In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 740-747, Providence, Rhode Island, August 1997

# 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 is "run" 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. (I.e., our inference algorithm terminates precisely when the stochastic program does.) 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.

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

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, 'bar)` denotes a "cons cell", i.e., a data structure of type `'cons`. The first field of this cons cell contains the constant `'foo` and the second field contains the constant `'bar`. All values in our language are either symbolic constants or data structures whose fields contain (recursively) values. An expression of the form $c_i^{-1}(e)$ extracts the 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 $\texttt{'cons}_1^{-1}$ and $\texttt{'cons}_2^{-1}$ correspond to the Lisp functions `car` and `cdr` respectively. We use the latter as shorthand in some of our code. The expression $c?(e)$ 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 $\texttt{if}(e_1,\ e_2,\ e_3)$ 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 $\texttt{flip}(\alpha)$ are stochastic. The expression $\texttt{flip}(\alpha)$ 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 = flip(.5); y = 'pair($x$, $x$); z = 'pair(flip(.5), flip(.5));

The variables $y$ and $z$ 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 $x_1 = e_1; x_2 = e_2; \ldots; x_n = e_n$;.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 $\langle D,\ N\rangle$ 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 $\langle D,\ N\rangle$ is the value of the last assignment in $N$. If the program diverges, it takes the value $\perp$.

A program is a stochastic model — it defines a probability distribution over the value 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

```
digit() = {output = if(flip(.5), 'one, 'zero);}
real() = {output = 'cons(digit(), real());}
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 = real(); output = if(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, a simple burglar-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 `output = 'vlist(`$x_1, \ldots, x_n$`)` 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 $\Pr(P \mid Q)$, where $P$ and $Q$ are observations about the function output, using the simple program

```
x = f(); output = if(Q(x), if(P(x), 'q-and-p, 'q-and-not-p), 'not-q);
```

The probability of the value `'q-and-p` equals $\Pr(P(x) \wedge Q(x))$ and similarly for the other values, so from `'q-and-p` and `'q-and-not-p` we can compute $\Pr(P(x) \mid Q(x))$.

In traditional Bayesian networks, the conditional probability tables contain an entry for every combination of values of a node and its parents. There has been much work on more compact representations of conditional probability tables, such as noisy-or models (Pearl 1988) and trees (Boutilier *et al.* 1996). The latter can be used to model situations where two variables are independent of each other given some values of a third variable and not others. Our language easily expresses both these representations.

Our language also supports significant and interesting extensions to the Bayesian network formalism. The basis is the observation that each node in a Bayesian network can be viewed as a stochastic function of its parents' values.[1] Thus, we can create a user-defined function representing a node. We can compose these functions, resulting in more complex functions that represent an entire network fragment with multiple inputs and multiple outputs. As we have recently shown (Koller and Pfeffer ), this capability provides the foundation for the definition of a *hierarchical* and even an *object-oriented* Bayesian network. For example, we can easily model fault diagnosis in component hierarchies (as in (Srinivas 1994)), where the inputs to a high-level component are passed to its subcomponents, which in turn return their output value.

Our language can be used to extend the framework of Bayesian networks even further. For example, we can easily model Bayesian networks where one function recursively calls another (or itself); our lazy semantics will provide semantics to such networks even when the recursion is infinite. A similar idea can be used to describe complex Markov processes. In a dynamic belief network (Dean and Kanazawa 1989), the state of the world at one instant in time is a stochastic function of the state at the previous instant. We can model this as a user-defined function that takes one state as an input and outputs the new state.

