Dan Scarafoni

ID- 27435144

Project 3

CSC242- Introduction to AI

Inference in Bayesian Networks in Java

Introduction

In variables in which relationships and outcomes are uncertain, predicting future events is a difficult endeavor. Relationships between variables can be complicated to the point of intractability in sufficiently large worlds. This is problematic because real world situations almost always involve degrees of improbability and unpredictability. Developing algorithms to help solve problems of this nature is essential to creating AI that can hope to function in real-world scenarios. Our classwork with Bayesian networks provides a framework for representing the sample space of a given problem which features probabilistic relationships between a set of variables. Within a Bayesian network, several key algorithms (including enumeration-ask and gibbs sampling) can be used to predict the potential values of a variable given observed phenomena in an environment.

Algorithms

A Bayesian network is a directed graph depicting the causal relationships between variables. The nodes of the graph are variables in the problem which can have one of several potential values (known as random variables). The connections in the graph are directed, and a connection from node A to node B represents that node A's state influences the state of node B (the two variables are said to be conditionally dependent). Figure 1 illustrates a simple Bayesian network used in experimentation for this project.

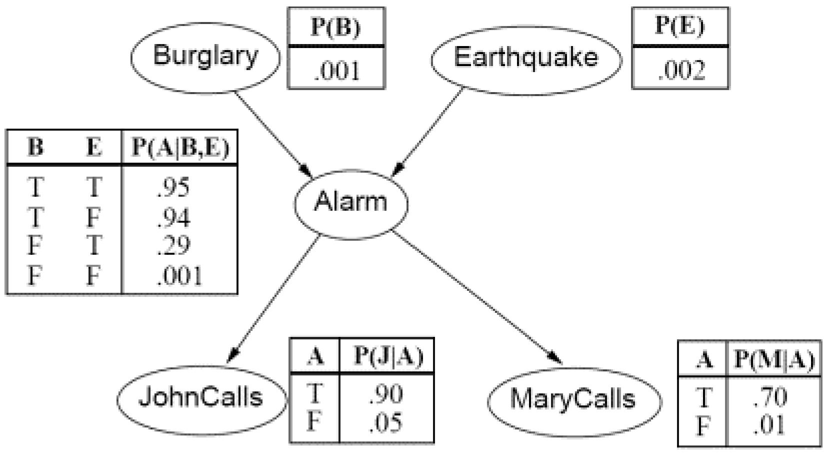


Figure 1- a Bayesian network

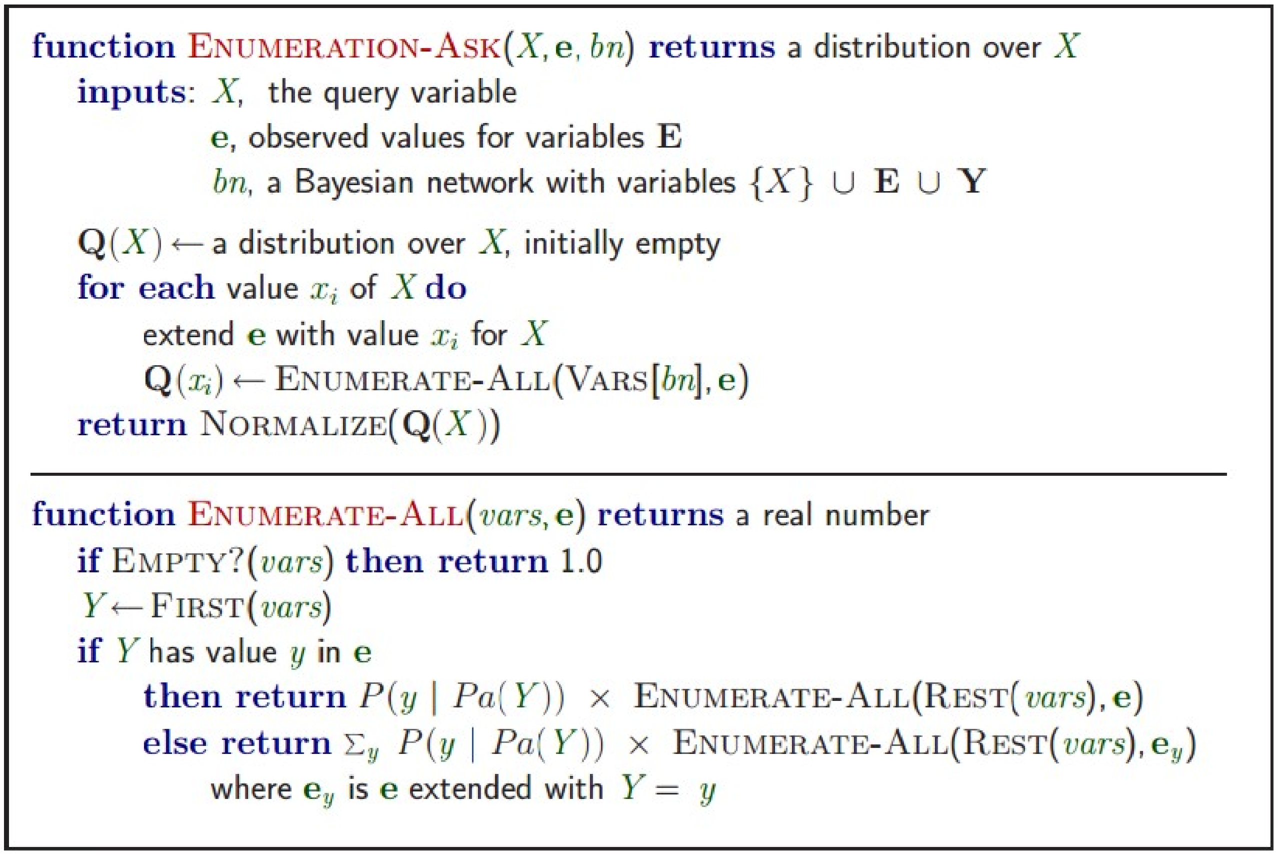
Note that Figure 1 also contains the conditional probability tables (CPT) for each variable; these represent the probable values of each node given the state of parent nodes.

