Report Number: WUCS-91-25

1991-04-01

CABeN: A Collection of Algorithms for Belief Networks

Steve B. Cousins, William Chen, and Mark E. Frisse

Belief networks have become an increasingly popular mechanism for dealing with uncertainty in systems. Unfortunately, it is known that finding the probability values of belief network nodes given a set of evidence is not tractable in general. Many different simulation algorithms for approximating solutions to this problem have been proposed and implemented. In this report, we describe the implementation of a collection of such algorithms, CABeN. CABeN contains a library of routines for simulating belief networks, a program for accessing the routines through menus on any 'tty' interface, and some sample programs demonstrating how the library would be used... Read complete abstract on page 2.

CABeN: A Collection of Algorithms for Belief Networks

Steve B. Cousins, William Chen, and Mark E. Frisse

Complete Abstract:

Belief networks have become an increasingly popular mechanism for dealing with uncertainty in systems. Unfortunately, it is known that finding the probability values of belief network nodes given a set of evidence is not tractable in general. Many different simulation algorithms for approximating solutions to this problem have been proposed and implemented. In this report, we describe the implementation of a collection of such algorithms, CABeN. CABeN contains a library of routines for simulating belief networks, a program for accessing the routines through menus on any 'tty' interface, and some sample programs demonstrating how the library would be used within an application. CABeN implements five algorithms: Logic Sampling, Likelihood, Weighting (Shachter's Basic algorithm), Self Importance, Pearl's algorithm, and Chavez's algorithm. In addition, we have implemented Markov scoring as an option to any of the above algorithms. We have compared these 10 variations with each other in a series of experiments in which we varied the graph topologies, the number of nodes provided with evidence, and the conditional probability values. A detailed description of each of the five algorithms is given.

CABeN: A Collection of Algorithms for Belief Networks

Steve B. Cousins, William Chen and Mark E. Frisse

WUCS-91-25

April 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Portions of this report have been submitted to the Seventh Conference on Uncertainty in AI (1991) and the Fifteenth Annual Symposium on Computer Applications in Medical Care (1991).

CABeN: A Collection of Algorithms for Belief Networks

Steve B. Cousins
William Chen
Mark E. Frisse

WUCS-91-25

Medical Informatics Laboratory
Washington University

October, 1990
Revised January 22, 1992

Correspond with: Steve Cousins
Medical Informatics Laboratory
Washington University School of Medicine
660 South Euclid Ave., Box 8121
St. Louis, MO 63110 USA
email: caben@informatics.WUSTL.EDU
phone: (314) 362-4322

Portions of this report have been published in the Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care (November, 1991).

Abstract

Belief networks have become an increasingly popular mechanism for dealing with uncertainty in systems. Unfortunately, it is known that finding the probability values of belief network nodes given a set of evidence is not tractable in general. Many different simulation algorithms for approximating solutions to this problem have been proposed and implemented. In this report, we describe the implementation of a collection of such algorithms, CABeN. CABeN contains a library of routines for simulating belief networks, a program for accessing the routines through menus on any 'tty' interface, and some sample programs demonstrating how the library would be used within an application.

CABeN implements five algorithms: Logic Sampling, Likelihood Weighting (Shachter's Basic algorithm), Self Importance, Pearl's algorithm, and Chavez's algorithm. In addition, we have implemented Markov scoring as an option to any of the above algorithms. We have compared these 10 variations with each other in a series of experiments in which we varied the graph topologies, the number of nodes provided with evidence, and the conditional probability values. A detailed description of each of the five algorithms is given.

Chapter 1

Introduction

A belief network is a formal knowledge representation and inference technique consisting of a directed graph and a set of conditional probabilities. Belief networks are an elegant, well-founded way to reason with uncertainty, but in general, inference with them is computationally intractable [1]. Fortunately, for many specific types of graphs, reasoning with belief networks in polynomial time is possible. For some graphs, algorithms exist to give an exact answer in a reasonable amount of time [2,3,4]. For many other graphs, approximation algorithms may be used. This report describes the implementation and use of stochastic simulation algorithms for doing approximate inference with belief networks. The following algorithms are implemented in the package:

- Logic Sampling
- Likelihood weighting
- Self Importance
- Pearl's Markov simulation algorithm
- Chavez's algorithm

Markov scoring has been implemented as an option to all algorithms in the package. In other words, CABeN implements Likelihood weighting, as well as Likelihood weighting with Markov scoring. The first three algorithms have been implemented from the description in Shachter [5], and we consulted Pearl [2] and Chavez [6,7] for details about their respective algorithms.

We have done some analysis of the relative performance of these algorithms under various circumstances, following Shachter. Some of our results have confirmed those of Shachter, but others contradict some of his initial findings. Although certainly not conclusive, we present our initial results and conclusions as additional empirical data for the field.

1.1 Background

Many approaches to reasoning with uncertain knowledge have been proposed (certainty factors, fuzzy logic, etc). The argument in favor of belief networks, made by Pearl and others, is that only belief networks are based on a really firm theoretical foundation: probability theory.

One use of belief networks has been in the field of expert systems [8]. A rule based system might contain a rule, "If A, then conclude B with certainty C." In probabilistic terms, this would correspond to the conditional probability statement $p(B \mid A) = C$. In general, without making conditional probability assumptions, $p(B \mid A)$ does not tell us much, because if there are any other variables in the system, we must consider them as well as B before we can determine the probability of B. Belief networks provide a graphical means of specifying which other variables a variable depends on, and more importantly, which variables can be ignored.

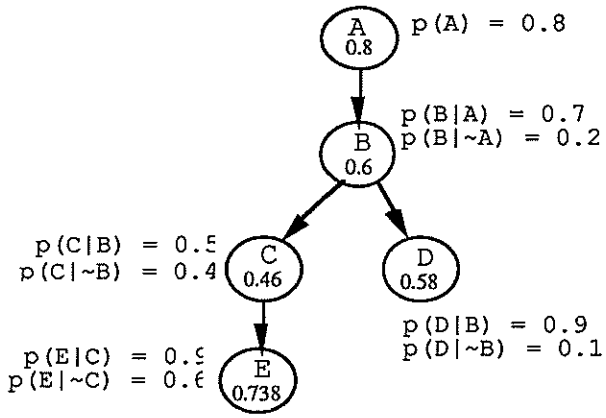


Figure 1.1: A simple belief network with its associated conditional probabilities and posterior marginal probabilities (on nodes).

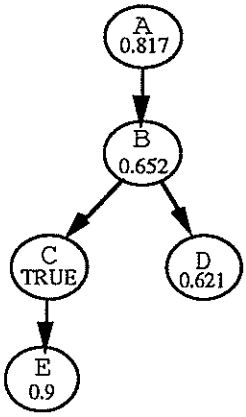


Figure 1.2: The node C has been set to 'TRUE', causing the probabilities of the other nodes in the network to increase according to the conditional probability relationships.

Inference in belief networks is performed by propagating the effects of evidence. After propagation, the "posterior marginal probability of X", for a node X in the network, corresponds to $p(X|E_1 = e_1, E_2 = e_2, ..., E_n = e_n)$, where $E_i = e_i$ indicates that evidence node E_i is in state e_i. We will informally use the term "probablity of X" for "posterior marginal probability of X", to indicate the resulting value of X once all evidence has been propagated.

For a brief example of belief network inference, consider the network in Figure 1.1. The numbers on the nodes indicate the initial probabilities of the variables represented by the nodes. In this case, the nodes are all boolean and the numbers represent the probability that the node state is true. These probabilities represent the posterior marginal probabilities of the nodes given no evidence in the network, and are calculated using the conditional probabilities given for the network.

Evidence can be given to a network in various ways. The most common means is to fix the state of nodes in the network. For example, if we know that node C is in state 'TRUE', we fix its state and recalculate the probabilities of the other nodes in the network (Figure 1.2). Naturally, C's probability will be 1.0 in the resulting configuration, but we are interested in the values of the other nodes in the network. Note that the probabilities of nodes near C in the network have increased, while nodes more distant from C have not changed as much. If we now set A's state to false, as in Figure 1.3, notice how the probabilities again shift to reflect the new evidence.

Another method of giving evidence to a belief network is in terms of ratios. If we have evidence that C is likely, but not certain, we can give feedback to the network to that effect through an

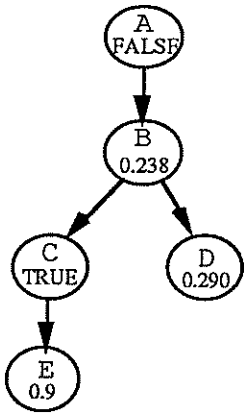


Figure 1.3: With both A and C fixed, the only probabilities of interest are those for B, D, and E. Setting A to 'FALSE' had no effect on E because of the graph structure, but caused a significant decrease in the values of B and D.

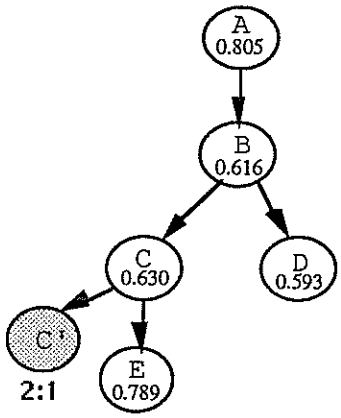


Figure 1.4: Evidence can also be specified in an odds-likelihood format. This is done by adding a virtual evidence node (C') whose relationship to the parent node is set according to the value of the ratio.

odds-likelihood ratio. Figure 1.4 shows the results of giving the network 2:1 odds in favor of C being true and no other evidence. This method is useful when belief networks are given partial feedback, as when they are used incrementally.

With a simple network, the kind of belief network inference described above can be done exactly. Algorithms for doing exact inference are described elsewhere [9,2]. In the rest of this introduction, the CABeN algorithms for computing approximate solutions are described and compared.

1.2 The Algorithms

In this section we provide an informal, high-level description of the algorithms we have implemented in this package. For a more rigorous, mathematical treatment see Shachter's recent paper [5]. Our intent here is to give enough detail for the reader to get an intuitive understanding of the algorithms.

We first describe a brute force algorithm for propagating belief network values. Consider a network with 2 nodes, A and B (Figure 1.5) where node A has three states and node B has two states (boolean). This network has $3 * 2 = 6$ states. For each state of the network, we can calculate a joint probability value for the state (Table 1.1).

The probability of any variable in any state can be calculated by simply summing over all rows

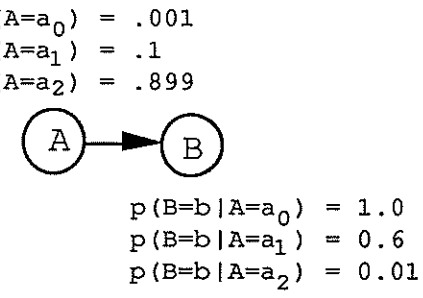


Figure 1.5: A trivial 2-node belief network.

Table 1.1: Exhaustive list of joint probabilities for a trivial belief network.

A	B	Value		
a_0	b	$p(A=a_0)p(B=b \mid A=a_0)$	$0.001 * 1.0 =$	0.001
a_0	$\bar{b}$	$p(A=a_0)p(B=\bar{b} \mid A=a_0)$	$0.001 * 0.0 =$	0.0
a_1	b	$p(A=a_1)p(B=b \mid A=a_1)$	$0.1 * 0.6 =$	0.06
a_1	$\bar{b}$	$p(A=a_1)p(B=\bar{b} \mid A=a_1)$	$0.1 * 0.4 =$	0.04
a_2	b	$p(A=a_2)p(B=b \mid A=a_2)$	$0.899 * 0.01 =$	0.00899
a_2	$\bar{b}$	$p(A=a_2)p(B=\bar{b} \mid A=a_2)$	$0.899 * 0.99 =$	0.89001

in which the variable is assigned the state. For example, $p(A = a_0)$ is the sum of the first two rows, or $0.001 + 0.0 = 0.001$. Evidence restricts the cases to those in which the evidence node is in the desired state, and then normalizing. We normalize a variable by dividing each of its possible states by the sum of its possible states. For example, if B is set to $\bar{b}$, we only consider the 3 rows where B is false. In this case, the calculated value for $A = a_2$ would be $\frac{0.89001}{0.0+0.04+0.89001} = 0.957$.

Unfortunately, the size of this table is exponential in the number of nodes, so this algorithm quickly becomes intractable as the number of nodes increases. Simulation algorithms select a subset of the rows and use the values calculated to estimate the values for the variables. The simulation algorithms we consider here differ primarily in the method they use to select rows. Each time a row is selected, the probability of selecting that row, p-selecting, is used to normalize the values.