The expressive power of our language is not restricted to extensions of Bayesian networks. For example, it easily models *stochastic context-free grammars (SCFG)*, a formalism which has been used in statistical natural language processing (Charniak 1993) and understanding biological structures (Sakakibara *et al.* 1995). A stochastic context-free grammar (SCFG) is the natural probabilistic extension of a context-free grammar. It contains sets of non-terminal and terminal symbols, where each non-terminal symbol is associated with a set of productions which transform it into strings of terminals and non-terminals. In a SCFG, we also have a probability distribution over the set of transitions associated with each nonterminal. The following is a simple example of a SCFG:

| | | | | | |
|---|---|---|---|---|---|
| $S \to AB$ | $(0.6)$ | $A \to BA$ | $(0.3)$ | $B \to AB$ | $(0.2)$ |
| $S \to BA$ | $(0.4)$ | $A \to a$ | $(0.7)$ | $B \to b$ | $(0.8)$ |

A SCFG describes a stochastic model in which non-terminal symbols are successively replaced by the right-hand side of a production with the given probability, thus generating a probability distribution over strings. A SCFG can be represented in our language by using a defined function for each non-terminal. For example, the program

```
S() = {output = if(flip(0.6), 'cons(A(), B()), 'cons(B(), A()));}
A() = {output = if(flip(0.3), 'cons(B(), A()), 'cons('a, 'nil));}
B() = {output = if(flip(0.2), 'cons(A(), B()), 'cons('b, 'nil));}
output = S();
```

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:[2]

```
match-suffix(t,s) =
  {output = if('nil?(t), s,
               if('nil?(s), 'false,
                  if('cons?(t), match-help(t,s),
                     if(t == last(s), butlast(s), 'false))));}
match-help(t,s) =
  {s' = match-suffix(cdr(t), cdr(s));
   output = if(s' == 'false, 'false,
               match-suffix(car(t), 'cons(car(s), s'))}
```

The function `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 `'false`. If $t$ is a parse for the entire string $s$, `match-suffix` will return `nil`.

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

[2] We assume that `last` and `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.

## 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 subexprexpressions of $e$ other than variables. For example, $x = \texttt{if}(y,\ z,\ w)$ is shallow but $x = \texttt{if}(\texttt{flip}(\alpha),\ z,\ w)$ 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:

earthquake = flip(0.01); burglary = flip(0.1); a-when-e-b = flip(0.99);
a-when-e-nb = flip(0.2); a-when-ne-b = flip(0.98);
a-when-ne-nb = flip(0.01); a-when-e = if(burglary, a-when-e-b, a-when-e-nb);
a-when-ne = if(burglary, a-when-ne-b, a-when-ne-nb);
alarm = if(earthquake, a-when-e, a-when-ne);

Our sampling algorithm is formulated as a recursive function SAMP which takes as input a network $N$ and a variable $x$ occuring in $N$ and produces as output a new network $N'$ where $N'$ contains an assignment of the form $x = c(y_1,\ \ldots,\ y_n)$. 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 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 SAMP is defined by the following conditions.

if $x = c(y_1,\ \ldots,\ y_n) \in N$ then SAMP($N$, $x$) = $N$.
if $x = c_i^{-1}(y) \in N$ then to compute SAMP($N$, $x$):
  let $N'$ be SAMP($N,\ y$)
  if $N'$ contains $y = c(z_1,\ \ldots,\ z_n)$
    then let $N''$ be SAMP($N',\ z_i$)
    and return $N''[x = v]$ where $z_i = v \in N''$
  else return $N'[x$='false]
if $x = c?(y) \in N$ then to compute SAMP($N$, $x$):
  let $N'$ be SAMP($N,\ y$)
  if $N'$ contains $y = c(z_1,\ \ldots,\ z_n)$
    then return $N'[x$='true]
    else return $N'[x$='false]
if $x$ = flip($\alpha$) $\in N$ then
  SAMP($N$, $x$) = $N[x$='true] with probability $\alpha$
  SAMP($N$, $x$) = $N[x$='false] with probability $1 - \alpha$
if $x$ = if($y, z, w$) $\in N$ then to compute SAMP($N,\ x$):
  let $N'$ be SAMP($N$, $y$)
  let $h$ be the variable $z$ if $y$ = 'true $\in N'$ and the variable $w$ otherwise
  let $N''$ be SAMP($N'$, $h$)
  SAMP($N$, $x$) = $N''[x = v]$ where $h = v \in N''$
if $x = f(y_1,\ \ldots,\ y_n) \in N$ then to compute SAMP($N,\ x$):
  let $M$ be the body of the definition of $f$ where the inputs
    have been renamed to $y_1, \ldots, y_n$ and all other variables
    renamed to fresh variables.
