New Advances in Logic-Based Probabilistic Modeling by PRISM

Taisuke Sato and Yoshitaka Kameya

Tokyo Institute of Technology, Ookayama Meguro Tokyo Japan,
`http://sato-www.cs.titech.ac.jp/`

Abstract. We review a logic-based modeling language PRISM and report recent developments including belief propagation by the generalized inside-outside algorithm and generative modeling with constraints. The former implies PRISM subsumes belief propagation at the algorithmic level. We also compare the performance of PRISM with state-of-the-art systems in statistical natural language processing and probabilistic inference in Bayesian networks respectively, and show that PRISM is reasonably competitive.

1 Introduction

The objective of this chapter is to review PRISM,[1] a logic-based modeling language that has been developed since 1997, and report its current status.[2]

PRISM was born in 1997 as an experimental language for unifying logic programming and probabilistic modeling [1]. It is an embodiment of the *distribution semantics* proposed in 1995 [2] and the first programming language with the ability to perform EM (expectation-maximization) learning [3] of parameters in programs. Looking back, when it was born, it already subsumed BNs (Bayesian networks), HMMs (hidden Markov models) and PCFGs (probabilistic context free grammars) semantically and could compute their probabilities.[3] However there was a serious problem: most of probability computation was exponential. Later in 2001, we added a tabling mechanism [4, 5] and "largely solved" this problem. Tabling enables both reuse of computed results and dynamic programming for probability computation which realizes standard polynomial time probability computations for singly connected BNs, HMMs and PCFGs [6].

Two problems remained though. One is the *no-failure condition* that dictates that failure must not occur in a probabilistic model. It is placed for mathematical consistency of defined distributions but obviously an obstacle against the use of constraints in probabilistic modeling. This is because constraints may be

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

[2] This work is supported in part by the 21st Century COE Program 'Framework for Systematization and Application of Large-scale Knowledge Resources' and also in part by Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B), 2006, 17300043.

[3] We assume that BNs and HMMs in this chapter are discrete.

unsatisfiable thereby causing failure of computation and the failed computation means the loss of probability mass. In 2005, we succeeded in eliminating this condition by merging the FAM (failure-adjusted maximization) algorithm [7] with the idea of logic program synthesis [8].

The other problem is inference in multiply connected BNs. When a Bayesian network is singly connected, it is relatively easy to write a program that simulates $\pi\lambda$ message passing [9] and see the correctness of the program [6]. When, on the other hand, the network is not singly connected, it has been customarily to use the junction tree algorithm but how to realize BP (belief propagation)[4] on junction trees in PRISM has been unclear.[5] In 2006 however, it was found and proved that BP on junction trees is a special case of probability computation by the IO (inside-outside) algorithm generalized for logic programs used in PRISM [10].

As a result, we can now claim that PRISM *uniformly* subsumes BNs, HMMs and PCFGs at the algorithmic level as well as at the semantic level. All we need to do is to write appropriate programs for each model so that they denote intended distributions. PRISM's probability computation and EM learning for these programs exactly coincides with the standard algorithms for each model, i.e. the junction tree algorithm for BNs [11, 12], the Baum-Welch (forward-backward) algorithm for HMMs [13] and the IO algorithm for PCFGs [14] respectively.

This is just a theoretical statement though, and the actual efficiency of probability computation and EM learning is another matter which depends on implementation and should be gauged against real data. Since our language is at an extremely high level (predicate calculus) and the data structure is very flexible (terms containing variables), we cannot expect the same speed as a C implementation of a specific model. However due to the continuing implementation efforts made in the past few years, PRISM's execution speed has greatly improved to the point of being usable for medium-sized machine learning experiments. We have conducted comparative experiments with Dyna [15] and ACE [16–18]. Dyna is a dynamic programming system for statistical natural language processing and ACE is a compiler that compiles a Bayesian network into an arithmetic circuit to perform probabilistic inference. Both represent the state-of-the-art approach in each field. Results are encouraging and demonstrate PRISM's competitiveness in probabilistic modeling.

That being said, we would like to emphasize that although the generality and descriptive power of PRISM enables us to treat existing probabilistic models uniformly, it should also be exploited for exploring new probabilistic models. One such model, *constrained HMM*s that combine HMMs with constraints, is explained in Section 5.

In what follows, we first look at the basics of PRISM [6] in Section 2. Then in Section 3, we explain how to realize BP in PRISM using logically described junc-

[4] We use BP as a synonym of the part of the junction tree algorithm concerning message passing.

[5] Contrastingly it is straightforward to simulate variable elimination for multiply connected BNs [6].

tion trees. Section 4 deals with the system performance of PRISM and contains comparative data with Dyna and ACE. Section 5 contains generative modeling with constraints made possible by the elimination of the no-failure condition. Related work and future topics are discussed in Section 6. We assume the reader is familiar with logic programming [19], PCFGs [20] and BNs [9, 21].

2 The Basic System

2.1 Programs as distributions

One distinguished characteristic of PRISM is its declarative semantics. For self-containedness, in a slightly different way from that of [6], we quickly define the semantics of PRISM programs, the distribution semantics [2], which regards programs as defining infinite probability distributions.

Overview of the distribution semantics: In the distribution semantics, we consider a logic program DB which consists of a set F of facts (unit clauses) and a set R of rules (non-unit definite clauses). That is, we have $DB = F \cup R$. We assume the *disjoint condition* that there is no atom in F unifiable with the head of any clause in R. Semantically DB is treated as the set of all ground instances of the clauses in DB. So in what follows, F and R are equated with their ground instantiations. In particular F is a set of ground atoms. Since our language includes countably many predicate and function symbols, F and R are countably infinite.

We construct an infinite distribution, or to be more exact, a probability measure P_{DB}[6] on the set of possible Herbrand interpretations [19] of DB as the denotation of DB in two steps.

Let a sample space Ω_F (resp. Ω_{DB}) be all interpretations (truth value assignments) for the atoms appearing in F (resp. DB). They are so called the "possible worlds" for F (resp. DB). We construct a probability space on Ω_F and then extend it to a larger probability space on Ω_{DB} where the probability mass is distributed only over the least Herbrand models made from DB. Note that Ω_F and Ω_{DB} are uncountably infinite. We construct their probability measures, P_F and P_{DB} respectively, from a family of finite probability measures using Kolmogorov's extension theorem.[7]

[6] A probability space is a triplet $(\Omega, \mathcal{F}, P)$ where Ω is a sample space (the set of possible outcomes), $\mathcal{F}$ a σ-algebra which consists of subsets of Ω and is closed under complementation and countable union, and P a probability measure which is a function from sets $\mathcal{F}$ to real numbers in $[0, 1]$. Every set S in $\mathcal{F}$ is said to be measurable by P and assigned probability $P(S)$.

[7] Given denumerably many, for instance, discrete joint distributions satisfying a certain condition, Kolmogorov's extension theorem guarantees the existence of an infinite distribution (probability measure) which is an extension of each component distribution [22].

Constructing P_F: Let $A_1, A_2, \ldots$ be an enumeration of the atoms in F. A truth value assignment for the atoms in F is represented by an infinite vector of 0s and 1s in such way that i-th value is 1 when A_i is true and 0 otherwise. Thus the sample space, F's all truth value assignments, is represented by a set of infinite vectors $\Omega_F = \prod_{i=1}^{\infty}\{0,1\}_i$.

We next introduce finite probability measures $P_F^{(n)}$ on $\Omega_F^{(n)} = \prod_{i=1}^{n}\{0,1\}_i$ ($n = 1, 2, \ldots$). We choose 2^n real numbers $p_\nu^{(n)}$ for each n such that $0 \leq p_\nu^{(n)} \leq 1$ and $\sum_{\nu \in \Omega_F^{(n)}} p_\nu^{(n)} = 1$. We put $P_F^{(n)}(\{\nu\}) = p_\nu^{(n)}$ for $\nu = (x_1, \ldots, x_n) \in \Omega_F^{(n)}$, which defines a finite probability measure $P_F^{(n)}$ on $\Omega_F^{(n)}$ in an obvious way. $p_\nu^{(n)} = p_{(x_1,\ldots,x_n)}^{(n)}$ is a probability that $A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}$ is true where $A^x = A$ (when $x = 1$) and $A^x = \neg A$ (when $x = 0$).

We here require that the *compatibility condition* below hold for the $p_\nu^{(n)}$s:

$$\sum_{x_{n+1} \in \{0,1\}} p_{(x_1,\ldots,x_n,x_{n+1})}^{(n+1)} = p_{(x_1,\ldots,x_n)}^{(n)} .$$

It follows from the compatibility condition and Kolmogorov's extension theorem [22] that we can construct a probability measure on Ω_F by merging the family of finite probability measures $P_F^{(n)}$ ($n = 1, 2, \ldots$). That is, there uniquely exists a probability measure P_F on the minimum σ-algebra $\mathcal{F}$ that contains subsets of Ω_F of the form

$$[A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F \overset{\text{def}}{=} \{\nu \mid \nu = (x_1, x_2, \ldots, x_n, *, *, \ldots) \in \Omega_F, * \text{ is either 1 or 0}\}$$

such that P_F is an extension of each $P_F^{(n)}$ ($n = 1, 2, \ldots$):

$$P_F([A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F) = p_{(x_1,\ldots,x_n)}^{(n)} .$$

In this construction of P_F, it must be emphasized that the choice of $P_F^{(n)}$ is arbitrary as long as they satisfy the compatibility condition.

Having constructed a probability space $(\Omega_F, \mathcal{F}, P_F)$, we can now consider each ground atom A_i in F as a random variable that takes 1 (true) or 0 (false). We introduce a probability function $P_F(A_1 = x_1, \ldots, A_n = x_n) = P_F([A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F)$. We here use, for notational convenience, P_F both as the probability function and as the corresponding probability measure. We call P_F a *base distribution* for DB.

Extending P_F to P_{DB}: Next, let us consider an enumeration $B_1, B_2, \ldots$ of atoms appearing in DB. Note that it necessarily includes some enumeration of F. Also let $\Omega_{DB} = \prod_{j=1}^{\infty}\{0,1\}_j$ be the set of all interpretations (truth assignments) for the B_is and $M_{DB}(\nu)$ ($\nu \in \Omega_F$) the least Herbrand model [19] of a program $R \cup F_\nu$ where F_ν is the set of atoms in F made true by the truth assignment ν. We consider the following sets of interpretations for F and DB, respectively:

$$[B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_F \overset{\text{def}}{=} \{\nu \in \Omega_F \mid M_{DB}(\nu) \models B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}\},$$

$$[B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_{DB} \overset{\text{def}}{=} \{\omega \in \Omega_{DB} \mid \omega \models B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}\}.$$

We remark that $[\cdot]_F$ is measurable by P_F. So define $P_{DB}^{(n)}$ $(n > 0)$ by:

$$P_{DB}^{(n)}([B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_{DB}) \overset{\text{def}}{=} P_F([B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_F).$$

It is easily seen from the definition of $[\cdot]_F$ that $[B_1^{y_1} \wedge \cdots \wedge B_n^{y_n} \wedge B_{n+1}]_F$ and $[B_1^{y_1} \wedge \cdots \wedge B_n^{y_n} \wedge \neg B_{n+1}]_F$ form a partition of $[B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_F$, and hence the following compatibility condition holds:

$$\textstyle\sum_{y_{n+1}\in\{0,1\}} P_{DB}^{(n+1)}([B_1^{y_1} \wedge \cdots \wedge B_n^{y_n} \wedge B_{n+1}^{y_{n+1}}]_{DB}) = P_{DB}^{(n)}([B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_{DB}).$$

Therefore, similarly to P_F, we can construct a unique probability measure P_{DB} on the minimum σ-algebra containing open sets of Ω_{DB}[8] which is an extension of $P_{DB}^{(n)}$ and P_F, and obtain a (-n infinite) joint distribution such that $P_{DB}(B_1 = y_1, \ldots, B_n = y_n) = P_F([B_1^{y_1} \wedge \cdots \wedge B_n^{y_n}]_F)$ for any $n > 0$. The *distribution semantics* is the semantics that considers P_{DB} as the denotation of DB. It is a probabilistic extension of the standard least model semantics in logic programming and gives a probability measure on the set of possible Herbrand interpretations of DB [2, 6].

Since $[G] \overset{\text{def}}{=} \{\omega \in \Omega_{DB} \mid \omega \models G\}$ is P_{DB}-measurable for any closed formula G built from the symbols appearing in DB, we can define the probability of G as $P_{DB}([G])$. In particular, quantifiers are numerically approximated as we have

$$\begin{aligned}
&\lim_{n\to\infty} P_{DB}([G(t_1) \wedge \cdots \wedge G(t_n)]) = P_{DB}([\forall x G(x)]),\\
&\lim_{n\to\infty} P_{DB}([G(t_1) \vee \cdots \vee G(t_n)]) = P_{DB}([\exists x G(x)]),
\end{aligned}$$

where $t_1, t_2, \ldots$ is an enumeration of all ground terms.

Note that properties of the distribution semantics described so far only assume the disjoint condition. In the rest of this section, we may use $P_{DB}(G)$ instead of $P_{DB}([G])$. Likewise $P_F(A_{i_1}=1, A_{i_2}=1, \ldots)$ is sometimes abbreviated to $P_F(A_{i_1}, A_{i_2}, \ldots)$.

PRISM programs: The distribution semantics has freedom in the choice of $P_F^{(n)}$ $(n = 1, 2, \ldots)$ as long as they satisfy the compatibility condition. In PRISM which is an embodiment of the distribution semantics, the following requirements are imposed on F and P_F w.r.t. a program $DB = F \cup R$:

– Each (ground) atom in F takes the form $\mathtt{msw}(i, n, v)$,[9] which is interpreted that "a switch named i randomly takes a value v at the trial n." For each switch i, a finite set V_i of possible values it can take is given in advance.[10]
– Each switch i chooses a value exclusively from V_i, i.e. for any ground terms i and n, it holds that $P_F(\mathtt{msw}(i, n, v_1), \mathtt{msw}(i, n, v_2)) = 0$ for every $v_1 \neq v_2 \in V_i$ and $\sum_{v\in V_i} P_F(\mathtt{msw}(i, n, v)) = 1$.

