Pless, D., Luger, G.F., and Stern, C.R. "A new Object-Oriented Stochastic Modeling Language". In *Proceedings of the IASTED International Conference*. Zurich: IASTED/ACTA Press (2000).

A NEW OBJECT-ORIENTED STOCHASTIC MODELING LANGUAGE

DAN PLESS & GEORGE LUGER
Department of Computer Science,
University of New Mexico
Albuquerque, NM 87131

CARL STERN
SandiaView Software
1009 Bradbury Dr. SE
Albuquerque, NM 87106

ABSTRACT

A new language and inference algorithm for stochastic modeling is presented. This work refines and generalizes the stochastic functional language originally proposed by [1]. The language supports object-oriented representation and recursive functions. It provides a compact representation for a large class of stochastic models including infinite models. It provides the ability to represent general and abstract stochastic relationships and to decompose large models into smaller components. Our work extends the language of [1] by providing object encapsulation and reuse and a new and effective strategy for caching. An exact and complete inference algorithm is presented here that is expected to support efficient inference over important classes of models and queries.

KEYWORDS

Bayesian Networks, Stochastic Modeling, Probabilistic Reasoning

INTRODUCTION

This paper describes a new object-oriented stochastic modeling language. The language is capable of representing a larger class of models than those expressible as Bayesian Networks. It supports a powerful form of object-oriented representation, allowing general probabilistic relationships over object classes to be expressed and computed. The inference algorithm for this language achieves efficiency through modular representation, lazy evaluation, goal-directed inference, and compact factoring of conditional probability tables.

The limitations of flat Bayesian Networks that use simple random variables has been noted by other researchers [2, 3]. These limitations have motivated a variety of recent research in hierarchical and composable Bayesian models [4, 5, 6, 7]. Most of these new Bayesian modeling formalisms support model decomposition, often based on an object-oriented approach. While these provide more expressive and succinct representational frameworks, few of these change the class of models that may be represented.

One important exception is the object-oriented functional stochastic modeling language proposed by [1]. Their language provides the ability to use objects and functions to represent general stochastic relationships. It provides Turing completeness by allowing the construction of objects representing infinite classes. Furthermore, they define an inference algorithm supporting effective computation over models embedding such objects through the use of lazy evaluation.

Our work extends and refines this proposed framework in a number of crucial ways. Their language has been modified to enhance usability and to support a more powerful object system. The objects defined by [1] are limited and provide no obvious way to encapsulate model components. On the other hand, the objects defined in our language provide the important capability of model encapsulation and reuse. We have also modified the language to support a more efficient implementation of the inference algorithm. The algorithm presented by [1] depends for its efficiency on the use of caching to avoid redundant computation. Unfortunately, caching in their language is difficult if not impossible to implement efficiently because it requires the recognition and retrieval of similar networks. The algorithm for our language requires only that identical networks be recognized and retrieved from the cache.

As just noted, our language is Turing complete. It supports construction of objects that represent infinite classes. This can be useful for pattern recognition and language processing. An example in the next section shows how this feature can be used to produce and recognize classes of expressions generated by stochastic context free grammars.

The next section presents the language with its component features and two examples. The following section describes the inference algorithm. It is presented incrementally in terms of inference algorithms for three languages of increasing complexity. The first language is equivalent in power to standard Bayesian Networks while the last is a Turing complete stochastic modeling language. Finally, implementation and efficiency issues are discussed., These include effective implementation of caching and lazy evaluation.

DESCRIPTION OF THE LANGUAGE

The elements of the outer language include variables, compound structures called objects, attribute chains, and functions. Statements in the language include assignment statements and object and function definitions. Two constructs, dist and case, are used to define new or dependent distributions.

A model is defined by a list of assignments to variables surrounded by brackets. The following is a simple network model with three variables, x, y, and z:

```
[x = dist true: 0.6, false: 0.4
 y = dist true: 0.6, false: 0.4
 z = x]
```

From the first assignment, the value of x is true with a probability of 0.6 and false with a probability of 0.4. Any value between 0 and 1 can be used as a probability in a dist statement as long as the distribution sums to 1. The second assignment does the same for y. However x and y remain statistically uncorrelated despite sharing the same probability distribution. The third assignment gives z the same value as x, thereby making x and z completely correlated in the model.