  let $N'$ be $N$ where $x = f(y_1,\ \ldots,\ y_n)$ is replaced by $M; x = v$
    where $o = v$ is the output statement in $M$.
  return 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, 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, SAMP ($N$, alarm) may return (as one possible outcome):

earthquake = 'true; burglary = 'false; a-when-e-b = flip(0.99);
a-when-e-nb = 'true; a-when-ne-b = flip(0.98);
a-when-ne-nb = flip(0.01); a-when-e = 'true;
a-when-ne = if(burglary, a-when-ne-b, a-when-ne-nb); alarm = 'true;

As a more complex example, consider the function `tree`

tree() = { flp = flip(.4); x = tree(); y = tree(); lf = 'leaf; pr = 'pair(x, y);
  output = if(flp, lf, pr);}

Let $N$ be the network consisting of the single assignment `output=tree()` and consider computing a value of SAMP($N$, `output`). The last case of the procedure applies and we continue by computing SAMP($N'$, `output`) where $N'$ is the following network.

flp = flip(.4); x = tree(); y = tree(); lf = 'leaf; pr = 'pair(x, y);
output = if(flp, lf, pr);

Now the conditional rule applies and we compute SAMP($N'$, `flp`). Depending on whether the flip yields `'true` or `'false` we either evaluate SAMP($N''$, `lf`) or SAMP($N''$, `pr`) where $N''$ is a network with the corresponding value for `flp`. In the case where `flp` is `'true`, the final network contains `output = 'leaf`. In the case where `flp` is `'false`, the final network contains `output = 'pair`$(x,\ y)$. In this example, the complete value may be infinite, but the lazy evaluation process only resolves values to the point where the top level constructor is known.

This semantics, whereby only the top level constructor of the value of a program is determined, does not prevent us from formulating nontrivial queries. The following program, for example, uses the procedure `has-depth` which determines whether the depth of a given tree is $\leq n$:[3]

has-depth(t, n) = {if('leaf?(t), 'true,
      if(zero?(n), 'false,
        and(has-depth(left(t), pred(n)),
          has-depth(right(t), pred(n)))))}
x = tree(); output=has-depth(x, 10);

This network has the two possible values `'true` and `'false`. But in order to determine the value, the sampling algorithm will have to "open up" the value of $x$, until it verifies whether its depth is more than 10 or not.

Note that the sampling algorithm SAMP ($N$, $x$) returns a network rather than a value for $x$. This property allows us to sample multiple variables defined in the same program. To understand this issue, consider a network $N$ of the form $x = \texttt{flip}(\alpha)$; $N_y$; $N_z$; $N_w$; $o = \texttt{if}(y,\ z,\ w)$ where $N_{var}$ is a network defining $var$. Assume that the variable $x$ is used in each of the networks $N_y$, $N_z$ and $N_w$. Now, computing SAMP($N$, $y$) can result in $x$ being sampled and assigned a value. If SAMP($N$, $y$) returned a value rather

[3]We assume that `zero?`, `and`, `pred`, `left`, and `right` have been given appropriate definitions.

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 SAMP$(N,\ z)$ 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 $\langle N,\ x\rangle$ 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 SAMP$(N,\ x)$ to diverge. Note that, due to the use of lazy evaluation, certain expressions, such as `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 $\langle N,\ x\rangle$, the set of networks which can be returned by 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 $\{\langle v_1,\ \alpha_1\rangle,\ \ldots,\ \langle v_n,\ \alpha_n\rangle\}$ where all $v_i$ must be distinct, all $\alpha_i$ must be real numbers in the interval $[0,\ 1]$, and the sum of all $\alpha_i$ must be 1. This table denotes the distribution where item $v_i$ has probability $\alpha_i$.

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 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 `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 DIST$(N,\ x)$ is identical to the procedure for computing 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 SAMP$(N,\ v)$.