[8] Each component space $\{0, 1\}$ of Ω_{DB} carries the discrete topology.

[9] `msw` is an abbreviation for *multi-ary random switch*.

[10] The intention is that $\{\mathtt{msw}(i, n, v) \mid v \in V_i\}$ jointly represents a random variable X_i whose range is V_i. In particular, $\mathtt{msw}(i, n, v)$ represents $X_i = v$.

- The choices of each switch made at different trials obey the same distribution, i.e. for each ground term i and any different ground terms $n_1 \neq n_2$, $P_F(\mathtt{msw}(i, n_1, v)) = P_F(\mathtt{msw}(i, n_2, v))$ holds. Hence we denote $P_F(\mathtt{msw}(i, \cdot, v))$ by $\theta_{i,v}$ and call it as a *parameter* of switch i.
- The choices of different switches or different trials are independent. The joint distribution of atoms in F is decomposed as $\prod_{i,n} P_F(\mathtt{msw}(i, n, v_{i1}) = x^n_{i1}, \ldots, \mathtt{msw}(i, n, v_{iK_i}) = x^n_{iK_i})$, where $V_i = \{v_{i1}, v_{i2}, \ldots, v_{iK_i}\}$.

The disjoint condition is automatically satisfied since `msw/3`, the only predicate for F, is a built-in predicate and cannot be redefined by the user.

To construct P_F that satisfies the conditions listed above, let $A_1, A_2, \ldots$ be an enumeration of the `msw` atoms in F and put $N^{(m)}_{i,n} \overset{\text{def}}{=} \{k \mid \mathtt{msw}(i, n, v_{ik}) \in \{A_1, \ldots, A_m\}\}$. We have $\{A_1, \ldots, A_m\} = \bigcup_{i,n}\{\mathtt{msw}(i, n, v_{ik}) \mid k \in N^{(m)}_{i,n}\}$. We introduce a joint distribution $P^{(m)}_F$ over $\{A_1, \ldots, A_m\}$ for each $m > 0$ which is decomposed as

$\prod_{i,n:N^{(m)}_{i,n} \neq \phi} P^{(m)}_{i,n}\left(\bigwedge_{k\in N^{(m)}_{i,n}} \mathtt{msw}(i, n, v_{ik}) = x^n_{ik}\right)$ where

$$P^{(m)}_{i,n}\left(\bigwedge_{k\in N^{(m)}_{i,n}} \mathtt{msw}(i, n, v_{ik}) = x^n_{ik}\right) \overset{\text{def}}{=} \begin{cases} \theta_{i,v_{ik}} & \text{if } x^n_{ik} = 1, x^n_{ik'} = 0 \ (k' \neq k) \\ 1 - \sum_{k\in N^{(m)}_{i,n}} \theta_{i,v_{ik}} & \text{if } x^n_{ik} = 0 \text{ for every } k \in N^{(m)}_{i,n} \\ 0 & \text{otherwise.} \end{cases}$$

$P^{(m)}_F$ obviously satisfies the second and the third conditions. Besides the compatibility condition holds for the family $P^{(m)}_F$ $(m = 1, 2, \ldots)$. Hence P_{DB} is definable for every DB based on $\{P^{(m)}_F \mid m = 1, 2, \ldots\}$.

We remark that the `msw` atoms can be considered as a syntactic specialization of assumables in PHA (probabilistic Horn abduction) [23] or atomic choices in ICL (independent choice logic) [24] (see also Chapter 9 in this volume), but without imposing restrictions on modeling itself. We also point out that there are notable differences between PRISM and PHA/ICL. First unlike PRISM, PHA/ICL has no explicitly defined infinite probability space. Second the role of assumptions differs in PHA/ICL. While the assumptions in Subsection 2.2 are introduced just for the sake of computational efficiency and have no role in the definability of semantics, the assumptions made in PHA/ICL are indispensable for their language and semantics.

Program example: As an example of a PRISM program, let us consider a left-to-right HMM described in Fig. 1. This HMM has four states $\{\mathtt{s0}, \mathtt{s1}, \mathtt{s2}, \mathtt{s3}\}$ where `s0` is the initial state and `s3` is the final state. In each state, the HMM outputs a symbol either '`a`' or '`b`'. The program for this HMM is shown in Fig. 2.

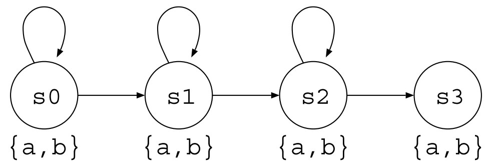

Fig. 1. Example of a left-to-right HMM with four states.

The first four clauses in the program are called *declarations* where `target`(p/n) declares that the observable event is represented by the predicate p/n, and `values`(i, V_i) says that V_i is a list of possible values the switch i can take (a `values` declaration can be seen as a 'macro' notation for a set of facts in F). The remaining clauses define the probability distribution on the strings generated by the HMM. `hmm`(Cs) denotes a probabilistic event that the HMM generates a string Cs. `hmm`(T, S, Cs') denotes that the HMM, whose state is S at time T, generates a substring Cs' from that time on. The comments in Fig. 2 describe a procedural behavior of the HMM as a string generator. It is important to note here that this program has no limit on the string length, and therefore it implicitly contains countably infinite ground atoms. Nevertheless, thanks to the distribution semantics, their infinite joint distribution is defined with mathematical rigor.

```
target(hmm/1).
values(tr(s0),[s0,s1]).
values(tr(s1),[s1,s2]).
values(tr(s2),[s2,s3]).
values(out(_),[a,b]).

hmm(Cs):- hmm(0,s0,Cs).

hmm(T,s3,[C]):- msw(out(s3),T,C). % If at the final state:
                                  %   output a symbol and then terminate.
hmm(T,S,[C|Cs]):- S\==s3,         % If not at the final state:
  msw(out(S),T,C),                %   choose a symbol to be output,
  msw(tr(S),T,Next),              %   choose the next state,
  T1 is T+1,                      %   Put the clock ahead,
  hmm(T1,Next,Cs).                %   and enter the next loop.
```

Fig. 2. PRISM program for the left-to-right HMM, which uses `msw/3`.

One potentially confusing issue in our current implementation is the use of `msw`(i, v), where the second argument is omitted from the original definition for the sake of simplicity and efficiency. Thus, to run the program in Fig. 2 in practice, we need to delete the second argument in `msw/3` and the first argument in `hmm/3`, i.e. the 'clock' variables `T` and `T1` accordingly. When there are multiple occurrences of $\mathtt{msw}(i, v)$ in a proof tree, we assume by default that their original second arguments differ and their choices, though they happened to be the same, v, are made independently.[11] In the sequel, we will use `msw/2` instead of `msw/3`.

2.2 Realizing generality with efficiency

Explanations for observations: So far we have been taking a model-theoretic approach to define the language semantics where an interpretation (a possible world) is a fundamental concept. From now on, to achieve efficient probabilistic inference, we take a proof-theoretic view and introduce the notion of *explanation*. Let us consider again a PRISM program $DB = F \cup R$. For a ground goal G, we can obtain $G \Leftrightarrow E_1 \vee E_2 \vee \cdots \vee E_K$ by logical inference from the completion [25] of R where each E_k $(k = 1, \ldots, K)$ is a conjunction of switches (ground atoms in F). We sometimes call G an *observation* and call each E_k an *explanation for* G. For instance, for the goal $G = \mathtt{hmm}([\mathtt{a},\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{a}])$ in the HMM program, we have six explanations shown in Fig. 3.

$$\begin{aligned}
E_1 = {} & \mathtt{m}(\mathtt{out}(\mathtt{s0}),\mathtt{a}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s0}),\mathtt{s0}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s0}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s0}),\mathtt{s0}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s0}),\mathtt{b}) \\
& \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s0}),\mathtt{s1}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s1}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s1}),\mathtt{s2}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s2}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s2}),\mathtt{s3}) \\
& \wedge \mathtt{m}(\mathtt{out}(\mathtt{s3}),\mathtt{a}) \\
E_2 = {} & \mathtt{m}(\mathtt{out}(\mathtt{s0}),\mathtt{a}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s0}),\mathtt{s0}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s0}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s0}),\mathtt{s1}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s1}),\mathtt{b}) \\
& \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s1}),\mathtt{s1}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s1}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s1}),\mathtt{s2}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s2}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s2}),\mathtt{s3}) \\
& \wedge \mathtt{m}(\mathtt{out}(\mathtt{s3}),\mathtt{a}) \\
& \vdots \\
E_6 = {} & \mathtt{m}(\mathtt{out}(\mathtt{s0}),\mathtt{a}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s0}),\mathtt{s1}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s1}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s1}),\mathtt{s2}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s2}),\mathtt{b}) \\
& \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s2}),\mathtt{s2}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s2}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s2}),\mathtt{s2}) \wedge \mathtt{m}(\mathtt{out}(\mathtt{s2}),\mathtt{b}) \wedge \mathtt{m}(\mathtt{tr}(\mathtt{s2}),\mathtt{s3}) \\
& \wedge \mathtt{m}(\mathtt{out}(\mathtt{s3}),\mathtt{a})
\end{aligned}$$

Fig. 3. Six explanations for $\mathtt{hmm}([\mathtt{a},\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{a}])$. Due to the space limit, the predicate name `msw` is abbreviated to `m`.

Intuitively finding explanations simulates the behavior of an HMM as a string analyzer, where each explanation corresponds to a state transition sequence. For example, E_1 indicates the transitions $\mathtt{s0} \rightarrow \mathtt{s0} \rightarrow \mathtt{s0} \rightarrow \mathtt{s1} \rightarrow \mathtt{s2} \rightarrow \mathtt{s3}$. It follows from the conditions on P_F of PRISM programs that the explanations

[11] As a result $P(\mathtt{msw}(i,v) \wedge \mathtt{msw}(i,v))$ is computed as $\{P(\mathtt{msw}(i,v))\}^2$ which makes the double occurrences of the same atom unequivalent to its single occurrence. In this sense, the current implementation of PRISM is not purely logical.

$$\begin{aligned}
\texttt{hmm}([\texttt{a},\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{hmm}(\texttt{0},\texttt{s0},[\texttt{a},\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{0},\texttt{s0},[\texttt{a},\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s0),a)} \ \wedge\ \texttt{m(tr(s0),s0)} \ \wedge\ \texttt{hmm}(\texttt{1},\texttt{s0},[\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) \\
&\quad \vee\ \texttt{m(out(s0),a)} \ \wedge\ \texttt{m(tr(s0),s1)} \ \wedge\ \texttt{hmm}(\texttt{1},\texttt{s1},[\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{1},\texttt{s0},[\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s0),b)} \ \wedge\ \texttt{m(tr(s0),s0)} \ \wedge\ \texttt{hmm}(\texttt{2},\texttt{s0},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) \\
&\quad \vee\ \texttt{m(out(s0),b)} \ \wedge\ \texttt{m(tr(s0),s1)} \ \wedge\ \texttt{hmm}(\texttt{2},\texttt{s1},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}])\ddagger \\
\texttt{hmm}(\texttt{2},\texttt{s0},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s0),b)} \ \wedge\ \texttt{m(tr(s0),s1)} \ \wedge\ \texttt{hmm}(\texttt{3},\texttt{s1},[\texttt{b},\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{1},\texttt{s1},[\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s1),b)} \ \wedge\ \texttt{m(tr(s1),s1)} \ \wedge\ \texttt{hmm}(\texttt{2},\texttt{s1},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}])\ddagger \\
&\quad \vee\ \texttt{m(out(s1),b)} \ \wedge\ \texttt{m(tr(s1),s2)} \ \wedge\ \texttt{hmm}(\texttt{2},\texttt{s2},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{2},\texttt{s1},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}])\dagger &\Leftrightarrow \texttt{m(out(s1),b)} \ \wedge\ \texttt{m(tr(s1),s1)} \ \wedge\ \texttt{hmm}(\texttt{3},\texttt{s1},[\texttt{b},\texttt{b},\texttt{a}]) \\
&\quad \vee\ \texttt{m(out(s1),b)} \ \wedge\ \texttt{m(tr(s1),s2)} \ \wedge\ \texttt{hmm}(\texttt{3},\texttt{s2},[\texttt{b},\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{3},\texttt{s1},[\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s1),b)} \ \wedge\ \texttt{m(tr(s1),s2)} \ \wedge\ \texttt{hmm}(\texttt{4},\texttt{s2},[\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{2},\texttt{s2},[\texttt{b},\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s2),b)} \ \wedge\ \texttt{m(tr(s2),s2)} \ \wedge\ \texttt{hmm}(\texttt{3},\texttt{s2},[\texttt{b},\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{3},\texttt{s2},[\texttt{b},\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s2),b)} \ \wedge\ \texttt{m(tr(s2),s2)} \ \wedge\ \texttt{hmm}(\texttt{4},\texttt{s2},[\texttt{b},\texttt{a}]) \\
\texttt{hmm}(\texttt{4},\texttt{s2},[\texttt{b},\texttt{a}]) &\Leftrightarrow \texttt{m(out(s2),b)} \ \wedge\ \texttt{m(tr(s2),s3)} \ \wedge\ \texttt{hmm}(\texttt{5},\texttt{s3},[\texttt{a}]) \\
\texttt{hmm}(\texttt{5},\texttt{s3},[\texttt{a}]) &\Leftrightarrow \texttt{m(out(s3),a)}
\end{aligned}$$

Fig. 4. Factorized explanations for $\texttt{hmm}([\texttt{a},\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}])$.

$E_1, \ldots, E_6$ are all exclusive to each other (i.e. they cannot be true at the same time), so we can compute the probability of G by $P_{DB}(G) = P_{DB}(\bigvee_{k=1}^{6} E_k) = \sum_{k=1}^{6} P_{DB}(E_k)$. This way of probability computation would be satisfactory if the number of explanations is relatively small, but in general it is intractable. In fact, for left-to-right HMMs, the number of possible explanations (state transitions) is ${}_{T-2}C_{N-2}$, where N is the number of states and T is the length of the input string.[12]

Efficient probabilistic inference by dynamic programming: We know that there exist efficient algorithms for probabilistic inference for HMMs which run in $O(T)$ time — forward (backward) probability computation, the Viterbi algorithm, the Baum-Welch algorithm [13]. Their common computing strategy is dynamic programming. That is, we divide a problem into sub-problems recursively with memoizing and reusing the solutions of the sub-problems which appear repeatedly. To realize such dynamic programming for PRISM programs, we adopt a two-staged procedure. In the first stage, we run tabled search to find

[12] This is because in each transition sequence, there are $(N-2)$ state changes in $(T-2)$ time steps since there are two constraints — each sequence should start from the initial state, and the final state should appear only once at the last of the sequence. For fully-connected HMMs, on the other hand, it is easily seen that the number of possible state transitions is $O(N^T)$.