Objects in this language add significant expressive power. Object definitions assign a model to a variable. The object's model consists of a series of assignments to variables. The variables on the left-hand side of these assignments comprise the attributes of the object.

```
[x = dist true: 0.3, false: 0.7
 y = dist true: 0.4, false: 0.6
 o = [head = x   tail = y]
 z = o.head]
```

Here the variable o is an object with two attributes: head and tail. The head and tail attributes in the object o are bound to values x and y respectively. The attributes of o can be accessed with an attribute chain as shown in the assignment to z. An attribute chain is a list of symbols separated by periods that indicate a path through a series of embedded objects.

The language includes a case statement that is a generalization of the if statement in [1]. Here is an example using a case statement:

```
[x = dist true: 0.6, false: 0.4
 y = dist true: 0.6, false: 0.4
 r = case x true: y, false: false]
```

The case statement in the definitions of r works similarly to case statements in other languages. When x has value true, then r has the same value as y, and when it is false, r is also false.

One can assign an object to a variable, for example z2 = o. Since variables can be distributions, they can also be distributions over any combination of symbols and objects. The language also supports the reuse of object definitions with the copy statement. z2 = copy o is an example. Here copy copies the definition of o into the current scope, creating a new uncorrelated instance of o.

There are a number of consequences deriving from the fact that models and objects share the same structural form. It means that one can model a piece of the domain of interest and then incorporate that model as an object into a larger model without modification.

The syntax for function definitions and calls is illustrated in the following example:

```
[or(a, b) = case a  true: true,  false: b
 x = dist true: 0.3, false: 0.7
 y = dist true: 0.4, false: 0.6
 z = or(x, y)]
```

The first assignment in the list contains the definition of the function or. Parentheses surrounding a set of arguments indicates a function. or takes two arguments and implements the *or* operation using a case statement.

In the outer language, function calls and case statements may be nested. dist statements may be nested within function calls and case statements but not vice versa. Objects too may be nested in function calls and case statements.

The inner language is identical to the outer language with the following restrictions. There are no functions, but the same capabilities can be obtained using objects as explained below. The rules for the nesting of elements are also much more limited. Finally, the inner language supports only a restricted form of nested case statements:

case s $\{r_{11}{:}e_{11},\ r_{12}{:}e_{12},\ldots\},\ \{r_{21}{:}e_{21},\ r_{22}{:}e_{22},\ldots\},\ \ldots$

To translate from the outer to the inner language, the model is first "flattened" by eliminating nested structures. This is done by defining new variables that represent the intermediate values from nested calculations. Next, functions are converted into objects that use but do not define their arguments. A special symbol is employed to define the variable that represents the output of the function. Finally, function calls are represented by creating objects that define the arguments to the function.

AN EXAMPLE

We next show object decomposition in a simple model of the electrical system of an automobile. The example contains three components (Electrical, Ignition, and Lighting) which have internal subcomponents. These three are modeled as objects with fields representing the subcomponents. There are two components, Headlight, and Engine, that are treated as simple variables. The Ignition system depends on the Electrical system since Ignition.Plugs depends on Ignition.Current which is set equal to Electrical.Current. This network demonstrates how a stochastic model can be broken up into interacting objects, each with its own internal structure:

```
[Electrical =
   [Battery = dist charged: 0.9, weak: 0.08, dead: 0.02
    Wires = dist ok: 0.99, broken: 0.01
    Current = case Wires
                   broken: off,
                   ok: case Battery
                          charged: strong,
                          weak: weak,
                          dead: off]
Ignition =
  [Current = Electrical.Current
   Starter =
     [Condition = dist good: 0.9, broken: 0.1
      Function = case Condition
        broken: doesnt_turn_over,
        good: case Current,
                 strong: turns_over,
                 otherwise: doesnt_turn_over]
Plugs = case Current
              strong: fires,
              otherwise: doesnt_fire]

Lighting =
  [Switches_good = dist true: 0.95, false: 0.05
   Bulbs_good = dist true: 0.8, false: 0.2
   Wires_good = dist true: 0.99, false: 0.01
   System_good = Switches_good and Bulbs_good
                    and Wires_good]
Headlight = case Lighting.System_good
                 true: Electrical.Current,
                 false: off
Engine = case Ignition.Starter
  doesnt_turn_over: dead,
  turns_over: case Ignition.Plugs
                 fires: runs,
                 doesnt_fire: turns_over]
```