The first three algorithms for simulating belief networks all have a common structure. The key steps for each simulation run are *selecting* a state for each node in the network and *scoring* the outcome of that selection. The algorithms differ in the method they use to select states for the nodes and how they score each iteration. The score is dependent on how the states were chosen, reflected in the variable p-selecting. In logic sampling, for example, the states are chosen randomly, and p-selecting is a constant, while in Likelihood weighting, the conditional probabilities are taken into account when selecting states, so p-selecting reflects the conditional probabilities used. Pseudocode for the top-level simulation algorithm is as follows:

```
Preprocess the network
Loop for the number of simulations
     Select states for nodes (while keeping track of p-selecting)
          (see individual algorithm descriptions)
     Calculate a score (total probabilities / p-selecting)
     Score the net
End Loop
Normalize the node values
Cleanup
```

The total probabilities value used in calculating the score is the product of the conditional probability values given the states chosen for all nodes in the network (except fixed evidence nodes). For example, in the very simple belief network in Figure 1.5, given a state selection $(A = a_1, B = b)$, the total probability would be $p(A = a_1) * p(B = b \mid A = a_1)$.

1.2.1 Logic Sampling

The simplest simulation algorithm, *logic sampling*, randomly chooses a state for each node in the network from among the possible states by giving an equal chance to all states. The value of p-selecting for this algorithm is a constant ($\prod \frac{1}{states_i}$ where $states_i$ is the number of states of the node i), but since normalization will negate the effects of this constant, we avoid the computation and use 1.0. The following pseudocode describes this algorithm.

```
Loop for the number of simulations
     For each non-evidence node in the network
         Set the state of the node to one of its possible states at random
     End For
     Set p-selecting to 1.0
     Calculate a score (total probabilities / p-selecting)
     Score the net (using traditional or Markov blanket scoring)
End Loop
Normalize the node values
```

For example, consider a very simple graph with two nodes, A and B. Node A represents the arrival time of a student a summer job, and is one of three mutually exclusive and exhaustive states: a_0 means the student arrives before 7:30, a_1 means the student arrives between 7:30 and 9:00, and a_2 means the student arrives after 9:00. Node B represents the proposition that the student will find a parking space within a ten minute walk of the laboratory, and is conditioned on node A. For simplicity, we will let a_i stand for the expression $A = a_i$, and b or $\bar{b}$ stand for the expressions $B = b$ or $B = \bar{b}$. The initial probabilities would then be:

$p(a_0)$	=	0.001	It is very unlikely the student will arrive before 7:30
$p(a_1)$	=	0.1	It is unlikely the student will arrive before 9:00
$p(a_2)$	=	0.899	Usually the student shows up after 9:00
$p(b \mid a_0)$	=	1.0	Before 7:30 there is always parking
$p(b \mid a_1)$	=	0.6	There is usually parking between 7:30 and 9:00
$p(b \mid a_2)$	=	0.01	It is hard to park close after 9:00

Scores for each node are stored in a table for that node, and are accumulated across all of the simulations before being normalized. At the beginning of the simulation, the scores are all zero. The initial scoring tables would look like this:

A:		B:	
a_0	0	b	0
a_1	0	$\bar{b}$	0
a_2	0		

The pseudo-code defines the score on an iteration in terms of the *total probabilities*. The total probabilities is the probability that the network is in a given state, which in this example is $p(A)p(B \mid A)$. If A is in state a_1 and B is in state b, then the total probability of the network in that state is $p(a_1)p(b \mid a_1) = 0.1 * 0.6 = 0.06$.

Stepping through Logic Sampling

We will now run through three iterations of the algorithm, and approximate the values of the network.

Iteration 1

A:		B:	
a_0	0	b	0
a_1	0	$\bar{b}$	0 ⇐
a_2	0 ⇐		

We randomly choose a state for each node in the network. We use the symbol ⇐ in the diagram above to denote our choices. On this iteration, we chose the case where the student arrived

after 9:00 (a_2) and did not find a parking space ($\bar{b}$). Intuitively, this is a likely possibility which should be given a high score. We calculate that

$$\text{score} = \text{total probability} = p(a_2)p(\bar{b} \mid a_2) = 0.899 * 0.99 = 0.89001$$

which we will round to 0.890. The score is actually the total probability divided by p-selecting, but in this algorithm we always use 1.0 for p-selecting because p-selecting would always be a constant, and we will normalize at the end anyway. (In this example, p-selecting would be $p(selectingA) * p(selectingB) = \frac{1}{3} * \frac{1}{2} = \frac{1}{6}$).

The final step on each iteration is to score the network, and is done by updating the tables. On this iteration, we add 0.890 to the bin for $A = a_2$ and to the bin for $B = \bar{b}$.

Iteration 2

A:		B:	
a_0	0	b	0
a_1	0 ⇐	$\bar{b}$	0.890 ⇐
a_2	0.89		

This time we choose $A = a_1$ and $B = \bar{b}$ as the states.

$$\text{score} = \text{total probability} = p(a_1)p(\bar{b} \mid a_1) = 0.04$$

If we were to stop after only two iterations, $p(a_0)$ would be zero and $p(b)$ would be 0 just because those states were never chosen. It is important, even in small networks, to take enough samples to ensure that the expected value of the stochastic error is minimized.

Iteration 3

A:		B:	
a_0	0	b	0 ⇐
a_1	0.04 ⇐	$\bar{b}$	0.93
a_2	0.89		

For our final iteration, we chose $A = a_1$ and $B = b$.

$$\text{score} = \text{total probability} = 0.06$$

Final score tables

A:		B:	
a_0	0	b	0.06
a_1	0.1	$\bar{b}$	0.93
a_2	0.89		

Normalization The final step, normalization, yields the values

A:		B:	
a_0	0.0	b	0.061
a_1	0.101	$\bar{b}$	0.939
a_2	0.899		

Once normalized, these numbers represent the probabilities of the node being in the given state, so $p(a_1) = 0.101$.

Stepping through Logic Sampling with evidence

Using this belief network, it is possible to estimate the time our student arrived given that we know that he found a parking space. We do this by fixing the state of node B to state b. Since the state of node B is known, we will not bother to update its table throughout this example.

Iteration 1

A:	
a_0	0
a_1	0 ⇐
a_2	0

$$\text{score} = \text{total probability} = p(a_1)p(b \mid a_1) = 0.06$$

Iteration 2

A:	
a_0	0
a_1	0.06
a_2	0 ⇐

score = total probability = $p(a_2)p(b \mid a_2) = 0.009$

Iteration 3

A:	
a_0	0
a_1	0.06
a_2	0.009 ⇐

score = total probability = $p(a_0)p(b \mid a_0) = 0.001$

Finally

A:	
a_0	0.001
a_1	0.06
a_2	0.009

Normalize

A:	
a_0	0.014
a_1	0.857
a_2	0.129

Note that these are the correct answers, since we selected each possible state exactly once. Unfortunately, for a non-trivial number of nodes, this calculation is intractable. Note also that these answers agree with our intuition. Given our initial probabilities, it was extremely unlikely that the student would arrive before 7:30 ($p(a_0) = 0.001$). Given that we know he found a parking space, the chances of early arrival increase ten-fold, but are still small. The most likely explanation, given that the student found parking, is that he arrived between 7:30 and 9:00. Finally, notice that although p-selecting has changed (it would now be $\frac{1}{3}$), it is still a constant, and would therefore be lost in the normalization anyway.

1.2.2 Likelihood weighting

Another simple algorithm is called the *Likelihood weighting*. We consider it more advanced than logic sampling because it uses more information to choose its states. Likelihood weighting differs from logic sampling by weighting the node states before selecting them. The node states are weighted by their prior probabilities. For example, node A in the previous example will be assigned to state a_0 in 0.1% of the trials, to state a_1 in 10% of the trials, and to state a_2 in all other trials. Node B in our example can not be weighted until A's state has been chosen. However, once we have chosen A state for node A, we can select a state for B. This implies that a preprocessing step is needed for this algorithm: we need to order the nodes such that parent states are always chosen before child node states are selected. This can always be done in linear time since belief networks are acyclic. Choosing states in this way affects p-selecting. We can calculate p-selecting for scoring purposes by taking the product of the probabilities of all of the states we chose. The Likelihood weighting algorithm is:

```
Sort the nodes of the graph so that parents always proceed children
Loop for the number of simulations
     p-selecting = 1.0
     For each non-evidence node N in the network (in graph order)
                                  where N has k parents P_1 ... P_k in states s_{P_i}
          Choose a state s_N according to the conditional probability of that state
          p-selecting = p-selecting * p(N = s_N | P_1 = s_{P_1}, ..., P_k = s_{P_k})
     End For
     Calculate a score (total probabilities / p-selecting)
     Score the net (using traditional or Markov blanket scoring)
End Loop
Normalize the node values
```

Stepping through Likelihood weighting

We will now run through five iterations of Likelihood weighting, using the same example as we used for logic sampling. Selecting states in Likelihood weighting requires more work than was required in Logic Sampling. To choose a state for A, we generate a random number rn between 0 and 1. We then select a state for A from its initial conditional probability table:

$$A = \begin{cases} a_0 & \text{if } rn \leq 0.001 \\ a_1 & \text{if } 0.001 < rn \leq 0.101 \\ a_2 & \text{if } rn > 0.101 \end{cases}$$

A will always be chosen using this method because it is a root node in the network. We can not define the equation for choosing B until we know the state of A. In the iterations that follow, we will note the random number that caused the selection of the given state.

Iteration 1

	A:	a_0	0		B:	b	0	⇐
	(0.03)	a_1	0	⇐	(0.03)	$\bar{b}$	0	
		a_2	0					

Our first random numbers was 0.03, so we chose $A = a_1$. Since $p(b \mid a_1) = 0.6$, we choose B according to the following equation:

$$B = \begin{cases} b & \text{if } rn \leq 0.6 \\ \bar{b} & rn > 0.6 \end{cases}$$

We happen to choose 0.03 for our second random number, so $B = b$. Then

$$\begin{aligned} \text{total probability} &= p(a_1)p(b \mid a_1) = 0.1 * 0.6 = 0.06 \\ \text{p-selecting} &= p(a_1)p(b \mid a_1) = 0.06 \\ \text{score} &= \frac{\text{total probability}}{\text{p-selecting}} = 1.0 \end{aligned}$$

Note that in this algorithm, total probability will always equal p-selecting unless there is evidence in the network. When evidence is present, p-selecting will change because the nodes with evidence will not factor into the p-selecting.

Iteration 2

	A:	a_0	0		B:	b	1	
	(0.37)	a_1	1		(0.074)	$\bar{b}$	0	⇐
		a_2	0	⇐				

We select $A = a_2$ using the same table as above (rn = 0.37), but for B we now select

$$B = \begin{cases} b & \text{if } rn \leq 0.01 \\ \bar{b} & rn > 0.01 \end{cases}$$

Since our second random number is 0.074, we will add 1 to the bins for $A = a_2$ and $B = \bar{b}$.

Iteration 3

	A:	a_0	0		B:	b	1	
	(0.56)	a_1	1		(0.58)	$\bar{b}$	1	⇐
		a_2	1	⇐				

Iteration 4

	A:	a_0	0		B:	b	1	
	(0.13)	a_1	1		(0.75)	$\bar{b}$	2	⇐
		a_2	2	⇐				

Iteration 5

A:		B:		
(0.014)	a_0 | 0	(0.703)	b | 1	
	a_1 | 1		$\bar{b}$ | 3	
	a_2 | 3			

Finally

A:		B:		
	a_0 | 0		b | 1	
	a_1 | 2		$\bar{b}$ | 4	
	a_2 | 3			

Normalize

A:		B:		
	a_0 | 0.0		b | 0.2	
	a_1 | 0.4		$\bar{b}$ | 0.8	
	a_2 | 0.6			

Stepping through Likelihood weighting with evidence

Now we will step through Likelihood weighting assuming our student finds a parking space ($B = b$).

Iteration 1

A:		
(0.24)	a_0 | 0	
	a_1 | 0	
	a_2 | 0	⇐

$$\begin{aligned} \text{total probability} &= p(a_2)p(b \mid a_2) = 0.899 * 0.01 = 0.009 \\ \text{p-selecting} &= p(a_2) = 0.899 \\ \text{score} &= \frac{\text{total probability}}{\text{p-selecting}} = 0.01 \end{aligned}$$

Note that the score will always be 0.01 as long as $A = a_2$.

