Bayesian Belief Network Simulation

Changyun Wang
Department of Computer Science
Florida State University

February 24, 2003

Abstract

A Bayesain belief network is a graphical representation of the underlying probabilistic relationships of a complex system. These networks are used for reasoning with uncertainty, such as in decision support systems. This requires probabilistic inference with Bayesian belief networks. Simulation schemes for probabilistic inference with Bayesian belief networks offer many advantages over exact inference algorithms. The use of randomly generated Bayesian belief networks is a good way to test the robustness and convergence of simulation schemes. In this report, we first present methods for random generations of Bayesian belief networks, then we implement stochastic simulation algorithms for probabilistic inference with such networks. Since random number generators play a critical role in random generations of belief networks, we explore the theoretical and practical backgrounds of random number generators and select suitable generators for our project.

Contents

Chapter 1

Introduction

Bayesian belief networks are also known as "belief networks", "causal probabilistic networks", "causal nets", and "graphical probability networks". These networks have attracted much attention recently as a possible solution to complex problems related to decision support under uncertainty. Although the underlying theory has been around for a long time, the possibility of building and executing realistic models has only been made possible because of recent improvements on algorithms and the availability of fast electronic computers.

Several algorithms exist for the calculation of the exact a-priori and posteriori probabilities of a network. However, exact algorithms are not generally applicable, because they are computationally expensive. Exact algorithms have difficulties with certain types of network structures, which is not surprising, since the task has been proven to be NP-hard [5].

A widely used method for handling the computational burden in decision theory and Bayesian statistics is the use of approximation methods. Instead of the exact calculation of probabilities, representative samples of the variables in the Bayesian networks can be generated via simulation schemes.

Monte Carlo methods for belief networks can be classified into two different groups: those based on independent sampling and those based on Markov chains. The first simulation algorithm was based on independent sampling. It is called probabilistic logic sampling and developed by Henrion[8]. It provides good results for a-priori probabilistic inference, that is when no evidence is given to the network. An improved algorithm, called likelihood weighting [7], performs better for posteriori probabilistic inference with evidence. The second classification of algorithms are simulation algorithms based on Markov

chains. The algorithms simulate with samples that are not independent, but they verify the Markov property. The first and best known approximate propagation algorithm using this technique is Pearl's algorithm [9].

Systemic sampling techniques are used in simulation schemes, such as stratified simulation and Latin hypercube sampling. A well-known method for selecting representative samples in statistics is the use of stratification. The stratified simulation method for Bayesian belief networks was initially suggested by Bouckaert, and the Latin Hypercube sampling method was suggested by Chen and Druzdeel.

The use of randomly generated Bayesian belief networks might be a good way to test the robustness and convergence of simulation schemes. In this report, we first present methods for random generations of Bayesian belief networks, then we implement stochastic simulation algorithms for probabilistic inference with such networks. Since random number generators play a critical role in random generations of belief networks, we explore the theoretical and practical backgrounds of random number generators and select suitable generators for our project.

The report is organized as follows: theory background of belief network is introduced in Chapter 2. Examples and definitions of Bayesian belief network are presented. Since random numbers play a critical role in simulation, we will also discuss how to generate pseudo-random numbers in Chapter 3. The random generation of a Bayesian belief network is studied in Chapter 4. Stochastic simulations and their implementations are carried out in Chapters 5 and 6, respectively. Finally, Chapter 7 concludes with a summary and a discussion of future work.

Chapter 2

Bayesian Networks

A basic familiarity with probability theory is assumed for the purpose of this report. However, for completeness we will give the following basic definitions.

2.1 Probability as a Measure of Uncertainty

When using the notion of probability, one may talk in terms of: the probability that a cancer patient will respond to a certain form of chemotherapy; the probability that a projectile might hit a region of space; the probability of observing a string of three identical outcomes in six dice throws. We shall use the general term sample point to refer to the "things" we are talking about; a abstraction of a cancer patient, a geometric point, a chance outcome.

A Sample Space, or universe, is the set of all possible sample points in a situation of interest. It is usual to use Ω to designate a specific space. The sample points in a sample space must be mutually exclusive and collectively exhaustive.

A probability measure, $p(.)$, is a function on subsets of a space Ω These subsets are called events. we can refer the values of $p(A)$, $p(A \cup B)$, $p(\Omega)$ as the probabilities of the respective events (for A, $B \subseteq \Omega$).

The function $p(.)$ is a measure with the following properties.

Definition 2.1.1 *A probability measure on a sample space Ω is a function mapping subsets of Ω to the interval $[0, 1]$ such that:*

1. *For each $A \subseteq \Omega$, $p(A) \geq 0$.*
2. *$p(\Omega) = 1$.*

3. *For any countable infinite collection disjoint subsets of* Ω, A_k, $k = 1, 2, \ldots,$

$$p(\bigcup_{k=1}^{\infty} A_k) = \sum_{k=1}^{\infty} p(A_k) \tag{2.1}$$

In general, we will need to check that the sets (events) themselves satisfy certain properties to ensure that they are measureable.

2.1.1 The Reliability of Measurements

Of course, any act of measurement has an element of imprecision associated with it. So, we would expect the probabilities of events obtained by measurement also to be imprecise; strictly, any physical probability should be represented by a distribution of possible values. In general, the more information we have, the tighter will be the distribution. Sometimes, however, we will have no direct physical measurements by which to estimate a probability. An example of such a case might be when one is asked to toss a coin one has never seen before and judge the probability that it will land heads up. if one believes the coin to be fair, an estimate of 1/2 for this physical probabililtiy would seem reasonable. Sometimes, a probability elicited in this way is taken as a measure of an expert's **belief** that a certain situation will arise. This can lead to extensive discussions as to whether experts and others do, or should, use the laws of probability to update their beliefs as new evidence becomes available. To avoid such discussions, one might just take the elicitation of such probabilities as an act of expert judgment. Whichever vies one takes, probability theory offers a gold standard by which the probability estimates may be revised in the light of experience.

2.1.2 The Bayesian Theorem

We have so far concentrated largely on the static aspects of probability theory. But probability is a dynamic theory. It provides a mechanism for coherently revising the probabilities of events as evidence becomes available. Conditional probability and the Bayesian theorem play a central role in this. Again for completeness, we include a brief discussion of these here.

We will write $p(A \mid B)$ to represent the probability of event A (an hypothesis) conditional on the occurrence of some event B (evidence). If we are counting sample points, we are interested in the fraction of events B for

which A is also true; We are switching our attention from the universe Ω to the universe B. From this it should be clear that (with the comma denoting the conjunction of events), we have

$$p(A \mid B) = \frac{p(A, B)}{p(B)} \tag{2.2}$$

This is often written in the form

$$p(A, B) = p(A \mid B)p(B) \tag{2.3}$$

and referred to as the "product rule", this is in fact the simple form of Bayes' Theorem. It is important to realize that this form of the rule is not, as often stated, a definition. Rather, it is a theorem derivable from simpler assumptions.

The Bayesian theorem can be used to tell us how to obtain a posterior probability of a hypothesis A after observation of some evidence B, given the *prior* probability of A and the likelihood of observing B were A to be the case:

$$p(A \mid B) = \frac{p(B \mid A)p(A)}{p(B)} \tag{2.4}$$

This simple formula has immense practical importance on a domain such as diagnosis. It is often easier to elicit the probability, for example, of observing a symptom given a disease than that of a disease given a symptom. Yet, operationally it is usually the latter which is required. In its general form, Bayesian Theorem is stated as follows.

Proposition 2.1.2 *Suppose* $\cup A_n = \Omega$ *is a partition of a sample space into disjoint sets. Then*

$$p(A_n \mid B) = \frac{p(B \mid A_n)p(A_n)}{\sum p(B \mid A_n)p(A_n)} \tag{2.5}$$

It is important to appreciate that the Bayesian theorem is as applicable at the "meta-level" as it is at the domain level. It can be used to handle the case where the hypothesis is a proposition in the knowledge domain and the evidence is observation of some condition. However, it can also handle the case where a hypothesis is that a parameter in a knowledge model has a certain value or that the model has a certain structure, and the evidence is some incoming case data.

2.1.3 Probability Distributions on Sets of Variables

The points in a sample space may be very concrete. If we are considering an epidemiological study, these points may consist of people. In the case of quality control of an assembly line, they may be specific electronic components. As such, points in a sample space may possess certain qualities about which we are interested and which may be observed or measured in some way. So, for example, an electronic logic gate may be functional or non-functional, or it may have a certain weight.

We will refer to such a distinction, about which we may be uncertain, as a *variable*. A variable has a set of *states* corresponding to a mutually exclusive and exhaustive set of events. It may be *discrete*, in which case it has a finite or countable number of states, or it may be *continuous*. For example, we may use a discrete (e.g. binary) variable to represent the possible functioning or otherwise of a logic gate selected from a production line, and a continuous variable to represent its weight. Strictly, a variable(on Ω) is a function, X say, from sample points to a domain representing the qualities or distinctions of interest. An element of randomness in $X(\varpi)$ is induced by selecting "at random" of the sample point ϖ; a specific logic gate, or a specific throw of dice. Once the sample point has been chosen, the outcome $X(\varpi)$ is fixed and can be measured, or otherwise determined. However, it is often the case that the elements of the underlying sample space can be implicitly understood, in which case an unadorned capital letter is used to represent the variable. Following this custom, in the remainder of this report, we will use uppercase letters to represent single variables (e.g. X, Y and Z). Lower case letters will be used for states, with, for example, $X = x$ denoting that variable X is in state x. Of course, some qualities of a sample point may be easier to determine than others. For example, although we can readily determine whether a logic gate is functional or not, determining the underlying cause of a non-functional gate is not normally possible without some invasive inspection of the device. But here experience might help us. Suppose we had carefully analysed a hundred non-functional devices and found sixty-five with a faulty bond between the chip. Then given a new observation of a non-functional gate, we can use these statistics to predict the chances of that gate having specific states for these not-so-easily observable qualities. Real world problems are typically more complex than this. To move a little closer to real example, Figure 2.1 lists a set of variables which will have specific states for some person of interest. The

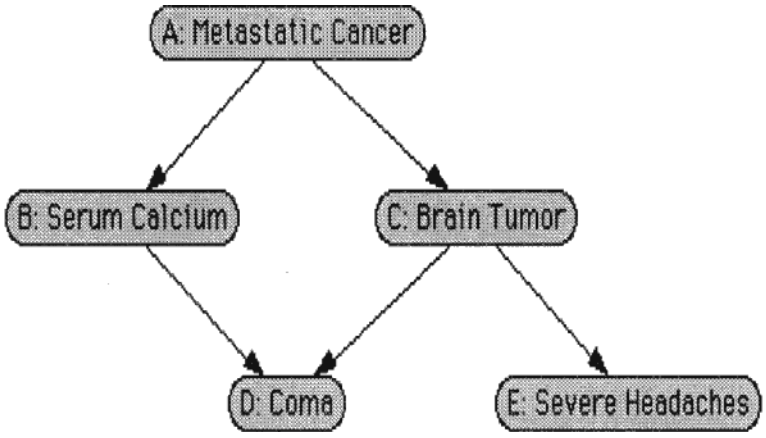

Figure 2.1: A Bayesian Belief Network Describing Influences Among Five Variables.

states of some of these variables, such as a (A:metastatic cancer).

Our goal is to predict the most likely states of those states of those variables that are harder to observe directly, such as D:coma or E:severe headaches, using the observed states. We can do this if we are able to elicit a probability distribution $p(A, B, C, D, E)$ over all the variables of interest. Yet even if each variable has only two states we will need to elicit $2^5 - 1$ distinct values in order to define the probability distribution completely This would require a massive data collection exercise if we were to hope to use physical probabilities, or alternatively make unreasonable demands on the domain experts if we were to think in terms of eliciting probabilities from them. Yet even this is a "toy" problem in relation to some of the real applications that are being built.

The problem is that in defining a *joint probability distribution*, such as $p(A, B, C, D, E)$, we need to assign probabilities to all possible events. We can, however, make the knowledge election problem much more tractable if we exploit the structure that is very often implicit in the domain knowledge. The next section will expand on this.

2.2 Bayesian Belief Networks

Bayesian belief networks have a qualitative part and a quantitative part, represented by a graph of discrete probabilisitic variables and tables with conditional probabilities for these variables, respectively.

2.2.1 Graph Theory

Many problem domains can be structured through using a graphical representation. Essentially, one identifies the concepts or items of information which are relevant to the problem at hand (nodes in a graph), and then makes explicit the *influences* between concepts. This section introduces some of the terminology associated with the use of graphs.