Next we show how a stochastic context free grammar (SCFG) can be implemented. Consider a simple SCFG:

```
A⇒xA   (p = 0.9)
A⇒y    (p = 0.1)
```

In this example, we define a function A returning the proper distribution of sentences in the grammar. Sentences are represented by objects that are trees. These objects contain a field called val that contains a terminal symbol at the leaves and the symbol compound internally. The internal nodes in this tree contain two other fields, left and right for the branches of the tree.

The Linear function takes such a tree and converts it to a linear form. The Sentence object (not specified here) is compared with this linear form in the variable Generated. Thus if a sentence is placed into the Sentence object, Generated will contain a distribution over true and false corresponding to the probability that the grammar will generate that sentence.

```
[A() = case (dist r1: 0.9, r2: 0.1)
        r1: [val = y]
        r2: [val = compound
             left = [val = x]
             right = A()]

Compare(s1,s2) = case s1.head
                   x: case s2.head
                        x: compare(s1.tail, s2.tail),
                        y: false,
                        z: false;
                   y: case s2.head
                        x: false,
                        y: compare(s1.tail, s2.tail),
                        z: false;
                   z: case s2.head
                        x: false,
                        y: false,
                        z: true

Linear1(t, xs) = case t.val
                   x: [head = x
                       tail = xs],
                   y: [head = y
                       tail = xs],
                   compound:
                       Linear1(t.left, Linear1(t.right, xs))
Linear(t) = Linear1(t, [val = z])
Sentence = [...]
Generated = Compare(Linear(A()), Sentence)]
```

THE INFERENCE ALGORITHM

Space limitations force us to present an abbreviated description of the inference algorithm For expositional purposes we first describe inference algorithms for two simpler languages. The first is equivalent in expressive power to Bayesian Networks. The second supports a weak form of objects that cannot import data and isn't Turing complete. We then provide the inference algorithm for the full language described above. All of these are versions of the inner language. Before going into the three languages separately, we describe some of the commonality in structure between them and define some helpful operations that are used in the algorithms.

The inference algorithm is based on one initially proposed by [1]. We follow them in dividing the algorithm into two functions, peval and pprocess. The higher level function, peval, is called directly, performing pre- and postprocessing for the second. The lower level function, pprocess contains methods for handling each construct in the language on a case by case basis.

Both functions take three arguments: a network model N, a variable x to be fully evaluated and a set of variables of interest, whose correlations must be tracked. They return a distribution of network models. A distribution, denoted $\{<N_1, p_1>, <N_2, p_2>, \ldots\}$, is a set of distinct networks, each weighted by a probability.

Two features of this inference algorithm are worth noting. First, it represents joint probability distributions internally as distributions over networks. By calculating joint and marginal distributions only as needed, the algorithm employs a form of *lazy evaluation* [8, 9]. Since networks generally provide a compact representation of joint distributions, this can provide significant savings in the space required for inference.

Second, the algorithm employs a recursive approach that reduces and augments the networks passed in and out of recursive calls. Network reduction and augmentation is done in such a way as to ensure that the networks evaluated in recursive calls contain exactly the information required to reconstruct distributions over specific subsets of variables. This approach is generally referred to as *goal-directed inference* [10, 11, 12].