Iteration 2

A:		
(0.96)	a_0 | 0	
	a_1 | 0	
	a_2 | 0.01	⇐

Iteration 3

A:		
(0.69)	a_0 | 0	
	a_1 | 0	
	a_2 | 0.02	⇐

Iteration 4

A:		
(0.16)	a_0 | 0	
	a_1 | 0	
	a_2 | 0.03	⇐

Iteration 5

A:		
(0.41)	a_0 | 0	
	a_1 | 0	
	a_2 | 0.04	⇐

Iteration 6

A:		
(0.33)	a_0 | 0	
	a_1 | 0	
	a_2 | 0.05	⇐

Iteration 7

A:		
(0.07)	a_0 | 0	
	a_1 | 0	
	a_2 | 0.06	⇐

In this case,

$$\begin{aligned} \text{total probability} &= p(a_1)p(b \mid a_1) = 0.1 * 0.6 = 0.06 \\ \text{p-selecting} &= p(a_2) = 0.1 \\ \text{score} &= \frac{\text{total probability}}{\text{p-selecting}} = 0.6 \end{aligned}$$

Finally

A:		
	a_0 | 0	
	a_1 | 0.6	⇐
	a_2 | 0.06	

Normalizing

A:		
	a_0 | 0	
	a_1 | 0.91	⇐
	a_2 | 0.09	

1.2.3 Self Importance

The *self importance algorithm* differs from Likelihood weighting in that it periodically updates its state selection criteria. This self importance sampling selects more likely states of the network more often, and then adjusts the final results to avoid bias. It does this by using an 'importance' distribution rather than the original conditional probabilities as Likelihood weighting does. In Self Importance, the original importance distribution is just the conditional probabilities, but it is regularly updated in a normalization step to take into account the results of the simulation so far.

Self Importance requires two additional tables with each node, one for a normalized-score (the evolving importance distribution), and one for a total-score. The total-score table for each node is initialized to zero, and on every update the cell corresponding to the state of the network is incremented by the score for that iteration. The pseudo-code for Self Importance is exactly like that of Likelihood weighting, except that rather than using the conditional probability tables, the normalized-score tables are used:

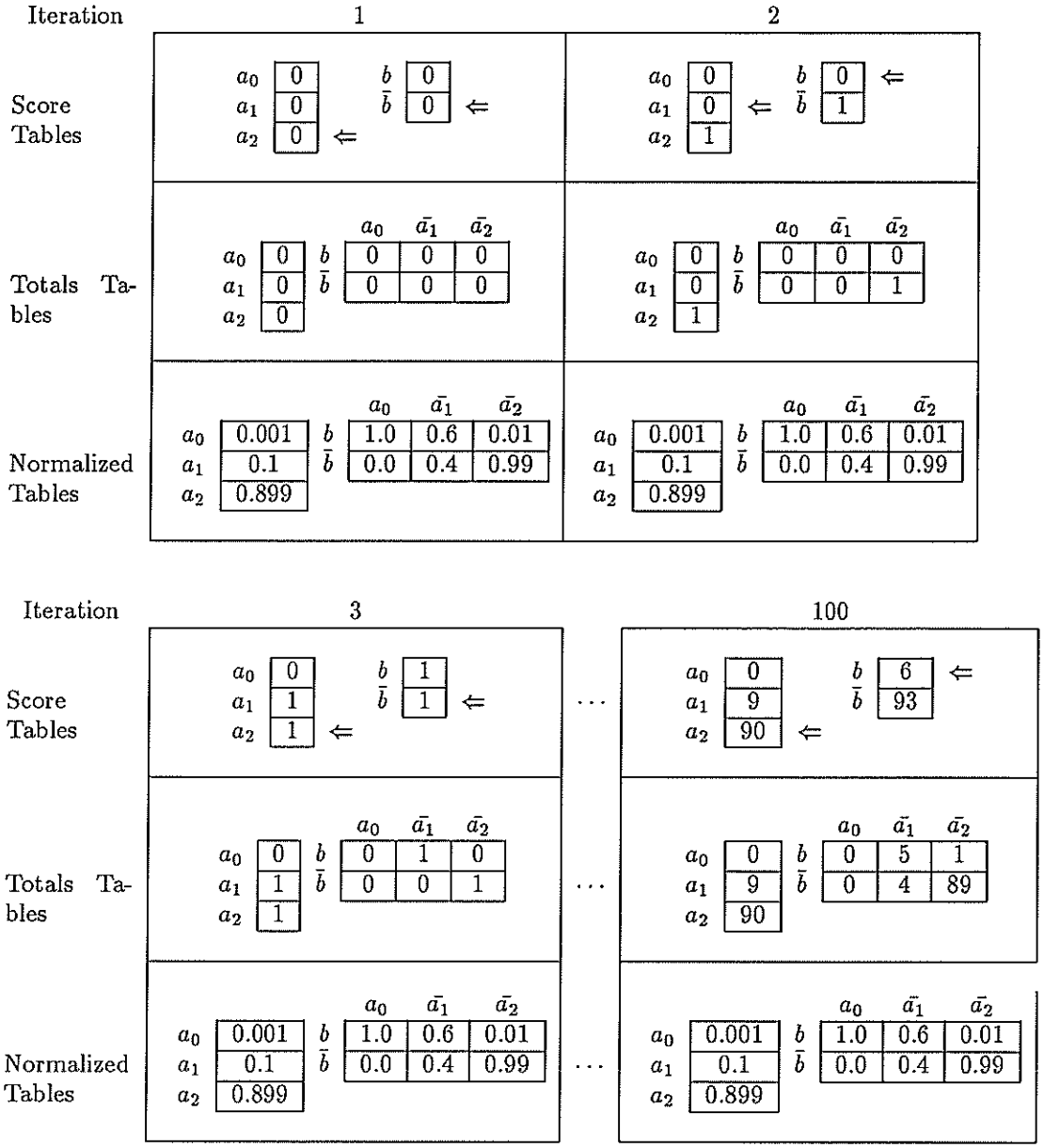
```
Sort the nodes of the graph so that parents always proceed children
Initialize the normalized-score and total-score tables
Loop for the number of simulations
     If this is the 100th iteration (this implementation), normalize (see below)
     p-selecting = 1.0
     For each non-evidence node in the network (in graph order)
         Choose a state according to the normalized table entry for that state
         p-selecting = p-selecting * normalized probability of chosen state
     End For
     Calculate a score (total probabilities / p-selecting)
     Score the net (using traditional or Markov blanket scoring)
End Loop
Normalize the node values
```

A normalizing step, invoked every 100 iterations in this implementation, replaces the values in normalized-score with a combination of the total score and the original conditional probabilities. The longer the algorithm runs, the smaller the weight the original conditional probabilities have in this combination. Pseudo-code for the normalization algorithm is given here:

```
Weight-ratio = constant / iteration number
For each non-evidence node in the network (in graph order)
    Copy the entries from total-score to normalized-score for this node
    Normalize normalized-score
    Calculate a weighted average between the accumulated score in
         normalized-score and the conditional probabilities, using Weight-ratio
    Re-Normalized normalized-score
End For
```

Stepping Through Self-Importance

Self-Importance keeps two additional tables per node as compared to Likelihood weighting, the totals tables and the normalized tables. The trials will give the same results as Likelihood weighting until iteration number 100.



During the first 100 iterations of Self-Importance, the values in the Score Tables are updated exactly as they were in Likelihood weighting. The Totals Tables are updated on each iteration by adding the score to the appropriate bin based on the state chosen (remember that in our notation we are showing the tables *before* each iteration, so the scores for iteration 1 are not noted until iteration 2). The algorithm is actually using the Normalized Tables for choosing states and computing p-selecting, but since the Normalized Tables are identical to the conditional probability tables, there is no difference in output from Likelihood weighting.

On iteration 100, the Normalized Tables are updated with a combination of the Totals Tables and the original conditional probability tables. The second parameter to Self-Importance (the first was how many iterations to go between normalization) is a weight parameter. In this implementation, we use a weight of 100. The weight parameter determines how much influence the conditional probabilities have compared to the Totals Tables in computing the new Normalized Tables. We calculate a ratio

$$R = \frac{\text{weight}}{\text{Number of Simulations}} = \frac{100}{100} = 1$$

Let $norm(T)$ represent a normalizing function for a table T, TT represent the Totals Tables, NT represent the Normalized Tables, and CP represent the original conditional probability tables. Then, the new normalized tables are computed as follows:

$$NT = norm(norm(TT) + (CP * R))$$

After 100 iterations, the Totals Tables were:

a_0	0
a_1	9
a_2	91

	a_0	$\bar{a_1}$	$\bar{a_2}$
b	0	5	2
$\bar{b}$	0	4	89

Normalizing these, we get

a_0	0.0
a_1	0.09
a_2	0.91

	a_0	$\bar{a_1}$	$\bar{a_2}$
b	1.0	0.58	0.015
$\bar{b}$	0.0	0.42	0.985

Adding these to the original conditional probability tables (times 1.0) and normalizing again, we get our new Normalized Tables:

a_0	0.0005
a_1	0.095
a_2	0.9045

	a_0	$\bar{a_1}$	$\bar{a_2}$
b	1.0	0.58	0.015
$\bar{b}$	0.0	0.42	0.985

Note that these tables are very close to the original conditional probability tables. This is because there is no evidence given to the network. The point of Self-Importance sampling is to use tables more relevant to the problem at hand, so Self-Importance will be more tailored to problems with evidence present.

With the normalization step complete, we can now continue for another 100 iterations. Remember that we now use the Normalized Tables when selecting states for A and B, so, for example, it is now only half as likely that we will select state a_0.

Iteration	101	102
Score Tables	a_0 0, a_1 9, a_2 91 ⇐ ; b 7, $\bar{b}$ 93 ⇐	a_0 0, a_1 9 ⇐, a_2 91.999 ; b 7 ⇐, $\bar{b}$ 93.999
Totals Tables	a_0 0, a_1 9, a_2 91 ; b: a_0 0, $\bar{a_1}$ 5, $\bar{a_2}$ 2; $\bar{b}$: 0, 4, 89	a_0 0, a_1 9, a_2 91.999 ; b: a_0 0, $\bar{a_1}$ 5, $\bar{a_2}$ 2; $\bar{b}$: 0, 4, 89.999
Normalized Tables	a_0 0.0005, a_1 0.095, a_2 0.9045 ; b: a_0 1.0, $\bar{a_1}$ 0.58, $\bar{a_2}$ 0.015; $\bar{b}$: 0.0, 0.42, 0.985	a_0 0.0005, a_1 0.095, a_2 0.9045 ; b: a_0 1.0, $\bar{a_1}$ 0.58, $\bar{a_2}$ 0.015; $\bar{b}$: 0.0, 0.42, 0.985

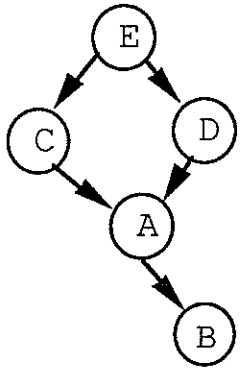


Figure 1.6: Simple example extended to 5 nodes.

The change to the Normalized Tables changed p-selecting, so the score added on each new iteration will no longer be 1.0. For convenience, we will refer to the probabilities in the Normalized Tables as $\tilde{p}$. The new score is calculated as follows:

$$\begin{aligned} \text{total probability} &= p(a_2)p(\bar{b} \mid a_2) = 0.890 \\ \text{p-selecting} &= \tilde{p}(a_2)\tilde{p}(\bar{b} \mid a_2) = .9045 * .985 = 0.06 \\ \text{score} &= \frac{\text{total probability}}{\text{p-selecting}} = .999 \end{aligned}$$

Note that we used the original conditional probability tables to calculate the total probability, but the Normalized Tables ($\tilde{p}$) to calculate p-selecting.

At any point, we can stop the simulation and calclate final probabilities by normalizing the Score Tables, just as we did in Likelihood weighting.

1.2.4 Markov Blanket Scoring

In our examples so far, we have been using a simple scoring method in which the calculated score is added to a single bin in each Score Table. Markov Blanket Scoring is an alternative method in which the score is broken up and spread over multiple bins according to some distribution. Markov Blanket scoring has a significant computational cost associated with it, but in many cases pays for itself with more accurate results.

The Markov blanket of a node takes into account the states of the parent nodes, child nodes, and parents of child nodes in order to derive a value for every state of the current node. Given node N with p_N parents and c_N children, let $P(N)_i$ represent the ith parent of node N ($1 \leq i \leq p_N$), $C(N)_i$ represent the ith child of node N ($1 \leq i \leq c_N$), and s_N represent a particular state of node N. A Markov blanket value is calculated for each state s_N of N as follows:

$$\begin{aligned} mb(s_N) = \; & p(N = s_N \mid P(N)_1 = s_{P(N)_1}, \ldots, P(N)_{p_N} = s_{P(n)_{p_N}}) * \\ & \prod_{i=1}^{c_N} p(C(N)_i = s_{C(N)_i} \mid P(C(N)_i)_1 = s_{P(C(N)_i)_1} \cdots \\ & P(C(N)_i)_{p_{C(N)_i}} = s_{P(C(N)_i)_{p_{C(N)_i}}} \end{aligned}$$

The values of $mb(s_N)$ for all states of N together constitute the Markov blanket of N. Note that the Markov blanket values based on the states of the neighbor nodes of N.