At its most abstract, a graph G is simply a collection of vertices V and edges E between vertices

$$G = (V, E)$$

We can associate a graph G with a set of variables $U = \{X_1, X_2, \ldots, X_n\}$ by establishing a one-to-one relationship between the node in the graph and the variables in U. One might, for example, label the nodes from 1 to n, with nodes being associated with the appropriately subscripted variable in U. An edge $e(i, j)$ might be *directed* from node i to one node j. In this case, the edge $e(j, i)$ cannot simultaneously belong to E and we say that node i is a *parent* of its *child* node j. If both $e(i, j)$ and $e(j, i)$ belong to E, we say the edge is *undirected*.

Definition 2.2.1 *An undirected graph G is an ordered pair*

$$G = (V(G), E(G))$$

where $V(G) = \{V_1, \ldots, V_n\}$, $n \geq 1$,is a finite set of vertices and $E(G)$ is a family of unordered pairs (V_i, V_j), $(V_i, V_j) \in (V(G))$, called edges. Two vertices V_i and V_j are called adjacent or neighboring vertices in G if $(V_i, V_j) \in (E(G))$. The set of all neighbors of vertes V_i in G is denoted by $v_G(V_i)$.

A graph which contains only directed edges is known as a *directed graph*. Those graphs which contain no directed cycles have been particularly studied in the context of probability expert systems. These are referred to as *directed acyclic graphs* (DAGs).

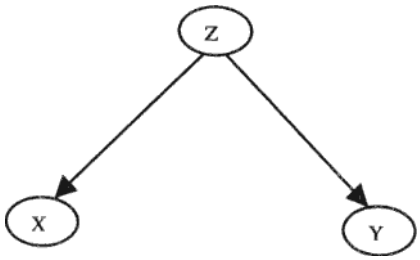

Figure 2.2: X is Conditionally Independent of Y Given Z.

As mentioned in the opening to this section, the important point about a graphical representation of a set of variables is that the edges can be used to indicate *relevance* or *influences* btween variables. Absence of an edge between two variables, on the other hand, provides some form of independence statement; nothing about the state of one variable can be inferred by the state of the other.

2.2.2 Independence

The notions of *independence* and *conditional independence* are a fundamental component of probability theory. It is this combination of qualitative information with the quantitative information of the numerical parameters that makes probability theory so expressive.

Let X and Y be variables. Then $X \parallel Y$ denotes independence of X and Y. The corresponding probabilistic expression of this is

$$p(x, y) = p(x)p(y) \tag{2.6}$$

Now we introduce another variable Z. Then $X \parallel Y \mid Z$ denotes that X is conditionally independent of Y given Z. One expression of this in terms of probability distribution is

$$p(x, y \mid z) = p(x \mid z)p(y \mid z) \tag{2.7}$$

We can draw a directed acycle graph that directly encodes this assertion of conditional independence. This is shown in Figure 2.2.

A significant feature of the structure in Figure 2 is that we can now decompose the joint probability distribution for the variables X, Y and Z

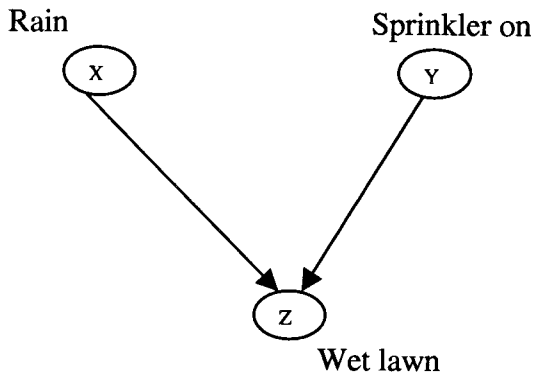

Figure 2.3: X and Y are Conditionally Dependent Given Z.

into the terms involving at most two variables

$$p(x, y, z) = p(x, y \mid z)p(z) = p(x \mid z)p(y \mid z)p(z) \tag{2.8}$$

As a concrete example, think of the variable Z as representing a disease such as measles. The variables X and Y represent distinct symptoms: perhaps "red spots" and "Kpplik's spots" respectively. Then if we observe that disease (measles) as present, the probability of either symptom being present is determined. Actual confirmation of one symptom being present will not alter the probability of the occurrence of the other.

A different situation is illustrated in Figure 2.3. Here X and Y are marginally independent, but conditional dependent given Z. This is best illustrated with another simple example. Both X="rain" and Y="sprinkler on" may cause the lawn to become wet. Before any observation of the lawn is made, the probability of rain and the probability of the sprinkler being on are independent. However, once the lawn is observed to be wet, confirmation of it raining may influence the probabilty of the sprinkler being on.

The probability distribution is

$$p(x, y, z) = p(z \mid x, y)p(x)p(y) \tag{2.9}$$

The final example completes the cases of interest Figure 2.4 In this case, the probability is

$$p(x, y, z) = p(y \mid z)p(z \mid x)p(x) \tag{2.10}$$

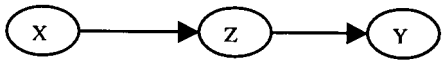

Figure 2.4: X and Y are Conditionally Independent Given Z.

2.3 Propagation in Bayesian Belief Networks

Th goal of building Bayesian networks is, given current observation, to answer a certain query about the probability distribution over the value of query variables. A fully specified Bayesian network contains the information needed to answer all probabilistic queries about these variables. The mechanism of drawing conclusions in Bayesian networks is called propagation of evidence, or simply propagation. There are two types of algorithms for propagation of algorithms for propagation: exact, and approximate. By an exact propagation algorithm we mean a method computes probability distribution of the nodes exactly. Meanwhile, an approximate propagation algorithm computes the probability approximately.

In this project, we focus on approximate methods for belief network inference and propagation with simulation algorithms. The simulation algorithms are tested with randomly generated networks. Since we can randomly produce probability distributions of the nodes of the networks, it is a good tool to test the performance of an approximate propagation algorithm.

Chapter 3

Pseudo-Random Numbers

Random numbers play central role in simulation algorithms for probabilistic inference with Bayesian belief networks. In practice, the pseudo-random number program is deterministic and produces a random sequence, called pseudo-random numbers. In Numerical Recipes, the author points out to "*be very very suspicious of system-supplied* `rand()`*, ANSI C libraryies is quite flawed; quite a number of implementations are in the category 'totally botched*"".

We briefly overview some popular basic random number generators listed below

- Linear Congruential Generators
  - Power of 2 Modulus
  - Prime Modulus
- Shift-Register Generators
- Lagged-Fibonacci Generators
- Inversive Congruential Generators
- Combination Generator

Based on properties of our simulation and the availability of codes, we decided to choose one type of shifted-register generator, called Mersenne Twister pseudo-random number generator.

We will give an overview of current random number generators and point out the theoretical safety-measures to avoid incorrect simulation results.

3.1 What Constitutes a Good Random Number Generator?

Basically, there is no quantitative measure of merit available fro RNGs that is able to guarantee improved results in our simulation. However, there do exist some measures to keep the risk of incorrect simulation results as small as possible. We list some desired properties of random number generators below [6]

- Randomness
- Reproducibility
- Speed
- Large cycle length

Well-designed algorithms for random number generation allow us to find conditions for their parameters that will guarantee a certain period of length of the output sequence. Futher, it will be possible to detect in advance, by theoretical analysis, certain weaknesses of the algorithm.

3.2 The Generalized Feedback Shift Register (GFSR)

Define $\oplus$ to be the *exclusive-or* operator which is equivalent to addition modulo 2.The idea behind the generalized feedback shift register pseudorandom algorithm (GFSR) [10, 12] is that the basic shift register sequence $\{a_i\}$ based on primitive trinomial $x^p + x^q + 1$ is set into j columns with a judiciously selected *delay* between columns.

An example will make the basic GFSR algorithm clear. Choose primitive trinimial $x^5 + x^2 + 1$. The basic sequence of $\{a_i\}$ is copied as follows [10]:

W_0	11010	W_{10}	01001	W_{20}	00111
W_1	10001	W_{11}	10000	W_{21}	01111
W_2	11011	W_{12}	10110	W_{22}	10010
W_3	11100	W_{13}	10100	W_{23}	01100
W_4	10011	W_{14}	01110	W_{24}	00101
W_5	00001	W_{15}	11111	W_{25}	10101
W_6	01101	W_{16}	00100	W_{26}	00011
W_7	01000	W_{17}	11000	W_{27}	10111
W_8	11101	W_{18}	01011	W_{28}	11001
W_9	11110	W_{19}	01010	W_{29}	00110
				W_{30}	00010

Since each column obeys the recurrence a_k=$a_{k-p+q} \oplus a_{k-p}$ each word must also obey $W_k = W_{k-p+q} \oplus W_{k-p}$. Observe that W_i occurred once and only once in the full period $2^5 - 1 = 31$ numbers. The GFSR algprithms is

1. If $k \neq 0$, go to 2 (k is initially zero)
2. Initialize $W_0, \ldots, Wp-1$ using a delayed basic sequence, $\{a_i\}$ to obtain each column of $W_0, \ldots, Wp-1$
3. $k \leftarrow k+1$
4. If $k > p$, then set $k \leftarrow 1$
5. $j \leftarrow k+p$
6. If $j > p$, then set $j \leftarrow j-p$
7. Store $W_k \leftarrow W_k \oplus W_i$

3.2.1 Advantages of GFSR

The algorithm has the following merits:

- The generation is very fast. Generation of one pseudorandom number requires only three memory references and one exclusive-or operation.
- The sequence has an arbitrarily long period independent of the word size of the machine.
- The implementation is independent of the word size of the machine.

3.2.2 Disadvantages of GFSR

The GFSR algorithms has the following drawbacks

- The selection of initial seeds is very critical and influential in the randomness, and good initialization is rather involved and time consuming.
- Each bit of a GFSR sequence can be regarded as an m-sequence based on the trinomial $t^n + t^m + 1$,which is known to have poor randomness
- The period of a GFSR sequence $2^n - 1$ is far smaller than the theoretical upper bound; i.e the number of possible states 2^{nw}.
- The algorithm requires n words of working area, which is memory consuming if a large number of generators is implemented simultaneously.

3.3 Mersenne Twister GFSR

A new generator, named the *twisted GFSR generator* (TGFSR) [12] does not exhibit the above four drawbacks. The TGFSR generator is the same as the GFSR generator, except that it is based on the linear recurrence

$$x_{l+n} := x_{l+m} \otimes x_l A \qquad (l = 0, 1, \ldots) \tag{3.1}$$

where A is a $w \times w$ matrix with $0 - 1$ components and x_l is regarded as row vector over $GF(2)$. With suitable choice of n, m, and A, the TGFSR generator attains the maximal period $2^{nw} - 1$.

TGFSR is a pseudorandom number generating algorithm with following properties

- Long period
- Good k-distribution property
- Efficient use of memory
- High speed

An implementation of TGFSR in C `mt19937.c` [3] has the following features

- The period is $2^{19937} - 1$

- 632-dimensionally equidistributed to 32-bit accuracy
- Consumes 624 words of 32 bits
- About four times faster than `rand()` in C

This code is available from `http://www.math.keio.ac.jp/matumoto/emt.html`.

In implementations, a little bit of modified x_k is given as follows [3]. MT generates the vectors of word size by the recurrence:

x_{k+n}=x_{k+m}+$(x_k^u|x_{k+1}^l)$A, (k=0,1,...). Here,n¿m are fixed positive integers,(x_k)k=0,1,... is a sequence of w-dimensional row vectors over the two element filed $\mathbf{F}_2$=0,1, $(x_k^u|x_{k+1}^l)$ is the w-dimensional vector obtained by concatenating the left w-r bits of x_k and the r bits of x_{k+1}(u for upper,l for lower). By multiplying a matrix A (called *twister*) from right,we get $(x_k^u|x_{k+1}^l)$A. Every arithmetic operation is modeulo 2, i.e,this is a linear recurrence of vectors in the two element field $\mathbf{F}_2$. Since A should be chosen to be quickly computable,we proposed the form called *companion matrix*:

$$\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ a_0 & a_2 & \ldots & a_{w-1} \end{pmatrix}$$

3.4 Summary

In our project, the random number generator plays an important part. When we either generate random Bayesian belief networks for testing and implement approximate propagation algorithms to test, a robust and reliable random number generator should be used. The random number generator built in any C or C++ Library is not reliable and we have to find another random number generator to replace it with. The Marsenne Twister [3] is a well-known and widely tested PRNG. The choice of PRNG is important, becauce the results in Chapter 5 show us it affects the performance of the approximate propagation algorithms that we tested.

