Effective Bayesian Inference for Stochastic Programs

Daphne Koller  
Stanford University  
koller@cs.stanford.edu

David McAllester  
AT&T Research  
dmac@research.att.com

Avi Pfeffer  
Stanford University  
avi@cs.stanford.edu

Abstract

In this paper, we propose a stochastic version of a general purpose functional programming language as a method of modeling stochastic processes. The language contains random choices, conditional statements, structured values, defined functions, and recursion. By imagining an experiment in which the program "runs" and the random choices made by sampling, we can interpret a program in this language as encoding a probability distribution over a (potentially infinite) set of objects. We provide an exact algorithm for computing conditional probabilities of the form \( \Pr(P(x) \mid Q(x)) \) where \( x \) is chosen randomly from this distribution. This algorithm terminates precisely when sampling \( x \) and computing \( P(x) \) and \( Q(x) \) terminates in all possible stochastic executions (under lazy evaluation semantics, in which only values needed to compute the output of the program are evaluated). We demonstrate the applicability of the language and the efficiency of the inference algorithm by encoding both Bayesian networks and stochastic context-free grammars in our language, and showing that our algorithm derives efficient inference algorithms for both. Our language easily supports interesting and useful extensions to these formalisms (e.g., recursive Bayesian networks), to which our inference algorithm will automatically apply.

1 Introduction

Over the past few years, there has been a growing consensus within the AI community that, as the real world is a noisy and nondeterministic place, it is often useful to model it as such. Modeling uncertainty has shown up in a variety of AI tasks as diverse as diagnosis (Heckerman et al. 1995), natural language processing (Charniak 1993), planning (Dean et al. 1993), and more. The different requirements of these tasks have resulted in the use of different stochastic modeling languages, such as Bayesian networks (Pearl 1988) and dynamic Bayesian networks (Dean and Kanazawa 1989), hidden Markov models (Rabiner and Juang 1986), and stochastic context-free grammars (SCFGs) (Charniak 1993).

In many respects, these formalisms appear quite different, and each of them has induced special-purpose probabilistic inference algorithms. Recently, however, there has been a growing understanding that the formalisms have a common basis, primarily the use of probabilistic independence as the key to compact representations and efficient inference. As a consequence, there has been a recent effort to relate the different formalisms to each other (Smyth et al. 1996; Pynadath and Wellman 1996).

In this paper, we utilize the same basic idea of probabilistic independence, but in the context of a very rich general-purpose stochastic modeling language. The language we propose subsumes all of these formalisms, but is significantly more expressive than any of them. As we will show, this additional expressive power does not prevent the formulation of an efficient inference algorithm.

The key idea behind our language is the use of stochastic programs to model systems. Specifically, we define a stochastic version of a general-purpose functional programming language. The language contains random choices, conditional statements, structured values, defined functions, and recursion. A program in this language can be viewed as defining a distribution over a potentially infinite set of objects. Intuitively, we can imagine an experiment in which the program is "run" and the random choices made by sampling (flipping random bits). This process defines a distribution over the various outputs that the program can produce.

Typically, the problem with moving to such a rich language is that we lose the ability to execute inference efficiently. In a Turing-complete language such as this one, it is not even clear that we can execute inference at all. This is not the case for our language. We provide a probabilistic inference algorithm which computes the exact value of a conditional probability expression of the form \( \Pr(P(x) \mid Q(x)) \) where \( x \) is chosen randomly from the distribution defined by the program, and where \( P \) and \( Q \) are any nonstochastic predicates defined in our language. We show that the algorithm terminates precisely when the stochastic program which samples \( x \) and computes \( P(x) \) and \( Q(x) \) terminates in all possible stochastic executions. In both cases, termination is with respect to lazy evaluation semantics, in which only values needed to compute the output of the program are evaluated. These semantics allows us to use our algorithm even when the program specifies a distribution over an infinite set of objects (and even when the objects themselves are infinite). Thus, for example, we can evaluate a query...
with respect to a distribution over infinitely many strings (as in a SCFG), if the answer to the query can be determined by a finite computation. In this case, we say that the query is evidence finite.

While termination is good, it is hardly enough. Our algorithm achieves efficient inference by utilizing the principles underlying efficient Bayesian network algorithms (Dechter 1996) and SCFG algorithms (Lari and Young 1990). Despite its generality, our algorithm is almost as efficient as these special-purpose algorithms. Thus, for example, when we run our algorithm on a stochastic program representing a Bayesian network, its computational behavior is the same as that of standard variable elimination algorithms for Bayesian networks. Similarly, it is possible to encode a SCFG as a stochastic program so that our algorithm, applied to this program, behaves essentially the same as the inside algorithm (an inference algorithm tailored for SCFGs).