The project required that queries to a given Bayesian network be calculated and answered. The notation to be used for these queries will be the same as used in the text book, repeated in Figure 2.

$$\boldsymbol{P}(X|E)$$

Figure 2: the probability of each value of variable X given evidence E

A number of techniques were usable in order to solve queries of this nature. The first to be implemented was enumeration-ask, which is shown in Figure 3.



```
function ENUMERATION-ASK($X$, $\mathbf{e}$, $bn$) returns a distribution over $X$
    inputs: $X$, the query variable
            $\mathbf{e}$, observed values for variables $\mathbf{E}$
            $bn$, a Bayesian network with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$

    $\mathbf{Q}(X) \leftarrow$ a distribution over $X$, initially empty
    for each value $x_i$ of $X$ do
        extend $\mathbf{e}$ with value $x_i$ for $X$
        $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL(VARS[$bn$], $\mathbf{e}$)
    return NORMALIZE($\mathbf{Q}(X)$)
```

```
function ENUMERATE-ALL($vars$, $\mathbf{e}$) returns a real number
    if EMPTY?($vars$) then return 1.0
    $Y \leftarrow$ FIRST($vars$)
    if $Y$ has value $y$ in $\mathbf{e}$
        then return $P(y \mid Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}$)
        else return $\Sigma_y\ P(y \mid Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_y$)
            where $\mathbf{e}_y$ is $\mathbf{e}$ extended with $Y = y$
```

Figure 3: enumeration-ask

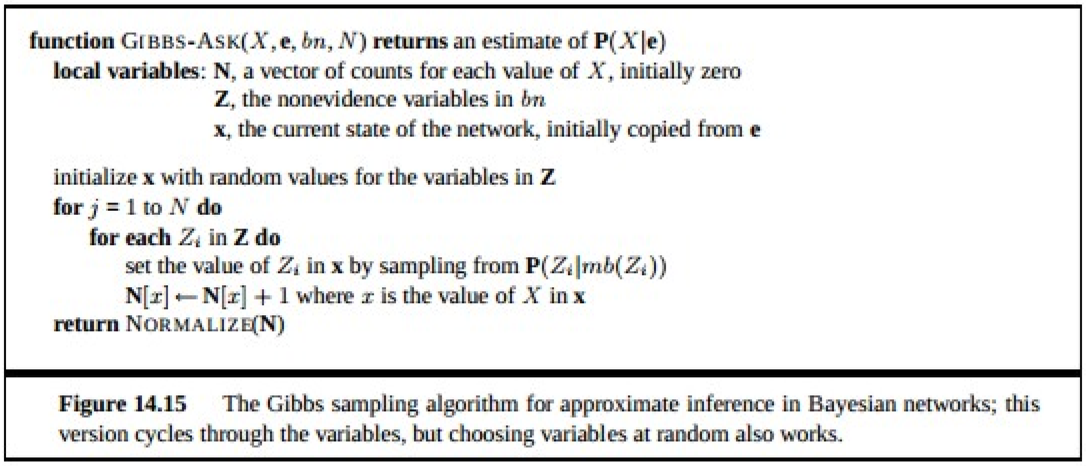
This algorithm solves the problem by enumerating through all unused (non-evidence and non-query) variables, enumerating all possible worlds (given the evidence). In each world, the program calculates the probability of that world's existence (full-joint distribution). The possibilities of each full-joint distribution are summed. This is repeated for all possible values of the query variable. The probability vector is then normalized such that the total probability of all variables sums to 1. Mathematically, the probability of a given variable can be represented a series of sums of products of network probabilities. This is represented in Figure 4.