Chapter 4

Random Generation of Bayesian Belief Networks

Since exact inference in Bayesian networks has been proved NP-hard, simulation schemes are becoming more popular for probabilistic inference in Bayesian belief networks. No one algorithm can be used for all networks. There are many simulation schemes are proposed. In order to test the properties of these simulation algorithms, belief networks generated randomly are useful to test these algorithms.

4.1 Random Generation

We followed the guidelines in Pearl's book [9] to create an algorithm that generates Bayesian belief networks randomly. That is, with a random graph structure and random conditional probability tables. Our codes are included in appendix I. The algorithm is controlled by two parameters, N and M. The parameter N is the number of variables and M is the number of at most how many parents which any one variable will have.

As for the (conditional) probabilities of the variables, we have two methods to produce the probability tables for the network. One method is to assign the probabilities by hand based on an existing network, for example. Another method is to generate them with random number generators. When we test one simulation method, we have test many possible probability distributions, especially ones with high variablility in small and large probabilities of nodes. One advantage of randomly generating probabilities for a network

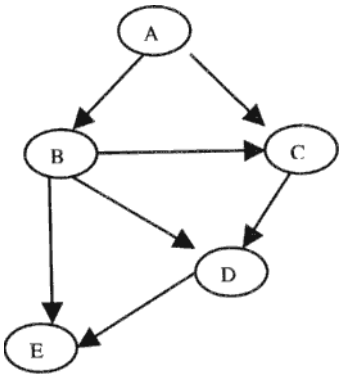

Figure 4.1: Random Generation of Bayesian Belief Network With $N = 5$ and $M = 2$.

is to help us to have uniform distribution but also more extreme probability distributions. We can generate distributions in a pointed interval, such as $[0.9, 1)$ or $(0, 0.1]$, see Table 4.2. In this way, we can test the convergence behavior of simulation algorithms to extreme probability distributions. We produced two probability distributions for the network in Figure 4.1.

The following figures depict Bayesian belief networks generated with our code. The Mersenne Twister random number generator is used to produce the random numbers. Capital letters or indexed capital letters, such as A, B, C, and A_i denote random variables. Lower case letters a, b, a denote particular instantiations of the variables A, B, C, respectively. Assume that every variable has binary values, for example $a = 1$ and $\tilde{a}$=0.

4.2 Summary

Theoretically speaking, our codes can produce a Bayesian belief network with any number of nodes and any probability distributions of nodes. Testing simulation algorithms on a network with extreme distributions is a good test to verify the convergence rate of the simulation.

In practice, we produced Bayesian networks with less than 20 nodes. We used two random number generators to create random networks: Mersenne twister RNG and `rand()` of the C library. There are no diffeneces between them. The main reason is that the number of nodes is relatively small.

Table 4.1: Probability Table for the Network in Figure 4.1 With $N = 5$.

Probability	Value	Probability	Value
P(a)	0.76	$P(\tilde{a})$	0.24
$P(b\|a)$	0.58	$P(\tilde{b}\|a)$	0.42
$P(b\|\tilde{a})$	0.56	$P(\tilde{b}\|\tilde{a})$	0.44
$P(c\|a,b)$	0.89	$P(\tilde{c}\|a,b)$	0.11
$P(c\|\tilde{a},b)$	0.47	$P(\tilde{c}\|\tilde{a},b)$	0.53
$P(c\|a,\tilde{b})$	0.02	$P(\tilde{c}\|a,\tilde{b})$	0.98
$P(c\|\tilde{a},\tilde{b})$	0.34	$P(\tilde{c}\|\tilde{a},\tilde{b})$	0.66
$P(d\|b,c)$	0.96	$P(\tilde{d}\|b,c)$	0.04
$P(d\|\tilde{b},c)$	0.85	$P(\tilde{d}\|\tilde{b},c)$	0.15
$P(d\|b,\tilde{c})$	0.18	$P(\tilde{d}\|b,\tilde{c})$	0.82
$P(d\|\tilde{b},\tilde{c})$	0.44	$P(\tilde{d}\|\tilde{b},\tilde{c})$	0.56
$P(e\|b,d)$	0.48	$P(\tilde{e}\|b,d)$	0.52
$P(e\|\tilde{b},d)$	0.84	$P(\tilde{e}\|\tilde{b},d)$	0.16
$P(e\|b,\tilde{d})$	0.80	$P(\tilde{e}\|b,\tilde{d})$	0.20
$P(e\|\tilde{b},\tilde{d})$	0.77	$P(\tilde{e}\|\tilde{b},\tilde{d})$	0.23

Table 4.2: Probability Table With Extreme Values for the Network in Figure 4.1 With $N = 5$.

Probability	Value	Probability	Value
P(a)	0.91	$P(\tilde{a})$	0.09
$P(b\|a)$	0.98	$P(\tilde{b}\|a)$	0.02
$P(b\|\tilde{a})$	0.09	$P(\tilde{b}\|\tilde{a})$	0.91
$P(c\|a,b)$	0.93	$P(\tilde{c}\|a,b)$	0.07
$P(c\|\tilde{a},b)$	0.07	$P(\tilde{c}\|\tilde{a},b)$	0.93
$P(c\|a,\tilde{b})$	0.91	$P(\tilde{c}\|a,\tilde{b})$	0.09
$P(c\|\tilde{a},\tilde{b})$	0.08	$P(\tilde{c}\|\tilde{a},\tilde{b})$	0.92
$P(d\|b,c)$	0.94	$P(\tilde{d}\|b,c)$	0.06
$P(d\|\tilde{b},c)$	0.03	$P(\tilde{d}\|\tilde{b},c)$	0.97
$P(d\|b,\tilde{c})$	0.93	$P(\tilde{d}\|b,\tilde{c})$	0.07
$P(d\|\tilde{b},\tilde{c})$	0.02	$P(\tilde{d}\|\tilde{b},\tilde{c})$	0.98
$P(e\|b,d)$	0.99	$P(\tilde{e}\|b,d)$	0.01
$P(e\|\tilde{b},d)$	0.03	$P(\tilde{e}\|\tilde{b},d)$	0.97
$P(e\|b,\tilde{d})$	0.96	$P(\tilde{e}\|b,\tilde{d})$	0.04
$P(e\|\tilde{b},\tilde{d})$	0.09	$P(\tilde{e}\|\tilde{b},\tilde{d})$	0.91

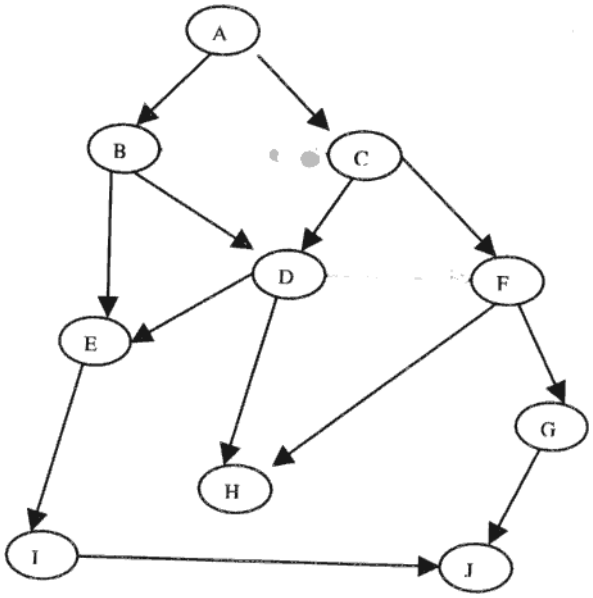

Figure 4.2: Randomly Generated Bayesian Belief Network for $N = 10$ and $M = 2$.

Table 4.3: Probability Table for the Network in Figure 4.2 With $N = 10$.

Probability	value	Probability	value
P(a)	0.09	$P(\tilde{a})$	0.91
$P(b\|a)$	0.46	$P(\tilde{b}\|a)$	0.54
$P(b\|\tilde{a})$	0.47	$P(\tilde{b}\|\tilde{a})$	0.53
$P(c\|a)$	0.74	$P(\tilde{c}\|a)$	0.26
$P(c\|\tilde{a})$	0.12	$P(\tilde{c}\|\tilde{a})$	0.88
$P(d\|b,c)$	0.83	$P(\tilde{d}\|b,c)$	0.17
$P(d\|\tilde{b},c)$	0.67	$P(\tilde{d}\|\tilde{b},c)$	0.33
$P(d\|b,\tilde{c})$	0.09	$P(\tilde{d}\|b,\tilde{c})$	0.91
$P(d\|\tilde{b},\tilde{c})$	0.11	$P(\tilde{d}\|\tilde{b},\tilde{c})$	0.89
$P(e\|b,d)$	0.12	$P(\tilde{e}\|b,d)$	0.88
$P(e\|\tilde{b},d)$	0.04	$P(\tilde{e}\|\tilde{b},d)$	0.96
$P(e\|b,\tilde{d})$	0.26	$P(\tilde{e}\|b,\tilde{d})$	0.74
$P(e\|\tilde{b},\tilde{d})$	0.26	$P(\tilde{e}\|\tilde{b},\tilde{d})$	0.74
$P(f\|c)$	0.92	$P(\tilde{f}\|c)$	0.04
$P(f\|\tilde{c})$	0.4	$P(\tilde{f}\|\tilde{c})$	0.60
$P(g\|f)$	0.3	$P(\tilde{g}\|f)$	0.70
$P(g\|\tilde{f})$	0.4	$P(\tilde{g}\|\tilde{f})$	0.60
$P(h\|d,f)$	0.56	$P(\tilde{h}\|d,f)$	0.44
$P(h\|\tilde{d},f)$	0.16	$P(\tilde{h}\|\tilde{d},f)$	0.84
$P(h\|d,\tilde{f})$	0.71	$P(\tilde{h}\|d,\tilde{f})$	0.29
$P(h\|\tilde{d},\tilde{f})$	0.27	$P(\tilde{h}\|\tilde{d},\tilde{f})$	0.73
$P(i\|e)$	0.13	$P(\tilde{i}\|e)$	0.87
$P(i\|\tilde{e})$	0.67	$P(\tilde{i}\|\tilde{e})$	0.33
$P(j\|g,i)$	0.57	$P(\tilde{j}\|g,i)$	0.43
$P(j\|\tilde{g},i)$	0.21	$P(\tilde{j}\|\tilde{g},i)$	0.79
$P(j\|g,\tilde{i})$	0.16	$P(\tilde{j}\|g,\tilde{i})$	0.84
$P(j\|\tilde{g},\tilde{i})$	0.87	$P(\tilde{j}\|\tilde{g},\tilde{i})$	0.13

Chapter 5

Approximate Algorithms and Their Implementations

There are two basic classes of approximate algorithms for Bayesian belief networks: independent sampling [2] and Markov chains algorithms [7]. The performance of both classes of algorithms depends on the properties of the underlying joint probability distribution represented by the model. Each algorithm has its advantages and disadvantages [13], that is, may work well on some but poorly on other networks. It is important to study the properties of real models and subsequently to be able to tailor or combine algorithms for each model utilizing its properties.

5.1 Random Sampling

In stochastic sampling algorithms (also called *Monte Carlo sampling*, *stochastic simulation*, or *random sampling*), the probability of an event of interest is estimated using the frequency with that occurs in a set of samples. Differences in the sampling algorithms are caused by the characteristics of the probability distribution from which they draw their samples. If the sampling distribution does not match the actual joint probability distribution, an algorithm may perform poorly.

We will use a simple two-node network presented in Figure (5.1)(the same as Figure (2.1) in Chapter 2) to illustrate the advantages and disadvantages of each algorithm. Both nodes are binary variables (denoted by upper case letter, such as A). The two outcomes will be represented by lower case letters

(for example a and $\bar{a}$).

We outline the proposed random sampling methods as follows

- **Logic sampling [8]**
  The simple and the first proposed sampling algorithm is the probabilistic sampling which works as follows: Each node is randomly instantiated to one of its possible states, according to the probability of this state given the instantiated states of its parents. This requires every instantiation to be performed in the topological order, that is, parents are sampled before their children. Nodes with observed states (evident nodes) are also sampled, but if the outcome of the sampling process is inconsistent with the observed state, the entire sample is discarded.

  Probablitic logic sampling produces probability distribution with very small absolute errors when no evidence has been observed. If there is evidence, and it is very unlikely, most samples generated will be inconsistent with it and will be discarded.

  Suppose that node B has been observed at an unlikely value b, which will have a very small probability. This means that most samples will be discarded. In prior probability of evidence is usually very small and, effectively, probabilistic logic sampling can perform poorly.

- **Likelihood weighting [7]**
  Likelihood weighting enhances the logic sampling in that it never generates samples for evidence nodes but rather weights each sample by the likelihood of evidence conditional on the sample. All samples are, therefore, consistent with the evidence and none are discarded.