In order to demonstrate Markov Blanket Scoring, we need to add some nodes to our sample two-node network. Our extended example, shown in Figure 1.6, has three additional nodes:

C - The boss is in town (boolean: c or $\bar{c}$)

D - The student stayed up late last night (d or $\bar{d}$)

E - Today is Monday (e or $\bar{e}$)

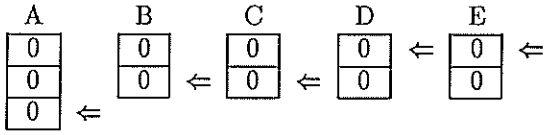
The full conditional probability table is as follows:

$p(e)$	=	0.2	One in five working days is Monday
$p(c \mid e)$	=	0.5	Half of the time the boss is gone on Monday
$p(c \mid \bar{e})$	=	0.85	Other days the boss is likely in town
$p(d \mid e)$	=	0.95	Student is almost always up late Sundays
$p(d \mid \bar{e})$	=	0.8	In fact, the student is up late a lot
$p(a_0 \mid c, d)$	=	0.001	Same as previous example
$p(a_1 \mid c, d)$	=	0.1	
$p(a_2 \mid c, d)$	=	0.899	
$p(a_0 \mid c, \bar{d})$	=	0.01	More likely to come in early if went to bed early
$p(a_1 \mid c, \bar{d})$	=	0.4	
$p(a_2 \mid c, \bar{d})$	=	0.59	
$p(a_0 \mid \bar{c}, d)$	=	0.0	If boss is away, no chance of early arrival
$p(a_1 \mid \bar{c}, d)$	=	0.01	
$p(a_2 \mid \bar{c}, d)$	=	0.99	
$p(a_0 \mid \bar{c}, \bar{d})$	=	0.001	Boss away, but went to bed early...
$p(a_1 \mid \bar{c}, \bar{d})$	=	0.2	slight chance of early arrival
$p(a_2 \mid \bar{c}, \bar{d})$	=	0.799	
$p(b \mid a_0)$	=	1.0	Same as before
$p(b \mid a_1)$	=	0.6	
$p(b \mid a_2)$	=	0.01	

Stepping Through Logic Sampling with Markov Blanket Scoring

We will use this scenario to demonstrate Markov Blanket Scoring. In this example, we will use Logic Sampling with no evidence present. The same technique is used to use Markov Blanket Scoring with any of the other algorithms.

Iteration 1



Since we are using Logic Sampling, the states are selected from a uniform distribution. We begin by computing a score as usual.

$$\begin{aligned} \text{total probability} &= p(e)p(\bar{c} \mid e)p(d \mid e)p(a_2 \mid \bar{c}, d)p(\bar{b} \mid a_2) \\ &= 0.2 * 0.5 * 0.95 * 0.99 * 0.99 \\ &\cong 0.093 \\ \text{p-selecting} &= 1.0 \\ \text{score} &= 0.093 \end{aligned}$$

With traditional scoring, we had to add the score to a state of each node. With Markov Blanket Scoring, the work at each node is more complex. The Markov Blanket (mb) for each node is calculated, and used to update the Score table for that node.

For A: The Markov Blanket contains C, D, B, and A. C, D, and B remain fixed in their states $(\bar{c}, d, \bar{b})$ and we calculate the mb vector for the states of A:

$$\begin{aligned} mb(a_0) &= p(a_0 \mid \bar{c}, d)p(\bar{b} \mid a_0) = 0.0 \\ mb(a_1) &= p(a_1 \mid \bar{c}, d)p(\bar{b} \mid a_1) = 0.004 \\ mb(a_2) &= p(a_2 \mid \bar{c}, d)p(\bar{b} \mid a_2) = 0.98 \end{aligned}$$

The Markov Blanket is then normalized, multiplied by the score, and added to the Score table for A. This ensures that the same total amount of score is added to the Score table, but the score will be distributed across the states of A based on the likelihood of those states given the states of A's neighbors.

For B: The Markov Blanket contains only A and B.

$$\begin{aligned} mb(b) &= p(b \mid a_2) = 0.01 \\ mb(\bar{b}) &= p(\bar{b} \mid a_2) = 0.99 \end{aligned}$$

Again, this vector is normalized, multiplied by the score, and added to the Score table for B.

For C: The Markov Blanket contains A, E, D, and C.

$$\begin{aligned} mb(c) &= p(c \mid e)p(a_2 \mid c, d) = 0.5 * 0.899 = 0.4495 \\ mb(\bar{c}) &= p(\bar{c} \mid e)p(a_2 \mid \bar{c}, d) = 0.5 * 0.99 = 0.495 \end{aligned}$$

For D: The Markov Blanket contains A, E, C, and D.

$$\begin{aligned} mb(d) &= p(d \mid e)p(a_2 \mid c, d) = 0.95 * 0.899 = 0.8541 \\ mb(\bar{d}) &= p(\bar{d} \mid e)p(a_2 \mid c, \bar{d}) = 0.05 * 0.59 = 0.0295 \end{aligned}$$

For E: The Markov Blanket contains C, D, and E.

$$\begin{aligned} mb(e) &= p(e)p(\bar{c} \mid e)p(d \mid e) = 0.2 * 0.5 * .95 = 0.095 \\ mb(\bar{e}) &= p(\bar{e})p(\bar{c} \mid \bar{e})p(d \mid \bar{e}) = 0.2 * 0.15 * 0.8 = 0.024 \end{aligned}$$

Iteration 2

A		B		C		D		E	
0.0	⇐	.00093		.0443		.0899	⇐	.0742	
.000372		.09207	⇐	.0487	⇐	.0031		.0188	⇐
.092628									

This scenario represents an impossible day for the student. After staying up late, he came in early but did not find a parking space. The boss is not even in town to see him. At least it is not Monday. This is the kind of state that can happen with Logic Sampling, but which is impossible using the weighted selection of Likelihood weighting.

$$\begin{aligned} \text{total probability} &= p(\bar{e})p(\bar{c} \mid \bar{e})p(d \mid \bar{e})p(a_0 \mid \bar{c}, d)p(\bar{b} \mid a_0) \\ &= 0.8 * 0.15 * 0.8 * 0.0 * 0.0 \\ &= 0.0 \\ \text{p-selecting} &= 1.0 \\ \text{score} &= 0.0 \end{aligned}$$

The score indicates the kind of weight we would give to such an impossible situation: 0. We will not go through the Markov steps for this example since they all involve multiplying by zero.

This example continues, and the final normalization step is performed on the Score Table as usual. Note that Markov Blanket Scoring could be turned on and off on a node-by-node basis, and the results would still converge. The challenge to using this feature is to know when Markov Blanket Scoring is appropriate, and when its expense does not justify its use.

1.2.5 Pearl

Pearl's Markov blanket algorithm is more complicated than the above algorithms because it requires the calculation of a Markov blanket for a node in all cases. Markov blanket *scoring* can be added to the other algorithms, but is integrally required in Pearl's algorithm. In addition, Pearl's algorithm can and should use Markov Blanket Scoring, because since the Markov Blanket is already calculated, the incremental cost of this method is very small.

Pearl's algorithm begins by assigning each node a random state, and thereafter using the Markov blanket to choose a new state for each node as it is processed:

```
Make a unbiased random state assignment for each non-evidence node
For each node, N, and each state of N, s_N, let bin(N,s_N) = 0

Repeat for desired number of iterations
       For each non-evidence node in the network, N
           Calculate mb(s_N) for all states s_N of N
           Add a score to bin(N,s_N), where s_N is the current state (see below)
           Set the new state of N based on the distribution in mb
       End For
End Repeat
Normalize the values in bin for all nodes
Use the normalized values as the node probabilities.
```

Pearl's algorithm can be implemented with or without Markov blanket scoring. Markov blanket scoring is especially useful with Pearl's algorithm because the Markov blanket is already calculated, so there is no added cost to using this scoring technique. To use Markov blanket scoring with Pearl, the scoring step is as follows:

```
For each state s_N of N
    bin(N,s_N) = bin(N,s_N) + mb(s_N)
End For
```

If Markov blanket scoring is not used, the scoring step reduces to simply

```
bin(N,s_N) = bin(N,s_N) + 1
```

where s_N is the current state of N.

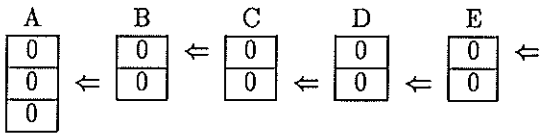
Stepping through Pearl

We will demonstrate this algorithm with the example we introduced in the previous section. Since this algorithm has significant state changes in an inner loop, we will use notation such as 1-A to indicate that we are on the first iteration, considering node A. In this example, we have given the network no evidence, and will use Markov Blanket Scoring.

Pearl's algorithm begins by assigning a random state to the network, using the same technique as Logic Sampling. For this example, the initial state chosen is

$$(a_1, b, \bar{c}, \bar{d}, e)$$

Iteration 1-A



	mb(A)		
a_0	0.0	=	$p(a_0 \mid \bar{c}, \bar{d})p(b \mid a_0)$
a_1	0.12	=	$p(a_1 \mid \bar{c}, \bar{d})p(b \mid a_1)$
a_2	0.00799	=	$p(a_2 \mid \bar{c}, \bar{d})p(b \mid a_2)$

Normalizing, we get

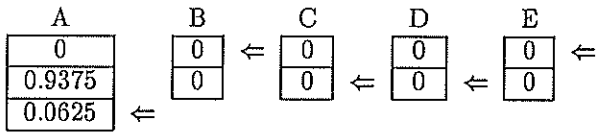
	$norm(mb(A))$
a_0	0
a_1	0.9375
a_2	0.0625

At this point, we both assign a new state to A, and update A's score bins. We use the normalized Markov Blanket to assign a state to A, as follows:

$$A = \begin{cases} a_0 & \text{if } rn < 0.0 \\ a_1 & \text{if } 0.0 \leq rn < 0.9375 \\ a_2 & \text{if } rn \geq 0.9375 \end{cases}$$

On this iteration, we generated $rn = 0.97$, so we chose $A = a_2$. Using Markov Blanket Scoring, we add the $norm(mb(A))$ vector directly to the scoring bins for A. If we had been using non-Markov Blanket Scoring, we would have scored by adding 1 to the a_2 bin, since we chose state $A = a_2$.

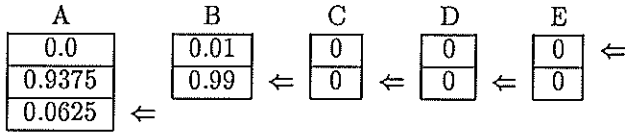
Iteration 1-B



	mb(B)		
b	0.01	=	$p(b \mid a_2)$
$\bar{b}$	0.99	=	$p(\bar{b} \mid a_2)$

This table happens to be normalized already ($norm(mb(B)) = mb(B)$)), so we proceed to choose a new state for B. We generate $rn = 0.57$, so we choose $B = \bar{b}$.

Iteration 1-C



	mb(C)		
c	0.295	=	$p(c \mid e)p(a_2 \mid c, \bar{d})$
$\bar{c}$	0.3995	=	$p(\bar{c} \mid e)p(a_2 \mid \bar{c}, \bar{d})$

Normalizing, we get

	$norm(mb(C))$
c	0.425
$\bar{c}$	0.575

We choose a new state for C given a random number rn

$$C = \begin{cases} c & \text{if } rn < 0.425 \\ \bar{c} & \text{if } rn > 0.425 \end{cases}$$

In this case, we choose $rn = .155$, so $C = c$.

Iteration 1-D

A	B	C	D	E
0.0	0.01	0.425	0	0
0.9375	0.99	0.575	0	0
0.0625				

mb(D)

d	0.854	=	$p(d \mid e)p(a_2 \mid c, d)$
$\bar{d}$	0.0295	=	$p(\bar{d} \mid e)p(a_2 \mid c, \bar{d})$

Normalizing, we get

	$norm(mb(D))$
d	0.966
$\bar{d}$	0.034

We choose $rn = .637$, so $D = d$.

Iteration 1-E

A	B	C	D	E
0.0	0.01	0.425	0.966	0
0.9375	0.99	0.575	0.034	0
0.0625				

mb(E)

e	0.095	=	$p(e)p(c \mid e)p(d \mid e)$
$\bar{e}$	0.544	=	$p(\bar{e})p(c \mid \bar{e})p(d \mid \bar{e})$

Normalizing, we get

	$norm(mb(E))$
e	0.149
$\bar{e}$	0.851

We choose $rn = .721$, so it is not Monday: $E = \bar{e}$.

At this point we have completed a single iteration of Pearl's algorithm. At the end of any iteration we can stop and normalize by simply dividing all scoring bins by the number of iterations. This simple normalization method works as long as we score each node on each iteration*.