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \sum_{\mathbf{y}} \prod_{i=1}^{n} P(x_i \mid parents(X_i))$$

Figure 4

This algorithm guarantees that an exact probability inference is found; however, the algorithm is complex to the point of intractability, running in $O(n2^n)$. This is simply unacceptable, particularly considering that the entire goal of probabilistic inference is to aid in solving real-world situations. As such, a trade off of accuracy in exchange for improved run-time would be acceptable.

Approximate inference algorithms provide this trade-off. The first algorithm looked at was Gibbs sampling. This algorithm, the pseudo-code of which is outlined in figure 5,



```
function GIBBS-ASK($X$, e, $bn$, $N$) returns an estimate of $\mathbf{P}(X|\mathbf{e})$
  local variables: N, a vector of counts for each value of $X$, initially zero
                   Z, the nonevidence variables in $bn$
                   x, the current state of the network, initially copied from e

  initialize x with random values for the variables in Z
  for $j$ = 1 to $N$ do
      for each $Z_i$ in Z do
          set the value of $Z_i$ in x by sampling from $\mathbf{P}(Z_i|mb(Z_i))$
          $\mathbf{N}[x] \leftarrow \mathbf{N}[x] + 1$ where $x$ is the value of $X$ in x
  return NORMALIZE(N)
```

Figure 14.15 The Gibbs sampling algorithm for approximate inference in Bayesian networks; this version cycles through the variables, but choosing variables at random also works.

Figure 5- Gibbs-Ask algorith

This algorithm is a Markov Chain Monte Carlo algorithm (MCMC), and is a form of local search not dissimilar to the hill-climbing algorithm sometimes implemented to solve the n-queens problem. The algorithm essentially assigns all non-evidence variables to arbitrary values, and then reassigns variables one at a time based on which value is most likely given the state of it's Markov blanket (to be explained soon). In effect, the algorithm “wanders” through state space, with a probability of transitioning from one state space to another determined by the probability of the next state. If the values of the query variable are noted on each step, the ratio of each of the observed values will closely match the expected probability of each state (thus giving a consistent answer). The sampling algorithm described above determines the probability of selecting the next assignment for a variable based on the probability of that variable (given it's parents) multiplied by the probability of each of its children multiplied by their respective parents. This set of a node, it's parents, children, and parents of those children is known as a Markov Blanket.

Though not as sophisticated as gibbs-sampling, rejection sampling and the slightly more complicated likelihood-weighting algorithms were also used in order to asses their advantages, disadvantages, and overall utility when compared to gibbs-sampling and exact-inference.

Rejection sort works by reading in all nodes in a Bayesian network in topological order and assigning a value to each one based on the probability of each assignment given the parents (much like in gibbs-sampling). If this world produces a state that contradicts the evidence, it is rejected. Otherwise, the value of the query variable is noted and another full-disjoint is chosen. At the end of a number of iterations, the vector which held the number of occurrences of each value of the query variable is normalized and returned.

This algorithm was simple to implement and did not require much code. However, it is very inefficient in that is rejects a very large number of samples. The program would create hundreds of enumerations only to throw out the results produced, and it would throw out even more as more evidence was added to the network.

Likelihood-weighting reduced this problem. Like rejection sampling, the algorithm enumerated possible worlds via sampling. However, it only enumerated possibilities which were consistent with the evidence, and assigned a weight to each result based on the evidence.

Implementation

Java was selected as the language of choice for this project. This program made use of many complex data structures (most notably array lists) for which java has a myriad of built-in classes. This allowed for easy implementation of the various methods and classes needed.

Although sample code was given for the problem, only the parsing class (for parsing in

Bayesian networks written in xml format) was utilized. A heavily edited topological sort algorithm from the code was also utilized for rejection-sampling likelihood-weighting. All other code was built from scratch. Classes were programmed for the basic components the Bayesian networks- random variables had their own class. CPT's were represented as trees, each layer of which represented all the possible values of a given parent variable. The “leaves” of the tree were the probabilities the CPT's random variable given the value of parent nodes. The data structure is illustrated in Figure 6.