  Also, likelihood sampling suffers from another problem. The likelihood sampling algorithm will set node A to $\bar{a}$ most of time, but will assign a small weight to every sample. It will rarely set A to a, but assign these samples a high weight. Effectively, the generated samples may not reflect the impact of evidence.

  These proportions may become more extreme in very large networks and with a tractable number of samples. It may happen that some states will never be sampled. It is popularly believed that the likelihood sampling suffers from unlikely evidence. This belief is inaccurate-likelihood sampling suffers mainly from a mismatch between the prior and the posterior probability distribution, as demonstrated in the example.

5.1.1 Enhancements to Sampling

There are several improvements on these two basic schemes, classified collectively as forward sampling because their order of sampling coincides with the direction of arcs in the network. Each node in the network is sampled after its parent have been sampled.

- **Stratified Sampling**
  One of the improvements to forward samplingis *stratified simulation* [1] (Bouckart 1994) that divides the whole sample space evenly into many parts, then picks one sample from each part. In other words, it allows for a systematic generation of samples without duplicates. The main problem in applying stratified sampling to large networks is that at each stage of the algorithm, we need to maintain the accumulated high and low bounds for each variable. In a network consisting of hundreds of variables, the high bound approaches the low bound as the sampling proceeds, and they will meet at some point due to the limit of the accuracy of number representations used to simulate the network. After this point variables will prevent the algorithm from generating desired samples, thus its performance will deteriorate.

- **Latin Hypercube sampling**
  Latin hypercube sampling [11]uses the idea of evenly dividing the sampling space, but it focus on the sample space of each node. It has been found to offer an improvement on any scheme, although the degree of improvement depends on the properties of the model [4] (Cheng and Druzdzel 1999).

- **Importance sampling**
  Importance sampling (Shachter and Peot 1990) uses samples from an "important distribution" rather than the original conditional distributions. This adds flexibility in devising strategies for instantiating a network during a simulation trial. It provides a way of choosing any sampling distribution, and compensating for this by adjusting the weight of each sample. This main difficulty related to this approach is defining a good importance sampling distribution. Self-importance sampling, for example, revises conditional probability table periodically in order to make the sampling distribution gradually approach the posterior distribution.

Another improvement is backward sampling. Backward sampling [7] (Fung and del Favor 1994) allows for generating samples starting from evidence nodes based on essentially any reasonable sampling distribution. Backward sampling will work better than forward sampling in the example presented in the section on likelihood sampling. In some case, however, both backward sampling and forward sampling will perform poorly.

5.2 Implementing the Simulation Algorithms

This section presents an implementation of the stochastic simulation algorithm based on the concept of Markov Blankets [15] and the logic sampling algorithm.

5.2.1 Stochastic Simulation With Markov Blankets

"*Metastatic cancer is a possible cause of a brain tumor. and is also an explanation for increased total serum calcium. In turn either of these could explain a patient falling into a coma. severe headache is also possibly associated with a brain tumor.*" The information about the qualitative dependences of this example are represented by the Bayesian network in Figure 5.1. The following probability distribution completely specify the example network

$$
\begin{array}{llllll}
p(a) & = & .20 & & & \\
p(b \mid a) & = & .80 & p(c \mid a) & = & .20 \\
p(b \mid \neg a) & = & .20 & p(c \mid \neg a) & = & 0.05 \\
p(d \mid b, c) & = & .80 & & & \\
p(d \mid \neg b, c) & = & .80 & & & \\
p(d \mid b, \neg c) & = & .80 & p(e \mid c) & = & .80 \\
p(d \mid \neg b, \neg c) & = & .05 & p(e \mid \neg c) & = & .60
\end{array}
$$

Given this distribution, our task is to compute the posterior probability of every proposition in the system, given that a patient is observed to be suffering from severe headaches, that is $E = e = 1$, but is definitely not in coma ($D = \neg d = 0$). The first step is to instantiate all the unobserved variables to some arbitrary initial state, say $A = B = C = 1$, and then let each variable, in turn, choose another state in accordance with the conditional probability of that variable, given the current state of all variables except A.

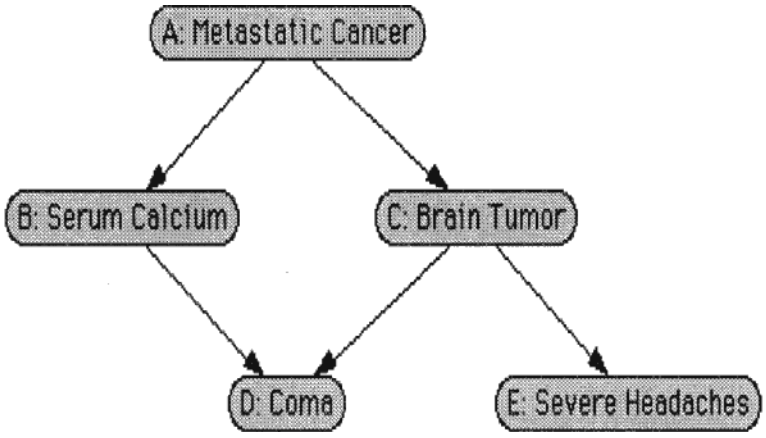

Figure 5.1: A Bayesian Network Describing Influences Among Five Variables.

That is, $w_A = \{B = 1, C = 1, D = 0, E = 1\}$. Then the next value of A will be chosen by tossing a coin that favors 1 to 0 by a ration of $p(a \mid w_A)$ to $p(\neg a \mid w_A)$. The distribution of each variable X conditioned on the values w_X of all other variables in the system, can be calculated by purely local computations. It is given simply as the product of the matrix of X times the link matrices of its children as follows

$$p(A \mid w_A) = p(A \mid B, C, D, E) = \alpha p(A)p(B \mid A)p(C \mid A) \quad (5.1)$$
$$p(B \mid w_B) = p(A \mid A, C, D, E) = \alpha p(B \mid A)p(D \mid B, C) \quad (5.2)$$
$$p(C \mid w_C) = p(C \mid A, B, D, E) = \alpha p(C \mid A)p(D \mid B, C)p(E \mid C) \quad (5.3)$$

where the α are normalizing constants. The probabilities associated with D and E are not needed because these variables are assumed to be fixed at $D = \neg d = 0$ and $E = e = 1$. Note that a variable X may determine its transition probability $p(X \mid w_X)$ by inspecting only parents, its children and those with which it shares children. This set of variables is called the *Markov Blanket of* X. For example, A needs to inspect only B and C.

Initial state

There are two ways to initialze the state

- To instantiate all the unobserved variables to some arbitrary initial state say $A = B = C = 1$.

- To instantiate the unobserved variables to some values according the given probability. For example, $p(a) = 0.2$ $p(\neg a) = 0.8$ using the uniform distribution to get the initial state. Since $A = a = 1$ or $A = \neg a = 0$ with only two values, we can use the Bernoulli algorithm to generate the value of A.

Markov Blanket Computations

Let $w_A = \{B = 1, C = 1, D = 0, E = 1\}$, then the next value of A will be chosen by tossing a coin that favors 1 to 0 by a ratio of $p(a \mid w_A)$ to $p(\neg a \mid w_A)$.

Note that variables X may determine its transition probability $p(X \mid w_X)$ by inspecting only its parents, its children and those with which it shares children. This set of variables is called the Markov Blanket of X.

$$p(A \mid w_A) = p(A \mid B, C, D, E) = \alpha p(A)p(B \mid A)p(C \mid A) \quad (5.4)$$
$$p(B \mid w_B) = p(A \mid A, C, D, E) = \alpha p(B \mid A)p(D \mid B)p(D \mid B, C) \quad (5.5)$$
$$p(C \mid w_C) = p(C \mid A, B, D, E) = \alpha p(C \mid A)p(D \mid B, C)p(E \mid C) \quad (5.6)$$

Judea Pearl had proved these three formula in his research note.

- Activating node A

$$\begin{aligned} p(A = 1 \mid B = 1, C = 1) &= \alpha p(a)p(b \mid a)p(c \mid a) = \alpha * 0.2 * 0.8 * 0.2 \\ &= \alpha * 0.032 \\ p(A = 0 \mid B = 1, C = 1) &= \alpha * 0.8 * 0.2 * 0.05 = \alpha * 0.008 \end{aligned}$$

  From this, we see that $\alpha = 1/0.032 + 0.008 = 25$. Thus, $p(A = 1 \mid w_A) = 25 * 0.032 = 0.80$ and $p(A = 0 \mid w_A) = 25 * 0.008 = 0.20$. Then, node A consults a random number generator that issues ones with probability .80 and zero with probability .20. Assuming the value sampled is 1, A adopts this value $A = 1$, and control shifts to node B.

- Activating node B
  Node B looks at its neighbors, with $A = 1$, $C = 1$, $D = 0$, $\frac{p(B=1)}{p(B=0)} = 4$.

$$\begin{aligned} p(B = 1 \mid A = 1, C = 1, D = 0) &= \alpha p(b \mid a)p(\neg d \mid b, c) \\ &= \alpha * 0.80 * (1 - 0.80) \\ p(B = 0 \mid A = 1, C = 1, D = 0) &= \alpha p(\neg b \mid a)p(\neg d \mid \neg b, c) \\ &= \alpha * (1 - 0.80) * (1 - 0.80) \end{aligned}$$

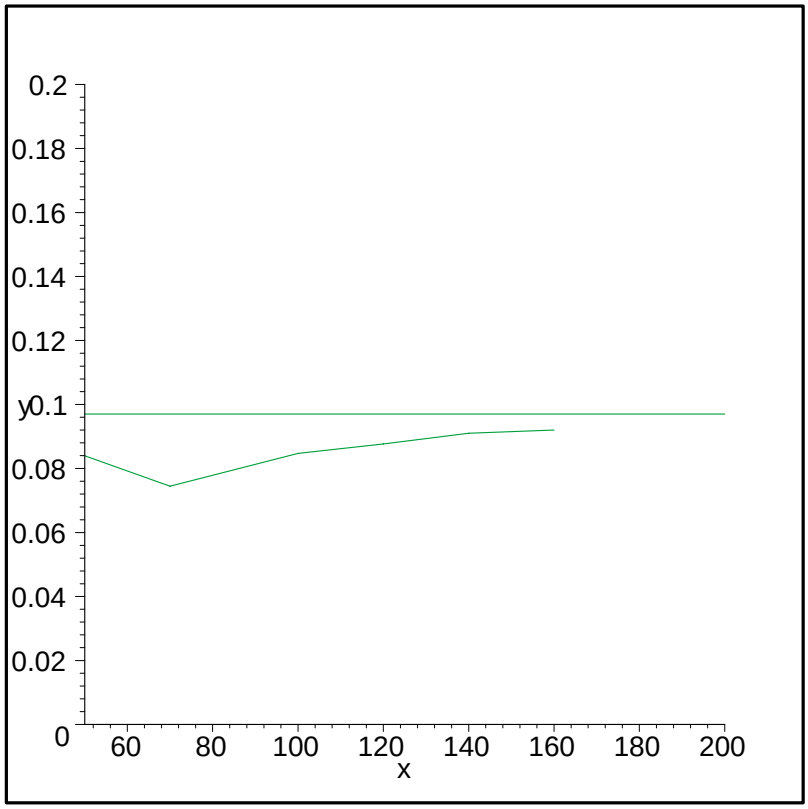

Figure 5.2: Stochastic Simulation: the Straight Line Represents the Exact Probability 0.097 of Node A.

As node A did in its turn, node B samples a random number generator favoring ones to zeros by a 4:1 ratio. Assuming this time B is set to 0 and gives control to node C.

- Activating node C
  The neighbors of node C are at the state $w_C = \{A = 1, B = 0, D = 0, E = 1\}$. Therefore, $\frac{p(C=1)}{p(C=0)} = 1/14.25$, Node C samples a random number generator favoring ones to zeros by a 14.15:1 ratio. Assuming 0 is sampled, node C adopts the value and gives control to node A.

The expected probability of node A is 0.097 and Figure (5.2) shows the simulation result for multiple iterations.