Iteration 2-A

A	B	C	D	E
0.0	0.01	0.425	0.966	0.149
0.9375	0.99	0.575	0.034	0.851
0.0625				

mb(A)

a_0	0.0	=	$p(a_0 \mid c, d)p(\bar{b} \mid a_0)$
a_1	0.04	=	$p(a_1 \mid c, d)p(\bar{b} \mid a_1)$
a_2	0.89	=	$p(a_2 \mid c, d)p(\bar{b} \mid a_2)$

Normalizing, we get

	$norm(mb(A))$
a_0	0
a_1	0.043
a_2	0.957

We pick $rn = 0.562$, so our new state for A is a_2. We also add $norm(mb(A))$ to the scoring bins for A, giving $a_0 = 0$, $a_1 = 0.9805$, and $a_2 = 1.0195$.

1.2.6 Chavez

The *Chavez's algorithm* essentially executes multiple separate runs of Pearl's algorithm. Scores are only updated each suite of runs of Pearl's algorithm. By interrupting and restarting the algorithm periodically, the Chavez algorithm avoids local minima, thereby allowing one to calculate *error bounds* for each node's value. These bounds allow computation to be directed on specific high-interest areas of a network. In some settings, this potential may offset the Chavez algorithm's

*Updating every node on every iteration is not necessary, but is the way CABeN implements Pearl's algorithm

generally higher computation cost, suggesting that it is not always ideal to select an algorithm solely on the basis of its execution time.

```
Repeat for desired number of iterations
        Make a unbiased random state assignment for each non-evidence node
        For each node, N, and each state of N, s_N, let bin(N,s_N) = 0
        Repeat for number of iterations of Pearl (before restart)
                For each non-evidence node in the network, N
                    Calculate mb(s_N) for all states s_N of N
                    Set the new state of N based on the distribution in mb
                End For
        End Repeat (Pearl)
        For each non-evidence node in the network, N (now score)
                Add a score to bin(N,s_N), where s_N is the current state (see below)
        End For
End Repeat
```

In addition to having multiple restarts, Chavez's algorithm differs from Pearl's algorithm in that scores are only computed before each restart (and at the end). Initially, the scoring tables for Chavez's algorithm look like those for Pearl:

A	B	C	D	E
0	0	0	0	0
0	0	0	0	0
0				

$\Leftarrow$ between A and B, $\Leftarrow$ between B and C, $\Leftarrow$ between C and D, $\Leftarrow$ between D and E.

The special parameter of this algorithm is how many Pearl iterations to do before each restart. CABeN uses 20 as the default number of Pearl iterations before scoring and restarting, so we will use 20 here.

Iteration 1.A Initially, we have $(a_1, b, \bar{c}, \bar{d}, e)$ (chosen at random using logic sampling, as in Pearl). We compute mb(A), normalize it, and use it to choose a state for A as in Pearl's algorithm.

	$norm(mb(A))$
a_0	0
a_1	0.9375
a_2	0.0625

This time, we have $rn = 0.671$, so we choose state $A = a_1$. Note that the scores remain zero, because unlike Pearl's algorithm, Chavez will not score until iteration 20.

Iteration 1.B $(a_1, b, \bar{c}, \bar{d}, e)$. Choose a new state for B based on $norm(mb(B))$.

$\vdots$

Iteration 20.E $(a_2, \bar{b}, c, d, \bar{e})$

	$norm(mb(E))$
e	0.149
$\bar{e}$	0.851

Let $rn = 0.558$, so choose $E = \bar{e}$

Scoring Step: If we are using non-Markov Blanket Scoring, we add 1 to each currently selected bin $(a_2, \bar{b}, c, d, \bar{e})$. For Markov Blanket Scoring, we compute the normalized Markov Blanket for each node, and add that to the scoring bins, exactly as we did in Pearl.

1.3 Algorithm Comparison

In this section, we describe an ongoing series of experiments designed to assess how the various algorithms perform under different circumstances. Although it is clear that more rigorous study is

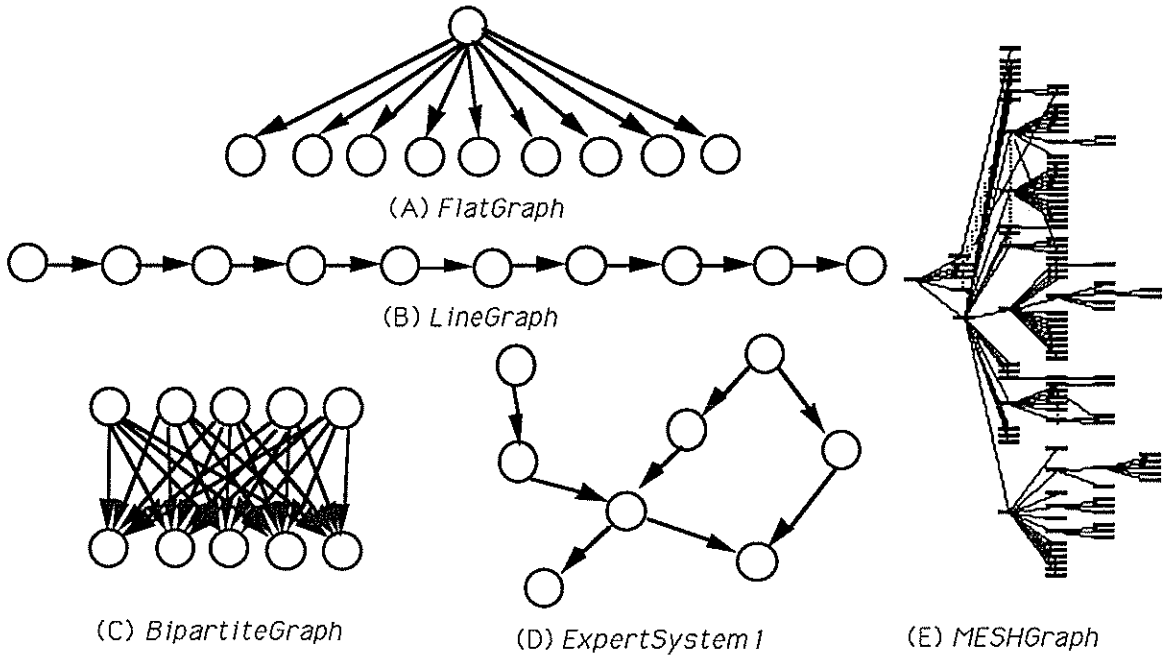


Figure 1.7: Five graph topologies used to compare the algorithms.

needed, it is clear that the biggest variable to algorithm performance is the structure of the belief network itself. Features of the belief network which we have studied are the topology, the extreme conditional probabilities, and the amount of evidence to consider. In addition, we have examined how well the algorithms perform in terms of the number of iterations required for convergence, the time required for convergence, and whether Markov scoring helps. Table 1.2 summarizes many of our findings. The following paragraphs describe the experiments which led to the conclusions in Table 1.2.

We compared the algorithms on the five network topologies shown in Figure 1.7. The networks are: *FlatGraph*, a hierarchical graph with one parent "root" node and nine child nodes; *LineGraph*, a serial order of ten nodes; *BipartiteGraph*, a two-layer graph with five parent nodes and five child nodes in which each node in the child node layer is connected to each node in the parent node layer; *ExpertSystem1*, a graph taken from an influential paper by Lauritzen and Spiegelhalter [3]; and *MeSHGraph*, a slightly modified 109-node graph taken from the MeSH cardiovascular diseases subtree. Probabilities were assigned to nodes in *BipartiteGraph*, *LineGraph*, *FlatGraph*, and *MeSHGraph* by assigning root nodes probabilities of 0.5 and assigning conditional probabilities between parent (p) nodes and child (c) nodes $p(c \mid p) = 0.8$ and $p(c \mid \bar{p}) = 0.05$. Conditional probabilities for nodes with more than one parent were computed using the *noisy-or* function [8]. Probabilities for *ExpertSystem1* were the same as those assigned by Lauritzen [3] with the exception that probabilities of 1 or 0 were modified slightly to simplify evaluation of the Pearl and Chavez algorithms (which require special treatment for values of 1 and 0).

Figures 1.8 and 1.9 present our experimental results. For each algorithm, absolute mean-squared-error (*mse*) is plotted against CPU-time. Each point is the average of at least 20 trials. The lines in the graphs represent the best-fit logarithmic models of the data, generated using Cricket Graph on a Macintosh. Each algorithm was run both with and without Markov blanket scoring. We present data only for the mode in which each algorithm performed the best (as determined by previous criteria [10]). In the illustrations, Markov scoring is indicated by a '/M' following the algorithm's name.

Algorithm performance was examined by plotting the *mse* value against CPU-time (Figures 1.8-

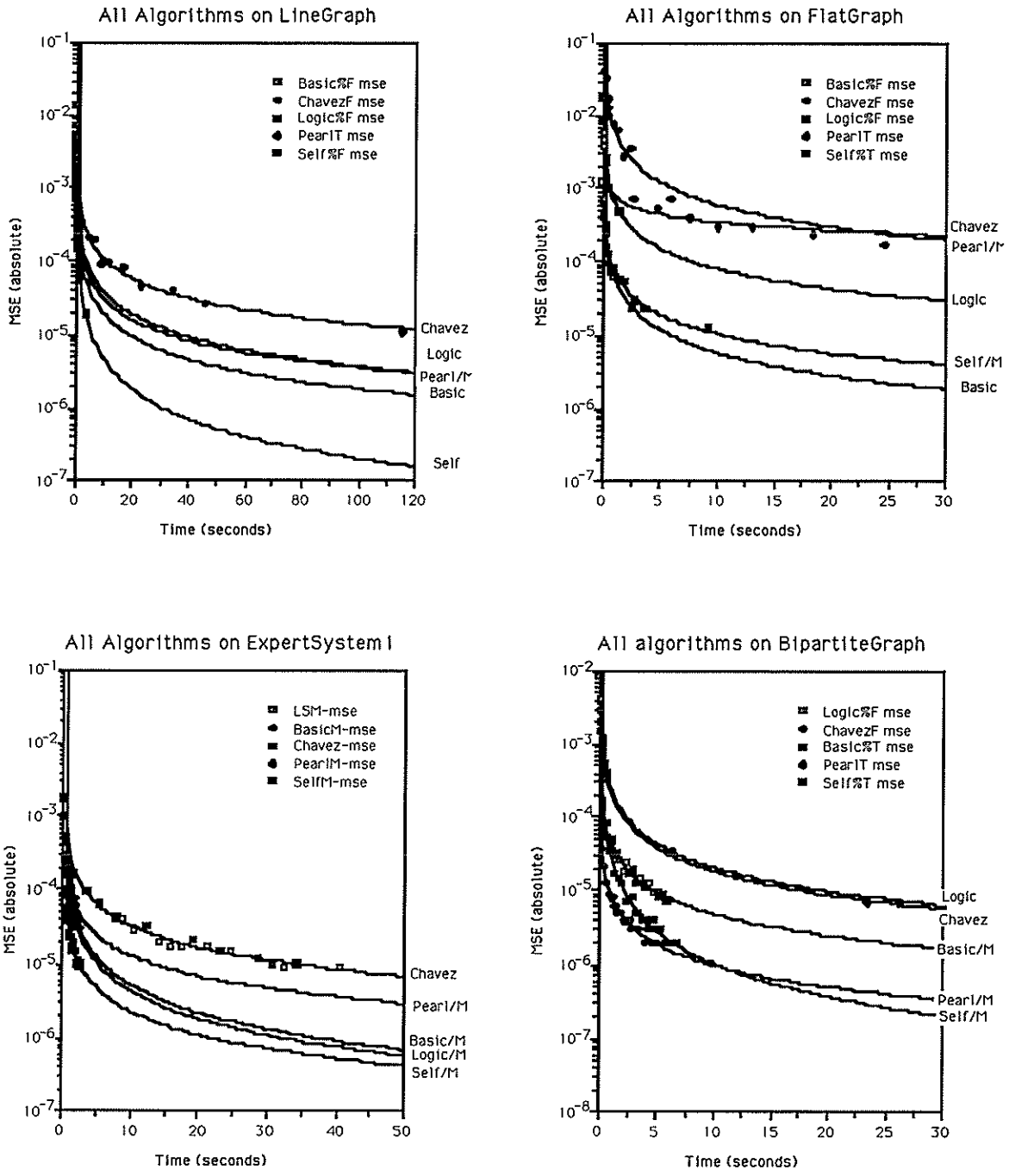


Figure 1.8: Overall performance evaluation on the four simple networks. The Self-Importance algorithm does well on each network topology.