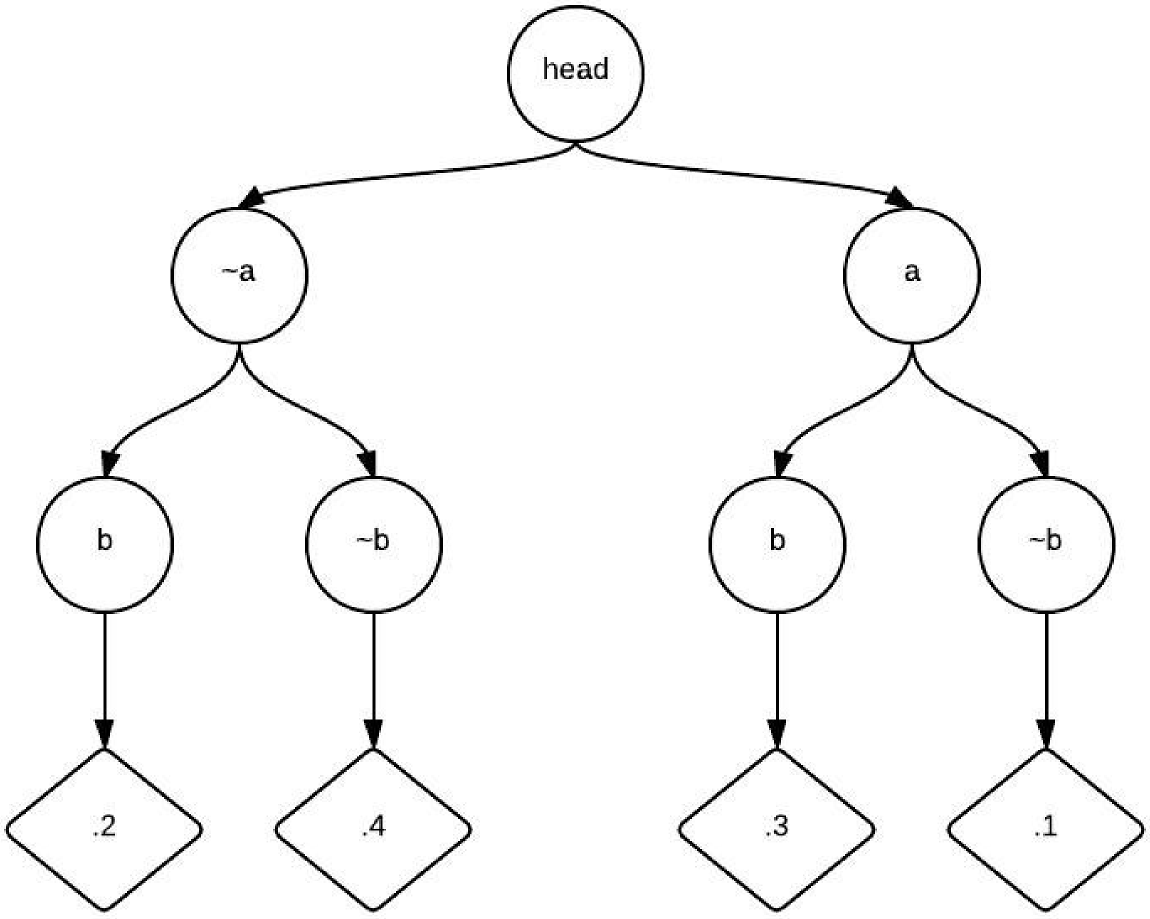


Figure 6: a CPT for a theoretical random variable A with parent B

A wrapper class for the random variable (which would contain the variables CPT and a list of parents) named Bnode was also implemented to safely house all important data to a given variable. A list of Bnodes was kept in the class BayesianNetwork, along with a number of accessing methods. All driver programs would specify a query given evidence and an inference algorithm to be used. All driver programs were subclasses of the class Inference. Figure 7 shows the general class hierarchy for the project.