if $x = c(y_1,\ \ldots,\ y_n) \in N$ then DIST$(N, x) = \{\langle N,\ 1\rangle\}$.
if $x = c_i^{-1}(y) \in N$ then DIST$(N, x)$ is
  the weighted sum over $N'$ from DIST$(N,\ y)$ of
    if $N'$ contains $y = c(z_1,\ \ldots,\ z_n)$
      then the weighted sum over $N''$ from DIST$(N',\ z_i)$ of
        $\{\langle N''[x = v],\ 1\rangle\}$ where $z_i = v \in N''$
      else $\{\langle N''[x = \texttt{`false}],\ 1\rangle\}$
if $x = \text{flip}(\alpha) \in N$ then DIST$(N,\ x)$ is
  $\{\langle N[x = \texttt{`true}],\ \alpha\rangle,\ \langle N[x = \texttt{`false}],\ 1-\alpha\rangle\}$
if $x = \text{if}(y, z, w) \in N$ then DIST$(N,\ x)$ is
  the weighted sum over $N'$ from DIST$(N, y)$ of
    let $h$ be $z$ if $y$ = 'true $\in N'$ and $w$ otherwise in
      the weighted sum over $N''$ from DIST$(N', h)$ of
        $\{\langle N''[x = v],\ 1\rangle\}$ where $h = v \in N''$

For example, when applied to our burglar-alarm network, DIST$(N,$ `alarm`) will start by evaluating the definition of `alarm` using the rule for evaluating `if`. The first step is the evaluation of DIST$(N,$ `earthquake`). This step generates two networks with weights: one network $N_t$ identical to $N$ except that the `earthquake` is assigned `` `true ``, and one identical to $N_f$ except that `earthquake` is assigned `` `false ``; $N_t$ has weight 0.01 and $N_f$ weight 0.99. The algorithm proceeds to evaluate DIST$(N_t,$ `a-when-e`) and DIST$(N_f,$ `a-when-ne`). The first of these results in a call to DIST$(N_t,$ `burglary`), which also returns in two networks each with its own weight—0.1 and 0.9 respectively. The second of these results in a separate but analogous call to DIST$(N_f,$ `burglary`) with similar output. These three distributions are then combined using the weighted sum operation to result in a distribution over four networks, corresponding to the four possible assignments to the variables `earthquake` and `burglary`.

As a result of the close parallel between the computations of `DIST` and of `SAMP`, we have the following theorem:

**Theorem:** The computation of DIST$(N,\ x)$ terminates exactly when the pair $\langle N,\ x\rangle$ terminates.

The procedure `DIST` is very inefficient. To understand why, consider the expression has-depth(tree(), $n$) for a given value of $n$. `DIST` returns a distribution over the networks returned by `SAMP`; `SAMP` gradually "unrolls" `tree()`, opening up recursive calls, and assigning values to the variables needed to determine the value of the computation. The networks returned contain variables for all the intermediate calculations used. The output networks always contain enough detail of the computation to determine the value of `has-depth`. The number of such verbose networks which are possible outputs of has-depth(tree(), $n$) is exponential in $n$. Therefore, its analysis using `DIST` takes exponential time and returns a distribution over an exponential number of networks. This is disappointing, since the distribution can actually be computed quite easily: for $n > 0$ the probability $\beta_n$ that has-depth(tree(), $n$) is `` `true `` is just $\alpha + \beta_{n-1}^2$ where $\alpha$ is the probability that tree() returns `` `leaf `` and $\beta_{n-1}$ is the probability that has-depth(tree(), $n-1$) is `` `true ``. Hence, there exists a method of computing a distribution over the value of has-depth(tree(), $n$) which runs in time linear in $n$.

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

[5] As 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 $\texttt{has-depth}(\texttt{tree}(),\ n)$.

## 5 The Final Procedure

The problem with the procedure `DIST` is that it returns a distribution over very long and complicated networks. For example, as we saw above, `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 `burglary` and `earthquake` relied on a common cause which does not directly affect `alarm`, the assignment to that variable would also have been part of the networks returned by `DIST`.