An alternative approach in the literature to developing rich stochastic modeling languages is probabilistic logic programming (Haddawy 1994; Poole 1993). While these formalisms have opened the way towards exploring the issues we discuss, they do not deal with structured values, lazy evaluation, and evidence finite computation. In addition, the algorithms that have been developed do not exploit the techniques we use to make our algorithm efficient.

2 Stochastic Programs

In this section we give the syntax and an informal semantics of a stochastic programming language. We start by giving a grammar for terms.

$$e ::= x | c(e_1, \ldots, e_n) | c^{-1}(e) | c?() | \text{if}(e_1, e_2, e_3) | \text{flip}(\alpha) | f(e_1, \ldots, e_n)$$

First we consider data structures. An expression of the form $c(e_1, \ldots, e_n)$ denotes a data structure whose components are the values of $e_1, \ldots, e_n$ respectively. The data structure denoted by $c(e_1, \ldots, e_n)$ also contains a type tag specifying that it was constructed by the constructor function $c$. Symbolic constants, such as 'true, 'false, and 'foo are represented by data constructor functions of no arguments. In general, a data constructor function is represented by a character string starting with the character '.

For example, the expression 'cons('foo, 'baz') denotes a "cons cell", i.e., a data structure of type 'cons. All values in our language are either symbolic constants or data structures whose fields contain (recursively) values. An expression of the form $c^{-1}(e)$ extracts the $i$th field from data structures of type $c$. If the data structure is not of type $c$, it returns 'false. For example, if 'cons is a constructor that takes two arguments, the functions 'cons^{-1} and 'cons correspond to the Lisp functions car and cdr respectively.

We use the latter as shorthand in some of our code. The expression $c?()$ has value 'true if $e$ is a data structure of type $c$ and value 'false otherwise. Note that we can use an expression of the form 'foo?($x$) to check whether the value of the variable $x$ is 'foo. We use $x == 'foo$ as a somewhat more natural notation.

The expression $\exists \epsilon(c_1, c_2, c_3)$ is a conditional expression — if the value of $e_1$ is the constant 'true then the value of the expression is the value of $e_2$, otherwise it is the value of $e_3$. Expressions of the form $\text{flip}(\alpha)$ are stochastic. The expression $\text{flip}(\alpha)$ has value 'true with probability $\alpha$ and 'false with probability $1 - \alpha$ ($\alpha$ must be a fixed constant in the open interval $(0, 1)$). An expression of the form $f(e_1, \ldots, e_n)$ is a call to the user defined function $f$.

In addition to terms, our programming language allows assignment statements of the form $x = e$ where $x$ is a variable and $e$ is an expression as defined by the above grammar. Assignments are important for stochastic modeling because they introduce correlations due to "common causes". Consider the following sequence of assignments:

$$x = \text{flip}(5); y = \text{pair}(x, z); z = \text{pair}(\text{flip}(5), \text{flip}(5));$$

The variables $x$ and $y$ are assigned values independently. The components of $z$ get independent values — there are four possible values of $z$, each equally likely. However, the values of the components of $y$ are not independent, and there are only two possible values of $y$.

We now define a network to be a sequence of assignment statements of the form $x_1 = c_1; x_2 = c_2; \ldots; x_n = c_n$. We require that each variable appearing in the network be assigned at most once, and that all uses of a variable assigned in the network occur after its definition. Variables appearing in the network which are not defined in it will be called inputs. A network with no inputs will be called closed.

A procedure definition is an expression of the form $f(x_1, \ldots, x_n) = \{N\}$ where $N$ is a network with inputs $x_1, \ldots, x_n$. The last assignment in $N$ is taken to define the output value of $f$. A program is a pair $(D, N)$ of a set $D$ of (user) definitions plus a network $N$ which is closed and where every user function is defined in $D$. The value of a program $(D, N)$ is the value of the last assignment in $N$. If the program diverges, it takes the value $1$.

A program is a stochastic model — it defines a probability distribution over the values of the program. There are two standard ways of computing values — strict evaluation and lazy evaluation. Under strict evaluation, if $e_i$ diverges (fails to terminate) then $c(e_1, \ldots, e_n)$ also diverges. Note that the divergent argument $e_i$ may not be needed by the remainder of the computation. Under lazy evaluation, if $e_i$ diverges then $c(e_1, \ldots, e_n)$ still terminates. For example, consider the program:

```
digit0 = {output = if (flip(S), 'one, 'zero);}
digit1 = {output = if (flip(S), 'zero, 'one);}
digit2 = {output = flip(S);}
real0 = {output = 'cons(digit(), real());}
real1 = {output = 'cons(real(), digit());}
real() = {output = 'cons(real0(), real1());}
output = real();
```

Under strict semantics this program will diverge. However, any run of the program will generate an infinite list of digits, which we may wish to interpret as defining a real number uniformly distributed over the interval $[0, 1]$. For example, we would like the value of

$$x = \text{real}(); \text{output} = \text{if}(\text{car}(x) == 'zero, 'true, 'false);$$

to be 'true or 'false, each with probability 1/2. A program like this, which only examines a finite fraction of an arbitrarily large stochastic value, is called evidence finite.