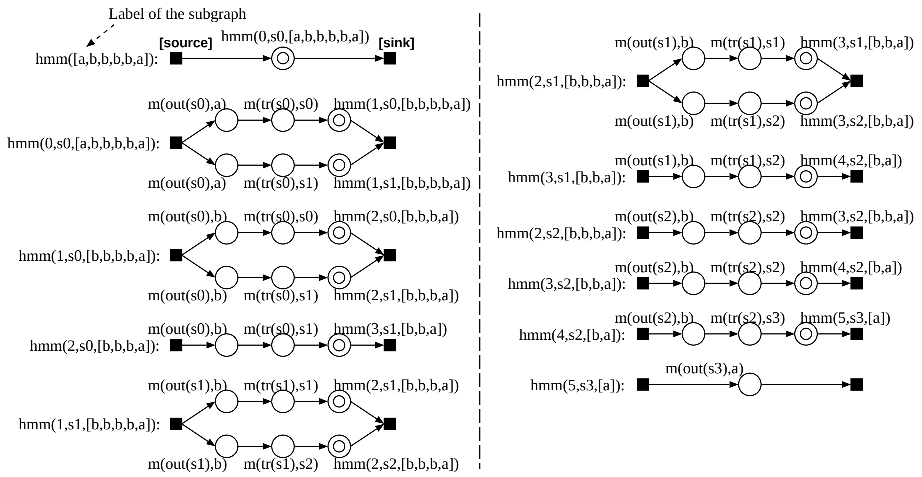

Fig. 5. Explanation graph

all explanations for an observation G, in which the solutions for a subgoal A are registered into a table so that they are reused for later occurrences of A. In the second stage, we compute probabilities while traversing an AND/OR graph called the *explanation graph for* G, extracted from the table constructed in the first stage.[13]

For instance from the HMM program, we can extract factorized iff formulas from the table as shown in Fig. 4 after the tabled search for $\texttt{hmm}([\texttt{a},\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}])$. Each iff formula takes the form $A \Leftrightarrow E'_1 \vee \cdots \vee E'_K$ where A is a subgoal (also called a *tabled atom*) and E'_k (called a *sub-explanation*) is a conjunction of subgoals and switches. These iff formulas are graphically represented as the explanation graph for $\texttt{hmm}([\texttt{a},\texttt{b},\texttt{b},\texttt{b},\texttt{b},\texttt{a}])$ as shown in Fig. 5

As illustrated in Fig. 4, in an explanation graph sub-structures are shared (e.g. a subgoal marked with † is referred to by two subgoals marked with ‡). Besides, it is reasonably expected that the iff formulas can be linearly ordered with respect to the caller-callee relationship in the program. These properties conjunctively enable us to compute probabilities in a dynamic programming fashion. For example, with an iff formula $A \Leftrightarrow E'_1 \vee \cdots \vee E'_K$ such that $E'_k = B_{k1} \wedge B_{k2} \wedge \cdots \wedge B_{kM_k}$, $P_{DB}(A)$ is computed as $\sum_{k=1}^{K} \prod_{j=1}^{M_k} P_{DB}(B_{kj})$ if $E'_1, \ldots, E'_6$ are exclusive and B_{kj} are independent. In the later section, we call $P_{DB}(A)$ the *inside probability* of A. The required time for computing $P_{DB}(A)$ is known to be linear in the size of the explanation graph, and in the case of the HMM program, we can see from Fig. 5 that it is $O(T)$, i.e. linear in the length of the

[13] Our approach is an instance of a general scheme called *PPC (propositionalized probability computation)* which computes the sum-product of probabilities via propositional formulas often represented as a graph. Minimal AND/OR graphs proposed in [26] are another example of PPC specialized for BNs.

input string. Recall that this is the same computation time as that of the forward (or backward) algorithm. Similar discussions can be made for the other types of probabilistic inference, and hence we can say that probabilistic inference with PRISM programs is as efficient as the ones by specific-purpose algorithms.

Assumptions for efficient probabilistic inference: Of course, our efficiency depends on several assumptions. We first assume that $obs(DB)$, a countable subset of ground atoms appearing in the clause head, is given as a set of observable atoms. In the HMM program, we may consider that $obs(DB) = \{\mathtt{hmm}([o_1, o_2, \ldots, o_T]) \mid o_t \in \{\mathtt{a}, \mathtt{b}\}, 1 \leq t \leq T, T \leq T_{\max}\}$ for some arbitrary finite $T_{\max}$. Then we roughly summarize the assumptions as follows (for the original definitions, please consult [6]):

Independence condition:
For any observable atom $G \in obs(DB)$, the atoms appearing in the sub-explanations in the explanation graph for G are all independent. In the current implementation of $\mathtt{msw}(i, v)$, this is unconditionally satisfied.

Exclusiveness condition:
For any observable atom $G \in obs(DB)$, the sub-explanations for each subgoal of G are exclusive to each other. The independence condition and the exclusiveness condition jointly make the sum-product computation of probabilities possible.

Finiteness condition:
For any observable atom $G \in obs(DB)$, the number of explanations for G is finite. Without this condition, probability computation could be infinite.

Uniqueness condition:
Observable atoms are exclusive to each other, and the sum of probabilities of all observable atoms is equal to unity (i.e. $\sum_{G \in obs(DB)} P_{DB}(G) = 1$). The uniqueness condition is important especially for EM learning in which the training data is given as a bag of atoms from $obs(DB)$ which are observed as true. That is, once we find G_t as true at t-th observation, we immediately know from the uniqueness condition that the atoms in $obs(DB)$ other than G_t are false, and hence in EM learning, we can ignore the statistics on the explanations for these false atoms. This property underlies a dynamic-programming-based EM algorithm in PRISM [6].

Acyclic condition:
For any observable atom $G \in obs(DB)$, there is no cycle with respect to the caller-callee relationship among the subgoals for G. The acyclicity condition makes dynamic programming possible.

It may look difficult to satisfy all the conditions listed above. However, if we keep in mind to write a terminating program that generates the observations (by chains of choices made by switches), with care for the exclusiveness among disjunctive paths, these conditions are likely to be satisfied. In fact not only popular *generative model*s such as HMMs, BNs and PCFGs but unfamiliar ones that have been little explored [27, 28] can naturally be written in this style.

Further issues: In spite of the general prospects of generative modeling, there are two cases where the uniqueness condition is violated and the PRISM's semantics is undefined. We call the first one "*probability-loss-to-infinity*," in which an infinite generation process occurs with a non-zero probability.[14] The second one is called "*probability-loss-to-failure*," in which there is a (finite) generation process with a non-zero probability that fails to yield any observable outcome. In Section 5, we discuss this issue, and describe a new learning algorithm that can deal with the second case.

Finally we address yet another view that takes explanation graphs as Boolean formulas consisting of ground atoms.[15] From this view, we can say that tabled search is a *propositionalization* procedure in the sense that it receives first-order expressions (a PRISM program) and an observation G as input, and generates as output a propositional AND/OR graph. In Section 6, we discuss advantages of such a propositionalization procedure.

3 Belief propagation

3.1 Belief propagation beyond HMMs

In this section, we show that PRISM can subsume BP (belief propagation). What we actually show is that BP is nothing but a special case of generalized IO (inside-outside) probability computation[16] in PRISM applied to a junction tree expressed logically [6, 10]. Symbolically we have

$$\text{BP} = \text{the generalized IO computation} + \text{junction tree}.$$

As far as we know this is the first link between BP and the IO algorithm. It looks like a mere variant of the well-known fact that BP applied to HMMs equals the forward-backward algorithm [31] but one should not miss the fundamental difference between HMMs and PCFGs.

Recall that an HMM deals with fixed-size sequences probabilistically generated from a finite state automaton, which is readily translated into a BN like the one in Fig. 6 where X_i's stand for hidden states and Y_i's stand for output symbols respectively. This translation is possible solely because an HMM

[14] The HMM program in Fig. 2 satisfies the uniqueness condition provided the probability of looping state transitions is less than one, since in that case the probability of all infinite sequence becomes zero.

[15] Precisely speaking, while switches must be ground, subgoals can be an existentially quantified atom other than a ground atom.

[16] Wang et al. recently proposed the generalized inside-outside algorithm for a language model that combines a PCFG, n-gram and PLSA (probabilistic latent semantic analysis) [29]. It extends the standard IO algorithm but seems an instance of the gEM algorithm used in PRISM [30]. In fact such combination can be straightforwardly implemented in PRISM by using appropriate `msw` switches. For example an event of a preterminal node A's deriving a word w with two preceding words (trigram), u and v, under the semantic content h is represented by `msw(`$[u,v,A,h]$`,`w`)`.

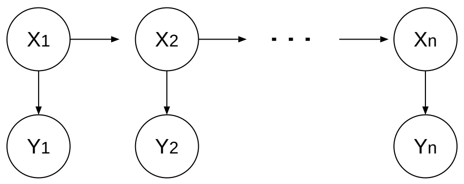

Fig. 6. BN for an HMM

has a finite number of states. Once HMMs are elevated to PCFGs however, a pushdown automaton for an underlying CFG has infinitely many stack states and it is by no means possible to represent them in terms of a BN which has only finitely many states. So any attempt to construct a BN that represents a PCFG and apply BP to it to upgrade the correspondence between BP and the forward-backward algorithm is doomed to fail. We cannot reach an algorithmic link between BNs and PCFGs this way.

We instead think of applying IO computation for PCFGs to BNs. Or more precisely we apply the *generalized IO computation*, a propositionally reformulated IO computation for PRISM programs [6], to a junction tree described logically as a PRISM program. Then we can prove that what the generalized IO computation does for the program is identical to what BP does on the junction tree [10], which we explain next.

3.2 Logical junction tree

Consider random variables $X_1, \ldots, X_N$ indexed by numbers $1, \ldots, N$. In the following we use Greek letters $\alpha, \beta, \ldots$ as a set of variable indices. Put $\alpha = \{n_1, \ldots, n_k\}$ ($\subseteq \{1, \ldots, N\}$). We denote by X_α the set (vector) of variables $\{X_{n_1}, \ldots, X_{n_k}\}$ and also by the lower case letter x_α the corresponding set (vector) of realizations of each variable in X_α.

A *junction tree* for a BN defining a distribution $P_{BN}(X_1 = x_1, \ldots, X_N = x_N) = \prod_{i=1}^{N} P_{BN}(X_i = x_i \mid X_{\pi(i)} = x_{\pi(i)})$, abbreviated to $P_{BN}(x_1, \ldots, x_N)$, where $\pi(i)$ denotes the indices of parent nodes of X_i, is a tree $T = (V, E)$ satisfying the following conditions [11, 12, 32].

- A node α ($\in V$) is a set of random variables X_α. An edge connecting X_α and X_β is labeled by $X_{\alpha\cap\beta}$. We use α instead of X_α etc to identify the node when the context is clear.
- A *potential* $\phi_\alpha(x_\alpha)$ is associated with each node α. It is a function consisting of a product of zero or more CPTs (conditional probability tables) like $\phi_\alpha(x_\alpha) = \prod_{\{j\}\cup\pi(j)\subseteq\alpha} P_{BN}(X_j = x_j \mid X_{\pi(j)} = x_{\pi(j)})$. It must hold that $\prod_{\alpha\in V} \phi_\alpha(x_\alpha) = P_{BN}(x_1, \ldots, x_N)$.
- RIP (*running intersection property*) holds which dictates that if nodes α and β have a common variable in the tree, it is contained in every node on the path between α and β.

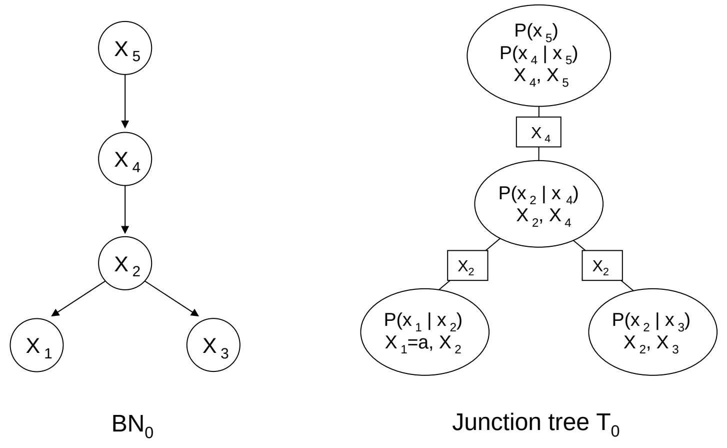

Fig. 7. Bayesian network BN_0 and a junction tree for BN_0

After introducing junction trees, we show how to encode a junction tree T in a PRISM program. Suppose a node α has K ($K \geq 0$) child nodes $\beta_1, \ldots, \beta_K$ and a potential $\phi_\alpha(x_\alpha)$. We use a *node atom* $nd_\alpha(X_\alpha)$ to assert that the node α is in a state X_α and δ to denote the root node of T. Introduce for every $\alpha \in V$ W_α (*weight clause for* α) and C_α (*node clause for* α), together with the *top clause* C_{top} by

$$\begin{array}{rrcl} W_\alpha : & weight_\alpha(X_\alpha) & \Leftarrow & \bigwedge_{P_{BN}(x_j \mid x_{\pi(j)}) \in \phi_\alpha} \texttt{msw}(\texttt{bn}(j, X_{\pi(j)}), X_j) \\ C_\alpha : & nd_\alpha(X_\alpha) & \Leftarrow & weight_\alpha(X_\alpha) \wedge nd_{\beta_1}(X_{\beta_1}) \wedge \cdots \wedge nd_{\beta_K}(X_{\beta_K}) \\ C_{top} : & top & \Leftarrow & nd_\delta(X_\delta). \end{array}$$

Here the X_j's denote logical variables, not random variables (we follow Prolog convention). W_α encodes ϕ_α as a conjunction of $\texttt{msw}(\texttt{bn}(j, X_{\pi(j)}), X_j)$s representing CPT $P_{BN}(X_j = x_j \mid X_{\pi(j)} = x_{\pi(j)})$. C_α is an encoding of the parent-child relation in T. C_{top} is a special clause to distinguish the root node δ in T.