If we produce simpler output networks, there would also be fewer of them, so that `DIST` would have to deal with smaller distributions. Network simplification is also crucial to efficient caching and reuse of computation, the other key to getting an efficient inference algorithm. We now show how to simplify both the networks provided as input to `SAMP` and the ones it returns as output (which are the ones over which `DIST` generates a distribution).

We say that a variable $x$ *uses* a variable $y$ in network $N$ if either $y$ is $x$ (every variable uses itself) or some variable on the right hand side of the assignment to $x$ recursively uses $y$. For any network $N$ and set of variables $V$ we define $N|_V$ to be the set of assignments in $N$ to variables used by variables in $V$. In the case of a Bayesian network, $N|_V$ includes the definitions of the variables in $V$ and of their ancestors in the network. For example, if $N$ is `z = 'a; y = 'b; x = 'f(y);` then $N|_{\{x\}}$ is `y = 'b; x = 'f(y);` The restriction operation $N|_V$ is our tool for simplifying networks.

For any basic networks $N$ and $M$ we let $N[M]$ be the network $M$ (as a set of assignments) plus those assignments in $N$ to variables unassigned in $M$. For example,

{x=flip(.4); y=flip(.1); z='cons(x,y);}[{x='true; z='false}]

is the network {`x='true; y=flip(.1); z='false;`}. One should think of $N[M]$ as a generalization of the notation $N[x=v]$ used in the procedure `SAMP`. Intuitively, $N[M]$ is the network $N$ modified by the more refined values in $M$.

Note that the computation of $\texttt{SAMP}(N,\ x)$ only assigns values to variables used by $x$. Thus, the effect of the sampling is contained within the subnetwork $N|_{\{x\}}$. The function `SAMP` satisfies the following equation:[6]

$$\texttt{SAMP}(N,\ x) = N[\texttt{SAMP}(N|_{\{x\}},\ x)]$$

The above equation allows the input network $N$ to be simplified to $N|_{\{x\}}$ before being passed as an argument to `SAMP`. The top level network is then modified to incorporate the result of $\texttt{SAMP}(N|_{\{x\}},\ x)$. Note that we may have to incorporate variables that were not present before, i.e., those originating from unrolling a user-defined function. Thus, for example, we may have {`y=g(); x=f();`}[{`x=c(z); z=h();`}], which is is defined to be {`y=g(); x=c(z); z=h();`}.

It is also possible to simplify output networks. The output networks are more complicated than necessary because they include all intermediate values, some of which may no longer be needed. However, we cannot consider just the output value of the network; after all, the whole reason for having `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 $x$ and $y$ which share a common stochastic input. We can sample pairs of values for $x$ and $y$ (in the network $N$) by computing $\texttt{SAMP}(\texttt{SAMP}(N,\ x),\ y)$. 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 $x$ from $N$ causes the variables above $x$ in $N$ to be assigned values. We must guarantee that $y$ uses the same assignments *for those variables that it cares about*. We define the set of variables *seen by* $y$ above $x$ (in network $N$), denoted $\texttt{SEENBY}(y,\ x,\ N)$, as follows. If $y$ is used by $x$ then $\texttt{SEENBY}(y,\ x,\ N)$ is $\{y\}$. If $y$ is not used by $x$ then $\texttt{SEENBY}(y,\ x,\ N)$ is the union over variables $z$ other than $x$ appearing in the right hand side of the definition of $y$ of $\texttt{SEENBY}(z,\ x,\ N)$. Informally, to compute $\texttt{SEENBY}(y,\ x,\ N)$ we "crawl up" from $y$ avoiding $x$ until we reach a variable used by $x$. In a Bayesian network, $\texttt{SEENBY}(y,\ x,\ N)$ consists of the variables that are in the "fringe" of the "cone" defining $x$ (the cone consisting of $x$ and its ancestors); more precisely, the minimal set of variables in $x$'s cone that d-separate the cone from $y$. We use $\texttt{SEENBY}(V,\ x,\ N)$ to denote the union, over $y \in V$, of the sets $\texttt{SEENBY}(y,\ x,\ N)$.

