmm(0,3,s1,[])`}

  For each subgoal G' that does not match the declarations will form a layer whose only member is G'. So, if we know in advance that there are no non-ground observations, we can omit the declaration with respect to `hmm/1`.

  The matching pattern (the 4th argument) in `layered/4` is shown in Table 4.2. If we want to make a matching with more than one argument, it is allowed to use the list form. To show the example of this specification, we can modify the second argument as follows (though is redundant in this example):

  ```
  layered(hmm,4,[1,4],[integer,length]).
  ```

Table 4.2: Matching pattern used in `layered/4`.

`integer`	integer
`atom`	atom
`compound`	compound term
`length`	list length
`d_length`	the length of the list represented by the d-list (the functor of d-list is assumed to be `'-'/2`)
`depth`	term depth

4.10.3 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, constant scaling (specified by '`const`') should be fastest since we only need to multiply a constant number for each occurrence of switch instances. On the other hand, the method specified by '`log_exp`' requires additional computation time for calls of the logarithmic and exponential functions. In parameter learning with layered scaling (specified by '`layer`'), probability computation can be slow since *inter-goal sharing* [12], a simple optimization to make explanation graphs compact, is not allowed to be applied. This comes from the fact that, in layered scaling, we need a different scaling factor for each goal.

4.11 Keeping the solution table*

In version 1.10, the '`clean_table`' flag is introduced for a (partial) control of the solution table. If this flag is set to '`on`', which is the default, the programming system will automatically clean up all past results of explanation search (say, solutions) in the solution table[11] when invoking a routine that executes the explanation search (i.e. learning (§4.7) and probability computations (§4.3, §4.5, §4.6)). On the other hand, if the flag is set to '`off`' (see §4.12), the programming system does not clean up the solutions at all. Keeping and reusing the past solutions can be significantly useful when we only attempt to 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).

4.12 Execution flags

4.12.1 Handling execution flags

Since version 1.9, the system provides more than a dozen 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

[11] 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 arbitraty timing.