To illustrate how our language captures very different formalisms, we show how both Bayesian networks and
stochastic context-free grammars can easily be described. These examples only scratch the surface of the expressive power of the language, but they should give a taste of the possibilities. Lack of space precludes us from presenting more examples.

A traditional Bayesian network (Pearl 1988) is a DAG in which each node is a random variable. Associated with each node is a conditional probability table defining the probability of each possible value of a node given each possible assignment of values to its parents. Such a network can easily be encoded in our language as a sequence of assignment statements, one for each node, making sure that parents are always assigned before their children. For example, for a simple burglary-alarm network could be written as

```
earthquake = flip(0.01); burglary = flip(0.1);
alarm = if(earthquake, if(burglary, flip(0.99), flip(0.2)), if(burglary, flip(0.98), flip(0.01)));
```

The restriction of our language to defining distributions over the last node of the network can be made without loss of generality. If we are interested in the distribution over \( x_1, \ldots, x_n \), we can simply add a line to the end of our program:

```
output = 'vlist(x1, ..., xn) to the end of our program.
```

If \( f \) is a function describing a stochastic model (e.g., a Bayesian network), we can answer any query of the form \( \text{Pr}(P \mid Q) \), where \( P \) and \( Q \) are observations about the function output, using the simple program:

```
\text{match-suffix}(t,s) = \begin{cases} \text{null}, & \text{if } t = \text{last}(s), \\
\text{match-suffix}(\text{butlast}(t), \text{butlast}(s)), & \text{otherwise}\end{cases}
```

This is a representation of the SCFG described above.

This program defines a distribution over parse trees for the grammar, and induces a distribution over strings. For a given string \( s \), represented as a list, we can check whether the parse tree \( t \) represents a parse for \( s \) using the following checker function:

```
\text{match-suffix}(t,s) = \begin{cases} \text{false}, & \text{if } t = \text{null}, \\
\text{match-help}(t,s), & \text{otherwise}\end{cases}
```

The function \( \text{match-suffix} \) attempts to match \( t \) with a suffix of \( s \). If it succeeds, it returns the prefix of \( s \) that was unmatched, otherwise it returns \( \text{false} \). If \( t \) is a parse for the entire string \( s \), \( \text{match-suffix} \) will return \text{null}.

```
\text{match-help}(t,s) = \begin{cases} \text{false}, & \text{if } t = \text{null}, \\
\text{match-suffix}(\text{car}(t), \text{car}(s)), & \text{otherwise}\end{cases}
```

```
\text{output} = \text{if } s = \text{nil}, \text{false}, \\
\text{match-suffix}(\text{car}(t), \text{car}(s))
```

We assume that \( \text{last} \) and \( \text{butlast} \) have been given the appropriate definitions. Also, our language does not include tests for equality between arbitrary variables, but since \( s \) is a particular string it can easily be implemented for this example.

---

1. This functional perspective is, in fact, the basis for Pearl's recent work on the causal semantics of Bayesian networks (Pearl 1994).
3 A Sampling Algorithm

In this section, we give an algorithm for sampling values of a stochastic program. This sampling algorithm will serve as a precise operational semantics for our programming language. The algorithm also serves as the starting point for the development of our Bayesian inference algorithm.

The first stage of the algorithm converts a network to shallow form. An assignment statement \( x = e \) is called shallow if there are no proper subexpressions of \( e \) other than variables. For example, \( x = \text{flip}(y, z, w) \) is shallow but \( x = \text{flip}(\text{flip}(y), z, w) \) is not. Any network can be mechanically converted into one in which all assignments are shallow in time proportional to the size of the network.