We now describe some functions and operators needed to implement this approach. cone(x, N) is defined as the set consisting of x and all of its ancestors in a network model N. $N|_V$ denotes the network N reduced to the variable set V, i.e. reduced to assignments to variables in V and their ancestors. It is the union of the cones of all of the variables in V. N[M] denotes the network N augmented with the assignments in M (pruning redundant assignments in N).

seenby(V, x, N) is used to maintain information necessary to track correlations between variables. The argument x is the variable of interest, N is a network, and V is the set of variables whose correlations to x must be tracked. Intuitively, this function finds the minimal set of variables in the cone of x that makes the cone of x conditionally independent of V. This allows the inference algorithm to focus on the single variable x while maintaining enough information about the set of correlated variables in the network to recover their full joint probability distribution later. In the first language this can be described fairly simply. seenby(V, x, N) returns the union of seenby(y, x, N) for all $y \in V$. For a single variable y, if y is in the cone of x, then seenby returns y. Otherwise it returns seenby over the parents of y.

The first language has no embedded objects or attribute chains. It supports only dist and case statements over simple variables. Despite these limitations, it is powerful enough to represent any Bayesian Network model.

This is the inference algorithm for the first language:

```
peval(N, x) = peval(N, x, {})
peval(N, x, V) = $\sum_{\langle M, P\rangle \in \text{pprocess}(N|\{x\}, x, \{x\} \cup \text{Seenby}(V, x, N))} P * N[M]|_{\{x\} \cup V}$

pprocess(N, x, V) = case over definition of x:
  x = dist $a_1{:}p_1, a_2{:}p_2, \ldots$
     return {<[x=$a_1$], $p_1$>, <[x=$a_2$], $p_2$>,...}
  x = case s $r_1{:}e_1, r_2{:}e_2, \ldots$
     return $\sum_{\langle M, P\rangle \in \text{peval}(N, s, V)} P * \text{peval}(M[x = e_i], x, V)$  where s=$r_i \in$ M

  x = s
   if s∈N then
     return $\sum_{\langle M, P\rangle \in \text{peval}(N, s, V)} P * M[x = e]$  where s=e∈M

   else
     return {<N, 1>}
```

peval first reduces the network N to just the variables needed to evaluate x. Using the seenby function, it calculates a subset of variables in the cone of x that need to be tracked in order to keep the proper distribution for the original V. It passes this set to pprocess for inference. pprocess examines the type of expression assigned to x, determines the parents of x (if any), then uses peval recursively to compute the distributions over the parents of x. The distribution for the parents are then incorporated into a set of networks and probabilities, <M,P>, that are returned by pprocess to the original peval call. Finally, peval uses this distribution of networks (including parent information) to augment the original network N, extracts the pertinent information about x, and then sums over the resulting distributions.

The second language adds objects and attribute chains to the first. However, the objects in this language are non-recursive and are not permitted to use variables from an outer context. To extend the algorithm to handle this richer language, a number of changes are needed.

First, a new version of pprocess is needed to handle the situation where x is not a variable, but an attribute chain. peval is applied the head of the chain and determines the distribution of objects that are denoted. Then peval is applied to the tail of the chain within each of the objects in the distribution. When entering an embedded object, some care must be taken to correctly track the correlations between variables within the set V. This is done by modifying the set V before the recursive call. Any attribute chain in that set that doesn't have the same head as x is dropped. The rest are retained, deleting their heads.

Second, cases for object expressions must be added to pprocess. This is simple, since an object cannot be evaluated further. Finally, a function called translate is added. When an attribute chain is evaluated, peval returns a distribution of networks. Each network contains the value referred to by the attribute chain. translate locates and returns those values. We define translate:

```
translate(N, x) = e where x=e∈N
translate(N, $x_1.x_2 \ldots x_n$) = case over definition of $x_1$ in N:
        $x_1 = y_1.y_2 \ldots$
           return translate(N, $y_1.y_2 \ldots y_m.x_2.x_3 \ldots$)
        $x_1 = [s_1{=}e_1, s_2{=}e_2, \ldots]$   (referred to as O)
           return translate(O, $x_2.x_3 \ldots$)
```

The full language adds two new features: recursion and interaction between objects. These two features provide Turing completeness.

The definition of seenby is now extended to handle the fact that objects can be parents of other objects. The result is that seenby uses delayed evaluation to avoid infinite loops in tracing dependencies between objects.