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).
```

- *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 (§4.10), we need to run `set_prism_flag(log_viterbi,on)` beforehand. Also we may run as a query `set_prism_flag(smooth,`*C*`)` in advance to make *smoothing* (i.e. MAP estimation) with the pseudo count *C*.

4.12.2 Available execution flags

Here we list the available execution flags:

- `verb` (possible values: `on` and `off`; default: `off`) — flag for enabling or disabling verbose mode.

- `warn` (possible values: `on` and `off`; default: `off`) — flag for enabling or disabling warning messages.

- `clean_table` (possible values: `on` and `off`; default: `on`) — flag for automatic cleaning of the solution table (see §4.11 for details). If this flag is set to '`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.

- `epsilon` (possible value: non-negative float; default: `1.0e-4`) — threshold ε for convergence in the EM algorithm (see §4.7.1).

- `smooth` (possible value: non-negative float; default: `0`) — pseudo count for MAP estimation (§4.7.2). If this flag is set to 0, the system will conduct ML estimation.

- `init` (possible values: `none`, `random` and `noisy_u`; default: `random`) — initialization method in the EM algorithm (§4.7.1). `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.

- `std_ratio` (possible value: non-negative float; default: `0.1`) — when we initialize the parameters in the EM algorithm (§4.7.1) with a k-valued switch according to a uniform distribution with Gaussian noises, where the noises are generated according to $N(1/k, (\texttt{std\_ratio} * (1/k))^2)$. The parameters will be normalized at the end of initialization.

- `restart` (possible value: positive integer; default: `1`) — number of restarts. Generally speaking, the EM algorithm (§4.7.1) 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.

- `max_iterate` (possible value: non-negative integer; default: `0`) — maximum number of EM iterations. In the EM algorithm (§4.7.1), 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.

- `log_viterbi` (possible values: `on` and `off`; default: `off`) — flag for enabling or disabling the log-valued version of Viterbi computation (§4.10). 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 (§4.10). If you wish to use some scaling method, use the `scaling` flag.

- `scaling` (possible values: `none`, `const`, `layer` and `log_exp`; default: `none`) — scaling methods. `none` means no scaling, `const` means doing the constant scaling, `layer` means doing the layered scaling, and `log_exp` means forcing 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 (§4.7.2) in parameter learning (this is because all relevant parameters should be non-zero to use `log_exp`). See §4.10 for a general description and the detailed usage on these scaling methods. If any value other than `none` is specified, the computed probabilities are obtained as log-valued. Please note that the value of this flag does not make any influence on the use of the log-valued version of Viterbi computation (§4.10 or §4.5). 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`) — scaling factor for constant scaling.

- `default_sw` (possible values: `none`, `uniform`, `f_geometric`, `f_geometric(`*Base*`)`, `f_geometric(`*Base*`,`*Type*`)`; default: `uniform`) — 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 to `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)`.

- `dynamic_default_sw` (possible values: `on` and `off`; default: `on`) — flag for 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 to '`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 (see above).

- `fix_init_order` (possible values: `on` and `off`; default: `on`) — flag for fixing the order of parameter initialization among switches. For an implementational reason, in the EM algorithm (§4.7.1), 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.

- `sort_hindsight` (possible values: `by_goal` and `by_prob`; default: `by_goal`) — flag for the mode on sorting the results of hindsight computation (§4.6). 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.

- `search_progress` (possible values: 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 to `0`, the message is suppressed.

- `em_progress` (possible values: non-negative integer; default: `10`) — the frequency of printing the progress message (i.e. the dot symbol) in the EM algorithm (§4.7.1). If this flag is set to `0`, the message is suppressed.

- `reduce_copy` (possible values: `on` and `off`; default: `off`) — flag for automatic copying of the Prolog terms returned by several built-ins (`probf/2`, `viterbif/3`, and so on; See §4.16). If this flag is set to '`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`', the copying will be skipped.

4.13 Random number generator

The following built-ins are provided to set information or retrieve information of the random number generator.[12] For sampling utilities based on discrete values, see §4.14.

- `random_float(`*Max*`,`*R*`)`: Generates a random number *R* in the range of `0...`*Max*.

- `get_seed(`*Seed*`)`: The seed used in the random generator is *Seed*.

- `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. This is equivalent to a sequential execution of `set_seed_time/0` and `get_seed(`*T*`)`.

[12] As a random number generator, the programming system uses *Mersenne Twister*, by incorporating the implementation of its authors' own available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html` . The development team is deeply grateful to the authors.

4.14 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 §4.13.

`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).

4.15 File IO

Basically, all B-Prolog's built-ins for file IO are also available for PRISM. In addition, since version 1.10, the programming system provides utilities for loading/saving clauses. The built-in `load_clauses(`*File*`,`*Clauses*`)` reads all clauses as a list *Clauses* from a file *File*, while `save_clauses(`*File*`,`*Clauses*`)` writes each element in *Clauses* as a clause into *File*. If you are only interested in some part of clauses, the following predicates would be useful:

- `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.
- `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.

4.16 Accessing to 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 the 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 can be safe if we are able to finish accessing to the terms before the next call of `viterbif/3`.

- `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 the 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

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 at the directories named 'exs' or 'exs_fail' in the released package.

5.1 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(`$\cdot$`)` chooses 'a' or 'b' as an emitted symbol at each state, and the state transition switches `tr(`$\cdot$`)` 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. 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` (§4.2) that calls `hmm/1` for *N* times.

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

```
% prism
   :
?- prism(hmm).
table hmm/1
```

[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.

```
table hmm/4
loading....hmm.psm.out
```

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

```
?- hmm_learn(100).

#goals: 0.........(97)
#graphs: 0.........(97)
#iterations: 0.........100.......(Converged: -689.116232627)
Finished learning
        Number of tabled subgoals: 1021
        Number of switches: 5
        Number of switch values: 10
        Number of iterations: 177
        Final log likelihood: -689.116233
        Total learning time: 0.210 seconds
        All solution search time: 0.060 seconds
        Total table space used: 402768 bytes
Type show_sw to show the probability distributions.
```

We can confirm the learned parameters by the built-in `show_sw/0` (§4.1.4):[2]

```
?- show_sw.

Switch init: unfixed: s0 (0.657062303207705) s1 (0.342937696792295)
Switch out(s0): unfixed: a (0.3257277231937) b (0.6742722768063)
Switch out(s1): unfixed: a (0.704817441866976) b (0.295182558133024)
Switch tr(s0): unfixed: s0 (0.284427965371372) s1 (0.715572034628628)
Switch tr(s1): unfixed: s0 (0.570367890086842) s1 (0.429632109913158)
```

Here we can make some probabilistic inferences based on the parameters estimated as above. To compute the most likely explanation (the Viterbi explanation) and its probability (the Viterbi probability) for a given observation, we can use the built-in `viterbif/1` (§4.5).