For example, the shallow version of the burglar-alarm network described above is:

\[
\text{earthquake} = \text{flip}(0.01); \text{burglary} = \text{flip}(0.1); \text{a-when-e-b} = \text{flip}(0.99);
\]
\[
\text{a-when-e-nb} = \text{flip}(0.2); \text{a-when-ne-b} = \text{flip}(0.98);
\]
\[
\text{a-when-ne-nb} = \text{flip}(0.01); \text{a-when-e} = \text{if}(\text{burglary}, \text{a-when-e-b}, \text{a-when-e-nb});
\]
\[
\text{a-when-ne} = \text{if}(\text{burglary}, \text{a-when-ne-b}, \text{a-when-ne-nb}); \text{alarm} = \text{true};
\]

Our sampling algorithm is formulated as a recursive function \( \text{SAMP} \) which takes as input a network \( N \) and a variable \( x \) occurring in \( N \) and produces as output a new network \( N' \) where \( x \) is replaced by \( c(y_1, \ldots, y_n) \). More precisely, for any network \( N \), variable \( x \), and "value expression" \( c(y_1, \ldots, y_n) \) we define \( N[x = c(y_1, \ldots, y_n)] \) to be the network that is identical to \( N \) except that the definition of \( x \) is replaced by \( x = c(y_1, \ldots, y_n) \). It is important to note that \( \text{SAMP} \) does not sample a complete value of \( x \). It only processes the network to the degree necessary to determine the top level constructor of \( x \). The procedure \( \text{SAMP} \) is defined by the following conditions.

\[
\text{if } x = c(y_1, \ldots, y_n) \in N \text{ then } \text{SAMP}(N, x) = N.
\]

\[
\text{if } x = c^{-1}(y) \in N \text{ then } \text{compute } \text{SAMP}(N, x):
\]
\[
\text{let } N'' \text{ be } \text{SAMP}(N', y)
\]
\[
\text{if } N'' \text{ contains } x = c(z_1, \ldots, z_n)
\]
\[
\text{and return } N''[x = c(z_1, \ldots, z_n)]
\]
\[
\text{else return } N''[x = 'false]
\]

\[
\text{if } x = \text{c}(y) \in N \text{ then } \text{compute } \text{SAMP}(N, x):
\]
\[
\text{let } N' \text{ be } \text{SAMP}(N, y)
\]
\[
\text{if } N' \text{ contains } y = c(z_1, \ldots, z_n)
\]
\[
\text{then return } N'[x = 'true]
\]
\[
\text{else return } N'[x = 'false]
\]

\[
\text{if } x = \text{flip}(o) \in N \text{ then } \text{compute } \text{SAMP}(N, x):
\]
\[
\text{let } N'' \text{ be } \text{SAMP}(N', y)
\]
\[
\text{let } h \text{ be the variable if } y = 'true \in N' \text{ and the variable } w \text{ otherwise}
\]
\[
\text{let } N''' \text{ be } \text{SAMP}(N', h)
\]
\[
\text{SAMP}(N, x) = N'''[x = v] \text{ where } h = v \in N'''
\]

\[
\text{if } x = f(y_1, \ldots, y_n) \in N \text{ then compute } \text{SAMP}(N, x):
\]
\[
\text{let } M \text{ be the body of the definition of } f \text{ where the inputs}
\]
\[
\text{have been renamed to } y_1, \ldots, y_n \text{ and all other variables}
\]
\[
\text{replaced by fresh variables.}
\]
\[
\text{let } N' \text{ be } N \text{ where } x = f(y_1, \ldots, y_n) \text{ is replaced by } M; x = v
\]
\[
\text{where } o = v \text{ is the output statement in } M.
\]

\[
\text{return } \text{SAMP}(N', x)
\]

To understand this program, consider its behavior when applied to a stochastic program corresponding to a standard Bayesian network. In this case, \( \text{SAMP}(N, x) \) will return a sequence of assignments where \( x \), all of \( x \)'s parents, and the relevant intermediate variables, have been assigned concrete values. The values of these variables are chosen randomly, according to the distribution specified in the program. For example, in our burglar-alarm network, \( \text{SAMP}(N, \text{alarm}) \) may return (as one possible outcome):

\[
\text{earthquake} = \text{true}; \text{burglary} = \text{false}; \text{a-when-e-b} = \text{flip}(0.99);
\]
\[
\text{a-when-e-nb} = \text{true}; \text{a-when-ne-b} = \text{flip}(0.09);
\]
\[
\text{a-when-ne-nb} = \text{flip}(0.01); \text{a-when-e} = \text{true};
\]
\[
\text{a-when-ne} = \text{if}(\text{burglary}, \text{a-when-ne-b}, \text{a-when-ne-nb}); \text{alarm} = \text{true};
\]

As a more complex example, consider the function \( \text{tree}(x) = \{ \text{flip}(4); x = \text{tree}() \} \).

\[
\text{Now, computing } \text{SAMP}(N', \text{output}) \text{ where } N' \text{ is the following network.}
\]
\[
\text{flip} = \text{flip}(4); x = \text{tree}(); y = \text{tree}(); \text{if} = \text{leaf}; \text{pr} = \text{pair}(x, y);
\]
\[
\text{output} = \text{if}(\text{flip, if, pr})
\]