Since we have $(\beta_i \setminus \alpha) \cap (\beta_j \setminus \alpha) = \phi$ if $i \neq j$ thanks to the RIP of T, we can rewrite $X_{(\bigcup_{i=1}^K \beta_i) \setminus \alpha}$, the set of variables appearing only in the right hand-side of C_α, to a disjoint union $\bigcup_{i=1}^K X_{\beta_i \setminus \alpha}$. Hence C_α is logically equivalent to

$$\begin{array}{l} nd_\alpha(X_\alpha) \Leftarrow \\ \quad weight_\alpha(X_\alpha) \wedge \exists X_{\beta_1 \setminus \alpha} nd_{\beta_1}(X_{\beta_1}) \wedge \cdots \wedge \exists X_{\beta_K \setminus \alpha} nd_{\beta_K}(X_{\beta_K}). \end{array} \quad (1)$$

Let F_T be the set of all ground msw atoms of the form $\texttt{msw}(\texttt{bn}(i, x_{\pi(i)}), x_i)$. Give a joint distribution $P_{F_T}(\cdot)$ over F_T so that $P_{F_T}(\texttt{msw}(\texttt{bn}(i, x_{\pi(i)}), x_i))$, the probability of $\texttt{msw}(\texttt{bn}(i, x_{\pi(i)}), x_i)$ being true, is equal to $P_{BN}(X_i = x_i \mid X_{\pi(i)} = x_{\pi(i)})$.

Sampling from $P_{F_T}(\cdot)$ is equivalent to simultaneous independent sampling from every $P_{BN}(X_i = x_i \mid X_{\pi(i)} = x_{\pi(i)})$ given i and $x_{\pi(i)}$. It yields a set S of true msw atoms. We can prove however that S uniquely includes a subset $S_{BN} = \{\mathtt{msw}(\mathtt{bn}(i, x_{\pi(i)}), x_i) \mid 1 \leq i \leq N\}$ that corresponds to a draw from the joint distribution $P_{BN}(\cdot)$.

Finally define a program DB_T describing the junction tree T as a union $F_T \cup R_T$ having the base distribution $P_{F_T}(\cdot)$:

$$DB_T = F_T \cup R_T \quad \text{where } R_T = \{W_\alpha, C_\alpha \mid \alpha \in V\} \cup \{C_{top}\}. \tag{2}$$

Consider the Bayesian network BN_0 and a junction tree T_0 for BN_0 shown in Fig. 7. Suppose evidence $X_1 = a$ is given. Then R_{T_0} contains node clauses listed in Fig. 8.

$$R_{T_0} \begin{cases} top \Leftarrow nd_{\{4,5\}}(X_4, X_5) \\ nd_{\{4,5\}}(X_4, X_5) \Leftarrow \mathtt{msw}(\mathtt{bn}(5, []), X_5) \wedge \mathtt{msw}(\mathtt{bn}(4, [X_5]), X_4) \wedge nd_{\{2,4\}}(X_2, X_4) \\ nd_{\{2,4\}}(X_2, X_4) \Leftarrow \mathtt{msw}(\mathtt{bn}(2, [X_4]), X_2) \wedge nd_{\{1,2\}}(X_1, X_2) \wedge nd_{\{2,3\}}(X_2, X_3) \\ nd_{\{2,3\}}(X_2, X_3) \Leftarrow \mathtt{msw}(\mathtt{bn}(3, [X_2]), X_3) \\ nd_{\{1,2\}}(X_1, X_2) \Leftarrow \mathtt{msw}(\mathtt{bn}(1, [X_2]), X_1) \wedge X_1 = a \end{cases}$$

Fig. 8. Logical description of T_0.

3.3 Computing generalized inside-outside probabilities

After defining DB_T which is a logical encoding of a junction tree T, we demonstrate that the generalized IO probability computation for DB_T with a goal *top* by PRISM coincides with BP on T. Let $P_{DB_T}(\cdot)$ be the joint distribution defined by DB_T. According to the distribution semantics of PRISM, it holds that for any ground instantiations x_α of X_α,

$$\begin{aligned} P_{DB_T}(weight_\alpha(x_\alpha)) &= P_{DB_T}\left(\textstyle\bigwedge_{P_{BN}(x_j \mid x_{\pi(j)}) \in \phi_\alpha} \mathtt{msw}(\mathtt{bn}(j, x_{\pi(j)}), x_j)\right) && (3) \\ &= \textstyle\prod_{P_{BN}(x_j \mid x_{\pi(j)}) \in \phi_\alpha} P_F(\mathtt{msw}(\mathtt{bn}(j, x_{\pi(j)}), x_j)) && (4) \\ &= \textstyle\prod_{P_{BN}(x_j \mid x_{\pi(j)}) \in \phi_\alpha} P_{BN}(x_j \mid x_{\pi(j)}) \\ &= \phi_\alpha(x_\alpha). \end{aligned}$$

In the above transformation, (3) is rewritten to (4) using the fact that conditional distributions $\{P_{F_T}(x_j \mid x_{\pi(j)})\}$ contained in ϕ_α are pairwise different and $\mathtt{msw}(\mathtt{bn}(j, x_{\pi(j)}), x_j))$ and $\mathtt{msw}(\mathtt{bn}(j', x_{\pi(j')}), x_{j'}))$ are independent unless $j = j'$.

We now transform $P_{DB_T}(nd_\alpha(x_\alpha))$. Using (1), we see it holds that between the node α and its children $\beta_1, \ldots, \beta_K$

$$\begin{aligned}
&P_{DB_T}(nd_\alpha(x_\alpha)) \\
&= \phi_\alpha(x_\alpha) \cdot P_{DB_T}\left(\textstyle\bigwedge_{i=1}^K \exists x_{\beta_i \setminus \alpha} nd_{\beta_i}(x_{\beta_i})\right) \\
&= \phi_\alpha(x_\alpha) \cdot \textstyle\prod_{i=1}^K P_{DB_T}\left(\exists x_{\beta_i \setminus \alpha} nd_{\beta_i}(x_{\beta_i})\right) && (5) \\
&= \phi_\alpha(x_\alpha) \cdot \textstyle\prod_{i=1}^K \sum_{x_{\beta_i \setminus \alpha}} P_{DB_T}(nd_{\beta_i}(x_{\beta_i})). && (6)
\end{aligned}$$

The passage from (5) to (6) is justified by Lemma 3.3 in [10]. We also have

$$P_{DB_T}(top) = 1 \qquad (7)$$

because *top* is provable from $S \cup R_T$ where S is an arbitrary sample obtained from P_{F_T}.

Let us recall that in PRISM, two types of probability are defined for a ground atom A. The first one is $inside(A)$, the *generalized inside probability* of A, defined by $inside(A) \overset{\text{def}}{=} P_{DB}(A)$. It is just the probability of A being true. (6) and (7) give us a set of recursive equations to compute generalized inside probabilities of $nd_\alpha(x_\alpha)$ and *top* in a bottom-up manner [6].

The second one is the *generalized outside probability* of A with respect to a goal G, denoted by $outside(G\ ;\ A)$, which is more complicated than $inside(A)$, but can also be recursively computed in a top-down manner using generalized inside probabilities of other atoms [6]. We here simply list the set of equations to compute generalized outside probabilities with respect to *top* in DB_T. We describe the recursive equation between a child node β_j and its parent node α in (9).

$$\begin{aligned}
&outside(top\ ;\ top) = 1 && (8) \\
&outside(top\ ;\ nd_{\beta_j}(x_{\beta_j})) \\
&= \sum_{x_{\alpha \setminus \beta_j}} \left(\phi_\alpha(x_\alpha) \cdot outside(top\ ; nd_\alpha(x_\alpha)) \prod_{i \neq j}^{K} \sum_{x_{\beta_i \setminus \alpha}} inside(nd_{\beta_i}(x_{\beta_i})) \right). && (9)
\end{aligned}$$

3.4 Marginal distribution

The important property of generalized inside and outside probabilities is that their product, $inside(A) \cdot outside(G\ ;\ A)$, gives the expected number of occurrences of A in a(-n SLD) proof of G from `msw`s drawn from P_F. In our case where $A = nd_\alpha(x_\alpha)$ and $G = top$, each node atom occurs at most once in an SLD proof of *top*. Hence $inside(A) \cdot outside(top\ ;\ A)$ is equal to the probability that *top* has a(-n SLD) proof containing A from $R_T \cup S$ where S is a sample from P_{F_T}. In this proof, not all members of S are relevant but only a subset $S_{BN}(x_1, \ldots, x_N) = \{\mathtt{msw}(\mathtt{bn}(i, x_{\pi(i)}), x_i) \mid 1 \leq i \leq N\}$ which corresponds to a sample $(x_1, \ldots, x_N)$ from P_{BN} (and vice versa) is relevant. By analyzing the proof, we know that sampled values x_α of X_α appearing in S_{BN} is identical to those in $A = nd_\alpha(x_\alpha)$ in the proof. We therefore have

$$\begin{aligned}
& \mathit{inside}(A) \cdot \mathit{outside}(\mathit{top}\ ;\ A) \\
& = \text{Probability of sampling } S_{BN}(x_1, \ldots, x_N) \text{ from } P_{F_T} \text{ such that} \\
& \qquad S_{BN}(x_1, \ldots, x_N) \cup R_T \vdash A \\
& = \text{Probability of sampling } (x_1, \ldots, x_N) \text{ from } P_{BN}(\cdot) \text{ such that} \\
& \qquad S_{BN}(x_1, \ldots, x_N) \cup R_T \vdash A \\
& = \text{Probability of sampling } (x_1, \ldots, x_N) \text{ from } P_{BN}(\cdot) \text{ such that} \\
& \qquad (x_1, \ldots, x_N)_\alpha = x_\alpha \text{ in } A = nd_\alpha(x_\alpha) \\
& = \text{Probability of sampling } x_\alpha \text{ from } P_{BN}(\cdot) \\
& = P_{BN}(x_\alpha).
\end{aligned}$$

When some of the X_i's in the BN are observed and fixed as evidence e, a slight modification of the above derivation gives $\mathit{inside}(A) \cdot \mathit{outside}(\mathit{top}\ ;\ A) = P_{BN}(x_\alpha, e)$. We summarize this result as a theorem.

Theorem 1. *Suppose DB_T is the PRISM program describing a junction tree T for a given BN. Let e be evidence. Also let α be a node in T and $A = nd_\alpha(x_\alpha)$ an atom describing the state of the node α. We then have*

$$\mathit{inside}(A) \cdot \mathit{outside}(\mathit{top}\ ;\ A) = P_{BN}(x_\alpha, e).$$

3.5 Deriving BP messages

We now introduce *message*s which represent messages in BP value-wise in terms of inside-outside probabilities of ground node atoms. Let nodes $\beta_1, \ldots, \beta_K$ be the child nodes of α as before and γ be the parent node of α in the junction tree T. Define a *child-to-parent message* by

$$\begin{aligned}
\mathit{msg}_{\alpha \triangleright \gamma}(x_{\alpha \cap \gamma}) &\stackrel{\text{def}}{=} \textstyle\sum_{x_{\alpha \setminus \gamma}} \mathit{inside}(nd_\alpha(x_\alpha)) \\
&= \textstyle\sum_{x_{\alpha \setminus \gamma}} P_{DB_T}(nd_\alpha(x_\alpha)).
\end{aligned}$$

The equation (6) for generalized inside probability is rewritten in terms of child-to-parent message as follows.

$$\begin{aligned}
\mathit{msg}_{\alpha \triangleright \gamma}(x_{\alpha \cap \gamma}) &= \textstyle\sum_{x_{\alpha \setminus \gamma}} P_{DB_T}(nd_\alpha(x_\alpha)) \\
&= \textstyle\sum_{x_{\alpha \setminus \gamma}} \left(\phi_\alpha(x_\alpha) \cdot \prod_{i=1}^{K} \mathit{msg}_{\beta_i \triangleright \alpha}(x_{\beta_i \cap \alpha}) \right).
\end{aligned} \tag{10}$$

Next define a *parent-to-child message* from the parent node α to the j-th child node β_j by

$$\mathit{msg}_{\alpha \triangleright \beta_j}(x_{\alpha \cap \beta_j}) \stackrel{\text{def}}{=} \mathit{outside}(\mathit{top}\ ;\ nd_{\beta_j}(x_{\beta_j})).$$

Note that $\mathit{outside}(\mathit{top}\ ;\ \mathit{nd}_{\beta_j}(x_{\beta_j}))$ looks like a function of x_{β_j}, but in reality it is a function of x_{β_j}'s subset, $x_{\alpha\cap\beta_j}$ by (9). Using parent-to-child messages, we rewrite the equation (9) for generalized outside probability as follows.

$$\begin{aligned}
& \mathit{msg}_{\alpha\triangleright\beta_j}(x_{\alpha\cap\beta_j}) \\
& = \mathit{outside}(\mathit{top}\ ;\ \mathit{nd}_{\beta_j}(x_{\beta_j})) \\
& = \sum_{x_{\alpha\setminus\beta_j}} \left(\phi_\alpha(x_\alpha) \cdot \mathit{msg}_{\gamma\triangleright\alpha}(x_{\gamma\cap\alpha}) \prod_{i\neq j}^{K} \mathit{msg}_{\beta_i\triangleright\alpha}(x_{\beta_i\cap\alpha}) \right) .
\end{aligned} \tag{11}$$

When α is the root node δ in T, since $\mathit{outside}(\mathit{top}\ ;\ \mathit{top}) = 1$, we have

$$\mathit{msg}_{\delta\triangleright\beta_j}(x_{\delta\cap\beta_j}) = \sum_{x_{\delta\setminus\beta_j}} \left(\phi_\delta(x_\delta) \cdot \prod_{i\neq j}^{K} \mathit{msg}_{\beta_i\triangleright\alpha}(x_{\beta_i\cap\alpha}) \right) . \tag{12}$$