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

$$\texttt{SAMP}(\texttt{SAMP}(N,\ x),\ y)|_{\{x,\ y\}} = \texttt{SAMP}(N',\ y)|_{\{x,\ y\}}$$
$$\text{where } N' = N[\texttt{SAMP}(N|_{\{x\}},\ x)|_V]$$
$$\text{and } V = \{x\} \cup \texttt{SEENBY}(y,\ x,\ N).$$

Note that the calculation of the intermediate network $N'$ involves simplifying both the input network $N$ to $N|_{\{x\}}$ and the output network $\texttt{SAMP}(N|_{\{x\}},\ x)$ to $\texttt{SAMP}(N|_{\{x\}},\ x)|_V$.

We now define $\texttt{PEVAL}(N,\ x,\ V)$ where $N$ is a network, $x$ is a variable in $N$, and $V$ is a set of variables in $N$ which contains $x$. Intuitively, like `DIST`, `PEVAL` returns a probability distribution over networks; however, in this case the networks are simplified ones, not the verbose ones returned by `SAMP`. Essentially, `PEVAL` returns networks that define only the variables in $V$. For any probability distribution $D$ over networks we define $D|_V$ to be the probability distribution induced by mapping every network $N$ in $D$ to $N|_V$. The function `PEVAL` satisfies the invariant $\texttt{PEVAL}(N,\ x,\ V) = \texttt{DIST}(N,\ x)|_V$. For example, if $N$ is the burglar-alarm network, we have that $\texttt{PEVAL}(N,\ \texttt{alarm},\ \{\texttt{alarm}\})$ is a probability distribution of the form $\{\langle$`alarm='true`, $\alpha\rangle$,

[6] This is an equation between expressions which sample distributions. The intended meaning of the equation is that the two sampling expressions are equivalent — a given value has the same probability of being generated when a sample is drawn from either of the two expressions. Other equations between sampling expressions are given below with the same intended meaning.

$\langle \texttt{alarm='false},\ 1-\alpha\rangle\}$ (as compared to the cumbersome distributions returned by DIST).

Initially, $V$ contains only $x$. As the function is called recursively, $V$ is increased to contain additional variables which are seen by other needed variables. At each point in the process, we maintain the invariant that $x \in V$ and that $x$ uses every element of $V$.

The function PEVAL is defined recursively by the conditions given below. Note that PEVAL starts by simplifying the input network and passing the simplified network to the "helper" function PHELP. Again, we omit the cases for $x = c?(y)$ and $x = f(y_1,\ \ldots,\ y_n)$.

PEVAL$(N,\ x,\ V)$ = PHELP$(N|_{\{x\}},\ x,\ V)$

if $x = c(y_1,\ \ldots,\ y_n) \in N$ then PHELP$(N,\ x,\ V) = \{\langle N|_V,\ 1\rangle\}$
if $x = c_i^{-1}(y) \in N$ then PHELP$(N, x, V)$ is
  the weighted sum over $M$ from PEVAL$(N,\ y,\ \texttt{SEENBY}(V,\ y,\ N))$ of
    let $N'$ be $N[M]$ in
      if $N'$ contains $y = c(z_1,\ \ldots,\ z_n)$
        then the weighted sum over $M'$ from
          PEVAL$(N',\ z_i,\ \texttt{SEENBY}(V,\ z_i,\ N))$ of
          $\{\langle N'[M'][x = e]|_V,\ 1\rangle\}$ where $z_i = e \in M'$
        else $\{\langle N'[x = \texttt{`false}]|_V,\ 1\rangle\}$
if $x = \texttt{flip}(\alpha) \in N$ then
  PHELP$(N,\ x,\ V) = \{\langle (x = \texttt{`true}),\ \alpha\rangle,\ \langle (x = \texttt{`false}),\ 1-\alpha\rangle\}$.
if $x = \texttt{if}(y,\ z,\ w) \in N$ then PHELP$(N,\ x,\ V)$ is
  the weighted sum over $M$ from PEVAL$(N,\ y,\ \texttt{SEENBY}(V,\ y,\ N))$ of
  let $N'$ be $N[M]$ in
    let $h$ be $z$ if $y$ = 'true $\in N'$ and $w$ otherwise in
      the weighted sum over $M'$ from PEVAL$(N,\ h,\ \texttt{SEENBY}(V,\ h,\ N'))$ of
        $\{\langle N'[M'][x = e]|_V,\ 1\rangle\}$ where $h = e \in M'$.