Let \( N \) be the network consisting of the single assignment \( \text{output} = \text{tree}() \) and consider computing a value of \( \text{SAMP}(N, \text{output}) \).

\[
\text{Let } N' = (\text{flip}(4); x = \text{tree}(); y = \text{tree}(); \text{if} = \text{leaf}; \text{pr} = \text{pair}(x, y);
\]
\[
\text{output} = \text{if}(\text{flip, if, pr})
\]

Now the conditional rule applies and we compute \( \text{SAMP}(N', \text{flip}) \).

\[
\text{Depending on whether the flip yields 'true' or 'false' we either evaluate } \text{SAMP}(N'', \text{if}) \text{ or }
\]
\[
\text{SAMP}(N'', \text{pr}) \text{ where } N'' \text{ is a network with the corresponding value for } \text{flip}.
\]

To understand this issue, consider a network \( N \) of variables. For example, uses the procedure \( \text{has-depth}(x) \).
than a network, then the sampled value for $x$, on which the value of $y$ depended, would be lost. In this case, the computation of $\text{SAMP}(N, x)$ could sample a different value for $x$ and the correlation between $y$ and $z$ due to the common input $x$ would be lost.

4 Computing Distributions

We now modify the sampling algorithm so that it computes an exact probability distribution rather than a sample from that distribution. That is, we will define a procedure DIST, which returns a distribution over the outputs of the procedure SAMP. The distribution will be the same distribution that SAMP induces on its outputs.

To characterize the conditions under which the distribution can be computed we need the following definition: a pair $(N, x)$ of a network $N$ and variable $x$ terminates if it is not possible to select the value of flip expressions so as to cause the computation of $\text{SAMP}(N, x)$ to diverge. Note that, due to the use of lazy evaluation, certain expressions, such as $\text{tree}()$, terminate even though in some sense they have an infinite set of possible values. Lazy evaluation makes it possible to build terminating models with infinite value sets. For any terminating pair $(N, x)$, the set of networks which can be returned by $\text{SAMP}(N, x)$ is finite.4

In this case, we can describe the output of DIST as a finite probability distribution. Such distributions will be written as “tables” of the form \( \{ (v_1, a_1), \ldots, (v_n, a_n) \} \) where all $v_i$ must be distinct, all $a_i$ must be real numbers in the interval $[0, 1]$, and the sum of all $a_i$ must be 1. This table denotes the distribution where item $v_i$ has probability $a_i$.

To understand DIST, it is important to recall that SAMP returns networks rather than values. Therefore, DIST will return distributions over objects that are, themselves, representations of other distributions. In order to eventually compute a single distribution, we have to combine these distributions into one. Therefore, we will often use a phrase of the form “the weighted sum over $M$ from $\text{DIST}(N, y)$ of $D_M$,” where $D_M$ is some distribution defined by the network $M$. This phrase denotes the probability distribution defined by sampling $M$ from the distribution (over networks) returned by DIST, and then sampling the distribution $D_M$ defined by $M$. The probability table defined by this phrase is computed by first (recursively) computing the table $D$ for $\text{DIST}(N, y)$, and then for each item $M$ which appears in this table with nonzero probability, a separate table $D_M$ is computed. The tables $D_M$ are then added together, where each table $D_M$ is weighted by the probability of $M$ under $D$.5

The procedure for computing $\text{DIST}(N, x)$ is identical to the procedure for computing $\text{SAMP}(N, x)$ except that it computes a distribution rather than a sample. We show only a few of the cases; the others are analogous variants of the corresponding cases in the definition of $\text{SAMP}(N, x)$.

4Suppose the set of networks which can be returned is infinite. The tree of possible computations has finite branching (each flip introduces a nondeterministic branch in the computation). By König’s lemma, any finitely branching tree with an infinite number of nodes must have an infinite path, i.e., a nonterminating computation.

5As we will see below, in the context of Bayesian networks this operation corresponds to the multiplication of factors.
In the next section we give a general method of computing distributions, which has the desired linear time performance when applied to \texttt{has-depth}(\texttt{tree}(n), n).

5 The Final Procedure

The problem with the procedure \texttt{DIST} is that it returns a distribution over very long and complicated networks. For example, as we saw above, \texttt{DIST} applied to a Bayesian network returns a distribution over networks that contain assignments to all intermediate variables, as well as to variables that were never used in the computation. More disturbingly, had \texttt{burglary} and \texttt{earthquake} relied on a common cause which does not directly affect \texttt{alarm}, the assignment to that variable would also have been part of the networks returned by \texttt{DIST}.