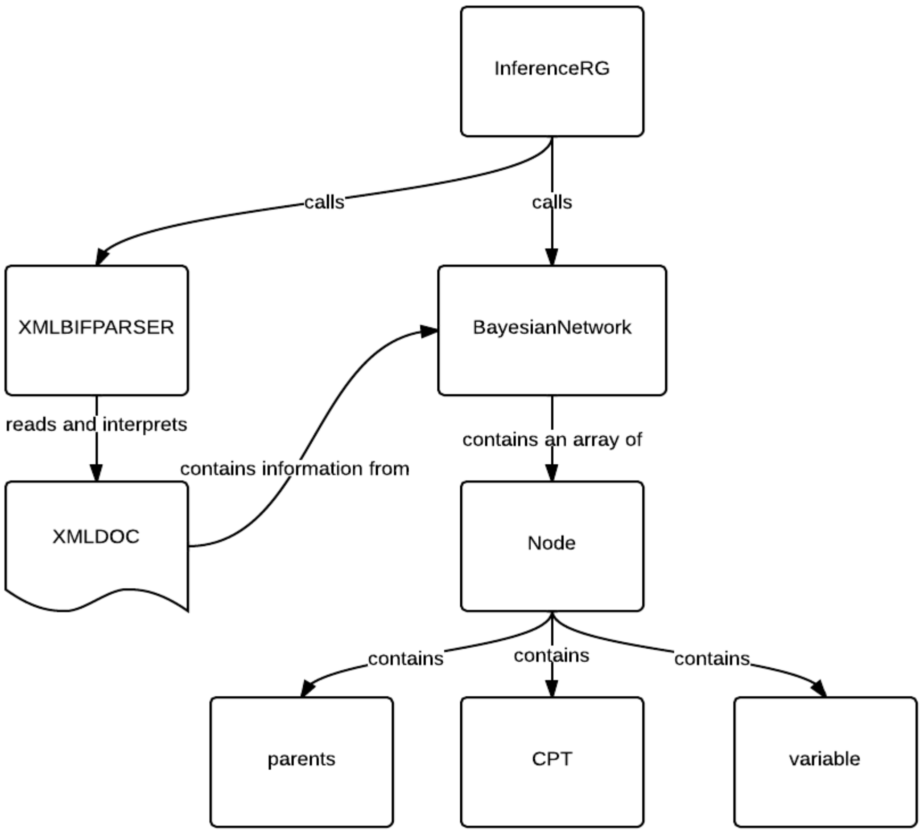


Figure 7

The exact Inference algorithm implemented utilized the pseudo-code found in the book (shown above in Figure 3). The arbitrary series of sums in the formula was implemented recursively using a tree, each layer of which represented the values of a given unused variable. Each leaf on the tree was a full disjoint of the problem. Full disjoints were calculated by multiplying the probability of each variable (which was found via the variable's CPT). This data implementation is not dissimilar to the implementation of the CPT tree, and as figure 8 (which represents a disjoint-tree) shows, the two share many common characteristics.