```
?- 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)
hmm(3,10,s0,[a,a,a,b,b,b,b,b])
     <= hmm(4,10,s1,[a,a,b,b,b,b,b]) & msw(out(s0),a) & msw(tr(s0),s1)
hmm(4,10,s1,[a,a,b,b,b,b,b])
     <= hmm(5,10,s0,[a,b,b,b,b,b]) & msw(out(s1),a) & msw(tr(s1),s0)

  ...omitted...

hmm(8,10,s1,[b,b,b])
     <= hmm(9,10,s0,[b,b]) & msw(out(s1),b) & msw(tr(s1),s0)
hmm(9,10,s0,[b,b])
```

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

Viterbi_P = 0.000117528
```

On the other hand, to compute the hindsight probabilities (§4.6) 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]): 0.000329700087289
  hmm(1,10,s1,[a,a,a,a,a,b,b,b,b,b]): 0.000316868405859
  hmm(2,10,s0,[a,a,a,a,b,b,b,b,b]): 0.000191994479969
  hmm(2,10,s1,[a,a,a,a,b,b,b,b,b]): 0.000454574013179
  hmm(3,10,s0,[a,a,a,b,b,b,b,b]): 0.000222026685023
  hmm(3,10,s1,[a,a,a,b,b,b,b,b]): 0.000424541808125

  ...omitted...

  hmm(8,10,s0,[b,b,b]): 0.000366678621664
  hmm(8,10,s1,[b,b,b]): 0.000279889871484
  hmm(9,10,s0,[b,b]): 0.000350254883354
  hmm(9,10,s1,[b,b]): 0.000296313609794
  hmm(10,10,s0,[b]): 0.000389511170922
  hmm(10,10,s1,[b]): 0.000257057322226
  hmm(11,10,s0,[]): 0.000257405112344
  hmm(11,10,s1,[]): 0.000389163380804
```

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

By specifying the execution flags (§4.12), 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 related flags as follows:

```
?- set_prism_flag(restart,10),set_prism_flag(smooth,0.5).
```

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

```
?- hmm_learn(100).

#goals: 0.........(94)
#graphs: 0.........(94)
[0]#iterations: 0.........100........(Converged: -686.955199159)
[1]#iterations: 0.........100.........200.....(Converged: -686.959803476)
[2]#iterations: 0.........100......(Converged: -686.955218214)
[3]#iterations: 0.........100.........(Converged: -686.955374352)
[4]#iterations: 0.........100.........200......(Converged: -686.958903786)
[5]#iterations: 0.........100.........200.....(Converged: -686.958217725)
[6]#iterations: 0.........100.........(Converged: -686.956113248)
[7]#iterations: 0.........100.......(Converged: -686.955115855)
```