The equations (10), (11) and (12) are identical to the equations for BP on T (without normalization) where (10) specifies the collecting evidence step and (11) and (12) specify the distributing evidence step respectively [12, 11, 33]. So we conclude

Theorem 2. *The generalized IO probability computation for DB_T describing a junction tree T with respect to the top goal top by (2) coincides with BP on T.*

Let us compute some messages in the case of the program in Fig. 8.

$$\begin{aligned}
\mathit{msg}_{\{4,5\}\triangleright\{2,4\}}(X_4) &= \mathit{outside}(\mathit{top}\ ;\ \mathit{nd}(X_2, X_4)) \\
&= \textstyle\sum_{X_5} P_{DB_{T_0}}(\mathit{nd}(X_4, X_5)) \\
&= \textstyle\sum_{X_5} P_{BN_0}(X_4 \mid X_5) P_{BN_0}(X_5) \\
&= P_{BN_0}(X_4) \\
\mathit{msg}_{\{1,2\}\triangleright\{2,4\}}(X_2) &= \mathit{inside}(\mathit{nd}(X_1 = a, X_2)) \\
&= P_{DB_{T_0}}(\mathit{nd}(X_1 = a, X_2)) \\
&= P_{BN_0}(X_1 = a \mid X_2) \\
\mathit{msg}_{\{2,3\}\triangleright\{2,4\}}(X_2) &= \textstyle\sum_{X_3} \mathit{inside}(\mathit{nd}(X_2, X_3)) \\
&= \textstyle\sum_{X_3} P_{DB_{T_0}}(\mathit{nd}(X_2, X_3)) \\
&= \textstyle\sum_{X_3} P_{BN_0}(X_3 \mid X_2) \\
&= 1
\end{aligned}$$

Using these messages we can confirm Theorem 1.

$$\begin{aligned}
& \mathit{inside}(\mathit{nd}(X_2, X_4)) \cdot \mathit{outside}(\mathit{top}\ ;\ \mathit{nd}(X_2, X_4)) \\
& = P_{BN_0}(X_2 \mid X_4) \cdot \mathit{msg}_{\{4,5\}\triangleright\{2,4\}}(X_4) \cdot \mathit{msg}_{\{1,2\}\triangleright\{2,4\}}(X_2) \cdot \mathit{msg}_{\{2,3\}\triangleright\{2,4\}}(X_2) \\
& = P_{BN_0}(X_2 \mid X_4) \cdot P_{BN_0}(X_4) \cdot P_{BN_0}(X_1 = a \mid X_2) \cdot 1 \\
& = P_{BN_0}(X_1 = a, X_2, X_4).
\end{aligned}$$

Theorem 2 implies that computation of generalized IO probabilities in PRISM can be used to implement the junction tree algorithm. If we do so, we will first unfold a junction tree to a boolean combination of ground node atoms and `msw` atoms which propositionally represents the junction tree and then compute required probabilities from that boolean combination. This approach is a kind of *propositionalized BN computation*, a recent trend in the Bayesian network computation [17, 26, 34, 35] in which BNs are propositionally represented and computed. We implemented the junction tree algorithm based on generalized IO probability computation. We compare the performance of our implementation with ACE, one of the propositionalized BN computation systems in the next section.

4 Performance data

In this section we compare the computing performance of PRISM with two recent systems, Dyna [15] and ACE [16–18]. Unlike PRISM, they are not a general-purpose programming system that came out of research in PLL (probabilistic logic learning) or SRL (statistical relational learning). Quite contrary they are developed with their own purpose in a specific domain, i.e. statistical NLP (natural language processing) in the case of Dyna and fast probabilistic inference for BNs in the case of ACE. So our comparison can be seen as one between a general-purpose system and a specific-purpose system. We first measure the speed of probabilistic inference for a PCFG by PRISM and compare it with Dyna.

4.1 Computing performance with PCFGs

Dyna system: Dyna[17] is a high-level declarative language for probabilistic modeling in NLP [15]. The primary purpose of Dyna is to facilitate various types of dynamic programming found in statistical NLP such as IO probability computation and Viterbi parsing to name a few. It is similar to PRISM in the sense that both can be considered as a probabilistic extension of a logic programming language but PRISM takes a top-down approach while Dyna takes a bottom-up approach in their evaluation. Also implementations differ. Dyna programs are compiled to C++ code (and then to native code[18]) while PRISM programs are compiled to byte code for B-Prolog.[19] We use in our experiment PRISM version 1.10 and Dyna version 0.3.9 on a PC having Intel Core 2 Duo (2.4GHz) and 4GB RAM. The operating system is 64-bit SUSE Linux 10.0.

[17] `http://www.dyna.org/`

[18] In this comparison, we added `--optimize` option to the `dynac` command, a batch command for compiling Dyna programs into native code.

[19] `http://www.probp.com/`

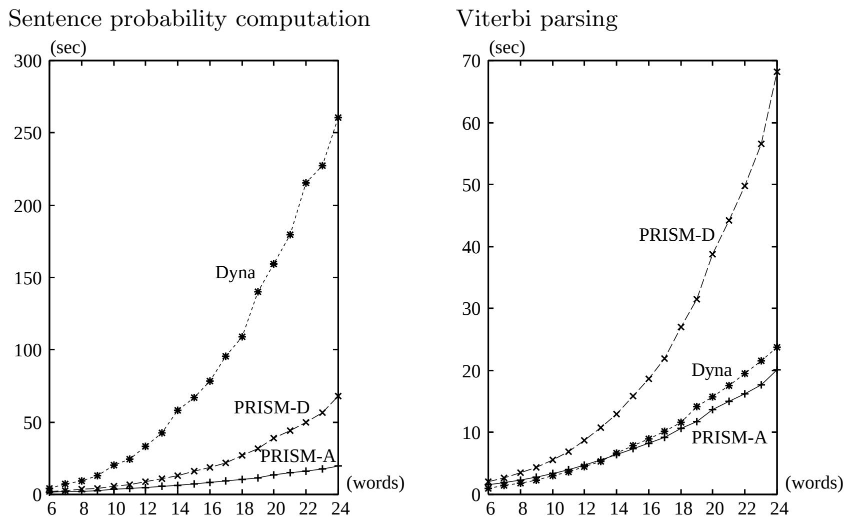

Fig. 9. Comparison between PRISM and Dyna on the speed of PCFG-related inference tasks

Computing sentence probability and Viterbi parsing: To compare the speed of various PCFG-related inference tasks, we have to prepare benchmark data, i.e. a set of sentences and a CFG (so-called a tree-bank grammar [36]) for them. We use the WSJ portion of Penn Treebank III[20] which is widely recognized as one of the standard corpora in the NLP community [37].

We converted all 49,208 raw sentences in the WSJ sections 00–24 to POS (part of speech) tag sequences (the average length is about 20.97) and at the same time extract 11,534 CFG rules from the labeled trees. These CFG rules are further converted to Chomsky normal form[21] to yield 195,744 rules with 5,607 nonterminals and 38 terminals (POS tags). Finally by giving uniform probabilities to CFG rules, we obtain a PCFG we use in the comparison.

Two types of PRISM program are examined for the derived PCFG. In the former, which is referred to as PRISM-A here, an input POS tag sequence $t_1, \ldots, t_N$ is converted into a set of ground atoms $\{\texttt{input}(0, t_1, 1), \ldots, \texttt{input}(N-1, t_N, N)\}$ and supplied (by the "`assert`" predicate) to the program. Each $\texttt{input}(d-1, t, d)$ means that the input sentence has a POS tag t is at position d ($1 \leq d \leq N$). The latter type, referred to as PRISM-D, is a probabilistic version of definite clause grammars which use difference lists. Dyna is closer to PRISM-A since in Dyna, we first provide the items equivalent to the above ground atoms to the chart (the inference mechanism of Dyna is based on a bottom-up chart parser). Compared to PRISM-A, PRISM-D incurs computational overhead due to the use of difference list.

[20] `http://www.ldc.upenn.edu/`

[21] We used the Dyna version of the CKY algorithm presented in [15].

We first compared time for computing the probability of a sentence (POS tag sequence) w.r.t. the derived PCFG using PRISM and Dyna. The result is plotted in Fig. 9 (left). Here X-axis shows sentence length (up to 24 by memory limitation of PRISM) and Y-axis shows the average time for probability computation[22] of randomly picked up 10 sentences. The graph clearly shows PRISM runs faster than Dyna. Actually at length 21 (closest to the average length), PRISM-A runs more than 10 times faster than Dyna.

We also conducted a similar experiment on Viterbi parsing, i.e. obtaining the most probable parse w.r.t. the derived CFG. Fig. 9 (right) show the result, where X-axis is the sentence length and Y-axis is the average time for Viterbi parsing of randomly picked up 10 sentences. This time PRISM-A is slightly slower than Dyna until length 13 but after that PRISM-A becomes faster than Dyna and the speed gap seems steadily growing.

One may notice that Dyna has a significant speed-up in Viterbi parsing compared to sentence probability computation while in PRISM the computation time remains the same between these two inference tasks. This is because, in Viterbi parsing, Dyna performs a best-first search which utilizes a priority-queue agenda. On the other hand, the difference between PRISM-A and Dyna in sentence probability computation indicates the efficiency of PRISM's basic search engine. Besides, not surprisingly, PRISM-D runs three times slower than PRISM-A at the average sentence length.

Thus in our experiment with a realistic PCFG, PRISM, a general-purpose programming system, runs faster than or competitively with Dyna, a specialized system for statistical NLP. We feel this is somewhat remarkable. At the same time though, we observed PRISM's huge memory consumption which might be a severe problem in the future.[23]

4.2 Computing performance with BNs

Next we measure the speed of a single marginal probability computation in BNs by PRISM programs DB_T described in Section 3.2 (hereafter called junction-tree PRISM programs) and compare it with ACE [16–18].

ACE[24] is a software package to perform probabilistic inference in a BN in three steps. It first encodes the BN by CNF propositionally, then transforms the CNF to yet another form d-DNNF (deterministic, decomposable negation normal form) and finally extracts from the d-DNNF, a graph called AC (arithmetic circuit). An AC is a directed acyclic graph in which nodes represent arithmetic operations such as addition and multiplication. The extracted AC is used to

[22] In the case of PRISM, the total computation time is the sum of the time for constructing the explanation graph and the time for computing the inside probability.

[23] In an additional experiment using another computer with 16GB memory, we could compute sentence probability for all of randomly picked up 10 sentences of length 43, but the sentences longer than 43 causes thrashing. It should be noted, however, that the PRISM system did not crash as far as we observed.

[24] `http://reasoning.cs.ucla.edu/ace/`

Table 1. Comparison between PRISM and ACE on the average computation time [sec] for single marginal probabilities.

		Junction-tree PRISM			ACE		
Network	#Nodes	Trans.	Cmpl.	Run	Cmpl.	Read	Run
Asia	8	0.1	0.03	0	0.53	0.14	0.02
Water	32	1.57	1.79	0.38	1.87	0.5	0.8
Mildew	35	6.48	11.2	12.5	4.27	1.95	3.42
Alarm	37	0.34	0.2	0.01	0.33	0.18	0.13
Pigs	441	2.52	9.38	5.38	2.48	0.57	1.95
Munin1	189	1.93	2.54	N/A (1)	2242	0.51	N/A (2)
Munin2	1,003	7.61	85.7	15	9.94	1.0	6.07
Munin3	1,044	8.82	70.3	15.9	8.12	0.95	4.11
Munin4	1,041	8.35	90.7	408	11.2	0.97	6.64

compute the marginal probability, given evidence. In compilation, we can make the resulting Boolean formulas more compact than a junction tree by exploiting the information in the local CPTs such as determinism (zero probabilities) and parameter equality, and more generally, CSI (context-specific independence) [38].

We picked up benchmark data from Bayesian Network Repository (`http://www.cs.huji.ac.il/labs/compbio/Repository/`). The network size ranges from 8 nodes to 1,044 nodes. In this experiment, PRISM version 1.11 and ACE 2.0 are used and run by a PC having AMD Opteron254(2.8GHz) with 16GB RAM on SUSE 64bit Linux 10.0. We implemented a Java translator from an XMLBIF network specification to the corresponding junction-tree PRISM program.[25] For ACE, we used the default compilation option (and the most advanced) `-cd06` which enables the method defined in [18].

Table 1 shows time for computing a single marginal $P(X_i|e)$.[26] For junction-tree PRISM, the columns "Trans.," "Cmpl.," and "Run" mean respectively the translation time from XMLBIF to junction-tree PRISM, the compilation time from junction-tree PRISM to byte code of B-Prolog and the inference time. For ACE, the column "Cmpl." is compile time from an XBIF file to an AC wheres the column "Read" indicates time to load the compiled AC onto memory. The column "Run" shows the inference time. N/A (1) and N/A (2) in Munin1 means PRISM ran out of memory and ACE stopped by run time error respectively.

When the network size is small, there are cases where PRISM runs faster than ACE, but in general PRISM does not catch up to ACE. One of the possible reasons might be that while ACE thoroughly exploits CSI in a BN to optimize computation, PRISM performs no such optimization when it propositionalizes

[25] In constructing a junction-tree, the elimination order basically follows the one specified in a `*.num` file in the repository. If such a `*.num` file does not exist, we used MDO (minimally deficient order). In this process, entries with 0 probabilities in CPTs are eliminated to shrink CPT size.

[26] For ACE, the computation time for $P(X_i \mid e)$ is averaged on all variables in the network. For junction-tree PRISM, since it sometimes took too long a time, the computation time is averaged on more than 50 variables for large networks.

a junction tree to an explanation graph. As a result, explanation graphs used by PRISM are thought to have much more redundancy than AC used by ACE. Translation to an optimized version of junction-tree PRISM programs using CSI remains as future work.

The results of two benchmark tests in this section show that PRISM is now maturing and *reasonably competitive* with state-of-the-art implementations of existing models in terms of computing performance, considering PRISM is a general-purpose programming language using general data structure – first order terms. We next look into another aspect of PRISM, PRISM as a vehicle for exploring new models.