5.2.2 Logic Sampling

Suppose we represent a Bayesian network by a sample of m deterministic *scenarios* $s = 1, 2, \ldots, m$. Suppose $L_s(x)$ is the truth of event x in scenario s. Then uncertainty about x can be represented by a *Logic Sample*, that is the vector of truth values for the sample of scenarios, which is

$$L(x) = [L_1(x), L_2(x), \ldots, L_m(x)] \tag{5.7}$$

If we are given the prior probability $p(x)$, we can use a random number generator to produce a logic sample for x. Given a logic sample, $L(x)$, we can estimate the probability of x as the *truth* fraction of the logic sample, i.e. the proportion of scenarios in which x is true

$$p(x) = \sum_{s=1}^{m} p(L_s(x))/m \tag{5.8}$$

For each conditional probability distribution given, such as $p(X \mid Y)$, we can generate a logic sample for each of its independent parameters using the corresponding probabilities. For example $p(x \mid y)$, $p(x \mid \neg y)$. We denote these conditional logic sample as $L(x \mid y)$, $L(x \mid \neg y)$. Ecah can be viewed as a vector of implication rules from the parent(s) to child. For a given scenario, s, the values of the two conditional logic samples specify the state of x for any state of its parent, y.

Figure 5.3 shows the example Bayesian belief network with each influence expressed as a conditional probability distribution in tabular form. Figure 5.4 shows a particular deterministic scenario from the sample with each of the corresponding influences is expressed as a truth table. Figure 5.5 presents the first few scenarios from a sample. Each horizontal vector of truth values represents a logic sample for the specified variable or conditional relation. Each of the columns of *truth* values represents one of the m scenarios. The first column of probabilities are those for the Bayesian belief network and used to generate the logic samples to their left.

It is straightforward to compute the truth of each variable given the state of its parents and the deterministic influence of its parents. For scenario s, we obtain the truth of b given its parent a, then

$$L_s(b) = L_s(b \mid a)L_s(b \mid \neg a) \vee L_s(b \mid \neg a)L_s(\neg a)$$

In this way, we can work down from the source nodes to their successive descendants, using simple logical operations to compute the truth for each

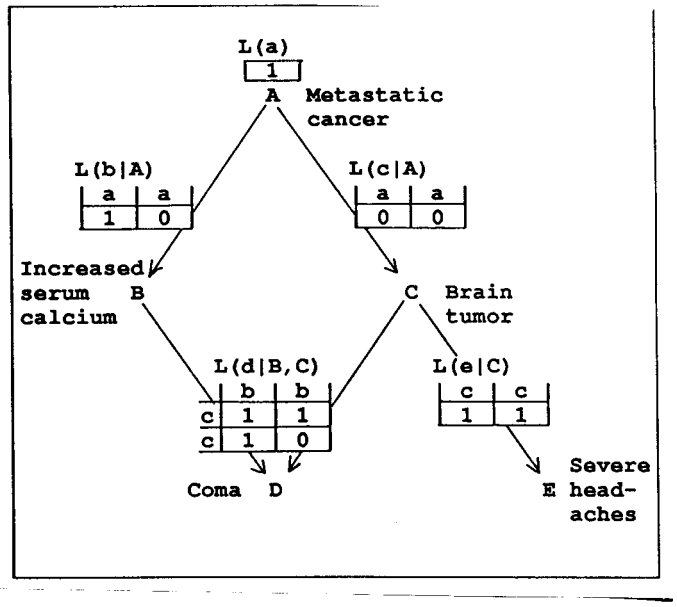

Figure 5.3: Bayesian Belief Network With Probabilities.

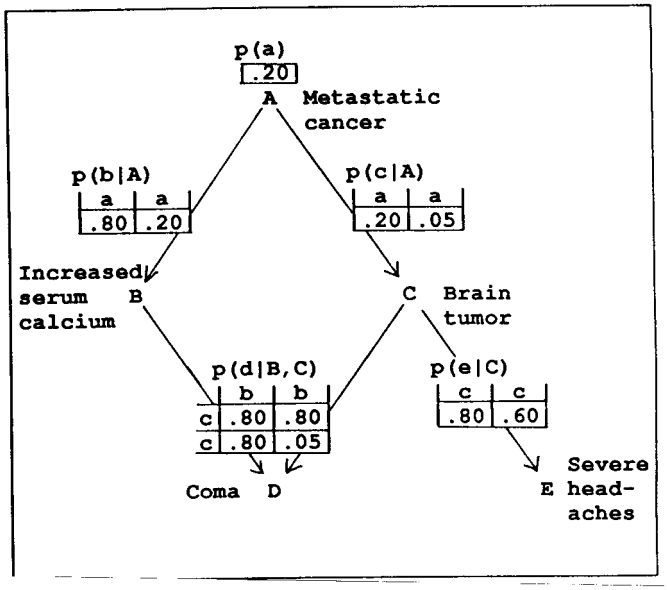

Figure 5.4: A Deterministic Scenario From the Example Network.

variable as follows

$$\begin{aligned} L_s(c) &= L_s(c \mid a)L_s(a) \vee L_s(c \mid \neg a)L_s(\neg a) \\ L_s(d) &= [L_s(d \mid b, c)L_s(c) \vee L_s(d \mid b, \neg c)L_s(\neg c)]L_s(b) \\ &\quad \vee [L_s(d \mid \neg b, c)L_s(c) \vee L_s(d \mid \neg b, \neg c)]L_s(\neg b) \\ L_s(e) &= L_s(e \mid c)L_s(c) \vee L_s(e \mid \neg c)L_s(\neg c) \end{aligned}$$

Note that identity is the deterministic counterpart of the probabilistic chain rule

$$p(b) = p(b \mid a)p(a) + p(b \mid \neg a)(1 - p(a))$$

The problem with performing simple probabilistic chaining in this example is that to compute $p(d)$ we need the joint distribution over its parents $p(B, C)$, but probabilistic chaining would only give us the marginal $p(B)$ and $p(C)$, and assuming independence would be incorrect in general. But in a deterministic cases, the individual truth values determine the joint truth value

$$L_s(b, c) = L_s(b)L_s(c)$$

Logic sample	Given Probs	Scenario # 1 2 3 4 ...m	Estimated probs
L(a)	.20	[1 0 0 0 ...]	
L(b|a)	.80	[1 0 1 1 ...]	
L(b|a)	.20	[0 1 0 1 ...]	
L(b)		[1 1 0 1 ...]	0.32
L(c|a)	.20	[0 1 0 0 ...]	
L(c|a)	.05	[0 0 1 0 ...]	
L(c)		[0 0 1 0 ...]	0.08
L(d|b,c)	.80	[1 0 1 1 ...]	
L(d|b,c)	.80	[0 1 1 1 ...]	
L(d|b,c)	.80	[1 1 1 0 ...]	
L(d|b,c)	.05	[0 0 0 1 ...]	
L(d)		[1 1 1 0 ...]	0.34
L(e|c)	.80	[0 1 0 0 ...]	
L(e|c)	.60	[0 0 1 0 ...]	
L(e)		[0 0 1 0 ...]	0.63
L(a&e)		[0 0 0 0 ...]	0.13

Figure 5.5: Logic Simulation Example.

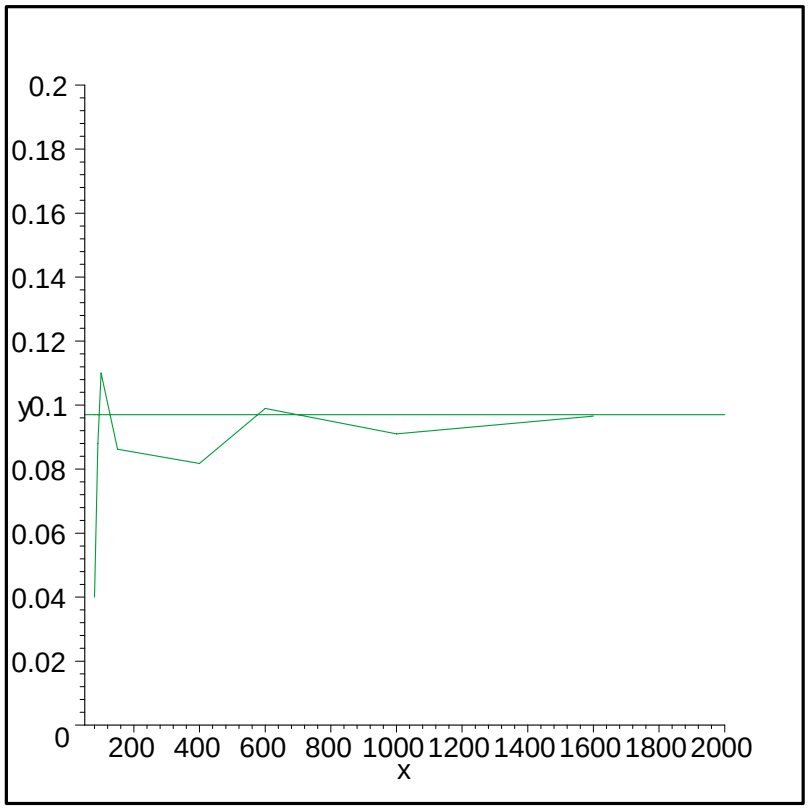

Figure 5.6: Logic Simulation: the Straight Line Represents the Exact Probability 0.097 of Node A.

While a single scenario says little about the probabilities, a sample of several scenarios can be used to estimate them. The marginal probability for each variable x, is estimated as the truth fraction of its Logic Sample, $T[L(x)]$. In Figure 5.6 the second column of probabilities are those estimated from the logic samples to their right. Similarly, we can estimate the joint probability of any set of variables, or indeed the probability of any Boolean combination of variables from the truth fraction of that Boolean combination of their Logic Sample.

In summary, probabilistic logic sampling proceeds as follows, assuming we start with a Bayesian network with priors specified for all source variables and conditional distributions for all others:

1. Use a random number generator to produce a sample truth value for each source variable, and a sample implication rule for each parameter of each conditional distribution, using the implication rules.

2. Proceed down through the network following the arrows from the source nodes, using the sample logical operations to obtain in the truth of each variable from its parents and the implication rules.

3. Repeat steps 2 and 3 m times to obtain a logic sample for each variable.

4. Estimate the prior marginal probability of any simple or compound event by the truth fraction of its logic sample, i.e. the fraction of scenario in which they are true.

5. Estimate the posterior probability for any event conditional on any set of observed variables as the fraction of sample scenarios in which the event occurs out of those in which the condition occurs.

5.3 The Effect of Random Number Generators on Simulation Convergence

We will discuss how the implementations of approximate algorithms is affected by using different Pseudo-Random Number Generators (PRNGs). These generators play a crucial role in implementing approximate algorithms for Bayesian networks. Convergence rate and robust of any approximate algorithms is damaged by using a bad generator.

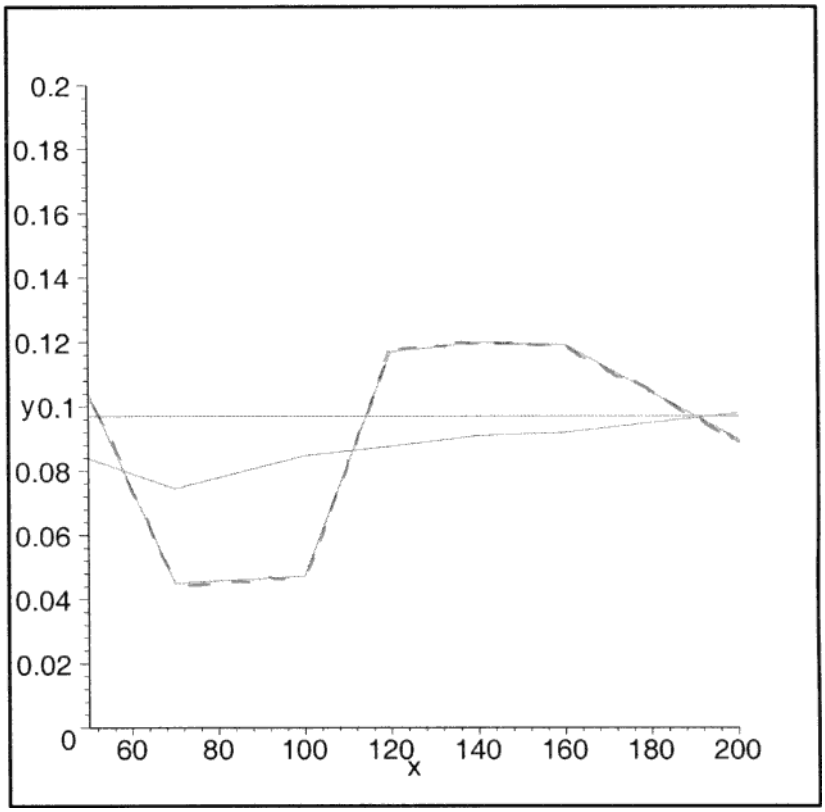

Figure 5.7: Convergence of the Stochastic Simulation Algorithm Using `rand()` (Dashed Line) and MT (Solid Line). The Exact Probability is 0.097.