If we produce simpler output networks, there would also be fewer of them, so that \texttt{DIST} would have to deal with smaller distributions. Network simplification is also crucial to efficient caching and reuse of computation, the other key reason for having \texttt{SAMP} return networks rather than values was that some variable assignments are relevant for other parts of the computation. The difficulty here is correlations induced by shared inputs. Consider two variables \texttt{x} and \texttt{y} which share a common stochastic input. We can sample pairs of values for \texttt{x} and \texttt{y} (in the network \texttt{N}) by computing \texttt{SAMP(SAMP(N, x), y)}. We are interested in simplifying the intermediate network \texttt{SAMP(N, x)} in a way that preserves the information needed about the shared input.

The process of sampling \texttt{x} from \texttt{N} causes the variables above \texttt{x} in \texttt{N} to be assigned values. We must guarantee that \texttt{y} uses the same assignments for those variables that it cares about. We define the set of variables \texttt{seen by} \texttt{y} above \texttt{x} in \texttt{N}, denoted \texttt{SEENBY(y, z, N)}, as follows. If \texttt{y} is used by \texttt{x} then \texttt{SEENBY}(y, x, N) is \{\texttt{y}\}. If \texttt{y} is not used by \texttt{x} then \texttt{SEENBY}(y, x, N) is the union over variables \texttt{z} other than \texttt{x} appearing in the right hand side of the definition of \texttt{y} in \texttt{SEENBY}(z, x, N). Informally, to compute \texttt{SEENBY}(y, x, N) we "crawl up" from \texttt{y} avoiding \texttt{x} until we reach a variable used by \texttt{x}. In a Bayesian network, \texttt{SEENBY}(y, x, N) consists of the variables that are in the "fringe" of the "cone" defining \texttt{x} (the cone consisting of \texttt{x} and its ancestors); more precisely, the minimal set of variables in \texttt{x}'s cone that d-separate the cone from \texttt{y}. We use \texttt{SEENBY}(V, x, N) to denote the union over \texttt{y} \in V, of the sets \texttt{SEENBY}(y, x, N).

In computing the value of \texttt{y} from the intermediate network \texttt{SAMP(N, x)} we need only be concerned with variables used by \texttt{y} above \texttt{x}. We have the equation:

\[
\texttt{SAMP}(\texttt{SAMP}(\texttt{N}, \texttt{x}), \texttt{y}) \mid \texttt{x}, \texttt{y} = \texttt{SAMP}(\texttt{N}', \texttt{y}) \mid \texttt{x}, \texttt{y}
\]

where \texttt{N}' = \texttt{N}[\texttt{SAMP}((\texttt{N}_x), \texttt{x})] and \texttt{V} = \{x\} \cup \texttt{SEENBY}(y, x, N).

Note that the calculation of the intermediate network \texttt{N}' involves simplifying both the input network \texttt{N} to \texttt{N}_x and the output network \texttt{SAMP}((\texttt{N}_x), \texttt{x}) to \texttt{SAMP}((\texttt{N}_x), \texttt{x})\mid \texttt{y}.

We now define \texttt{PEVAL}(N, x, V) where \texttt{N} is a network, \texttt{x} is a variable in \texttt{N}, and \texttt{V} is a set of variables in \texttt{N} which contains \texttt{x}. Intuitively, like \texttt{DIST}, \texttt{PEVAL} returns a probability distribution over networks; however, in this case the networks are simplified ones, not the verbose ones returned by \texttt{SAMP}. Essentially, \texttt{PEVAL} returns networks that define only the variables in \texttt{V}. For any probability distribution \texttt{D} over networks we define \texttt{D}_\texttt{V} to be the probability distribution induced by mapping every network \texttt{N} in \texttt{D} to \texttt{N}_\texttt{V}. The function \texttt{PEVAL} satisfies the invariant \texttt{PEVAL}(N, x, V) = \texttt{DIST}(N, x)\mid \texttt{V}. For example, if \texttt{N} is the burglary-alarm network, we have that \texttt{PEVAL}(N, \texttt{alarm}, \{\texttt{alarm}\}) is a probability distribution of the form \{(\texttt{alarm} = \texttt{true}, \alpha),
which are seen by other needed variables. At each point in
recursively, uses every element of \(x = c(y)\) and \(x = f(y_r, \ldots, y_n)\).