5 Generative modeling with constraints

5.1 Loss of probability mass

Probabilistic modeling in PRISM is generative. By generative we mean that a probabilistic model describes a sequential process of generating an outcome in which nondeterminacy is resolved by a probabilistic choice (using `msw/2` in the case of PRISM). All of BNs, HMMs and PCFGs are generative in this sense.

A generative model is described by a *generation tree* such that each node represents some state in a generation process. If there are k possible choices at node N and if the i-th choice with probability $p_i > 0$ ($1 \leq i \leq k$, $\sum_{i=1}^{k} p_i = 1$) causes a state transition from N to a next state N'_i, there is an edge from N to N'_i. Note that we neither assume the transition is always successful nor the tree is finite. A leaf node is one where an outcome o is obtained. $P(o)$, the probability of the outcome o is that of an occurrence of a path from the root node to a leaf generating o. The generation tree defines a distribution over possible outcomes *if* $\sum_{o \in obs(DB)} P(o) = 1$ (*tightness condition* [39]).[27]

Generative models are intuitive and popular but care needs to be taken to ensure the tightness condition.[28] There are two cases in which the danger of violating the tightness condition exists. The first case is *probability-loss-to-infinity*. It means infinite computations occur *and* the probability mass assigned to them is non-zero.[29] In fact this happens in PCFGs depending on parameters associated with CFG rules in a grammar [39, 40].

The second case is *probability-loss-to-failure* which occurs when a transition to the next state fails. Since the probability mass put on a choice causing the transition is lost without producing any outcome, the total sum of probability mass on all outcomes becomes less than one. Probability-loss-to-failure does not

[27] The tightness condition is part of the *uniqueness condition* introduced in Section 2 [6].

[28] We call distributions *improper*if they do not satisfy the tightness condition.

[29] Recall that mere existence of infinite computation does not necessarily violate the tightness condition. Take a PCFG $\{p : S \to a, q : S \to SS\}$ where S is a start symbol, a a terminal symbol, $p + q = 1$ and $p, q \geq 0$. If $q > 0$, infinite computations occur, but the probability mass assigned to them is 0 as long as $q \leq p$.

happen in BNs, HMMs or PCFGs but can happen in more complex modeling such as probabilistic unification grammars [41, 42]. It is a serious problem because it prevents us from using constraints in complex probabilistic modeling, the inherent target of PRISM. In the sequel, we detail our approach to generative modeling with constraints.

5.2 Constraints and improper distributions

We first briefly explain constraints. When a computational problem is getting more and more complex, it is less and less possible to completely specify every bit of information flow to solve it. Instead we specify conditions an answer must meet. They are called *constraints*. Usually constraints are binary relations over variables such as equality, ordering, set inclusion and so on. Each of them is just a partial declarative characterization of an answer but their interaction creates information flow to keep global consistency.

Constraints are mixed and manipulated in a constraint handling process, giving the simplest form as an answer just like a polynomial equation is solved by reduction to the simplest (but equivalent) form. Sometimes we find that constraints are inconsistent and there is no answer. When this happens, we stop the constraint handling process with failure.

Allowing the use of constraints significantly enhances modeling flexibility as exemplified by unification grammars such as HPSGs [43]. Moreover since they are declarative, they are easy to understand and easy to maintain. The other side of the coin however is that they can be a source of probability-loss-to-failure when introduced to generative modeling as they are not necessarily satisfiable. The loss of probability mass implies $\sum_{o \in obs(DB)} P(o) < 1$, an improper distribution, and ignoring this fact would destroy the mathematical meaning of computed results.

5.3 Conditional distributions and their EM learning

How can we deal with such improper distributions caused by probability-loss-to-failure? It is apparent that what we can observe is an outcome o of some successful generation process specified by our model and thus should be interpreted as an realization of a conditional distribution, $P(o \mid \texttt{success})$ where `success` denotes the event of generating some outcome.

Let us put this argument in the context of distribution semantics and let $q(\cdot)$ be a target predicate in a program $DB = F \cup R$ defining the distribution $P_{DB}(\cdot)$. Clauses in R may contain constraints such as $X < Y$ in their body. The event `success` is representable as $\exists X \, q(X)$ because the latter says there exists some outcome. Our conditional distribution is therefore written as $P_{DB}(q(t) \mid \exists X \, q(X))$. If R in DB satisfies the modeling principles stated in Subsection 5.1, which we assume hereafter, it holds that $P_{DB}(\exists X \, q(X)) = \sum_{q(t) \in obs(DB)} P_{DB}(q(t))$ where $obs(DB)$ is the set of ground target atoms provable from DB, i.e. $obs(DB) = \{q(t) \mid DB \vdash q(t)\}$. Consequently we have

$$P_{DB}(q(t) \mid \texttt{success}) = \frac{P_{DB}(q(t))}{P_{DB}(\exists X\, q(X))} = \frac{P_{DB}(q(t))}{\sum_{q(t') \in obs(DB)} P_{DB}(q(t'))} \ .$$

So if there occurs probability-loss-to-failure, what we should do is to normalize $P_{DB}(\cdot)$ by computing $P_{DB}(\texttt{success}) = \sum_{q(t') \in obs(DB)} P_{DB}(q(t'))$. The problem is that this normalization is almost always impossible. First of all there is no way to compute $P_{DB}(\texttt{success})$, the normalizing constant, if $obs(DB)$ is infinite. Second even if $obs(DB)$ is finite, the computation is often infeasible since there are usually exponentially many observable outcomes and hence so many times of summation is required.

We therefore abandon unconditional use of constraints and use them only when $obs(DB)$ is finite and an efficient computation of $P_{DB}(\texttt{success})$ is possible by dynamic programming. Still, there remains a problem. The *gEM (graphical EM)* algorithm, the central algorithm in PRISM for EM learning by dynamic programming, is not applicable if failure occurs because it assumes the tightness condition. We get around this difficulty by merging it with the FAM algorithm (failure-adjusted maximization) proposed by Cussens [7]. The latter is an EM algorithm taking failure into account.[30].

Fortunately the difference between the gEM algorithm and the FAM algorithm is merely that the latter additionally computes expected counts of `msw` atoms in a failed computation of $q(X)$ $(\exists X\, q(X))$. It is therefore straightforward to augment the gEM algorithm with a routine to compute the required expected counts in a failed computation, *assuming a failure program is available* which defines `failure` predicate that represents all failed computations of $q(X)$ w.r.t. DB. The augmented algorithm, *the fgEM algorithm* [8], works as the FAM algorithm with dynamic programming and implemented in the current PRISM.

So the last barrier against the use of constraints is the construction of a failure program. Failure must be somehow "reified" as a failure program for dynamic programming to be applicable. However how to construct it is not self-evident because there is no mechanism of recording failure in the original program DB. We here apply FOC (*first order compiler*), a program synthesis algorithm based on deductive program transformation [44]. It can derive, though not always, automatically a failure program from the source program DB for the target predicate $q(X)$ [8]. Since the synthesized failure program is a usual PRISM program, $P_{DB}(\texttt{failure})$ is computed efficiently by dynamic programming.

In summary, in our approach, generative modeling with constraints is possible with the help of the fgEM algorithm and FOC, provided that a failure program

[30] The FAM algorithm [7] assumes there occur failed computations before an outcome is successfully generated. It requires to count the number of occurrences of each `msw` atom in the failed computation paths but [7] does not give how to count them. Usually there are exponentially many failure paths and naive computation would take exponential time. We solved this problem by merging FAM with gEM's dynamic programming.

is successfully synthesized by FOC and the computation tree (SLD tree) for `failure` is finite (and not too large). We next look at some examples.

5.4 Agreement in number

A small example: We here take a small example of generative modeling with constraints and see its EM learning.

```
values(subj,[sg,pl]).     % introduce msw/2 named sbj and obj
values(obj,[sg,pl]).      % with outcomes = {sg,pl}
target(agree/1).

agree(A):-
  msw(subj,A),            % flip the coin subj
  msw(obj,B),             % flip the coin obj
  A=B.                    % equality constraint
```

Fig. 10. `agree` program describing agreement in number

A program in Fig. 10 models agreement in number in some natural language using two biased coins `subj` and `obj`. `values` clauses declare we use multi-ary random switches `msw(`a`,`v`)` and `msw(`b`,`v`)` where v is `sg` or `pl`. `target(agree/1)` declares a target predicate and what we observe are atoms of the form `agree(`$\cdot$`)`.

For a top-goal `:-sample(agree(X))` which starts a sampling of the defined distribution for `agree/1`, `msw(subj,A)` is executed simulating coin tossing of `subj` which probabilistically instantiates `A` either to `sg` or `pl`. Similarly for `msw(obj,B)`. Hence an outcome is one of $\{\texttt{agree(sg)}, \texttt{agree(pl)}\}$ and it is observable only when both coins agree (see the equality constraint `A=B`). When the two coins disagree, `A=B` fails and we have no observable outcome from this model.

Parameter learning by the fgEM algorithm: Given the program in Fig. 10 and a list of observations such as `[agree(sg),agree(pl),...]`, we estimate parameters, i.e. probabilities of each coin showing `sg` or `pl`. In what follows, to simplify notation and discussion, we treat logical variables `A` and `B` as random variables and put parameters by $\theta_{\texttt{a}} = P(\texttt{A} = \texttt{sg}) = P(\texttt{msw(subj,sg)})$, $\theta_{\texttt{b}} = P(\texttt{B} = \texttt{sg}) = P(\texttt{msw(obj,sg)})$ and $\boldsymbol{\theta} = (\theta_{\texttt{a}}, \theta_{\texttt{b}})$.

Parameters are estimated from the observable distribution $P(\texttt{A} \mid \texttt{success}, \boldsymbol{\theta})$ $= \sum_{\texttt{B} \in \{\texttt{sg},\texttt{pl}\}} P(\texttt{agree(A)}, \texttt{A} = \texttt{B} \mid \boldsymbol{\theta}) / P(\texttt{success} \mid \boldsymbol{\theta})$ (hereafter we omit $\boldsymbol{\theta}$ when obvious). Because the normalizing constant $P(\texttt{success}) = P(\texttt{agree(sg)}) + P(\texttt{agree(pl)}) = \theta_{\texttt{a}}\theta_{\texttt{b}} + (1 - \theta_{\texttt{a}})(1 - \theta_{\texttt{b}})$ is not necessarily equal to unity, the defined model becomes log-linear.

Obtaining a failure program: Thanks to the deep relationship between failure and negation in logic programming, a failure program can be derived automatically by 'negating' the target atom in the original program. Let `q(X)` be a target atom. We add `failure:-` $\forall$ `X` `not(q(X))` to the original program. `failure` says that there is no observable outcome of `q(X)`. As it is not executable, we 'compile' it using FOC to obtain a negation-free executable program. For the `agree` program in Fig. 10 we add two clauses shown in Fig. 11.[31]

```
failure :- not(success).
success :- agree(_).       % agree(_) = ∃X agree(X)
```

Fig. 11. Clauses defining `failure`

FOC compiles the `failure` predicate into executable code shown in Fig. 12. As can be seen, it eliminates negation in Fig. 11 while introducing two new predicates and one new function symbol.[32] We would like to point out that negation elimination is just one functionality of FOC. It can compile a much wider class of formulas into executable logic programs.

```
failure:- closure_success0(f0).   % f0 is initial continuation
closure_success0(C):- closure_agree0(C).
closure_agree0(C):-
    msw(subj,A),
    msw(obj,B),
    \+A=B.                        % \+ is Prolog's negation
```

Fig. 12. Compiled `failure` program

Using the failure program in Fig. 12 we conducted a learning experiment with artificial data sampled from the `agree` program. The sample size is 100 and the original and learned parameters (by fgEM and by gEM) are shown below.

parameters	*original*	*fgEM*	*gEM*
$\theta_{\mathtt{a}}$	0.4	0.4096	0.48
$\theta_{\mathtt{b}}$	0.6	0.6096	0.48

As seen clearly, parameters estimated by the gEM algorithm that does not take failure into account are widely off the mark. Worse yet it cannot even dis-

[31] We here decompose `failure :-` $\forall$ `X` `not(agree(X))` into two clauses for readability.

[32] They convey 'continuation' (data representing the remaining computation). `f0` is an initial continuation and bound to `C` in `closure_success0(C)` and `closure_agree0(C)`.

tinguish between two parameters. We suspect that such behavior always occurs when failure is ignored though data is generated from a failure model.

5.5 Constrained HMMs

As an instance of generative modeling with constraints that may fail, we introduce *constrained HMMs*, a new class of HMMs that have constraints over the states and emitted symbols [8]. Constraints are arbitrary and can be global such as the total number of emitted symbols being equal to a multiple of three. Constrained HMMs define conditional distributions just like CRFs (conditional random fields) [45] but generatively. Our hope is that they contribute to the modeling of complex sequence data and will be applied to, say bioinformatics, computer music etc.

We illustrate constrained HMMs by an example borrowed from [8]. It models the probabilistic behavior of a person on a diet. The situation is like this. He, the person, takes lunch at one of two restaurants '`r0`' and '`r1`' which he probabilistically chooses at lunch time. He also probabilistically chooses pizza (900) or sandwich (400) at '`r0`', and hamburger (500) or sandwich (500) at '`r1`' (numbers are calories). He is ordered by his doctor to keep calories for lunch in a week less than 4000 in total. Furthermore he is asked to record what he has eaten in a week like `[p,s,s,p,h,s,h]` and show the record to the doctor. He however is a smart person and preserves it only when he has succeeded in satisfying the constraint. Our task is to estimate his behavioral probability from the list of preserved records.

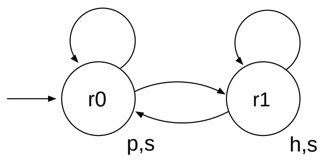

Fig. 13. HMM for the dieting person

An HMM in Fig. 13 represents the probabilistic behavior of the dieting person except the constraint on total calories for lunch in a week. A program in Fig. 14 is a usual HMM program corresponding to Fig. 13 but augmented with a constraint atom `C < 4000` where `C` stands for accumulated calories. If the accumulated calories, `C`, is not less than 4,000 on the seventh day, `C < 4000` in the last clause fails and so does the execution of the program (sampling).