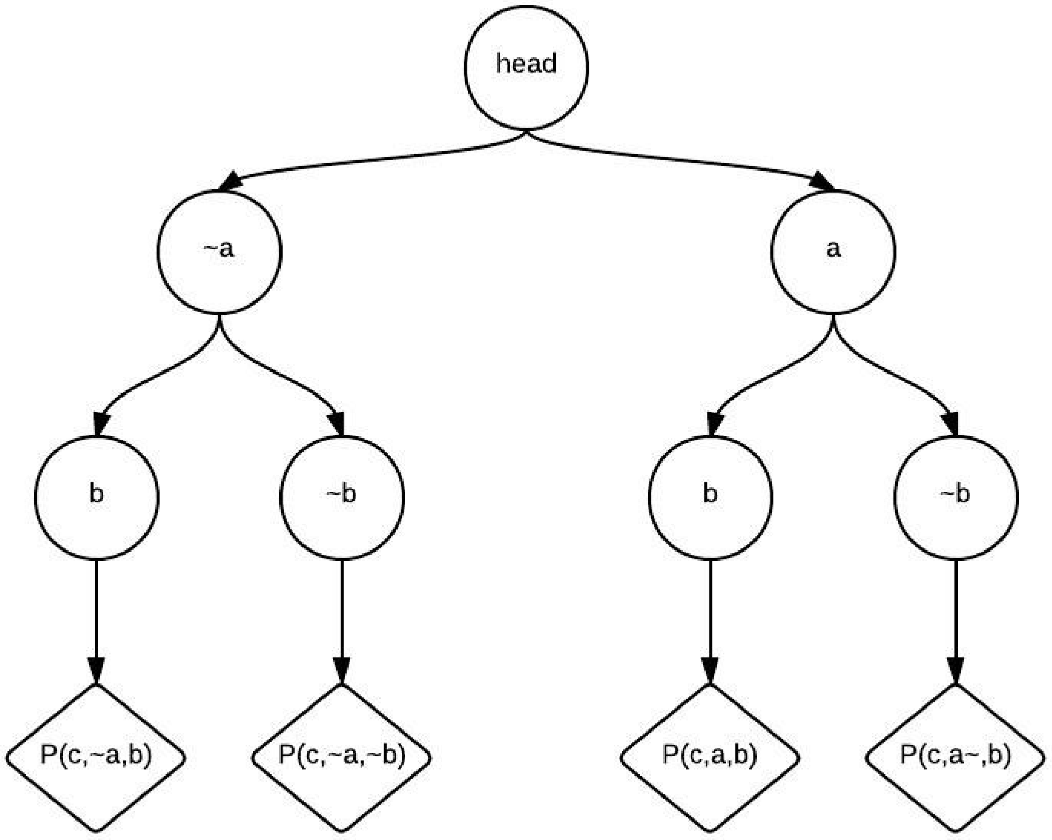
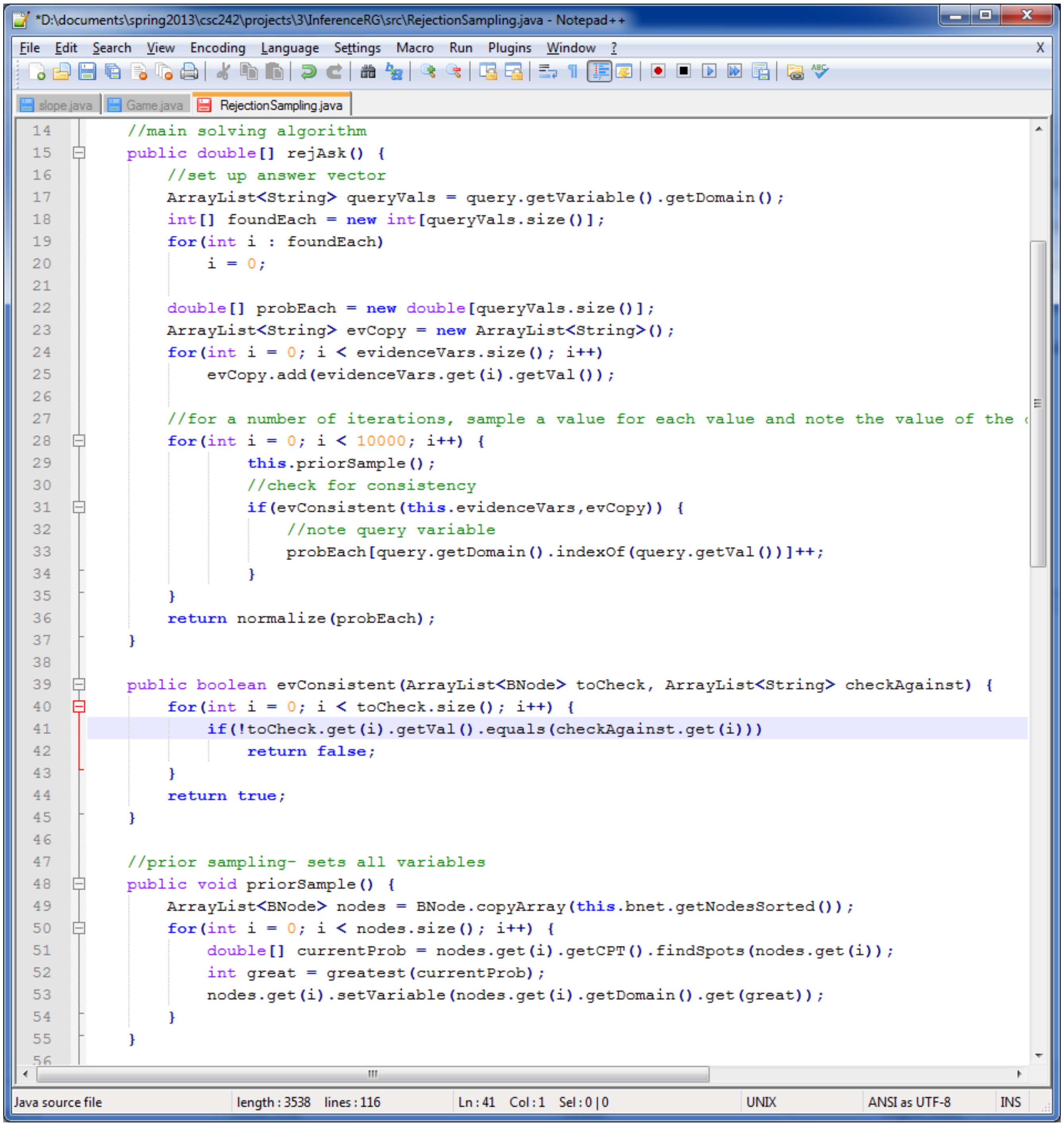


Figure 8- disjoint-tree for P(c) given no evidence

The Gibbs-Ask algorithm differed considerably. The algorithm first gathered all non-evidence variables, then iterated through each one individually. Each iteration sample a value for the variable given the Markov Blanket (as stated above). The value of the query variable was then noted in a vector which kept a running tally of the number of times each value was encountered. The ratio between the end values after a given number of iterations through all variables (typically 1000) was then normalized to give an approximation of the answer.