We choose two PRNGs to test the Markov Blanket and logic sampling algorithms: the Mersenne twister (MT) and the PRNG of the C library, which is a linear congruential generator (LCG). The standard LCG has recurrence

$$x_{n+1} = ax_n + c \pmod{p}$$

Because the choices of a, p, and c are not optimal, the PRNG of the C library is not reliable.

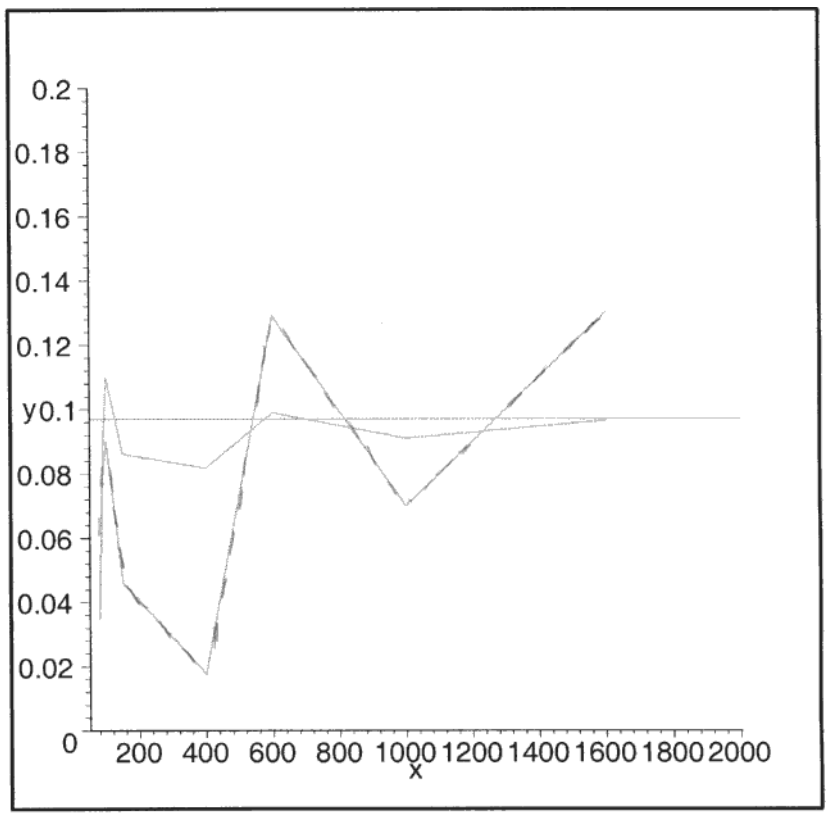

Figure 5.8: Convergence of the Logic Sampling Algorithm Using `rand()` (Dashed Line) and MT (Solid Line). The Exact Probability is 0.097.

5.4 Summary

We introduced the background of approximate algorithms. Implementations of two typical approximate algorithms in the Coma network are described. Two different PRNGs are chosen to implement the two approximate algorithms. The choice of PRNGs affects the performance of approximate algorithms. Reliable PRNGs, such as MT for example, is a good choice for any approximate algorithm.

Chapter 6

Putting the Approximate Algorithms to the Test

In this chapter, two of classical approximate algorithms are tested with randomly generated Bayesian belief networks.

6.1 Logic Sampling With Randomized Network

In Chapter 5, we used an existing network to run our logic sampling algorithm. In this chapter, we use our random generation of networks to test this algorithm. The network we used is shown in Figure 4.1. The result is depicted in Figures 6.1 and 6.2.

The propagation of evidence $E = 1$ and $D = 0$ was tested. The exact probability of node A is 0.7 in Figure 6.1 and the exact probability of node A is 0.42 in Figure 6.2. From both of these graphs, we can see that the graph in Figure 6.1 is convergent because its probability table is not extreme. However, the graph in Figure 6.2 is not convergent. It is hard to tell when the simulated value will get close to exact probability value.

This conclusion is consistent with that logic sampling. It is hard in general to deal with networks with extreme probability distributions.

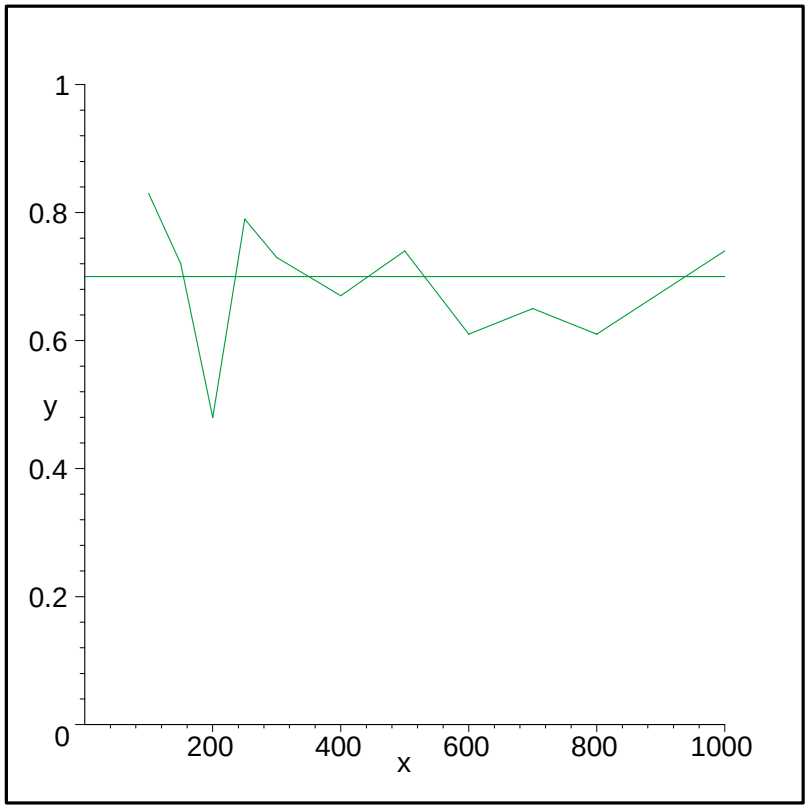

Figure 6.1: Simulation Result of Logical Sampling for Network With $N = 5$ and $M = 2$ Shown In Figure 4.1 and Table 4.1. The Straight Line Represents the Exact Probability 0.7.

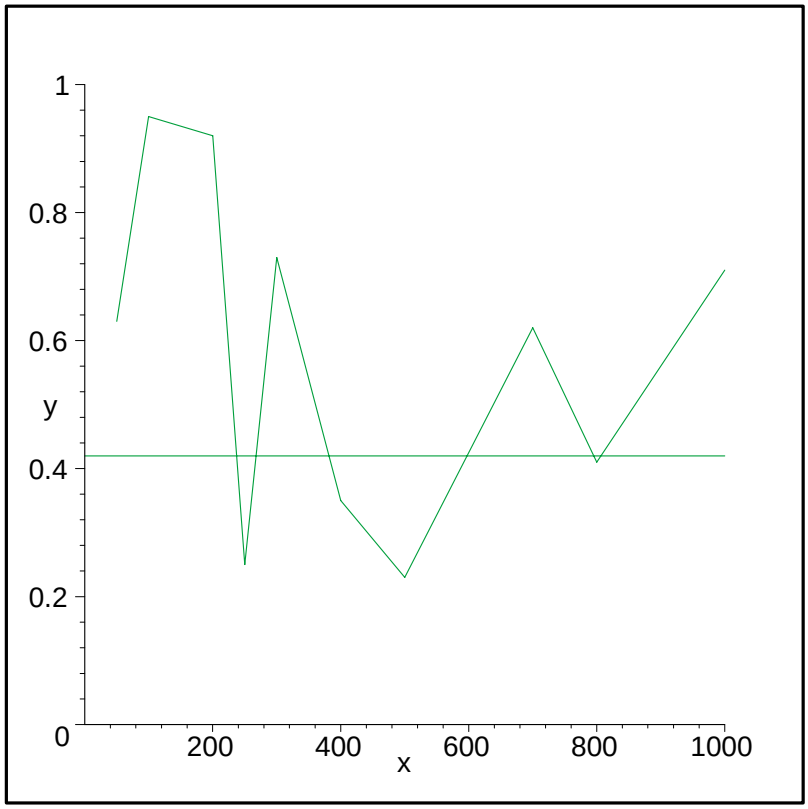

Figure 6.2: Simulation Result of Logical Sampling for Network With $N = 5$ and $M = 2$ Shown in Figure 4.1 and Table 4.2. The Straight Line Represents the Exact Probability 0.42.

6.2 Markov Blanket With Randomized Network

In Chapter 5, we used an existing network to run our Stochastic simulation algorithm with Markov blankets. In this chapter, we use our random generation of networks to test this algorithm. The network we used is shown in Figure 4.1. The result is depicted in Figures 6.3 and 6.4.

The propagation of evidence is $E = 1$ and $D = 0$ was tested. The exact probability of node A is 0.7 in Figure 6.3 and the exact probability of node A is 0.42 in Figure 6.4. We see similar results as with the logic sampling tests. We can see from Figure 6.3 that the graph is covergent because its probability table is not extreme. The graph in Figure 6.4 is not convergent.

6.3 Summary

A good approximation algorithm has a fast convergence rate. Because the random generations of Bayesian Networks can produce very different graph and probability distributions for a Bayesian belief network, it is perfect to use to test the approximate algorithms.

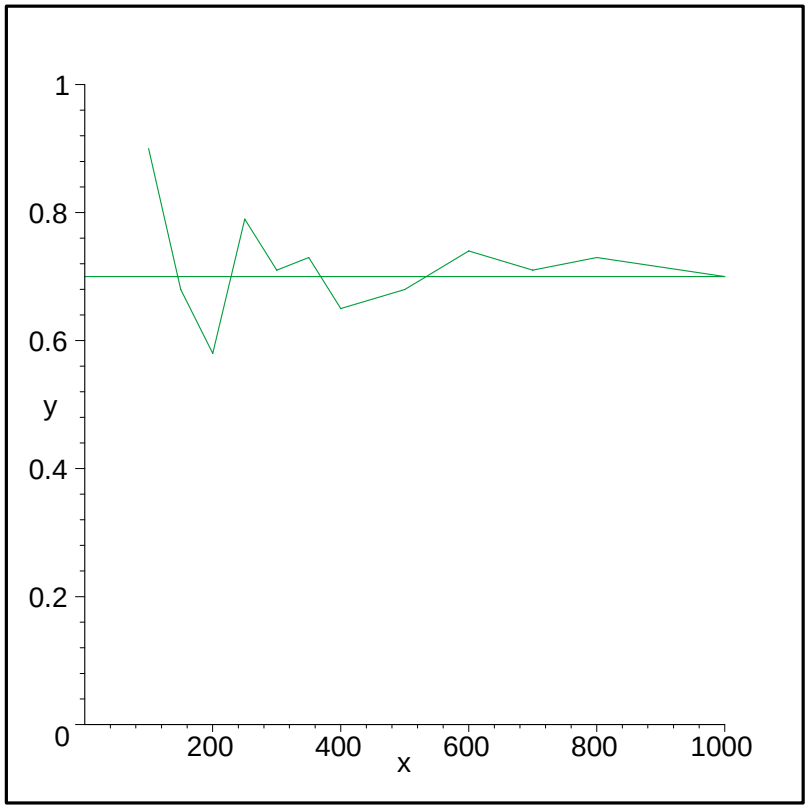

Figure 6.3: Simulation Result of Markov Blanket for Network With $N = 5$ and $M = 2$ Shown in Figure 4.1 and Table 4.1. The Straight Line Represents the Exact Probability 0.7.