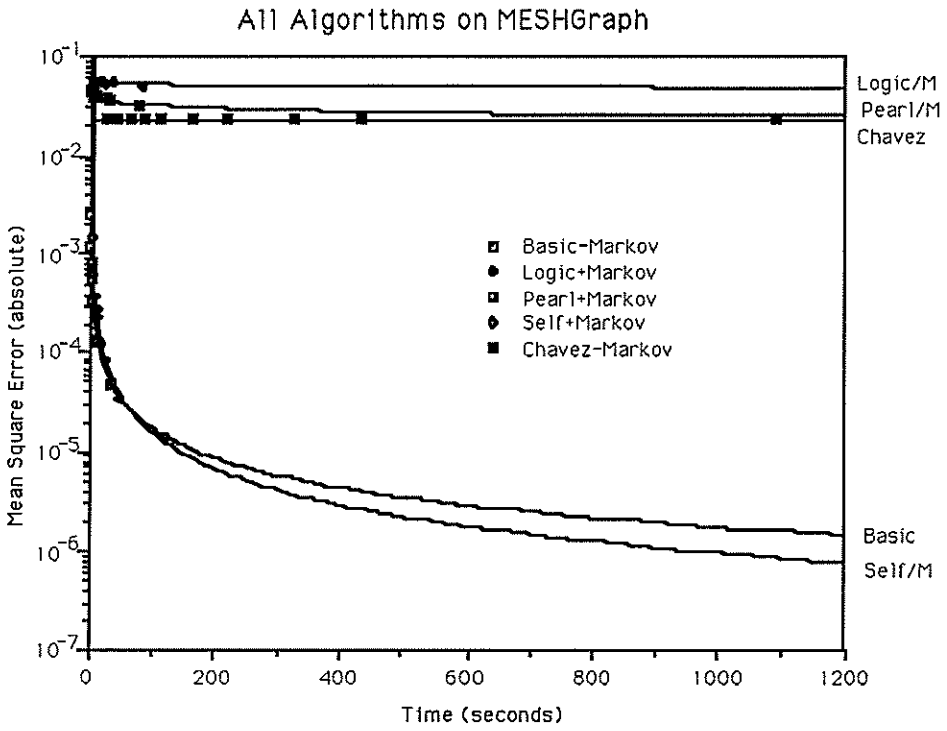


Figure 1.9: Performance of the algorithms on *MeSHGraph*. Likelihood weighting and Self-Importance algorithms both perform very well.

Table 1.2: Comparison of the algorithms under various circumstances.

	Topology	Markov Scoring	Conditional Probabilities near 0 or 1
Logic Sampling	Performs well on simple topologies	May hurt	No effect
Likelihood weighting	Performs well on simple topologies	Improvement	No Effect
Self-Importance	Slow when many parents	Big Improvement	No Effect
Pearl	Performs well with complex networks	Helps, and has no cost	Does not converge to exact solution
Chavez	Exponential with largest Markov blanket	No Help (but no cost)	Does not converge to exact solution, but performs better than Pearl

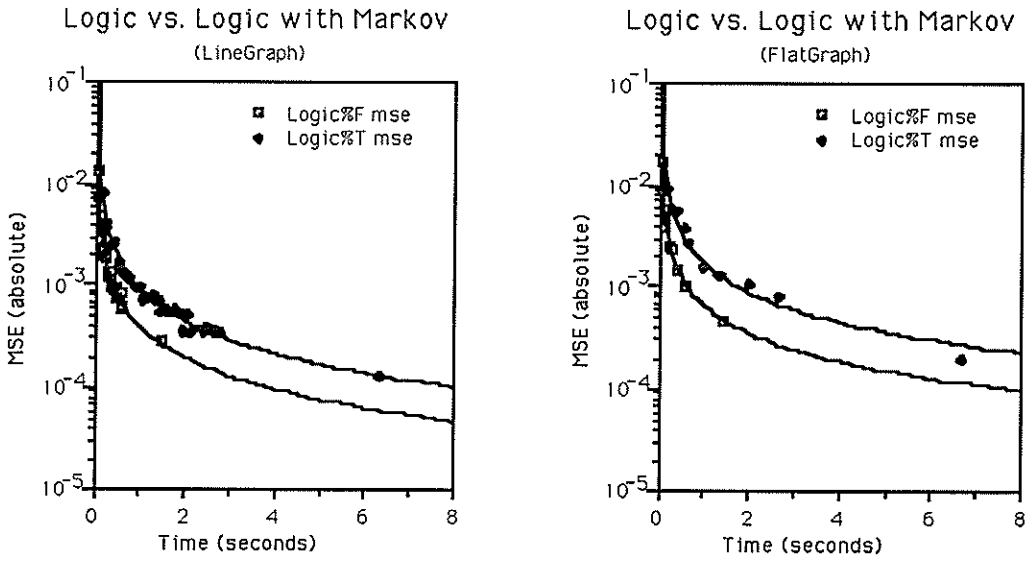


Figure 1.10: Markov blanket scoring actually does worse on very simple network topologies with Logic Sampling.

1.9). All algorithms appear to have similar performance profiles; the *mse* decreases as a function of the square root of the number of iterations. Lower values for *mse* indicate better performance. For the *MeSHGraph* network, both the *Self-Importance* and *Likelihood weighting* algorithms performed well. The *Chavez* algorithm required eight times as much CPU time to achieve the same *mse* as the *Likelihood weighting* algorithm, but error bounds obtained during the Chavez algorithm calculation suggest that the algorithm may help identify where more extensive computations are needed to achieve better probability estimates. Other, more qualitative conclusions about algorithm performance are listed in Table 1.2.

We compared scoring methods (Markov blanket or traditional) for each algorithm for each network. *ExpertSystem1* and *BipartiteGraph* shared similar results when compared with the two trivial networks, *LineGraph* and *FlatGraph*. For the Logic Sampling algorithm, there was no significant difference with or without Markov blanket scoring on complex graphs (*ExpertSystem1*, *BipartiteGraph*)

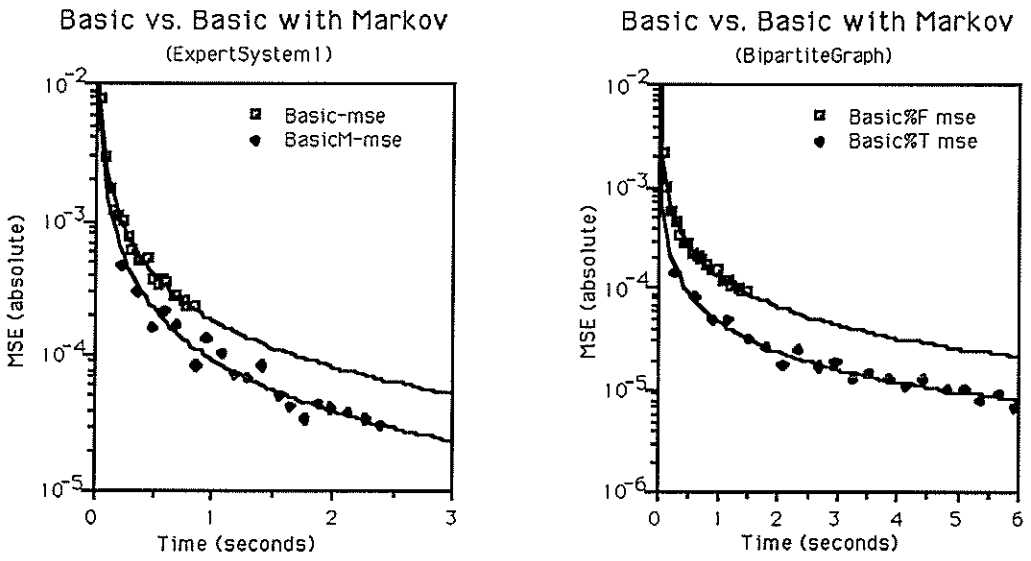


Figure 1.11: Markov blanket scoring helps Likelihood weighting on relatively complex graph topologies.

and Markov blanket scoring actually performed worse on simple graphs (Figure 1.10). Markov blanket scoring did not seem to affect the results of Likelihood weighting on simple graphs, and appeared to do better on the two complex graphs (Figure 1.11). Self-Importance algorithm behavior was similar to that of Likelihood weighting: Markov scoring did little to improve performance on simple graphs but was a great aid when applied to complex graphs (Figure 1.12). Markov scoring slightly improved Pearl algorithm performance when applied to the *ExpertSystem1* network and very significantly improved performance when applied to the *BipartiteGraph* network (Figure 1.13). We note that Markov scoring imposed no additional computation burden when applied to Pearl's algorithm because this calculation is an intrinsic part our implementation. The Chavez algorithm derives no benefit from Markov scoring in any of the cases examined.

Deteriorating performance is noted in some approximation algorithms when individual node values approach zero or one. To examine this phenomenon, we varied the conditional probabilities at a node in the *ExpertSystem1* network while keeping the number of iterations constant (Figure 1.14). As expected, the results clearly show a degradation of performance in the Pearl and Chavez algorithms as conditional probabilities approach zero.

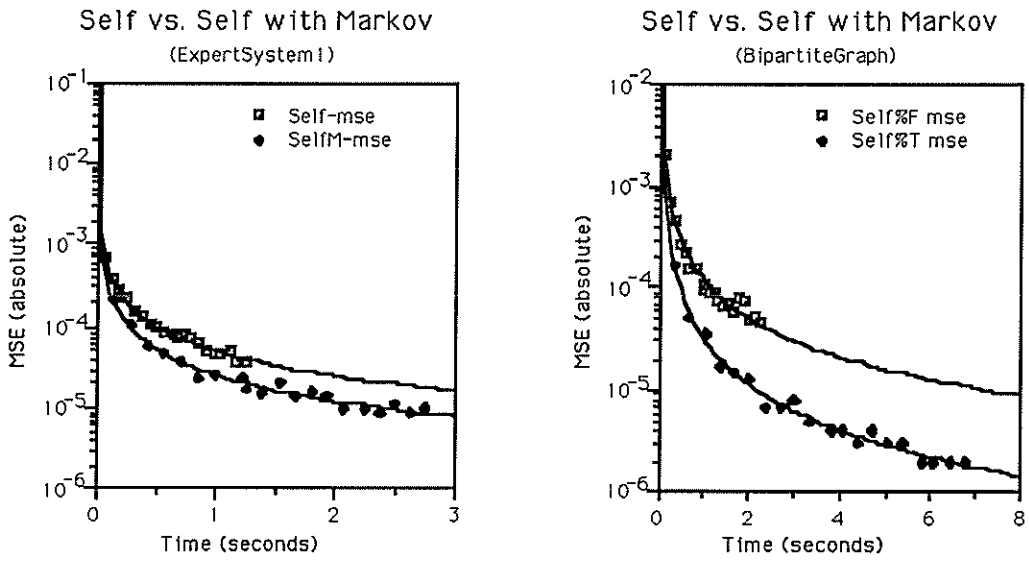


Figure 1.12: Self-Importance does better on complex networks with Markov blanket scoring.

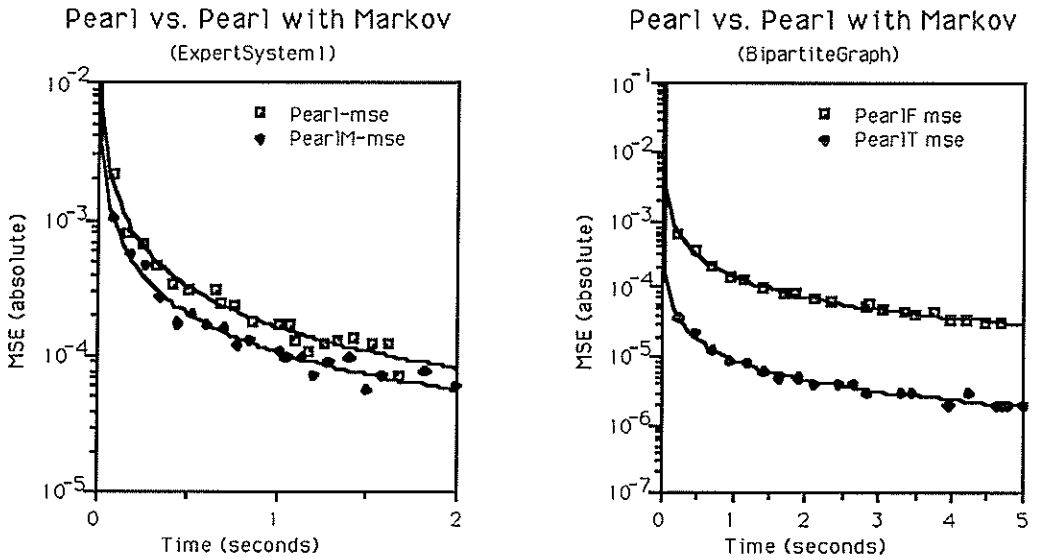


Figure 1.13: Pearl's algorithm does very well when Markov blanket scoring is used.