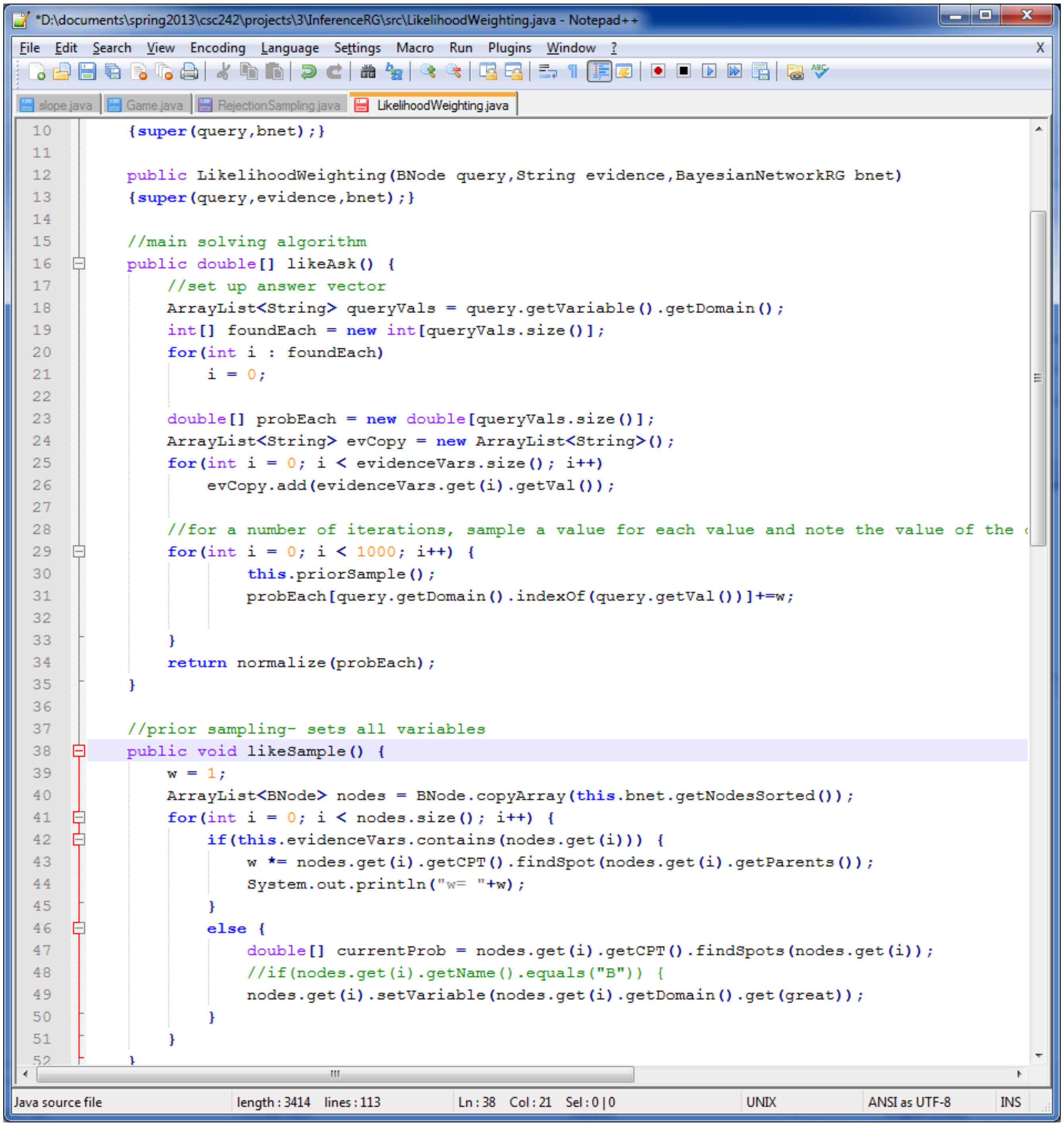
Rejection sampling used a much simpler implementation. No recursion was needed, only three methods were written. RejAsk was used to gather probabilistic values for the query based on worlds enumerated, and priorSample() was used to enumerate possible worlds. EvConsistent was also used in order to determine if an enumerated disjoint was consistent given the evidence. Figure 9 shows a screen capture of the source code.