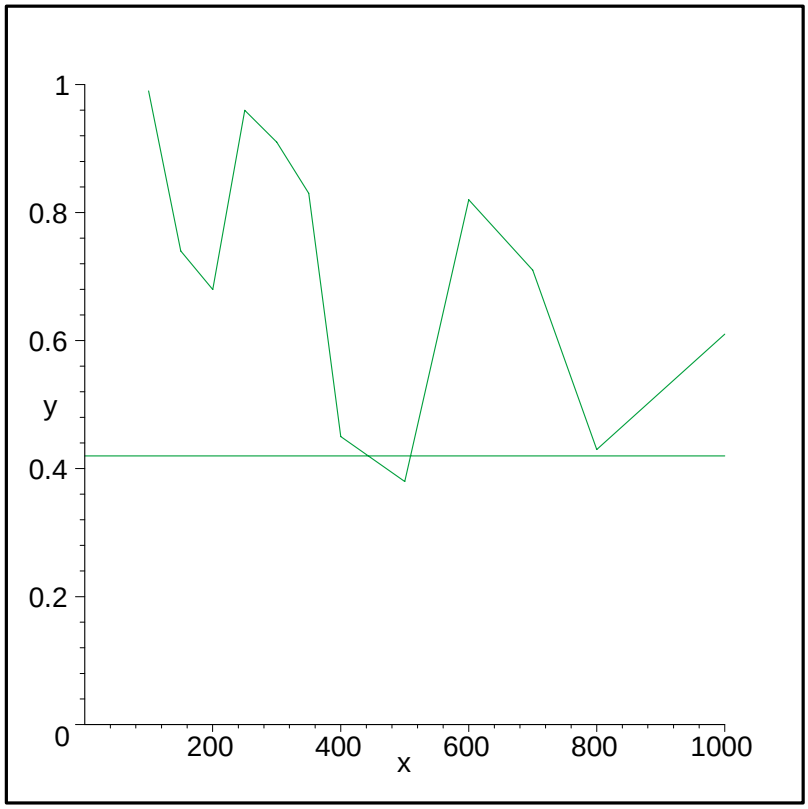

Figure 6.4: Simulation Result of Markov Blanket for Network With $N = 5$ and $M = 2$ Shown in Figure 4.1 and Table 4.2. The Straight Line Represents the Exact Probability 0.42.

Chapter 7

Conclusions and Further Work

In this report we presented how to generate randomized Bayesian belief networks and use these to test simulation algorithms.

The first goal of this project is to generate random networks, which includes its graph and probability distribution. The second goal of this project is to test simulation algorithms by using these random generation network.

Random numbers plays critical part in our project. We reviewed the theoretical and practical background of random number generation. We found that TGFSR is the best choice for our simulation. TGFSR has been implemented by a fast and effective method. Also it has a long period and past all available statistical tests.

We chose two famous approximate algrithms to test

- Stochastic simulation (with Markov blankets) proposed by Pearl[9]
- Logic sampling proposed by Henrion[8]

As for future work, many new simulation algorithm methods are proposed, such as Latin Hypercube Sampling and Systemic sampling (Stratified sampling), which are better to deal with extreme distributions, if we can use these random network as benchmark to test and we can improve our codes for more complicated network.

Bibliography

[1] Remco R. Bouckaert. A stratified simulation scheme for inference in bayesian belief networks. *Proceedings of the Tenth Conference on Untertainty in Artificial Intelligence*, Seattle,Morgan Kaufman,San Francisco:56–62, 1995.

[2] R.Martin Chavez and G.F.Cooper. A randomized approxumation algorithm for inference in bayesian belief networks. *Networks*, 20:661–685, 1990.

[3] R.Martin Chavez and G.F.Cooper. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generators. *ACM Trans.Modling and Computer simulations*, 8:3–30, 1998.

[4] Jian Cheng and Marek J.Druzdeel. Latin hypercubes samlping in bayesian networks. *In proceedings of the Uncertain Reasoning in Artificial Intelligence track of the Thirteenth International Florida Artificial Intelligence Research Symposium Conference*, FLAIRS-2000:287–292, 2000.

[5] G.F. Cooper. The computational complexity of probabilistic inference bayesian belief networks. *Artificial Intelligence*, 42:393–348, 1990.

[6] D.E.Knuth. *Seminumerical Algorithm 3rd ed.. Vol 2,The art of computer programming*. Addison Wesley, Reading,MA, 1997.

[7] R. Fung and K.C. Chang. Weighting and integrating evidence for stochastic simulation in bayesian networks. *Uncertainty in Artificial Intellegence*, 5:209–220, 1990.

[8] M. Henrion. Propagating uncertainty by logic sampling in bayesian networks. *Uncertainty in Artificial Intellegence*, 2:317–324, 1988.

[9] J.Pearl. *Probabilistic Reasoning in Intelligent Syatems: Networks of Plausible Inference*. Morgan Kaufman, San Mateo, 1988.

[10] T.G. Lewis and W.H. Payne. Generalized feedback shift register pseudorandom number algorithms. *Journal of ACM*, 20:456–468, 1973.

[11] Conover W.J Mckey, M.D and R.J. Beckman. A comparision of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21:239–245, 1979.

[12] M.Matsuoto and Y.Kurita. Twister gfsr generators. *ACM Trans.Modling and Computer simulations*, 2:179–194, 1992.

[13] P.Dagum and E.Horvitz. A bayesian analysis of simulation algorithms for inference in belief networks. *Networks*, 23:499–516, 1993.

[14] J. Pearl. Fusion,propagation,and structuring in belief networks. *Artificial Intelligence*, 28-20:241–282, 1986.

[15] J. Pearl. Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32:247–257, 1987.

Appendix A

Codes for Random Network Generation

CONST.h

```
#define N  10
#define M  2

typedef struct InfoNode{
  int NumOfPar;
  int WhoArePar[M];
};
```

genbbn.cpp

```
#include "CONST.h"
#include<iostream.h>
#include<stdlib.h>
#include<time.h>

class bbn {
 private:
   InfoNode node[N];

 public:
       bbn();
 void  bbnGeneration();
```

```
 void  printNode();

 private:
   void  putNumOfPar(int);
   void  putNumOfPar();
   void  putWhoArePar(int);
   int  comparePar(int ,int);
};

//-------------------------
bbn::bbn()
{
    node[0].NumOfPar=0;
    node[0].WhoArePar[0]=-1;

     node[1].NumOfPar=1;
     node[1].WhoArePar[0]=0;

for (int i=2;i<N;i++)
  {
   node[i].NumOfPar= 0;
   for(int j=0;j<M;j++)
     node[i].WhoArePar[j]=-1;
  }
}


//------------------------
void bbn::printNode()
{

for (int i=0;i<N;i++)
  {
  cout<<"node="<<i+1<<"  ";

   cout<<"NumOfPar="<<node[i].NumOfPar<<" Parents= ";
   for(int j=0;j<node[i].NumOfPar;j++)
    cout<< node[i].WhoArePar[j]<<" ";
```

```
    cout<<endl;
  }
}
//-----------------------------------------
 void bbn::bbnGeneration()
{
 cout<<"Algorithm I:"<<endl;
   putNumOfPar();
  for(int i=2;i<N;i++)
    putWhoArePar(i);
   printNode();

 cout<<endl<<"Algorithm II:"<<endl;
 for(int i=2;i<N;i++)
 {
   putNumOfPar(i);
   putWhoArePar(i);
 }
  printNode();
}
//-----------------------------------------
void bbn::putNumOfPar()
{
 int np[N/2+1];
 int ct;
 for (int i=0;i<N/2+1;i++)
   { srand(time(0));
     np[i]=rand()%N;
     ct=0;
     for(int j=0;j<i;j++)
       while(np[i]==np[j]&&ct<100000)
       { ct++;
         srand(time(0));
         np[i]=rand()%N;
       }
     }
 for(int i=0;i<N/2+1;i++)
if(np[i]!=0&&np[i]!=1)node[np[i]].NumOfPar=2;
```

```
  for(int i=2;i<N;i++)
   if(node[i].NumOfPar!=2)node[i].NumOfPar=1;
}


//-------------------------------------------
void bbn::putNumOfPar(int ii)   //ii>=2
{
    srand(time(0));
    int s=rand()%1000;

    if(s>=500)
     node[ii].NumOfPar=1;
    else
     node[ii].NumOfPar=2;
}

//---------------------------------
int bbn::comparePar(int ii, int d)
{
 int currentPar=node[ii].WhoArePar[d];

  for(int i=0;i<d;i++)
  if( node[ii].WhoArePar[i]==currentPar)
  return 1;

  return 0;
}

//-----------------------------------------

void bbn:: putWhoArePar(int ii ) //ii>=2
{
 int p,q,ct;
 int n=node[ii].NumOfPar;

    if(n==1)
```

```
    {
     srand(time(0));
     p=rand()%ii;
     node[ii].WhoArePar[0]=p;
    }
  else
    {
    for(int j=0;j<node[ii].NumOfPar;j++){
     srand(time(0));
     node[ii].WhoArePar[j]=rand()%ii;
     ct=0;
      while(j>0 && comparePar(ii,j)==1){
  ct++;
           // if(ct>10000) break;
            srand(time(0));
            node[ii].WhoArePar[j]=rand()%ii;
         } //end while.
    }     //end for.

 } //else
}

//====================MAIN=====

void  main()
{
 bbn ob1;
  ob1.bbnGeneration();
}
```

Appendix B

Codes for the Random Number Generator

Random Number Generator(MT) /* A C-program for MT19937: Real number version */ /* genrand() generates one pseudorandom real number (double) */ /* which is uniformly distributed on [0,1]-interval, for each */ /* call. sgenrand(seed) set initial values to the working area */ /* of 624 words. Before genrand(), sgenrand(seed) must be */ /* called once. (seed is any 32-bit integer except for 0). */ /* Integer generator is obtained by modifying two lines. */ /* Coded by Takuji Nishimura, considering the suggestions by */ /* Topher Cooper and Marc Rieffel in July-Aug. 1997. */

/* Copyright (C) 1997 Makoto Matsumoto and Takuji Nishimura. */ /* Any feedback is very welcome. For any question, comments, */ /* see http://www.math.keio.ac.jp/matumoto/emt.html or email */ /* matumoto@math.keio.ac.jp */

/* Modified by Changyun Wang 10/7/01 */

```
#include<stdio.h>

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */
```

```
/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializing the array with a NONZERO seed */
void
sgenrand( unsigned long seed)
{
    /* setting initial seeds to mt[N] using         */
    /* the generator Line 25 of Table 1 in          */
    /* [KNUTH 1981, The Art of Computer Programming */
    /*    Vol. 2 (2nd Ed.), pp102]                  */
    mt[0]= seed & 0xffffffff;
    for (mti=1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

double /* generating reals */
/* unsigned long */ /* for integer generation */
genrand()
{
    unsigned long y;
    static unsigned long mag01[2]={0x0, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used   */
```

```
        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

        mti = 0;
    }

    y = mt[mti++];
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);

    return ( (double)y / (unsigned long)0xffffffff ); /* reals */
    /* return y; */ /* for integer generation */
}
```