The function \(\text{PEVAL}\) is defined recursively by the condi-
tions given below. Note that \(\text{PEVAL}\) starts by simplifying
the input network and passing the simplified network to the
"helper" function \(\text{PEHELP}\). Again, we omit the cases for
\(x = c(y)\) and \(x = f(y, z, w)\).

\[
\text{PEVAL}(N, x, V) = \text{PEHELP}(N, x, V)
\]

if \(x = c(y_1, \ldots, y_n) \in N\) then \(\text{PEHELP}(N, x, V) = \{(N\{V\}, 1)\}
\]

if \(x = c^{-1}(y) \notin N\) then \(\text{PEHELP}(N, x, V) = \frac{\text{the weighted sum over } M \text{ from } \text{PEVAL}(N, y, \text{SEENBY}(V, y, N)) \text{ of}}{\text{let } N' \text{ be } N[M] \text{ in}}
\]

if \(N'\) contains \(y = c(z_1, \ldots, z_n)\) then the weighted sum over \(M'\) from
\(\text{PEVAL}(N', z_i, \text{SEENBY}(V, z_i, N))\) of
\(\{(N'[M']|x = e)\{V, 1\}\} \text{ where } z_i = e \in M'\)

\(\text{else } \{(N'[x = \text{'false'}]V, 1)\}\}

if \(x = \text{'false'}\) \(\in N\) then \(\text{PEHELP}(N, x, V) = \{(N \{x = \text{'false'}\} V, 1)\}\)

\(\text{else } \{(N \{x = \text{'true'}\} V, 1)\}\)}

To understand this code, let us examine its behavior for
a Bayesian network.\(^7\) Consider evaluating \(\text{if}(y, z, w)\).

Here \(\text{PEVAL}(N, y, \text{SEENBY}(V, y, N))\) is a distribution
over the possible assignments to the variables in
\(\text{SEENBY}(V, y, N)\), i.e., a factor over these variables. These
variables are the fringe of \(y\)'s cone, i.e., the variables in \(y\)’s
cone that cannot be “summed out” (in a Bayesian network
algorithm) since they are used by other variables in the net-
work. Note that \(V\) contains \(x\) and therefore also \(z\) and \(w\). Thus, the assignments needed to maintain the correlations
between \(y\) and \(z, w\) are maintained in the factor. As in \(\text{DIST}\),
we now proceed to examine each network \(M\) in turn, ana-
lyzing either \(z\) or \(M\) as appropriate. In this case, however,
\(M\) is first reintegrated into the network \(N\), which contains
the part of the network eliminated in the analysis for \(y\).
The resulting factors over \(z\) and \(w\) are then multiplied by the
factor over \(y\), in the weighted sum computation.

The bulk of the computation is done over simplified net-
works. In these networks, we eliminate a large part of the
"trace" of the computation. Hence, many different compu-
tations can result in the same simplified network. For exam-
ple, in a more complicated burglary-alarm example, where
there are additional assignments on which earthquake and
burglary depend, all of these assignments are eliminated by
the simplification process, so that \(\text{PEVAL}\) always returns
a simple factor over this pair of variables. Therefore, we can
often obtain significant computational savings if we cache
the results of \(\text{PEHELP}\) applied to the various networks, and
reuse it whenever a similar call is made.

When applied to a Bayesian network, our algorithm es-
entially mimics a standard efficient inference algorithm
for Bayesian networks, one based on variable elimination
(e.g., (Dechter 1996)). It follows from our explanation of
the algorithm above that \(\text{PEVAL}(N, x, V)\) returns a distri-
bution over networks that corresponds to the factor (a
product of conditional probability tables) over \(V\) obtained
by eliminating all other variables in \(x\)'s cone. The caching
of these distributions (factors) guarantees that each one is
only computed once. It can be shown that, applied to a
Bayesian network, our algorithm mimics the standard vari-
able elimination algorithm, using the elimination ordering
implied by the lazy evaluation behavior of the algorithm.

Unfortunately, this elimination algorithm might not be
the optimal ordering for a given Bayesian network; a differ-
ent ordering might result in smaller intermediate factors. In
some cases, the predetermined elimination ordering does no
harm. In particular, we can prove that \(\text{PEVAL}\) achieves linear
time performance (modulo a small overhead for caching)
for polytree (singly connected) Bayesian networks. (See
the full paper for details.) In general, the extent to which
\(\text{PEVAL}\)'s elimination ordering is suboptimal cannot be de-
termined theoretically. However, we believe that \(\text{PEVAL}\) can
be modified to allow for more flexibility in the evaluation
order, thereby circumventing this problem. We are in the
process of investigating such an extension.

On the other side, \(\text{PEVAL}\) is significantly more flexible
than the standard Bayesian network inference algorithms.
The algorithm automatically exploits both the causal inde-
dependence induced by noisy-or interactions and the context-
specific independence induced by tree-structured condi-
tional probability tables, which have been shown to sup-
port more efficient inference (Heckerman and Breese 1994;
Boutilier et al. 1996).