```
[8]#iterations: 0.........100.........200(Converged: -686.955300503)
[9]#iterations: 0.........100.........200.......(Converged: -686.960238496)
Finished learning
        Number of tabled subgoals: 1000
        Number of switches: 5
        Number of switch values: 10
        Number of iterations: 179
        Final log of a posteriori prob: -686.955116
        Total learning time: 1.131 seconds
        All solution search time: 0.060 seconds
        Total table space used: 394476 bytes
Type show_sw 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(smooth,0.5).
```

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)
#graphs: 0....(49)
[0]#iterations: 0.........100........(Converged: -347.352030044)
[1]#iterations: 0.........100...(Converged: -347.347902763)
[2]#iterations: 0.........(Converged: -347.353010697)
[3]#iterations: 0.........100.....(Converged: -347.352436731)
[4]#iterations: 0.........100.........200...(Converged: -347.353537932)
[5]#iterations: 0.........100.........200......(Converged: -347.285900532)
[6]#iterations: 0.........100.......(Converged: -347.352426257)
[7]#iterations: 0.........100.........200..(Converged: -347.351954980)
[8]#iterations: 0......(Converged: -347.345817648)
[9]#iterations: 0.........100.........200.........300....(Converged: -347.2
87105601)
Finished learning
        Number of tabled subgoals: 585
        Number of switches: 5
```

```
        Number of switch values: 10
        Number of iterations: 264
        Final log of a posteriori prob: -347.285901
        Total learning time: 0.511 seconds
        All solution search time: 0.030 seconds
        Total table space used: 230896 bytes
Type show_sw 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 the other facilities in a shell script.

5.2 Discrete 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 arcs indicate conditional independences 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 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: $v_{i,j}$ ($j = 1, \ldots, k$).[3] Then it is easily seen that each switch parameter corresponds to one entry of the CPT.

For illustration, let us consider an example from [13], shown in Figure 5.1. 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:

$$p(\mathit{Fire}, \mathit{Tampering}, \mathit{Smoke}, \mathit{Alarm}, \mathit{Leaving}, \mathit{Report}).$$

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

$$\begin{aligned} &p(\mathit{Fire}, \mathit{Tampering}, \mathit{Smoke}, \mathit{Alarm}, \mathit{Leaving}, \mathit{Report}) \\ &= p(\mathit{Fire})p(\mathit{Tampering})p(\mathit{Smoke} \mid \mathit{Fire}) \cdot \\ &\quad p(\mathit{Alarm} \mid \mathit{Fire}, \mathit{Tampering})p(\mathit{Leaving} \mid \mathit{Alarm})p(\mathit{Report} \mid \mathit{Leaving}). \end{aligned} \tag{5.1}$$

The factored probabilities in the RHS will be stored in CPTs, where $P(\mathit{Fire})$ and $P(\mathit{Tampering})$ 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:

$$\begin{aligned} &p(\mathit{Smoke}, \mathit{Report}) \\ &= \sum_{\mathit{Fire},\ \mathit{Tampering},\ \mathit{Alarm},\ \mathit{Leaving}} p(\mathit{Fire}, \mathit{Tampering}, \mathit{Smoke}, \mathit{Alarm}, \mathit{Leaving}, \mathit{Report}). \end{aligned} \tag{5.2}$$

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

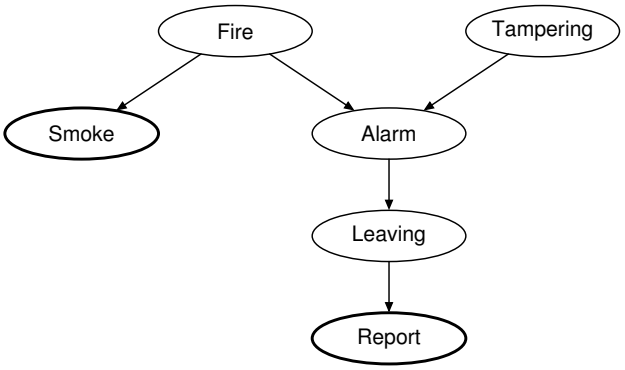

Figure 5.1: Example of a discrete Bayesian network.

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 [23] 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 5.1. 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* | *Fire*), and so on. Here we represent such a coin flipping by `msw(`I`,`V`)`, and define the joint distribution (Eq. 5.1) with a predicate `world/6`:

```
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 $\theta_{\texttt{sm(yes),no}}$, the probability of a switch instance `msw(sm(yes),no)` being true corresponds to the conditional probability *P*(*Smoke* = *no* | *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. 5.2, so we can say the model is valid. 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 (§5.1). 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; §4.1.2), `get_samples/3` (which generate training data; §4.2), and `learn/1` (§4.7.3) are used similarly to those in the batch routine for the experiments with HMMs (§5.1). `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` (§4.1.3).

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)
#graphs: 0(4)
```

```
#iterations: 0(Converged: -101.272680727)
Finished learning
        Number of tabled subgoals: 68
        Number of switches: 12
        Number of switch values: 24
        Number of iterations: 2
        Final log likelihood: -101.272681
        Total learning time: 0.020 seconds
        All solution search time: 0.010 seconds
        Total table space used: 24752 bytes
Type show_sw to show the probability distributions.
```

We can confirm the learned parameters as follows:

```
?- show_sw.