In the full language, unlike the second, variables and attribute chains cannot always be fully evaluated to terminal symbols and objects. This is because objects can use non-terminal symbols that are defined in the enclosing context. When the algorithm reaches such a situation, it must drop back to the enclosing object.

Case statement are also extended. Since objects can now have variables defined in the enclosing object, fully evaluating a variable may not be possible. If this happens to the head of a case statement, a difficulty arises. To handle this, the translate function copies the statement from the inner to the outer context, modifying object and variable references as appropriate.

The third algorithm requires some new notations. The symbol % is used for the attribute chain set dereferencing operation. A set V is dereferenced by a symbol x (V%x) by first removing all attribute chains in V that don't have x as their head. All the rest of the chains are retained, but without their heads.

The variable set representation is also expanded. It is sometimes necessary to delay part of the computation of a variable set, particularly in the case of infinitely long attribute chains arising from the recursive interactions of objects. This is handled by a delay operator that produces a structure representing a future computation. A statement of the form V.delay(*calculation*) indicates that every attribute chain in V is appended with the result of *calculation*. The delayed calculation finally takes place when an attribute chain is dereferenced to the point at which the head of the chain is the delay object.

The seenby function must be implemented in a more sophisticated way in order to handle potentially infinite attribute chains. Here is an algorithmic description of the third algorithm version of seenby:

```
seenby($y_1.y_2.y_3\ldots$, x, N) =
    if $y_1 \in$ cone(x, N) then {$y_1.y_2.y_3\ldots$}
    else if $y_1 \notin$ N then {}
    else case over definition of $y_1$ in N:
        $y_1$ = dist $a_1{:}p_1$, $a_2{:}p_2$,... : {}
        $y_1 = z_1.z_2\ldots$ : seenby($z_1.z_2\ldots y_2.y_3\ldots$, x, N)
        $y_1$ = case s of {$r_{11}{:}e_{11}$...},... :
            seenby({s, $e_{11}.y_2.y_3$, ...}, x, N)
        $y_1$ = [$s_1$=$e_1$, $s_2$=$e_2$,...] (referred to as O) :
            $\bigcup_{z \in Pa(O)}$ seenby(z, x, N).delay(seenby($y_2.y_3\ldots$, z, O)%z)
```