The program looks self-explanatory but we add some comments for readability. `R` is the current restaurant and `msw(tr(R),R2)` is used for a random choice of the restaurant next day, `R2`. Likewise `msw(lunch(R),D)` stands for a random choice of dish '`D`' for lunch at '`R`'. We accumulate calories in `C` and record the chosen dish in `L`. FOC eliminates `not` in the program and produces a failure program that runs linearly in the number of days ('`N`').

```
failure:- not(success).
success:- chmm(L,r0,0,7).

chmm(L,R,C,N):- N>0,
  msw(tr(R),R2),              % choose a restaurant
  msw(lunch(R),D),            % choose lunch
  ( R = r0,
    ( D = p, C2 is C+900      % pizza:900, sandwich:400
    ; D = s, C2 is C+400 )    % hanburger:400, sandwich:500
  ; R = r1,
    ( D = h, C2 is C+400
    ; D = s, C2 is C+500 ) ),
  L = [D|L2],
  N2 is N-1,
  chmm(L2,R2,C2,N2).          % recursion for next day
chmm([],_,C,0):- C < 4000.   % calories must be < 4,000
```

Fig. 14. Constrained HMM program

Table 2. Estimated probabilities for the dieting person

sw name	*original value*		*estimation(average)*	
tr(r0)	r0 (0.7)	r1 (0.3)	r0 (0.697)	r1 (0.303)
tr(r1)	r1 (0.7)	r0 (0.3)	r1 (0.701)	r0 (0.299)
lunch(r0)	p (0.4)	s (0.6)	p (0.399)	s (0.601)
lunch(r1)	h (0.5)	s (0.5)	h (0.499)	s (0.501)

With this synthesized failure program, we conducted a learning experiment using artificial data sampled from the original program. After setting parameters as shown on the left column of *original value* in Table 2, we generated 500 examples and let the fgEM algorithm estimate parameters from these 500 examples. The right column in Table 2 shows shows averages of 50 experiments. The result says for example that the probability of `msw(tr(r0),r0)` be true, i.e. the probability of visiting the restaurant `r0` from `r0`, is originally 0.7 and the average of estimation is 0.697. It seems we have reasonably succeeded in recovering original parameters.

In this section, we explained how to overcome probability-loss-to-failure in generative modeling and introduced constrained HMMs, a class of HHMs with constraints, as an application. We also introduced *finite PCFGs* in [46] as a preliminary step toward approximate computation of probabilistic HPSGs. They are PCFGs with a constraint on the size of parse trees and only a finite number of parse trees are allowed for a sentence. Their unique property is that even if the original PCFG suffers probability-loss-to-infinity, parameters can be estimated from the constrained one. We conducted a learning experiment with a real corpus and observed that the deterioration of parsing accuracy by truncating parse trees to a certain depth is small [46].

6 Related work and discussion

There are three distinctive features which jointly characterize PRISM as a logic-based probabilistic modeling language. They are the *distribution semantics* [2], *two-staged probability computation*, i.e. probability computation combined with tabled search [6, 47] and EM learning by the *fgEM algorithm* [8]. Since each topic has a body of related work of its own, we only selectively state some of them.

6.1 Semantic aspects

First we deal with semantic aspects. When looking back on probabilistic reasoning in AI, one can observe at least two research streams. One of them focuses on the inference of probabilistic intervals, in a deductive style [48–50] or in a logic programming style [51–54]. The other aims to define distributions. The latter is further divided, for the sake of explanation, into three groups in view of how distributions are defined. The first group uses undirected graphical models [55, 56] and define discriminative models. The second group is based on directed graphs, i.e. BNs and their combination with KBs (knowledge bases) called KBMC (knowledge-based model construction) [57–67]. The third group does not rely on graphs but relies on the framework of logic programming [2, 6, 7, 23, 24, 68–72] to which PRISM belongs, or on the framework of functional programming [73].

Semantically the most unique feature of PRISM's declarative semantics, i.e. the distribution semantics, is that it defines *unconditionally* a global probability measure over the set of uncountably many Herbrand interpretations for any program in a first order language with countably many function and predicate symbols. In short, it always uniquely defines a *global and infinite* distribution for any program, even if it is a looping program such as `p(X):-p(X),q(Y)` or even if it contains negation in some clause body [8]. Note that we are not claiming that PRISM, an embodiment of the distribution semantics, can always compute whatever probability defined by its semantics. Apparently it is impossible. All it can do is to compute computable part of the distribution semantics.

Unconditional global distribution: The unconditional existence of global distributions by the distribution semantics sharply differs from the KBMC approach. Basically in the KBMC approach, a BN is reconstructed from a KB and an input every time the input is given. For example in the case of PRMs (probabilistic relational models) [61], when a new record is added to a RDB, a new BN is constructed. Since different inputs yield different BNs, there is no global distribution defined by the KB, which makes it difficult to consider observed data as iid data from the KB.

Our semantics also sharply differs from SLP (stochastic logic programming) [68, 7] which defines a distribution over SLD proofs, not over Herbrand interpretations. Furthermore for a distribution to be definable in SLP, programs must satisfy a syntactic condition called "range-restrictedness" which excludes many ordinary programs such as the member program. The same range-restrictedness condition is imposed on BLPs (Bayesian logic programs) [62] and LBNs (logical Bayesian networks) [67].

Infinite distribution: Concerning infinite distributions (infinite BNs), there are attempts to construct them from infinitely many BNs. Kersting and De Raedt showed the existence of infinite BNs defined by BLPs. They assume certain conditions including the acyclicity of the ground level caller-callee relation defined by a BLP program. Under the conditions, local BNs are constructed for each ground atom in the least model and distributions defined by them are pasted together to construct an infinite distribution using the Kolmogorov extension theorem [65].

Similar conditions were placed when Laskey proposed a probabilistic logic called MEBN (multi-entity BN) [74]. In MEBN local BNs are defined by schemes containing logical variables and constraints. Assuming local BNs, when combined, create no infinitely many parents (in terms of the CPT) or ancestors for each node, the existence of a global infinite distribution satisfying each scheme is proved by Kolmogorov's extension theorem.

BLOG (Bayesian logic) proposed by Milch et al. [75] defines a distribution over possible worlds consisting of objects through an infinite BN. In a BLOG model (program), objects are generated according to certain statements containing CPTs. These statements generate structured random variables in conjunction with their dependency as a local BN. CBNs (contingent BNs) are introduced to precisely specify such dependency [76]. They are BNs with constraints attached to edges such that edges are removed when constraints are not satisfied. The existence of a global distribution satisfying these local BNs is proved when a CBN satisfies conditions ensuring that every node has finitely many ancestors and parents.

These approaches all unify infinitely many distributions defined by local BNs, and hence need conditions to ensure the possibility of their unification. The distribution semantics of PRISM on the other hand does not attempt to unify local BNs to construct an infinite distribution. Instead it starts from a simple infinite distribution (over `msw` atoms) that surely exists and extends it by way of fixpoint operation which is always possible, thereby achieving the unconditional definability of infinite distributions.

Independent vs. dependent choice: In the distribution semantics, P_{DB}, the distribution defined by a program DB, is parameterized with a base distribution P_F. PRISM implements the simplest P_F, a product measure of independent choices represented by `msw` atoms like PHA [23], aiming at efficiency of probability computation. At first one may feel that `msw` atoms are not enough for complex modeling because they cannot represent dependent choices. In reality, however, they can, because the name of an `msw` switch is allowed to be any term and hence, one choice `msw(`s`,`X`)` can affect another choice `msw(`$t[X]$`,`Y`)` through their common variable X. The realization of dependent choice by this "name trick" is used to write a naive BN program and also used to implement an extension of PCFGs called pseudo context sensitive models [6]. We note that mutually dependent choices can be implementable by substituting a Boltzmann machine for P_F.

6.2 Probability computation

Two-staged probability computation: The second unique feature of PRISM is a way of probability computation. To compute a probability $P_{DB}(G)$ of a top-goal atom (query) G from a distribution P_{DB} defined by a program DB at the predicate level, we first reduce G by tabled search to an explanation graph $\mathit{Expl}(G)$ for G such that $\mathit{Expl}(G) \Leftrightarrow G$. $\mathit{Expl}(G)$ is a compact propositional formula logically equivalent to G and consists of ground user atoms and `msw` atoms. Then we compute $P_{DB}(\mathit{Expl}(G))(= P_{DB}(G))$ as the generalized inside probability by dynamic programming assuming the exclusiveness of disjuncts and the independence of conjuncts in $\mathit{Expl}(G)$.

The reduction from a predicate level expression (goal) to a propositional expression (explanation graph) can have three favorable effects though it sometimes blows up the size of the final expression. The first one is that of pruning, detecting impossible conditions corresponding to zero probability. The second one is to be able to explicitly express exclusiveness as a disjunction and independence as a conjunction. In an attempt by Pynadath and Wellman that formulates a PCFG by a BN [77], they end up in a very loopy BN partly because zero probability and exclusiveness are not easy to graphically express in BNs. To state the last one, we need some terminology.

Value-wise vs. variable-wise: We say that random variables are used *variable-wise* in an expression if they are treated uniformly in the expression regardless of their values. Otherwise we say they are used *value-wise*. For example BNs are variable-wise expressions and their standard computation algorithm, BP, is a variable-wise algorithm because uniform operations are performed on variables regardless of their values whereas, for instance, d-DNNFs after optimization used in ACE are value-wise. The point is that value-wise expressions have a bigger chance of sharing subexpressions (and hence sharing subcomputations in dynamic programming) than variable-wise expressions because for sharing to occur in a value-wise expression, it is enough that only some values, not all values of a random variable, are shared by some subexpressions. Also we should note that value-wise dependency is always sparser than variable-wise dependency because 0 probability cuts off the chain of value-wise dependency.

Returning to PRISM, in a program, a random variable X is expressed as a logical variable X in an `msw` atom like $\mathtt{msw}(id, X)$, but a search process instantiates it, differently depending on the context of its use, to ground terms like $\mathtt{msw}(id, t_1), \ldots, \mathtt{msw}(id, t_n)$ resulting in a value-wise explanation graph. The reduction to a value-wise expression is a key step to make possible $O(L^3)$ computation of a sentence probability by PRISM [6] where L is the sentence length.

Moreover, it was empirically shown that for moderately ambiguous PCFGs, the probability computation by PRISM is much faster than the one by the standard Inside-Outside algorithm [6]. Another example of value-wise computation is the left-to-right HMMs shown in Fig. 1. Their direct implementation by a BN (Fig. 6) would require $O(N^2)$ time for probability computation where N is the number of states though their transition matrix is sparse. Contrastingly PRISM, exploiting this sparseness by value-wise computation, realizes $O(N)$ computation as can be seen from Fig. 4.

Propositionalized BNs: It is interesting to note that probability computation of BNs by way of explanation graphs such as the one in Section 3 agrees with the emerging trend of "propositionalized BNs" which computes probabilities of a BN by converting it to a value-wise propositional level formula [17, 34, 35]. For example in [17] Chavira and Darwiche considered a BN as a multi-variate polynomial composed of binary-random variables representing individual values of a random variable and compile the BN into an arithmetic circuit. They empirically showed that their approach can efficiently detect and take advantage of CSI (context-specific independence) [38] in the original BN. McAllester et al. proposed CFDs (case factor diagrams) which are formulas representing a "feasible" set of assignments for infinitely many propositional variables. They can compute probabilities of linear Boolean models, a subclass of log-linear models [34]. In a CFD, subexpressions are shared and probabilities are computed by dynamic programming, thus realizing cubic order probability computation for PCFGs. Mateescu and Dechter introduced AND/OR search trees representing variable elimination of BNs propositionally [26]. Value dependency of nodes in a BN is expressed as an AND/OR search tree but identical subtrees are merged to produce a "minimal AND/OR graph" which realizes shared probability computation.

The main difference between PRISM and these approaches is that they manipulate propositional level expressions and predicate level expressions are out of concern. In the case of CFDs for example, programming for a PCFG starts from encoding parse forests of sentences. Contrastingly in PRISM, we do not encode parse forests but encode the PCFG itself using high level expressions in predicate logic. A subsequent search process automatically reduces a sentence to propositional formulas representing parse forests. Our approach thus makes compatible the ease and flexibility of high level programming and the computational efficiency in low level probability computation. However hitting the right balance between generality and efficiency in designing a programming language is always a difficult problem. PRISM is one extreme aiming at generality. A recent proposal of LOHMMs (logical hidden Markov models) by Kersting et al. [71] takes an intermediate approach by specializing in a logical extension of HMMs.

Eliminating conditions: Finally we discuss the possibility of eliminating some conditions in Subsection 2.2. The first candidate is the exclusiveness condition on disjunctions. It can be eliminated by appealing to general computation schemes such as the inclusion-exclusion principle generalizing $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$ and the sum-of-disjoint products generalizing $P(A \vee B) = P(A) + P(\neg A \wedge B)$, or BDDs (binary decision diagrams) [78]. ProbLog, seeking efficiency, uses BDDs to compute probabilities of nonexclusive disjunctions [72]. Although it seems possible in principle to introduce BDDs to PRISM's explanation graphs at the cost of increasing time and memory, details are left as future work.

The second candidate is the acyclicity condition. When eliminated, we might have a "loopy" explanation graph. Such a graph makes mathematical sense if, like loopy BP, loopy probability computation guided by the graph converges. There is a general class of loopy probability computation that looks relatively simple and useful; prefix computation of PCFGs. Given a string $s = w_1, \ldots, w_k$ and a PCFG, we would like to compute the probability that s is an initial string of some complete sentence $s = w_1, \ldots, w_k, \ldots, w_n$ derived from the PCFG. There already exists an algorithm for that purpose [79] and we can imagine a generalized prefix computation in the context of PRISM. We however need to consider computation cost as the resulting algorithm will heavily use matrix operations to compute "loopy inside probability."

6.3 EM learning

The advantage of EM learning by PRISM is made clear when we are given the task of EM learning for N new probabilistic model classes like BN, HMMs, PCFGs etc. We write N different programs and apply the same algorithm, the (f)gEM algorithm, to all of them, instead of deriving a new EM algorithm N times. The differences in model classes are subsumed by those in their explanation graphs and do not affect the gEM algorithm itself. The cost we have to pay for this uniformity however is time and space inefficiency due to the use of predetermined data structure, explanation graphs, for all purposes. For example, HMMs in PRISM require memory proportional to the input length to compute forward-backward probabilities while a specialized implementation only needs a constant space.