Switch fi: fixed: yes (0.1) no (0.9)
Switch ta: unfixed: yes (0.519399313310633) no (0.480600686689367)
Switch le(no): unfixed: yes (0.384721321533468) no (0.615278678466532)
Switch le(yes): unfixed: yes (0.404738457815134) no (0.595261542184866)
Switch re(no): unfixed: yes (0.289857819297058) no (0.710142180702942)
Switch re(yes): unfixed: yes (0.21427119569658) no (0.78572880430342)
Switch sm(no): unfixed: yes (0.159674369834434) no (0.840325630165566)
Switch sm(yes): unfixed: yes (0.162930671951014) no (0.837069328048986)
Switch al(no,no): unfixed: yes (0.518638757519486) no (0.481361242480514)
Switch al(no,yes): unfixed: yes (0.50764550015705) no (0.49235449984295)
Switch al(yes,no): unfixed: yes (0.491645541377827) no (0.508354458622173)
Switch al(yes,yes): unfixed: yes (0.557537160970952) no (0.442462839029048)
```

It is also possible to get the frequencies of the sampled goals:

```
?- show_goals.

Goal world(no,no) (count=67, freq=67.000%)
Goal world(yes,yes) (count=9, freq=9.000%)
Goal world(yes,no) (count=7, freq=7.000%)
Goal world(no,yes) (count=17, freq=17.000%)
Total_count=100
```

Furthermore, for the Bayesian network program described in this section, conditional probabilities can be computed as conditional hindsight probabilities (§4.6). 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 3rd argument in `world/6` (which corresponds to *Alarm*) as a query argument:[4]

```
?- chindsight_agg(world(_,_,_,yes,_,no),world(_,_,query,yes,_,no)).
```

[4]In this computation, it is assumed that the parameters are set by `set_params/0` in advance.

```
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 (like the one proposed in [20]) of Bayesian networks which takes into account the computational effort for conditional hindsight probabilities, compared to the sophisticated algorithms for Bayesian networks proposed so far.

5.3 Statistical analysis

PRISM is a suitable tool for analyzing statistical data. In this section, we present two examples. The first 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 second example attempts to obtain statistics that can be used to tune the unification procedure.

5.3.1 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 wining 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.

5.3.2 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(t1),car(t2)) && unify(cdr(t1),cdr(t2));
        else return false;
    } else if (t1 is a structure){
        if (t2 is variable) bind t2 to t1;
        else if (t2 is a structure) {
            let t1 be f(a1,...,an) and t2 be g(b1,...,bm);
            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)
#graphs: 0(3)
#iterations: 0(Converged: -9.704060528)
Finished learning
        Number of tabled subgoals: 23
        Number of switches: 4
        Number of switch values: 16
        Number of iterations: 2
        Final log likelihood: -9.704061
        Total learning time: 0.030 seconds
        All solution search time: 0.030 seconds
        Total table space used: 6860 bytes
Type show_sw to show the probability distributions.

yes
?- show_sw.

Switch s1: unfixed: var (0.625) atom (0.125) list (0.0) struct (0.25)
Switch s2(atom): unfixed: var (0.0) atom (1.0) list (0.0) struct (0.0)
Switch s2(struct): unfixed: var (0.0) atom (0.0) list (0.0) struct (1.0)
Switch s2(var): unfixed: var (0.2) atom (0.8) list (0.0) struct (0.0)
```

From this result, we know how to order the tests of types so that the unification procedure performs the best on the samples.

5.4 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 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 5.2, as a basic model that captures the professor's probabilistic behavior. We then try to write a

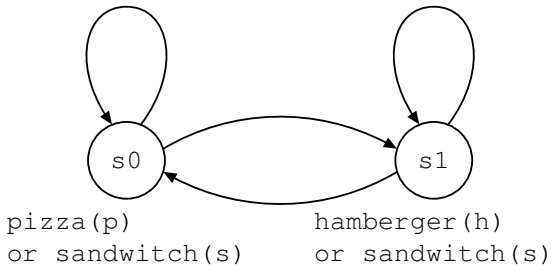

Figure 5.2: State transition diagram of the dieting professor.

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 5.2, we can see that four switches are required as follows:

```
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(·)` determine the next restaurant, and those named `lunch(·)` determine the menu of lunch at the corresponding 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` (§5.1), 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 (§5.1) and discrete Bayesian networks (§5.2) — 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

table failure/0
table failure/1
table success/0
table success/1
table closure_success0/1
table closure_chmm0/5
table closure_success0/2
table closure_success1/1
table closure_chmm1/4
table chmm/4
compiled in 90 milliseconds
loading....temp.out
```

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) [17] 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 (0.4) s (0.6)
Switch lunch(s1): unfixed: h (0.5) s (0.5)
Switch tr(s0): unfixed: s0 (0.7) s1 (0.3)
Switch tr(s1): unfixed: s1 (0.7) s0 (0.3)
```

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:

```
?- prob(success,Ps),prob(failure,Pf),X is Ps+Pf.

Pf = 0.348592596784
Ps = 0.651407403216
X = 1.0 ?
```

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 = 0.99862357726
Ps = 0.00137642274
L = [s,s,s,h,h,s,p]
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]).