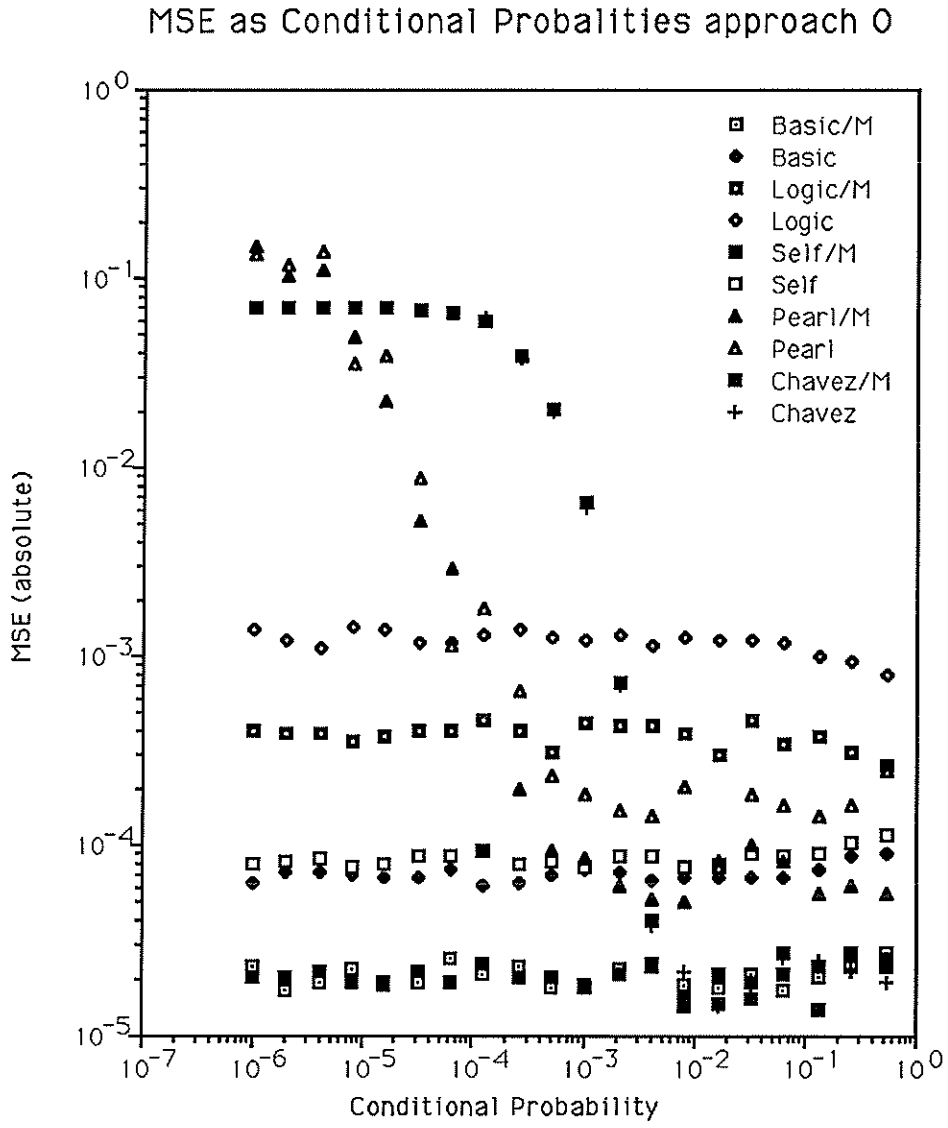


Figure 1.14: The effects of varying one node's conditional probability value on the convergence of the various algorithms.

Chapter 2

User's Manual

CABeN can be used in a variety of environments. The default environment is a menu-based system for a tty environment. This system runs on PC and unix platforms at the current time, and should be easily portable to any environment which has a C compiler. We are also working on an interactive LISP-based environment, which would be interfaced with a LISP-based graph- browsing tool [Cousins88a].

We have defined a belief-network file format which can be used to move belief-networks between applications. The details of the format are included in Appendix A.

2.1 The Menu Interface

The menu interface provides interactive access to the features of CABeN on virtually any screen. Networks can be created, edited, saved, and loaded; probabilities can be set or defaults may be used; and any of the algorithms can be run using this interface. An overview of the menu interface is shown in figure 2.1.

To access any of the menu items, simply type the letter next to the action you with to perform, and then press the return key. When you are prompted for information, values in square brackets '[]' indicate the current value. This is also the value which will be used if you press return rather than typing a response to a prompt.

The rest of this section will describe the features of the various menus. In this interface, every menu has an 'X- Exit' option. Exiting from a submenu returns to the parent menu; exiting from the top level menu terminates the program.

2.1.1 Belief Network Options

The top level 'Belief Network Options' menu provides basic network management functions:

- 'Start new network' clears out the current network.
- 'Modify network' leads to another menu ('Change Network Menu', described below) which allows the network (including node probabilities) to be edited.
- 'Save network' writes the current network to disk in our belief network file format (see Appendix A). If answers have been generated with the correct answers (brute force) algorithm or a confidence test, they are saved along with the network.
- 'Read network from disk' loads a file in belief network file format, replacing any network in memory.
- 'Add a node type' allows new node types to be added. Node types define the states which nodes can have. For example, nodes of type 'Boolean' have two states, 'T' and 'F'. When this menu option is chosen, the system prompts for the name of the new type, the number of states

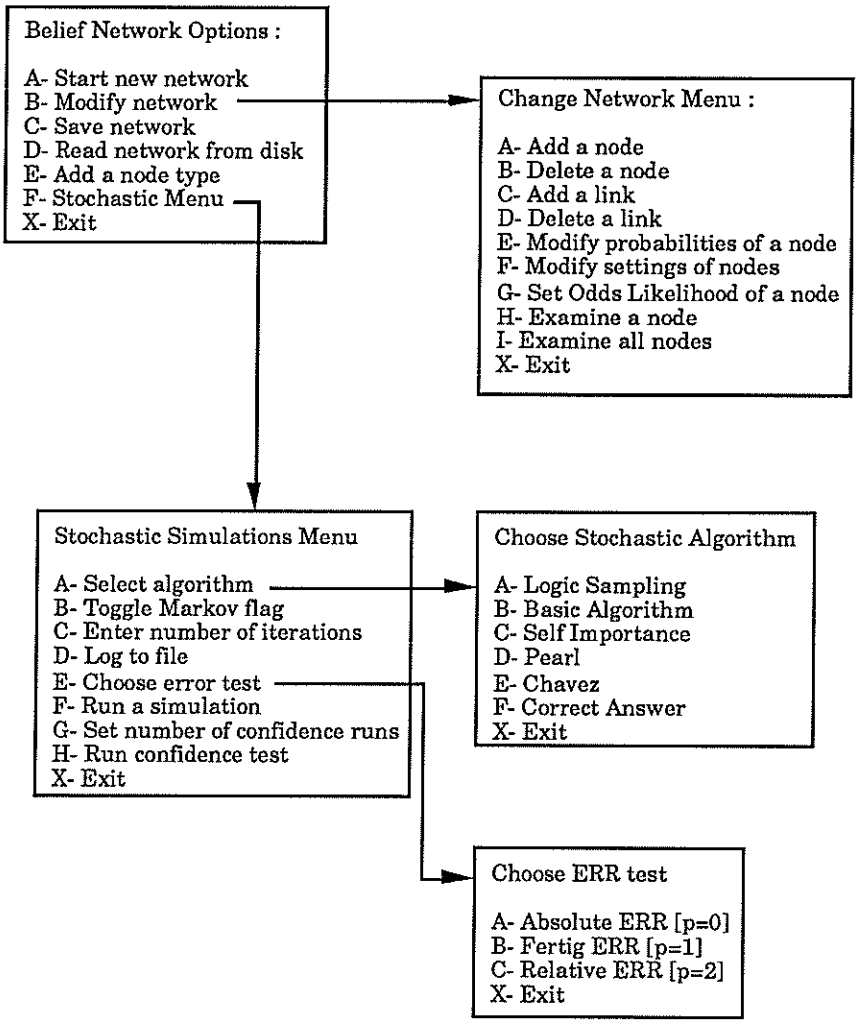


Figure 2.1: Overview of the menu interface.

it has, and the names of the states. The type definitions of all nodes in a network are saved along with the network in the belief network file format. Type names may not contain white space characters (such as spaces, tabs or newline characters)

- The 'Stochastic Menu' option gives access to another menu from which simulation algorithms and their parameters can be set, and simulations can be run.

2.1.2 Change Network Menu

The 'Change Network Menu' is a simple network editor. It allows the user to change the structure of the network, as well as the probabilities of nodes. In addition, nodes may be fixed in one of their states, or given 'evidence' in favor of one of their states:

- 'Add a node' requests a name for a new node in the network. The name must be unique among nodes in the network. Node names may contain spaces: they are terminated by a newline character.
- 'Delete a node' removes a node from the current network. The node name must be specified exactly.
- 'Add a link' requests a parent node and a child node, and creates a link from parent to child. Whenever a node's parents change in a belief network, its conditional probabilities may have to be updated. CABeN provides a default probability by making a conditional independence assumption. When the link is added, this default may be accepted, or rejected. If the default is rejected, CABeN asks for values for all affected probabilities.
- 'Delete a link' removes a link between two nodes. Once again, probabilities for the child node will change, and CABeN gives the choice of taking default reconditioned probabilities or explicitly specifying new ones.
- 'Modify probabilities of a node' iterates through the conditional probabilities of the node named, and allows each to be set.
- 'Modify settings of nodes' allows a node to be fixed in a given state (with possible states defined by the node's type).
- 'Set Odds Likelihood of a node' gives evidence to a node in favor of one or more of its states. In the limit, by giving enough evidence to a particular state, the results of this operation are the same as using the previous option.
- 'Examine a node' prints a report describing a node.
- 'Examine all nodes' prings a report describing all nodes in the network.

2.1.3 Stochastic Simulations Menu

The 'Stochastic Simulations Menu' allows simulation parameters to be set (such as which algorithm to run, the number of iterations, etc.) and allows a simulation to be run. A confidence test runs a simulation multiple times and generates an estimate of the correct answers. It is intended to convey a degree of confidence in the answers given by running an algorithm a certain number of times.

- 'Select algorithm' leads to another menu (which will not be described in more detail) which simply allows one of the available algorithms to be chosen.
- 'Toggle Markov flag' changes the state of the Markov flag. The Markov flag indicates whether or not Markov scoring will be used with the selected algorithm.
- 'Enter number of iterations' sets the number of iterations the selected algorithm will run for.

- 'Log to file' allows the results of the simulation to be written to a file.
- 'Choose error test' lets the error test be changed. Note that an error test is only relevant if the correct answer for the run is known in advance. This option is useful for comparing algorithms only.
- 'Run a simulation' starts the selected algorithm with the network in memory and the parameters set in the rest of this menu. The posterior marginal probabilities of all nodes in the network are printed when the algorithm terminates, along with the running time and an error estimate. The error estimate is based on the correct answers stored in memory. If no answers are available, the error will be reported as unknown. Answers may be loaded with a graph, may be generated using the 'Correct Answer' brute force algorithm (if the graph is small), or may be approximated by running a confidence test.
- 'Set number of confidence runs' sets the number of times to run a simulation during a confidence test.
- 'Run confidence test' runs a selected simulation multiple times and computes an expected mean squared error.

2.2 The LISP Interface

We have been working on an interface to a dynamic graphical browser in LISP, but this work is not yet completed.

2.3 The Macintosh Interface

We are working on an interface for the Macintosh which allows more direct manipulation of the network, probabilities, and parameter settings, but this interface is not available yet.

Chapter 3

Programmer's Manual

This section describes the functions a C-programmer would need to access to implement a new interface to the CABeN functionality. We divide the operations on belief networks into four basic groups: operations on graphs, operations on probabilities, operations which initiate the belief-network algorithms, and operations which compute statistics about algorithm performance. The graph operations are pretty standard, so we will not spend much time here describing them. The other operations, while they also operate on graphs, are specific to belief networks, and we will spend more time describing each algorithm here.

3.1 Manipulating Networks

Most of the functions for manipulating graphs are in `net.c` (functions for saving and loading belief networks are in `file.c`). The file `caben.h` must be included to access these operations.

3.1.1 Data Structures

We will not go into the details of the data structures here, except to enumerate them and point out things which may not be obvious. Further documentation may be found in the comments in `caben.h`.

A `State` defines a state type, (e.g. 'Boolean'), including what values the state may hold (e.g. 'T' or 'F'). A `Node` is a vertex of a belief network, and contains in addition to a `name` field, the node's state, its parents and children, its conditional probabilities and current probabilities. A `NodeList` is a list of nodes, currently implemented as an array. Finally, a `Network` holds a belief network.

3.1.2 Graph Operations

- Create
  `Network *malloc_net()`
  A network is created by calling malloc_net(). The resulting pointer to a Network will have no nodes and one state (Boolean).