Likewise peval and translate are be updated for the third algorithm as follows:

```
peval(N, x, V) =
    if x∈N then
        $\sum_{<M,\, P> \in \text{pprocess}(N!\{x\},\, x,\, \{x\} \cup \text{seenby}(V,\, x,\, N))} P * N[M]!_{\{x\} \cup V}$
    else
        {<N, 1>}
pprocess(N, $x_1.x_2.x_3\ldots$, V) = $\sum_{<M,\, P> \in \text{peval}(N,\, x1,\, V)} W$
    where W = case over definition of $x_1$ in M:
        $x_1 = y_1.y_2\ldots y_m$
            return peval(M, $y_1.y_2\ldots y_m.x_2.x_3\ldots$, V)
        $x_1$ = [$s_1$=$e_1$, $s_2$=$e_2$,...]   (referred to as O)
            return $\sum_{<M,\, P> \in \text{peval}(O,\, x2.x3\ldots,\, V\%x1)} P * M[x_1 = O']$
        otherwise
            return {<M, 1>}

pprocess(N, x, V) = case over definition of x in N:
    x = dist $a_1{:}p_1$, $a_2{:}p_2$,...
        return {<[x=$a_1$], $p_1$>, <[x=$a_2$], $p_2$>,...}
    x = case s
        return peval(N[x=s], x, V)
    x = case s {$r_{11}{:}e_{11}$, $r_{12}{:}e_{12}$, ...}, {$r_{21}{:}e_{21}$, $r_{22}{:}e_{22}$, ...},
               {$r_{31}{:}e_{31}$, $r_{32}{:}e_{32}$, ...},...
        return $\sum_{<M,\, P> \in \text{peval}(N,\, s,\, V)} W$
```

```
        where W =
            let h = translate(M, s) in case over definition of h
                h = $r_{1i}$
                    peval(M[x=case $e_i${$r_{21}{:}e_{21}$, $r_{22}{:}e_{22}$, ...},
                    {$r_{31}{:}e_{31}$, $r_{32}{:}e_{32}$, ...},...], x, V)
                h = s
                    {<M, 1>}
                h = [$s_1$=e1, $s_2$=$e_2$, ...]
                    throw error
                h = case s'{$r'_{11}{:}e'_{11}$, $r'_{12}{:}e'_{12}$, ...},
                            {$r'_{21}{:}e'_{21}$, $r'_{22}{:}e'_{22}$, ...}, ...
                    peval(M[x= case s' {$r'_{11}{:}e'_{11}$, $r'_{12}{:}e'_{12}$, ...},
                                       {$r'_{21}{:}e'_{21}$, $r'_{22}{:}e'_{22}$, ...}, ...,
                                         {$r_{11}{:}e_{11}$, $r_{12}{:}e_{12}$, ...}, {$r_{21}{:}e_{21}$,
                                       $r_{22}{:}e_{22}$, ...}, {$r_{31}{:}e_{31}$, $r_{32}{:}e_{32}$,
                                       ...},...], x, V)
                h = s'
                    peval(M[x = case s'  {$r_1{:}e_{11}$, $r_{12}{:}e_{12}$, ...},{$r_{21}{:}e_{21}$,
                    $r_{22}{:}e_{22}$, ...}, {$r_{31}{:}e_{31}$, $r_{32}{:}e_{32}$, ...},...], x, V)

    x = [$s_1$=$e_1$, $s_2$=$e_2$,...]
        return {<N, 1>}
    x = $s_1.s_2.s_3\ldots$ (potentially a single symbol)
        return $\sum_{<M,\, P> \in \text{peval}(N,\, s1.s2.s3\ldots,\, V)} W$

          where W =
              let h = translate(M, $s_1.s_2.s_3\ldots$) in
                  if h = [$s_1$=$e_1$, $s_2$=$e_2$,...] or h = $s_1.s_2.s_3\ldots$ then
                      {<M, 1>}
                  else
                      peval(M[x=h], x, V)

translate(N, $x_1.x_2.x_3\ldots x_n$) =
    if $x_1 \notin$ N then
        return $x_1.x_2.x_3\ldots x_n$
    else if n = 1 then
        return h where $x_1$=h∈N
    else
        case over definition of $x_1$ in N:
            $x_1 = y_1.y_2\ldots$
                return translate(N, $y_1.y_2\ldots y_m.x_2.x_3\ldots$)
            $x_1$ = [$s_1$=$e_1$, $s_2$=$e_2$,...]   (referred to as O)
                let h = translate(O, $x_2.x_3\ldots$) but modify:
                    for each attribute chain a in the definition of
                    h whose head is defined in O replace by $x_1$.a
                    return the modified h
            $x_1$ = case s {$r_{11}{:}e_{11}$, $r_{12}{:}e_{12}$, ...}, {$r_{21}{:}e_{21}$, $r_{22}{:}e_{22}$, ...}...
                return case s {$r_{11}{:}e_{11}$, $r_{12}{:}e_{12}$, ...}, {$r_{21}{:}e_{21}$, $r_{22}{:}e_{22}$, ...}...
```

This completes the third inference algorithm.

IMPLEMENTATION AND EFFICIENCY

There are three major implementation features that affect computational efficiency. The first is caching. For this algorithm to run efficiently on sizable problems, it must cache and reuse results of intermediate calculations. The simplest form of this is to cache resulting distributions returned by pprocess, keyed by the triplet of the network, variable, and variable set passed in to it. This is slightly different from the caching proposed in [1]. There it is necessary for caching to identify similar networks, which often is difficult to do efficiently. In our approach, the caching need only recognize exact matches.

We are also developing more advanced caching that can identify for reuse all calls involving a subset of the same attribute chains in the variable set of interest. This may seem at first difficult to do efficiently. However one can cache lists of such sets keyed on the exact network and variable to be evaluated. These lists are expected to be small, and thus a sequential check is sufficiently fast, given that the subsets can be matched quickly.