Appendix C

Implementation of Stochastic Sampling

```
// This code is written to complete Markov blanket algorithm
// which is proposed by Pearl.
#include<iostream.h>
#include<math.h>
#define M 100

void sgenrand( unsigned long );
double genrand();

main(){
int num[]={1,2,2,4,2};
float pr[][4]={{0.2},{0.8,0.2},{0.2,0.05},{0.8,0.8,0.8,0.05},{0.8,0.6}};
float temp1,temp2, probA[M], valueA[M];
int i,j,k,A,B,C,D,E;
double ii, P, nP;

D=0; E=1; temp1=0;temp2=0;

sgenrand(98765);
// Instantiate all unobserved variables A,B,C
   ii=genrand();
   if(ii<=0.2)A=1;
    else A=0;
```

```
  if (A==1 && genrand()<=0.8)B=1;
   else B=0;

  if (A==1 && genrand()<=0.2)C=1;
   else C=0;


for(i=0;i<M;i++){
  // Activating A

  if( B==1 && C==1){
    P=pr[0][0]*pr[1][0]*pr[2][0];
    nP=(1-pr[0][0])*pr[1][1]*pr[2][1];
   }
  if( B==0 && C==0){
     P=pr[0][0]*(1-pr[1][0])*(1-pr[2][0]);
     nP=(1-pr[0][0])*(1-pr[1][1])*(1-pr[2][1]);
    }
  if( B==1 && C==0){
    P=pr[0][0]*pr[1][0]*(1-pr[2][0]);
    nP=(1-pr[0][0])*pr[1][1]*(1-pr[2][1]);
   }
  if( B==0 && C==1){
   P=pr[0][0]*(1-pr[1][0])*pr[2][0];
   nP=(1-pr[0][0])*(1-pr[1][1])*pr[2][1];
   }
  ii=1/(P+nP);
  P=ii*P; nP=ii*nP;

//Sample A;
  ii=genrand();
  if(ii<=P) A=1;
   else A=0;
  valueA[i]=A; probA[i]=P;
  temp1 += A; temp2 += P;

 //Activating B
```

```
  if( A==1 && C==1 && D==0){
     P=pr[1][0]*(1-pr[3][0]);
     nP=(1-pr[1][0])*(1-pr[3][1]);
   }

  if( A==1 && C==0 && D==0){
     P=pr[1][0]*(1-pr[3][2]);
     nP=(1-pr[0][0])*(1-pr[3][3]);
   }

  if( A==0 && C==1 && D==0){
     P=pr[1][1]*(1-pr[3][0]);
     nP=(1-pr[1][1])*(1-pr[3][1]);
   }

  if( A==0 && C==0 && D==0){
     P=pr[1][1]*(1-pr[3][2]);
     nP=(1-pr[1][1])*(1-pr[3][3]);
   }

   ii=1/(P+nP);
  P=ii*P; nP=ii*nP;

 //Sample B;
  ii=genrand();
  if(ii<=P) B=1;
   else B=0;

//Activating C
 if(A==1 && B==1){
  P = pr[2][0]*(1-pr[3][0])*pr[4][0];
  nP= (1-pr[2][0])*(1-pr[3][2])*pr[4][1];
 }

 if(A==1 && B==0){
  P = pr[2][0]*(1-pr[3][0])*pr[4][0];
  nP= (1-pr[2][0])*(1-pr[3][2])*pr[4][1];
 }
```

```
if(A==0 && B==1){
  P = pr[2][1]*(1-pr[3][0])*pr[4][0];
  nP= (1-pr[2][1])*(1-pr[3][2])*pr[4][1];
 }

 if(A==0 && B==0){
  P = pr[2][1]*(1-pr[3][1])*pr[4][0];
  nP= (1-pr[2][1])*(1-pr[3][2])*pr[4][1];
 }

  ii=1/(P+nP);
  P=ii*P; nP=ii*nP;

 //Sample C;
  ii=genrand();
  if(ii<=P) C=1;
   else C=0;
}// end of for

cout<<"computed by frequency of A=1 is "<< temp1/M<<endl;
cout<<"computed by conditional probility A=1 is" << temp2/M<<endl;

}


\pagebreak

// This code is written to complete Logic Sampling algorithm
// which is proposed by Henrion.

#include<stdio.h>
#include<math.h>
#define M 140

void sgenrand( unsigned long );
double genrand();
```

```
main(){
int num[]={1,2,2,4,2};
float pr[][4]={{0.2},{0.8,0.2},{0.2,0.05},{0.8,0.8,0.8,0.05},{0.8,0.6}};
int temp[M][4], final[M][5];
int i,j,k, aDe, De;
double ii;

 sgenrand(98765);
//generate samples for A
 for (i=0;i<M;i++){
   ii=genrand();
   if(ii<=0.2)final[i][0]=1;
    else final[i][0]=0;
 }

 //generate samples for B
  for (i=0;i<M;i++){
    ii=genrand();
    if(ii<=0.8)temp[i][0]=1;
    else temp[i][0]=0;

    ii=genrand();
    if(ii<=0.2)temp[i][1]=1;
    else temp[i][1]=0;

    final[i][1]=(temp[i][0]&&final[i][0])||(temp[i][1]&&(1-final[i][0]));
  }

 //generate samples for C
  for (i=0;i<M;i++){
    ii=genrand();
    if(ii<=0.2)temp[i][0]=1;
    else temp[i][0]=0;

    ii=genrand();
    if(ii<=0.05)temp[i][1]=1;
    else temp[i][1]=0;
```

```
    final[i][2]=(temp[i][0]&&final[i][0])||(temp[i][1]&&(1-final[i][0]));
  }

 //generate samples for D
  for (i=0;i<M;i++){
    ii=genrand();
    if(ii<=0.8)temp[i][0]=1;
    else temp[i][0]=0;

    ii=genrand();
    if(ii<=0.8)temp[i][1]=1;
    else temp[i][1]=0;

    ii=genrand();
    if(ii<=0.8)temp[i][2]=1;
    else temp[i][2]=0;

    ii=genrand();
    if(ii<=0.05)temp[i][3]=1;
    else temp[i][3]=0;


   k=(temp[i][0]&&final[i][2]&&final[i][1]) || (temp[i][1]&&(1-final[i][1])&&fin
al[i][2]);
   j=(temp[i][2]&&final[i][1]&&(1-final[i][2])) || (temp[i][3]&&(1-final[i][1])&
&(1-final[i][2]));
   final[i][3]=k||j;
  }

//generate sample for E
  for (i=0;i<M;i++){
    ii=genrand();
    if(ii<=0.8)temp[i][0]=1;
    else temp[i][0]=0;

    ii=genrand();
    if(ii<=0.6)temp[i][1]=1;
    else temp[i][1]=0;
```

```
    final[i][4]=(temp[i][0]&&final[i][2])||(temp[i][1]&&(1-final[i][2]));
  }


//estimate aDe/De
aDe=0;De=0;
for (i=9;i<M;i++){
 aDe += final[i][0]*(1-final[i][3])*final[i][4];
 De += (1-final[i][3])*final[i][4];
}

ii=double(aDe)/double(De);

printf(" estimated probility of p(a|D,e)=%f\n",ii);

}
```

Appendix D

Codes to Test the Algorithms

sto.cpp

```
#include "structofbn.h"

class beliefNet
{
 private:
 nodeInfo A[20];
 void setAvalue();
 int proA(int, int);
 int proB(int,int);
 int proC(int,int);
 int bernouli(float p);

 public:
 void inputFun();

 void blanketPro(int, int, int);

};
//----------------------------------------

void beliefNet::inputFun()
{
  setAvalue();
```

```
}


int beliefNet::bernouli(float p)
 {
   int i;
   i=rand()%100;
   if(p*100>i)
   return 1;
   else
   return 0;
 }
 void beliefNet::setAvalue()
 {
  A[0].parentNum=0;
  A[0].pro[0]=0.20;
  A[1].parentNum=1;
  A[1].parentOfver[0]=1;
  A[1].pro[0]=0.80;
  A[1].pro[1]=0.20;
  A[2].parentNum=1;
  A[2].parentOfver[0]=1;
  A[2].pro[0]=0.20;
  A[2].pro[1]=0.05;
  A[3].parentNum=2;
  A[3].parentOfver[0]=2;
  A[3].parentOfver[1]=3;
  A[3].pro[0]=0.80;
A[3].pro[1]=0.80;
  A[3].pro[2]=0.80;
  A[3].pro[3]=0.05;
  A[4].parentNum=1;
  A[4].parentOfver[0]=3;
  A[4].pro[0]=0.80;
  A[4].pro[1]=0.60;

}
```

```
 int beliefNet::proA(int b,int c)
 {
    double norm;
    float PA[2];
   if(b==1&&c==1)
{
  PA[1]=A[0].pro[0]*A[1].pro[0]*A[2].pro[0];
     PA[0]=(1-A[0].pro[0])*A[1].pro[1]*A[2].pro[1];
     norm=1/(PA[0]+PA[1]);
     PA[1]=PA[1]*norm;
     PA[0]=PA[0]*norm;
     cout<<PA[1]<<"  "<<PA[0]<<"A,b=1,c=1"<<endl;
    return bernouli(PA[1]);
}


if(b==0&&c==1)
{
     PA[1]=A[0].pro[0]*(1-A[1].pro[0])*A[2].pro[0];
     PA[0]=(1-A[0].pro[0])*(1-A[1].pro[1])*A[2].pro[1];
     norm=1/(PA[0]+PA[1]);
     PA[1]=PA[1]*norm;
     PA[0]=PA[0]*norm;
   PA[0]=PA[0]*norm;
     cout<<PA[1]<<"  "<<PA[0]<<"A,b=0,c=1"<<endl;
     return bernouli(PA[1]);

}
  if(b==1&&c==0)
{
     PA[1]=A[0].pro[0]*A[1].pro[0]*(1-A[2].pro[0]);
     PA[0]=(1-A[0].pro[0])*A[1].pro[1]*(1-A[2].pro[1]);
     norm=1/(PA[0]+PA[1]);
     PA[1]=PA[1]*norm;
     PA[0]=PA[0]*norm;
      cout<<PA[1]<<"  "<<PA[0]<<"A,b=1,c=0"<<endl;
```

```
    return bernouli(PA[1]);


}
if(b==0&&c==0)
{
 PA[1]=A[0].pro[0]*(1-A[1].pro[0])*(1-A[2].pro[0]);
     PA[0]=(1-A[0].pro[0])*(1-A[1].pro[1])*(1-A[2].pro[1]);
     norm=1/(PA[0]+PA[1]);
     PA[1]=PA[1]*norm;
     PA[0]=PA[0]*norm;
     cout<<PA[1]<<"  "<<PA[0]<<"A,b=0,c=0"<<endl;
    return bernouli(PA[1]);



}
}

 int beliefNet::proB(int a,int c)

 {
 float PB[2];
   double norm;
  if(a==0&&c==1)
    {
    PB[0]=(1-A[1].pro[1])*(1-A[3].pro[0]);
    PB[1]=A[1].pro[1]*(1-A[3].pro[0]);
    norm=1/(PB[0]+PB[1]);
    PB[0]=norm*PB[0];
    PB[1]=norm*PB[1];
    cout<<PB[1]<<" "<<PB[0]<<"B,a=0,c=1"<<endl;
    return bernouli(PB[1]);
   }
   if(a==1&&c==0)
   {
    PB[0]=(1-A[1].pro[0])*(1-A[3].pro[1]);
    PB[1]=A[1].pro[0]*(1-A[3].pro[0]);
```

```
    norm=1/(PB[0]+PB[1]);
 PB[0]=norm*PB[0];
    PB[1]=norm*PB[1];
       cout<<PB[1]<<" "<<PB[0]<<"B,a=1,c=0"<<endl;
    return bernouli(PB[1]);
}


  if(a==0&&c==0)
{
    PB[0]=(1-A[1].pro[1])*(1-A[3].pro[3]);
    PB[1]=A[1].pro[1]*(1-A[3].pro[2]);
    norm=1/(PB[0]+PB[1]);
    PB[0]=norm*PB[0];
    PB[1]=norm*PB[1];
        cout<<PB[1]<<" "<<PB[0]<<"B,a=0,c=0"<<endl;
    return bernouli(PB[1]);

}
  if(a==0&&c==1)
{

    PB[0]=(1-A[1].pro[1])*(1-A[3].pro[1]);
    PB[1]=A[1].pro[1]*(1-A[3].pro[0]);
    norm=1/(PB[0]+PB[1]);
    PB[0]=norm*PB[0];
    PB[1]=norm*PB[1];
     cout<<PB[1]<<" "<<PB[0]<<"B,a=0,c=1"<<endl;
    return bernouli(PB[1]);
}
}

int beliefNet::proC(int a, int b)
{
  double norm;
 float PC[2];
 if(a==1&&b==0)
{
```

```
  PC[0]=(1-A[2].pro[0])*(1-A[3].pro[3])*A[4].pro[1];
  PC[1]=A[2].pro[0]*(1-A[3].pro[1])*A[a].pro[0];
  norm=1/(PC[0]+PC[1]);
  PC[0]=norm*PC[0];
  PC[1]=norm*PC[1];
    cout<<PC[1]<<" "<<PC[0]<<"C,a=1,b=0"<<endl;
  return bernouli(PC[1]);

}
 if(a==1&&b==1)
{
  PC[0]=(1-A[2].pro[0])*(1-A[3].pro[2])*A[4].pro[1];
  PC[1]=A[2].pro[0]*(1-A[3].pro[0])*A[4].pro[0];
  norm=1/(PC[0]+PC[1]);
 PC[0]=norm*PC[0];
  PC[1]=norm*PC[1];
   cout<<PC[1]<<" "<<PC[0]<<"C,a=1,b=1"<<endl;
  return bernouli (PC[1]);
}
if(a==0&&b==0)
{
  PC[0]=(1-A[2].pro[1])*(1-A[3].pro[3])*A[4].pro[1];
  PC[1]=A[2].pro[1]*(1-A[3].pro[1])*A[4].pro[0];
  norm=1/(PC[0]+PC[1]);
  PC[0]=norm*PC[0];
  PC[1]=norm*PC[1];
    cout<<PC[1]<<" "<<PC[0]<<"C,a=0,b=0"<<endl;
  return bernouli(PC[1]);

}
if(a==0&&b==1)
{
 PC[0]=(1-A[2].pro[1])*(1-A[3].pro[2])*A[4].pro[1];
  PC[1]=A[2].pro[1]*(1-A[3].pro[0])*A[4].pro[0];
  norm=1/(PC[0]+PC[1]);
  PC[0]=norm*PC[0];
  PC[1]=norm*PC[1];
    cout<<PC[1]<<" "<<PC[0]<<"C,a=0,b=1"<<endl;
```

```
  return bernouli(PC[1]);



}
}

 void beliefNet::blanketPro(int a,int b,int c)
 {
    int valueA;
int valueB;
    int valueC;
    valueA=proA(b,c);
    valueB=proB(valueA,c);
    valueC=proC(valueA,valueB);
    a=valueA;
    b=valueB;
    c=valueC;
}
```

structbn.h

```
struct nodeInfo {

 int parentNum;
 int parentOfver[2];
 float pro[4];
 };
```