Another problem is that when we attempt EM learning of a generative model with failure, we have to synthesize a failure program that can represent all failed computation paths of the original program for the model. When models are variants of HMMs like constrained HMMs in Section 5, this synthesis is always possible. However for other cases including PCFGs with constraints, the synthesis is future work.

7 Conclusion

PRISM is a full programming language system equipped with rich functionalities and built-in predicates of Prolog enhanced by three components for probabilistic modeling. The first one is the *distribution semantics* [2], a measure theoretical semantics for probabilistic logic programs. The second one is *two-staged probability* *computation* [6, 47], i.e. generalized IO computation after tabled-search for explanation graphs. The third one is an EM algorithm, *the fgEM algorithm* [8], for generative models allowing failure. PRISM not only uniformly subsumes three representative model classes, i.e. BNs, HMMs, and PCFGs as instances of the distribution semantics at the semantic level but uniformly subsumes their probability computation with the same time complexity, i.e. BP on junction trees for BNs [10], the forward-backward algorithm for HMMs, and IO probability computation for PCFGs respectively as instances of generalized IO probability computation for logic programs [6].

Despite the generality of computational architecture, PRISM runs reasonably fast compared to the state-of-art systems as demonstrated in Section 4 as long as we accept memory consumption for tabling. We also emphasize that PRISM facilitates the creation and exploration of new models such as constrained HMMs as exemplified in Section 5. Hence we believe PRISM is now a viable tool for prototyping of various probabilistic models.

There remains much to be done. The biggest problem is memory consumption. Currently terms are created dynamically by pointers and every pointer occupies 64 bits. This is a very costly approach from a computational viewpoint though it gives us great flexibility. Restricting the class of admissible programs to make it possible to introduce array is one way to avoid the memory problem. The second one is to make PRISM more Bayesian. Currently only MAP estimation is possible though we are introducing built-in predicates for BIC [80] and the Cheeseman-Stutz criterion [81]. Probably we need a more powerful Bayesian computation such as variational Bayes to cope with data sparseness. Also parallelism is inevitable to break computational barrier. Although an initial step was taken toward that direction [82], further investigation is needed.

Acknowledgments

We are grateful to Yusuke Izumi and Tatsuya Iwasaki for generously providing all graphs in this chapter.

References

1. Sato, T., Kameya, Y.: PRISM: a language for symbolic-statistical modeling. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97). (1997) 1330–1335
2. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Proceedings of the 12th International Conference on Logic Programming (ICLP'95). (1995) 715–729
3. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. Royal Statistical Society **B39**(1) (1977) 1–38
4. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Proceedings of the 3rd International Conference on Logic Programming (ICLP'86). Volume 225 of Lecture Notes in Computer Science., Springer (1986) 84–98

5. Zhou, N.F., Sato, T.: Efficient fixpoint computation in linear tabling. In: Proceedings of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03). (2003) 275–283
6. Sato, T., Kameya, Y., Abe, S., Shirai, K.: Fast EM learning of a family of PCFGs. Technical Report (Dept. of CS) TR01-0006, Tokyo Institute of Technology (2001)
7. Cussens, J.: Parameter estimation in stochastic logic programs. Machine Learning **44**(3) (Sept. 2001) 245–271
8. Sato, T., Kameya, Y., Zhou, N.F.: Generative modeling with failure in PRISM. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05). (2005) 847–852
9. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann (1988)
10. Sato, T.: Inside-Outside probability computation for belief propagation. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). (2007) 2605–2610
11. Lauritzen, S., Spiegelhalter, D.: Local computations with probabilities on graphical structures and their applications to expert systems. Journal of the Royal Statistical Society, B **50** (1988) 157–224
12. Jensen, F.V.: An Introduction to Bayesian Networks. UCL Press (1996)
13. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2) (1989) 257–286
14. Baker, J.K.: Trainable grammars for speech recognition. In: Proceedings of Spring Conference of the Acoustical Society of America. (1979) 547–550
15. Eisner, J., Goldlust, E., Smith, N.: Compiling Comp Ling: Weighted dynamic programming and the Dyna language. In: Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP). (2005) 281–290
16. Darwiche, A.: A compiler for deterministic, decomposable negation normal form. In: Proceedings of the 18th national conference on Artificial intelligence (AAAI'02). (2002) 627–634
17. Chavira, M., Darwiche, A.: Compiling Bayesian networks with local structure. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05). (2005) 1306–1312
18. Chavira, M., Darwiche, A., Jaeger, M.: Compiling relational bayesian networks for exact inference. International Journal of Approximate Reasoning **42** (2006) 4–20
19. Doets, K.: From Logic to Logic Programming. The MIT Press (1994)
20. Manning, C.D., Schütze, H.: Foundations of Statistical Natural Language Processing. The MIT Press (1999)
21. Castillo, E., Gutierrez, J.M., Hadi, A.S.: Expert Systems and Probabilistic Network Models. Springer-Verlag (1997)
22. Chow, Y., Teicher, H.: Probability Theory (3rd ed.). Springer (1997)
23. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence **64**(1) (1993) 81–129
24. Poole, D.: The independent choice logic for modeling multiple agents under uncertainty. Artificial Intelligence **94**(1-2) (1997) 7–56
25. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Databases. Plenum Press (1978) 293–322
26. Mateescu, R., Dechter, R.: The relationship between AND/OR search spaces and variable elimination. In: Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI'05). (2005) 380–387
27. Sato, T.: Modeling scientific theories as PRISM programs. In: Proceedings of ECAI'98 Workshop on Machine Discovery. (1998) 37–45

28. Mitomi, H., Fujiwara, F., Yamamoto, M., Sato, T.: Bayesian classification of human custom based on stochastic context-free grammar (in Japanese). IEICE Transaction on Information and Systems **J88-D-II**(4) (2005) 716–726
29. Wang, S., Wang, S., Greiner, R., Schuurmans, D., Cheng, L.: Exploiting syntactic, semantic and lexical regularities in language modeling via directed Markov random fields. In: Proceedings of the 22th International Conference on Machine Learning (ICML'05). (2005) 948–955
30. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of Artificial Intelligence Research **15** (2001) 391–454
31. Smyth, P., Heckerman, D., Jordan, M.: Probabilistic independence networks for hidden Markov probability models. Neural Computation **9**(2) (1997) 227–269
32. Kask, K., Dechter, R., Larrosa, J., Cozman, F.: Bucket-tree elimination for automated reasoning. ICS Technical Report Technical Report No.R92, UC Irvine (2001)
33. Shafer, G., Shenoy, P.: Probability propagation. Annals of Mathematics and Artificial Intelligence **2** (1990) 327–352
34. McAllester, D., Collins, M., Pereira, F.: Case-factor diagrams for structured probabilistic modeling. In: Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence (UAI'04), Arlington, Virginia, AUAI Press (2004) 382–391
35. Minato, S., Satoh, K., Sato, T.: Compiling bayesian networks by symbolic probability calculation based on zero-suppressed bdds. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). (2007) 2550–2555
36. Charniak, E.: Tree-bank grammars. In: Proceedings of the 13th National Conference on Artificial Intelligence(AAAI'96). (1996) 1031–1036
37. Marcus, M., Santorini, B., Marcinkiewicz, M.: Building a large annotated corpus of English: the Penn Treebank. Computational Linguistics **19** (1993) 313–330
38. Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D.: Context-specific independence in Bayesian networks. In: Procceding of the 12th Conference on Uncertainty in Artificial Intelligence (UAI'96). (1996) 115–123
39. Chi, Z., Geman, S.: Estimation of probabilistic context-free grammars. Computational Linguistics **24**(2) (1998) 299–305
40. Wetherell, C.S.: Probabilistic languages: a review and some open questions. Computing Surveys **12**(4) (1980) 361–379
41. Abney, S.: Stochastic attribute-value grammars. Computational Linguistics **23**(4) (1997) 597–618
42. Schmid, H.: A generative probability model for unification-based grammars. In: Proceedings of the 21st International Conference on Computational Linguistics (COLING'02). (2002) 884–896
43. Sag, I., Wasow, T.: Syntactic Theory: A Formal Introduction. Stanford: CSLI Publications (1999)
44. Sato, T.: First Order Compiler: A deterministic logic program synthesis algorithm. Journal of Symbolic Computation **8** (1989) 605–627
45. Lafferty, J., McCallum, A., Pereira, F.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the 18th International Conference on Machine Learning (ICML'01). (2001) 282–289
46. Sato, T., Kameya, Y.: Negation elimination for finite PCFGs. In: Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation 2004 (LOPSTR'04). (2004) 119–134

47. Kameya, Y., Sato, T.: Efficient EM learning for parameterized logic programs. In: Proceedings of the 1st Conference on Computational Logic (CL'00). Volume 1861 of Lecture Notes in Artificial Intelligence., Springer (2000) 269–294
48. Nilsson, N.J.: Probabilistic logic. Artificial Intelligence **28** (1986) 71–87
49. Frish, A., Haddawy, P.: Anytime deduction for probabilistic logic. Journal of Artificial Intelligence **69** (1994) 93–122
50. Lukasiewicz, T.: Probabilistic deduction with conditional constraints over basic events. Journal of Artificial Intelligence Research **10** (1999) 199–241
51. Ng, R., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation **101** (1992) 150–201
52. Lakshmanan, L.V.S., Sadri, F.: Probabilistic deductive databases. In: Proceedings of the 1994 International Symposium on Logic Programming (ILPS'94). (1994) 254–268
53. Dekhtyar, A., Subrahmanian, V.S.: Hybrid probabilistic programs. In: Proceedings of the 14th International Conference on Logic Programming (ICLP'97). (1997) 391–405
54. Saad, E., Pontelli, E.: Toward a more practical hybrid probabilistic logic programming framework. In: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'05), vol. 3350, LNCS. (2005) 67–82
55. B.Taskar, P.Abbeel, D.Koller: Discriminative probabilistic models for relational data. In: Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI'02). (2002) 485–492
56. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning **62** (2006) 107–136
57. Breese, J.S.: Construction of belief and decision networks. Computational Intelligence **8**(4) (1992) 624–647
58. Wellman, M., Breese, J., Goldman, R.: From knowledge bases to decision models. Knowledge Engineering Review **7**(1) (1992) 35–53
59. Koller, D., Pfeffer, A.: Learning probabilities for noisy first-order rules. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97). (1997) 1316–1321
60. Ngo, L., Haddawy, P.: Answering queries from context-sensitive probabilistic knowledge bases. Theoretical Computer Science **171** (1997) 147–177
61. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99). (1999) 1300–1309
62. Kristian Kersting, K., De Raedt, L.: Bayesian logic programs. In: Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming (ILP'00). (2000) 138–155
63. Jaeger, J.: Complex probabilistic modeling with recursive relational Bayesian networks. Annals of Mathematics and Artificial Intelligence **32**(1-4) (2001) 179–220
64. Getoor, L., Friedman, N., Koller, D.: Learning probabilistic models of relational structure. In: Proceedings of the 18th International Conference on Machine Learning (ICML'01). (2001) 170–177
65. Kersting, K., De Raedt, L.: Basic principles of learning bayesian logic programs. Technical Report Technical Report No. 174, Institute for Computer Science, University of Freiburg (2002)
66. Chavira, M., Darwiche, A., Jaeger, M.: Compiling relational bayesian networks for exact inference. In: Proceedings of the Second European Workshop on Probabilistic Graphical Models (PGM'04). (2004) 49–56

67. Fierens, D., Blockeel, H., Bruynooghe, M., Ramon, J.: Logical Bayesian networks and their relation to other probabilistic logical models. In: Proceedings of the 15th International Conference on Inductive Logic Programming (ILP'05), volume 3625 of Lecture Notes in Computer Science. (2005) 121–135
68. Muggleton, S.: Stochastic logic programs. In de Raedt, L., ed.: Advances in Inductive Logic Programming. IOS Press (1996) 254–264
69. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Proceedings of the 20th International Conference on Logic Programming (ICLP'04). Lecture Notes in Computer Science 3132 (2004) 431–445
70. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. In: Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04). Volume 2923 of LNAI., Springer (2004) 21–33
71. Kersting, K., De Raedt, L., Raiko, T.: Logical hidden Markov models. Journal of Artificial Intelligence Research **25** (2006) 425–456
72. De Raedt, L., Angelika, K., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discoverry. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). (2007) 00–00
73. Pfeffer, A.: IBAL: A probabilistic rational programming language. In: Proceedings of the 17th International Conference on Artificial Intelligence (IJCAI'01). (2001) 733–740
74. Laskey, K.: MEBN: A logic for open-world probabilistic reasoning. C4I Center Technical Report C4I06-01, George Mason University Department of Systems Engineering and Operations Research (2006)
75. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05). (2005) 1352–1359
76. Milch, B., Marthi, B., Sontag, D., Russell, S., Ong, D., Kolobov, A.: Approximate Inference for Infinite Contingent Bayesian Networks. In: Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics (AISTATS'05). (2005) 1352–1359
77. Pynadath, D.V., Wellman, M.P.: Generalized queries on probabilistic context-free grammars. IEEE Transaction on Pattern Analysis and Machine Intelligence **20**(1) (1998) 65–77
78. Rauzy, A., Chatelet, E., Dutuit, Y., Berenguer, C.: A practical comparison of methods to assess sum-of-products. Reliability Engineering and System Safety **79** (2003) 33–42
79. Stolcke, A.: An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Computational Linguistics **21**(2) (1995) 165–201
80. Schwarz, G.: Estimating the dimension of a model. Annals of Statistics **6**(2) (1978) 461–464
81. Cheeseman, P., Stutz, J.: Bayesian classification (AutoClass): Theory and results. In Fayyad, U., Piatesky, G., Smyth, P., Uthurusamy, R., eds.: Advances in Knowledge Discovery and Data Mining. The MIT Press (1995) 153–180
82. Izumi, Y., Kameya, Y., Sato, T.: Parallel EM learning for symbolic-statistical models. In: Proceedings of the International Workshop on Data-Mining and Statistical Science (DMSS'06). (2006) 133–140