To understand this code, let us examine its behavior for a Bayesian network.[7] Consider evaluating $\texttt{if}(y,\ z,\ w)$. Here PEVAL$(N,\ y,\ \texttt{SEENBY}(V,\ y,\ N))$ is a distribution over the possible assignments to the variables in 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 network. 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 DIST, we now proceed to examine each network $M$ in turn, analyzing either $z$ or $w$, 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 networks. In these networks, we eliminate a large part of the "trace" of the computation. Hence, many different computations can result in the same simplified network. For example, in a more complicated burglar-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 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 PHELP applied to the various networks, and reuse it whenever a similar call is made.

[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.

When applied to a Bayesian network, our algorithm essentially 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 PEVAL$(N, x,\ V)$ returns a distribution 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 variable 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 different ordering might result in smaller intermediate factors. In some cases, the predetermined elimination ordering does no harm. In particular, we can prove that 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 PEVAL's elimination ordering is suboptimal cannot be determined theoretically. However, we believe that 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, PEVAL is significantly more flexible than the standard Bayesian network inference algorithms. The algorithm automatically exploits both the *causal independence* induced by noisy-or interactions and the *context-specific independence* induced by tree-structured conditional probability tables, which have been shown to support more efficient inference (Heckerman and Breese 1994; Boutilier *et al.* 1996).

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

has-depth(left('pair(tree(), tree())), n);

and a similar number for expressions of the form

has-depth(right('pair(tree(), tree())), n).

Each such call returns a distribution over two networks representing the values `true and `false. The total number of calls is linear in $n$.

As another example consider the match-suffix function from Section 2. If $t$ is a parse tree (generated by a SCFG) and $s$ is a string of terminal symbols then match-suffix$(t,\ s)$ returns `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

if(match-suffix(S(),s) == 'nil, 'true, 'false).

This procedure does a case analysis on the tree $t$. In the 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.

```
s' = match-suffix(r, cdr(s))
s" = cons(car(s), s")
if(s' == 'false, 'false, match-suffix(l, s")
```

The inputs to this network are $s$, $l$ and $r$. Here $l$ and $r$ will be nonterminals form the 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 $O(n^2)$. 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 $O(n^2)$. 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 inefficient 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 a 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 implementions 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 that call each other recursively (Koller and Pfeffer ). 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 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

C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *UAI*, 1996.

E. Charniak. *Statistical Language Learning*. MIT Press, 1993.

T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *AAAI*, 1993.

R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *UAI*, 1996.

P. Haddawy. Generating bayesian networks from probability logic knowledge bases. In *UAI*, 1994.

D. Heckerman and J. S. Breese. A new look at causal independence. In *UAI*, 1994.

D. Heckerman, J. Breese, and K. Rommelse. Decision-theoretic troubleshooting. *CACM*, 38(3):49–57, 1995.

D. Koller and A. Pfeffer. Object-oriented bayesian networks. Submitted for publication.

K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

J. Pearl. A probabilistic calculus of actions. In *UAI*, 1994.

D. Poole. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64(1), November 1993.

D. V. Pynadath and M. P. Wellman. Generalized queries in probabilistic context-free grammars. In *AAAI*, 1996.

L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, January 1986.

Y. Sakakibara, M. Brown, R. C. Underwood, I. S. Mian, and D. Haussler. Stochastic context-free grammars for modeling RNA. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, 1995.

P. Smyth, D. Heckerman, and M. Jordan. Probabilistic independence networks for hidden Markov probability models. MSR-TR-96-03, Microsoft Research, 1996.

S. Srinivas. A probabilistic approach to hierarchical model-based diagnosis. In *UAI*, 1994.