#goals: 0.........100.........200......(261)
#graphs: 0.........100.........200......(261)
```

```
#iterations: 0.....(Converged: -2964.779121734)
Finished learning
        Number of tabled subgoals: 2275
        Number of switches: 4
        Number of switch values: 8
        Number of iterations: 55
        Final log likelihood: -2964.779122
        Total learning time: 0.220 seconds
        All solution search time: 0.090 seconds
        Total table space used: 778384 bytes
Type show_sw to show the probability distributions.
Gs = [success([s,h,s,s,s,h,s]),success([s,p,h,h,s,s,s]),
      ... omitted ...
      success([s,p,s,s,s,s,s]),success([s,h,h,h,h,p,s])] ?
```

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 (0.417373118524406) s (0.582626881475594)
Switch lunch(s1): unfixed: h (0.492000452846571) s (0.507999547153429)
Switch tr(s0): unfixed: s0 (0.705730869553732) s1 (0.294269130446268)
Switch tr(s1): unfixed: s1 (0.710909192666213) s0 (0.289090807333787)
```

Bibliography

[1] N. Angelopoulos. Extending the CLP engine for reasoning under uncertainty. In *14th International Symposium on Methodologies for Intelligent Systems (ISMIS2003)*, pages 365–373, 2003.

[2] E. Charniak. *Statistical Language Learning*. The MIT Press, 1993.

[3] 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.

[4] D. Chickering and D. Heckerman. Efficient approximation for the marginal likelihood of Bayesian networks with hidden variables. *Machine Learning*, 29:181–212, 1997.

[5] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[6] J. F. Crow. *Basic Concepts in Population Quantitative and Evolutionary Genetics*. W. H. Freeman and Company, 1988. Translated into Japanese.

[7] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44:245–271, 2001.

[8] 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.

[9] M. Jaeger. Ignorability for categorical data. *The Annals of Statistics*, 33(4):1964–1981, 2005.

[10] 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.

[11] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. In *Proc. of the 1st International Conference on Computational Logic (CL2000)*, pages 269–294, 2000.

[12] Y. Kameya, T. Sato, and Zhou N.-F. Yet more efficient EM learning for parameterized logic programs by inter-goal sharing. In *Proc. of the 16th European Conference on Artificial Intelligence (ECAI2004)*, pages 490–494, 2004.

[13] D. Poole. Probabilistic Horn abduction. *Artificial Intelligence*, 64(1):81–129, 1993.

[14] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proc. of IEEE*, volume 77, pages 257–286, 1989.

[15] D. B. Rubin. Inference and missing data. *Biometrika*, 63:581–592, 1976.

[16] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[17] T. Sato. First Order Compiler: a deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8:605–627, 1989.

[18] T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proc. of the 12th International Conference on Logic Programming (ICLP95)*, pages 715–729, 1995.

[19] T. Sato. Modeling scientific theories as PRISM programs. In *Proc. of ECAI-98 Workshop on Machine Discovery*, pages 37–45, 1998.

[20] T. Sato. Inside-Outside probability computation for belief propagation. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007. To appear.

[21] T. Sato and Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 1330–1335, 1997.

[22] T. Sato and Y. Kameya. A Viterbi-like algorithm and EM learning for statistical abduction. In *Proc. of UAI-2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support*, 2000.

[23] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[24] 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.

[25] 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.

[26] 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.

[27] G. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, 6(2):461–464, 1978.

[28] L. Sterling and E. Sharpiro. *The Art of Prolog*. The MIT Press, 1986.

[29] N.-F. Zhou and T. Sato. Efficient fixpoint computation in linear tabling. In *Proc. of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP-03)*, pages 275–283, 2003.

[30] N.-F. Zhou, T. Sato, and K. Hashida. Toward a high-performance system for symbolic and statistical modeling. In *Proc. 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