- Destroy
  `int InitNet(Network *net)`
  To free the space used by a network, call InitNet.

- Save (in `file.c`)
  `int SaveNet(Network *net)`
  `int SaveNetFile(Network *net,char *filename)`
  SaveNet is interactive: it asks the standard input for a filename. SaveNetFile can be used programmatically. Both write a file in belief network file format (see Appendix A).

- Load (in `file.c`)
  `int ReadNet(Network *net)`
  `int ReadNetFile(Network *net,char *filename)`
  ReadNet is interactive: it asks the standard input for a filename. ReadNetFile can be used programmatically. Both read a file in belief network file format (see Appendix A).

3.1.3 Node Operations

- Add
  `Node* add_node(Network *net, char *name)`
  Creates a new node named `name`, adds it to `net`, and returns it.

- Delete
  `void kill_node(Network *net, Node *node)`
  Removes `node` from `net`.

- Node names
  `node->name`
  `Node* find_node(Network *net, char *name)`
  A node's name is accessed through its name field, as in `node->name`. Given a node name (as a string), the corresponding node can be found with `find_node`.

- List
  There is no function to directly return a list of nodes. Instead, a macro is provided to iterate over the available nodes. The following code fragment would print the names of all of the nodes in the network:

```
LOOPNODES(net)
    printf("%s\n",node->name);
ENDNODES
```

- Set & Get Evidence
  Two different types of evidence settings are posssible. The simplest method is to set the state of the node directly, by changing `node->PRIOR`. The statement:
  `node->PRIOR = s;`
  sets the node to state s-1 (i.e. the state with name `node->state->names[s-1]`), for $s > 0$. If $s \leq 0$ no direct evidence is set. `node->PRIOR` may also be accessed as a variable.

  Another method is by setting an odds likelihood ratio for the nodes in a given state:
  `net->proto->setodds(net, node, double *odds);`
  where odds[s] consists of the ratio for state s. The odds array does not have to be normalized to sum to 1. For example if T is state 0 and F is state 1, odds[0] = 2, odds [1] = 1 would set a 2:1 ratio for the node being T. The odds likelihood ratio (unnormalized) is stored in `node->odds`.

3.1.4 Edge Operations

- Add
  `void connect(Node *mom, Node *kid)`
  Add a link from `mom` to `kid`.

- Delete
  `disconnect(Node *mom, Node *kid)`
  Removes an edge between `mom` and `kid`.

- Test for link
  `int is_link(Node *mom, Node *kid)`
  Returns 0 if there is no link between `mom` and `kid`, non-zero otherwise.

- List
  As with nodes, listing edges is done by executing a macro. Two macros are available, one for listing the child nodes (outgoing edges) and one for listing the parent nodes (incoming edges). The following code fragments print all the child nodes and parent nodes, respectively:

```
LOOPKIDS(node)
    printf("%s\n",kid->name);
ENDKIDS

LOOPMOMS(node)
    printf("%s\n",mom->name);
ENDMOMS
```

3.2 Manipulating Probabilities

Probabilities in CABeN are manipulated through a "probability prototype" (even though at the current time only one is implemented). All of these functions are accessed through `net->proto`, which is a structure containing pointers to the appropriate functions. To access a specific conditional probability value, $p(c \mid p_1, ...p_n)$, an index containing numeric values for the states of c and $p_1...p_n$ is passed to `net->proto->proto(node,ind)`, with ind[0] being the state of the node, ind[1] being the state of the first parent, and so on.

The easiest way to understand this may be to look at the code in the file `modimenu.c`. The function `ModiProb(net)` there calls `net->proto->modify()` which is a pointer to `full_modify(node)` in `proto.c`. `full_modify(node)` in turn calls `ask_probs(node)` which interactively prompts a user for conditional probability values.

An array called `node->calc_probs` is used to store posterior marginal probabilities calculated by the algorithms, where the i^{th} element of the array is the probability that the node is in the i^{th} state.

Default conditional probabilities are given to nodes without parents by assigning equal probability to each possible state of the node. Default probabilities are given when new parents are added to a node using the probabilities the node had before the parent was added. For example, if boolean node B has no parents, its default probability will be $p(B = TRUE) = 0.5$. If A is added as a parent of B, $p(B \mid A)$ will be:

$$p(B = TRUE \mid A = TRUE) = p(B = TRUE \mid A = FALSE) = p(B = TRUE) = 0.5$$

3.3 Belief Network Algorithms

The CABeN package implements many different belief network algorithms. In this section, we describe how to call the routines which implement the algorithms.

There are some parameters common to all of the algorithms, which we will describe first, and then refer to when discussing particular algorithms.

Markov The scoring part of most of the algorithms can be done either by computing a score value for each node individually, or by using the Markov blanket of the nodes for scoring. Using Markov blanket scoring takes more time on each iteration, but tends to cause the algorithms to converge in less iterations. This boolean parameter selects whether or not Markov scoring is used.

N-iterations The normal way to run these algorithms is to specify a number of iterations in advance. This integer parameter is used for this purpose. The number of iterations needed for convergence varies with each algorithm.

The following sample program performs logic sampling on a network read from a file in belief network file format:

```
#include "be_all.h"

extern Selecting *logic_sampling;
extern Proto test;

main(argc,argv)
      int argc;
      char *argv[];
{
   Network *net;
   int numsims;
   double error = 0.0;

   if (argc < 4){
      printf("Usage:  sample <net> <numsims>\n");
      exit(1);
   }

   sscanf(argv[2],"
   net = malloc_net();
   net->proto = &test;

   ReadNetFile (net,argv[1]);
   CopyPriors(net);

   if (simulate(net,(Selecting *) &logic_sampling,numsims,FALSE)) {
      printf("\n\n Error in simulation routine!");
      return ERR;
   };

/* The nodes of net now contain simulated answers in node->calc_probs */
}
```

3.4 Statistics on Algorithm Performance

The relevant statistics regarding algorithm performance are run time and error. The easiest way to calculate these parameters is by calling `SimRun()`. SimRun takes a StochParms structure which contains an algorithm and set of relevant parameters, and a number of times to repeat the experiment, and returns the time for one simulation, the error (as mean squared error), and a performance number which is the product of error and time.

```
SimRun(StochParms *sp, int numdo, double *rtime, double *mse, double *perform);
```

A good example of the use of this function is in the example program `compare.c`.

Chapter 4

Conclusions

The CABeN toolkit is a vehicle for experimenting with belief networks. No theory yet exists to predict whether or not a specific belief network is computationally tractable, or to suggest which algorithm might work best. Our simple experiments provide further guidelines on the applicability of stochastic simulation algorithms to a number of Bayesian network topologies. Our area of primary research interest is the applicability of these techniques to information retrieval problems where node values reflect the degree of belief in the value of a specific index term and conditional probabilities reflect the probability that a child index term will be relevant given that documents indexed under the parent node term are deemed relevant [11,12]. These networks resemble larger versions of the *ExpertSystem1* network. Our experiments suggest that from the perspective of response time, the Self Importance algorithm may perform best. Although our experiments provide support for reliable and rapid computation using stochastic simulation algorithms, we believe that other factors (e.g. conditional probability values, amount of evidence expected, degree of acceptable estimation error) must be considered with graph topology to determine the optimal algorithm for each application.

Acknowledgements

This work was supported in part by grants from Apple Computer, Inc. and the Center for Intelligent Computing at Washington University. We would like to thank Tim Oren of Apple Computer, Inc. and Roberto Lozano of Technology Resources Incorporated, a subsidiary of Southwestern Bell for their support, and Ron Loui and Michael Kahn for help with earlier drafts of this paper. Dr. Frisse is a Teaching and Research Scholar of the American College of Physicians.

The authors appreciate any comments, criticisms or enhancements. Please let us know what you think.*

*The authors can be reached at the Medical Informatics Laboratory, Washington University School of Medicine, 660 South Euclid Ave, Box 8121, St. Louis, MO 63110, or by electronic mail at caben@informatics.wustl.edu

Appendix A

Belief Network File Format

A file containing a belief network has three parts: A list of state types; A list of nodes and their types; and a list of edges and probabilities. The list of state types begins with a line of the form

```
STATES <# of states>
```

where ⟨# of states⟩ is the number which will be subsequently defined, and then that many lines of the form

```
<State type name> <# states> <State 1> ... <State n>
```

For example, the following would define a single state type, boolean:

```
STATES 1
Boolean 2 T F
```

The list of nodes and types is simple also. It begins with

```
NODES <# of nodes>
```

and then has ⟨# of nodes⟩ lines of the form

```
<State type name> <Node name>
```

For example, we could define 3 boolean nodes as

```
NODES 3
Boolean A
Boolean B
Boolean William Chen
```

Note that the third node has a name which contains a space. State names may not contain white space, but node names may – The node name contains all characters after the type and some white space, up to but not including the carriage return. These nodes are numbered implicitly by their position in the list, from 0 through n-1.

The final part of the file, the edges and probabilities, begins immediately after the NODES. Each node number should appear once, followed on the same line by the number of parents it has, the numbers of those parents, and the number of probabilities which will be specified:

```
<node num> <# parents> <p1> ... <pn> <# probs>
```

The probabilities are specified as they were in the state types, implicitly in the order of the parents:

```
<node state> <p1 state> ... <pn state> <value of p(node|p1...pn)>
```

If node "William Chen" (number 2) has parents A and B, with probabilities:

```
p(A=T) = 0.1
p(B=T) = 0.2
p("William Chen"=T | A=T B=T) = 0.3
p("William Chen"=T | A=T B=F) = 0.4
p("William Chen"=T | A=F B=T) = 0.5
p("William Chen"=T | A=F B=F) = 0.6
```

then the probability section would look like

```
0 0 1
T 0.1
F 0.9
1 0 1
T 0.2
F 0.8
2 2 0 1 4
T T T 0.3
F T T 0.7
T T F 0.4
F T F 0.6
T F T 0.5
F F T 0.5
T F F 0.6
F F F 0.4
```

A belief net file can contain other, optional fields as well. Lines beginning with "Answers" store the answer values for nodes for use in error tests. A line beginning with "Answer_source" indicates the source of the answers. For example,

```
Answer_Source 0 Direct Calculation
Answer 0 0.4 0.6
Answer 1 0.7 0.3
```

indicates that the correct answer for node 0 (True) is 0.4 and for node 1 (True) is 0.7 (assuming nodes 0 and 1 are boolean), and that the source for those answers is direct calculation (the brute force algorithm).

Evidence can also be specified in the belief net file using the "Set" command, as in

```
Set 0 T
```

Bibliography

[1] Cooper GF. Probabilistic inference using belief networks is NP-hard. Technical Report KSL-87-27. Medical Computer Science Group, Stanford University, Stanford, California. 1987.

[2] Pearl J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann, 1988.

[3] S.L.Lauritzen, Spiegelhalter D. Local computations with probabilities on graphical structures and their application to expert systems. J. Royal Statistical Society 1988;*9*:157–224.

[4] Neapolitan RE. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. New York, NY: Wiley Interscience, 1990.

[5] Shachter R, Peot M. Simulation approaches to general probabilistic inference on belief networks. In: Henrion M, Shachter R, Kanal L, Lemmer J, eds. *Uncertainty in Artificial Intelligence 5*. Amsterdam: Elsevier Science Publishers B.V. (North-Holland), 1990:221–31.

[6] Chavez RM, Cooper GF. A fully polynomial randomized approximation scheme for the bayesian inference problem (working paper). Technical report. Stanford University, Stanford California. Fall, 1988.

[7] Chavez RM, Cooper GF. A randomized approximation algorithm for probabilistic inference on bayesian belief networks. Networks 1990;*20*:661–85.

[8] Cooper GF. Current research directions in the development of expert systems based on belief networks. Applied Stochastic Models and Data Analysis 1989;*5*:39–52.

[9] Henrion M. An introduction to algorithms for inference in belief nets. In: Henrion M, Shachter R, Kanal L, Lemmer J, eds. *Uncertainty in Artificial Intelligence 5*. Amsterdam: Elsevier Science Publishers B.V. (North-Holland), 1990:129–38.

[10] Cousins SB, Chen W, Frisse ME. CABeN: A collection of algorithms for belief networks. Technical Report wucs-91-25. Washington University, St. Louis, MO. 1991.

[11] Frisse ME, Cousins SB. Information retrieval from hypertext: Update on the Dynamic Medical Handbook Project. In: Meyrowitz N, ed. *Proceedings of Hypertext 89*. New York: ACM Press, 1989:199–212.

[12] Cousins SB, Frisse ME. Query networks for medical information retrieval—assigning probabilistic relationships. In: Miller RA, ed. *Proceedings, Symposium on Computer Applications in Medical Care*. New York, NY: IEEE Computer Society, 1990:800–4.