A second feature affecting both efficiency and completeness of inference is lazy evaluation. The inference algorithm does not compute any distribution until it is needed. This not only improves efficiency; it also serves as the basis for exact inference over a larger (Turing complete) class of models. Without lazy evaluation, inference on models with infinite objects would fail to terminate.

Finally, the inference algorithm obtains efficiency by exploiting the modular domain representation supported by objects. It evaluates one object at a time independent of the outer context, only referencing the outer context to acquire needed information. This type of decomposition has been found effective for reducing computation time in other systems [6].

There are other lesser efficiencies implicit in our algorithm design. For example, the use of case statements rather than explicit conditional probability tables (CPTs) allows implicit factoring of CPTs. Because the inference algorithm operates directly on these case statements, it can take advantage of the computational reductions that factored CPTs offer. Other efficiencies derive from the implicit query optimization and goal-directed strategies that are incorporated into the inference algorithm.

CONCLUSION

A new stochastic modeling language has been presented. This language supports object-oriented features that are effective in the development of component-based hierarchical models. Object-oriented abstraction and recursive functions give this language significant expressive power (e.g., Turing completeness). These additional capabilities are paired with an exact efficient inference algorithm that can exploit the domain decomposition implicit in object-oriented representations.

In the future, this algorithm will be extended to include an approximation scheme that further enhances efficiency. In addition, parallel distributed computation will be supported, where the cache is the only data structure that needs to be shared between threads. Finally, the outer language will be extended to include additional high level constructs that simplify knowledge representation.

ACKNOWLEDGEMENT

This work was supported by NSF Grant 115-9800929 and NSF Phase I SBIR #DMI-996-937

REFERENCES

[1] D. Koller, D. McAllester, and A. Pfeffer, Effective Bayesian Inference for Stochastic Programs, *Proceedings of The Fourteenth National Confererence on Artificial Intelligence*, (Cambridge: MIT Press, 1997).

[2] Y. Xiang, D. Poole, and M. Beddoes, Multiply Sectioned Bayesian Networks and Junction Forests for Large Knowledge-Based Systems, *Computational Intelligence*, 9(2), 1993, 171-220.

[3] K. Laskey and S. Mahoney, Network Fragments: Representing Knowledge for Constructing Probabilistic Models, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, (San Francisco: Morgan Kaufmann, 1997).

[4] D. Koller and A. Pfeffer, Object-oriented Bayesian Network, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, (San Francisco: Morgan Kaufmann, 1997).

[5] D. Koller and A. Pfeffer, Probabilistic Frame-Based Systems, *Proceedings of The Fifteenth National Confererence on Artificial Intelligence*, Cambridge: (MIT Press, 1998).

[6] A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa, 1999. SPOOK: A System for Probabilistic Object-Oriented Knowledge Representation, *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence,* (San Francisco: Morgan Kaufmann, 1999).

[7] Y. Xiang, K.G. Olesen, and F.V. Jensen, Practical Issues in Modeling Large Diagnostic Systems with Multiply Sectioned Bayesian Networks, *International Journal of Pattern Recognition and Artificial Intelligence*, 14 (1), 2000, 59-71.

[8] A. Madsen, and F. Jensen LAZY Propagation: A Junction Tree Inference Algorithm Based on Lazy Evaluation, *Artificial Intelligence*, 113, 1999, 203-245.

[9] Y. Xiang and F. Jensen, Inference in Multiply Sectioned Bayesian Networks with Extended Shafer-Shenoy and Lazy Propagation, *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence,* (San Francisco: Morgan Kaufmann, 1999).

[10] R.D. Shachter, An Ordered Examination of Influence Diagrams, *Networks*, 20, 1990, 535-563.

[11] E. Castillo, J.M. Gutierrez, and A.S. Hadi, Goal Oriented Symbolic Propagation in Bayesian Networks, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, (Cambridge: MIT Press, 1996).

[12] M. Baker and T.E. Boult, Pruning Bayesian Networks for Efficient Computation, *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence*, (Amsterdam: North Holland, 1990).