The algorithm \(\text{PEVAL}\), augmented with caching, auto-
matically induces efficient algorithms for many problems.
For example, the calculation of \(\text{PEVAL}\) over the network
\(\text{has-depth}(\text{tree}(), n)\) calls \(\text{PEHELP}\) on a linear number of
networks representing expressions of the form

\(\text{has-depth}(\text{left}(\text{pair}(\text{tree}(),\text{tree}())), n)\); and

a similar number for expressions of the form

\(\text{has-depth}(\text{right}(\text{pair}(\text{tree}(),\text{tree}())), n)\);

Each such call returns a distribution over two networks repre-
senting the values \('\text{true}'\) and \('\text{false}'\). The total number
of calls is linear in \(n\).

As another example consider the \(\text{match-suffix}\) func-
tion from Section 2. If \(t\) is a parse tree (generated
by a SCFG) and \(s\) is a string of terminal symbols then
\(\text{match-suffix}(t, s)\) returns \(\text{'false'}\) if the fringe of \(t\) is
not a suffix of \(s\) and otherwise returns the prefix of \(s\) that
results from removing the fringe of \(t\) form the end of \(s\).
We can compute the probability that a given grammar generates
the string \(s\) by evaluating

\(\text{if}(\text{match-suffix}(S(), s) = \text{'true'}, \text{'false'})\).

This procedure does a case analysis on the tree \(t\). In the

\(^7\) In this discussion, we utilize some standard terminology from
Bayesian network inference. Space constraints prohibit us from
providing a full explanation. We hope that the main ideas will be
clear even to readers who are unfamiliar with these concepts.
case where $t$ is a tree with left and right subtrees $l$ and $r$ respectively the procedure evaluates networks which have essentially the following form.

$$
\begin{align*}
  s' &= \text{match-suffix}(r, \text{cdr}(s)) \\
  s^2 &= \text{unsn}(\text{car}(s), s') \\
  \text{if}(t) &= \text{'false, 'false, match-suffix(l, s')})
\end{align*}
$$

The inputs to this network are $s$, $l$ and $r$. Here $l$ and $r$ will be nonterminals form a grammar. So for a fixed grammar there are only $O(1)$ possible values of $l$ and $r$. But there are $O(n^2)$ possible values of $s$. Hence the total number of networks of this form which need to be evaluated is $O(n^6)$. The evaluation of the if expression at the bottom of the network will iterate over the values of $s'$, and for each value other than 'false, will then iterate over the values of match-suffix($l$, $s'$). There are $O(n)$ possible values of $s'$ and for each of these $O(n)$ possible values of the match-suffix($l$, $s'$). So the number of operations involved in evaluating each of these networks is $O(n^5)$. This gives a total number of operations is $O(n^4)$.

Alternatively, one could construct a procedure for testing whether the fringe of $t$ equals the string $s$ by iterating through the ways of splitting $s$ into two nonempty substrings $s_1$ and $s_2$ and recursively testing if the fringe of the left branch of $t$ is $s_1$ and the fringe of the right branch of $t$ is $s_2$. This procedure is efficient if $t$ is a fixed tree. However, it produces the correct answer. Furthermore, it runs in $O(n^3)$ operations when used to compute the probability that the fringe of a parse tree generated by a given SCFG will be a given string. This computation is essentially the inside algorithm for SCFGs. It is interesting to note that the analysis remains polynomial time under a variety of implementations of tests for $t$ having fringe $s$.

### 6 Conclusion

We have presented a powerful language for representing stochastic processes, and an efficient Bayesian inference algorithm for models specified in this language. We have also shown that, via its use of independence and caching, our algorithm mimics several efficient inference algorithms for special-purpose representation languages. We can easily imagine the language being used to represent more complex models than the ones discussed in the paper. For example, it is easy to represent Bayesian networks with defined subnetworks which call each other recursively (Koller and Pfeffer 1997). Our approach thus provides clear and coherent semantics for hierarchically structured Bayesian networks, as well as an effective inference algorithm that exploits the existence of repeated network fragments to speed up inference. Our approach can also be used to encode stochastic versions of richer grammars, including context-sensitive grammars, and grammars in which attributes are passed to non-terminal symbols via productions. We believe that our algorithm will transfer well to new models; and while it is unlikely to be the most efficient algorithm for all these model classes, it will provide a useful starting point for studying them.

**Acknowledgements** We thank Fernando Pereira and Lewis Stiller for useful discussions. Some of this work was done while Daphne Koller and Avi Pfeffer were visiting AT&T. This work was also supported through the generosity of the Powell Foundation, by ONR grant N00014-96-1-0718, and by DARPA contract DACA76-93-C-0025, under subcontract to Information Extraction and Transport, Inc.

**References**