*D:\documents\spring2013\csc242\projects\3\InferenceRG\src\RejectionSampling.java - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
slope.java
Game.java
RejectionSampling.java
Java source file
length : 3538 lines : 116
Ln : 41 Col : 1 Sel : 0 | 0
UNIX
ANSI as UTF-8
INS

```java
    //main solving algorithm
    public double[] rejAsk() {
        //set up answer vector
        ArrayList<String> queryVals = query.getVariable().getDomain();
        int[] foundEach = new int[queryVals.size()];
        for(int i : foundEach)
            i = 0;

        double[] probEach = new double[queryVals.size()];
        ArrayList<String> evCopy = new ArrayList<String>();
        for(int i = 0; i < evidenceVars.size(); i++)
            evCopy.add(evidenceVars.get(i).getVal());

        //for a number of iterations, sample a value for each value and note the value of the 
        for(int i = 0; i < 10000; i++) {
                this.priorSample();
                //check for consistency
                if(evConsistent(this.evidenceVars,evCopy)) {
                    //note query variable
                    probEach[query.getDomain().indexOf(query.getVal())]++;
                }
        }
        return normalize(probEach);
    }

    public boolean evConsistent(ArrayList<BNode> toCheck, ArrayList<String> checkAgainst) {
        for(int i = 0; i < toCheck.size(); i++) {
            if(!toCheck.get(i).getVal().equals(checkAgainst.get(i)))
                return false;
        }
        return true;
    }

    //prior sampling- sets all variables
    public void priorSample() {
        ArrayList<BNode> nodes = BNode.copyArray(this.bnet.getNodesSorted());
        for(int i = 0; i < nodes.size(); i++) {
            double[] currentProb = nodes.get(i).getCPT().findSpots(nodes.get(i));
            int great = greatest(currentProb);
            nodes.get(i).setVariable(nodes.get(i).getDomain().get(great));
        }
    }
```

Figure 9

The algorithm for likelihood-weighting was constructed in a very similar manner, but without the evidence checking method (since no generated states were ruled out. Figure 10 shows the source code for this algorithm. The value for the weight of a variable was stored as a global variable which was initialized offscreen.


*D:\documents\spring2013\csc242\projects\3\InferenceRG\src\LikelihoodWeighting.java - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
slope.java
Game.java
RejectionSampling.java
LikelihoodWeighting.java
Java source file
length : 3414 lines : 113
Ln : 38 Col : 21 Sel : 0 | 0
UNIX
ANSI as UTF-8
INS

```java
    {super(query,bnet);}

    public LikelihoodWeighting(BNode query,String evidence,BayesianNetworkRG bnet)
    {super(query,evidence,bnet);}

    //main solving algorithm
    public double[] likeAsk() {
        //set up answer vector
        ArrayList<String> queryVals = query.getVariable().getDomain();
        int[] foundEach = new int[queryVals.size()];
        for(int i : foundEach)
            i = 0;

        double[] probEach = new double[queryVals.size()];
        ArrayList<String> evCopy = new ArrayList<String>();
        for(int i = 0; i < evidenceVars.size(); i++)
            evCopy.add(evidenceVars.get(i).getVal());

        //for a number of iterations, sample a value for each value and note the value of the
        for(int i = 0; i < 1000; i++) {
                this.priorSample();
                probEach[query.getDomain().indexOf(query.getVal())]+=w;

        }
        return normalize(probEach);
    }

    //prior sampling- sets all variables
    public void likeSample() {
        w = 1;
        ArrayList<BNode> nodes = BNode.copyArray(this.bnet.getNodesSorted());
        for(int i = 0; i < nodes.size(); i++) {
            if(this.evidenceVars.contains(nodes.get(i))) {
                w *= nodes.get(i).getCPT().findSpot(nodes.get(i).getParents());
                System.out.println("w= "+w);
            }
            else {
                double[] currentProb = nodes.get(i).getCPT().findSpots(nodes.get(i));
                //if(nodes.get(i).getName().equals("B")) {
                nodes.get(i).setVariable(nodes.get(i).getDomain().get(great));
            }
        }
    }
```

Figure 10

Results

The first tests run focused on whether or not there was a correlation between the run-time of the program with respect to the amount of evidence given in the query. Although the random algorithms do not alter the amount of calculations they do based on the amount of evidence, enumeration-ask does. Thus, if any changes were to be found, they would most likely be found on this algorithm. The following graphs (Figures 11 through 13) show the results of this experiment for various algorithms. Note that the problem used was the aima-alarm.xml for each one. Note also that the following are the averages after ten runs each.

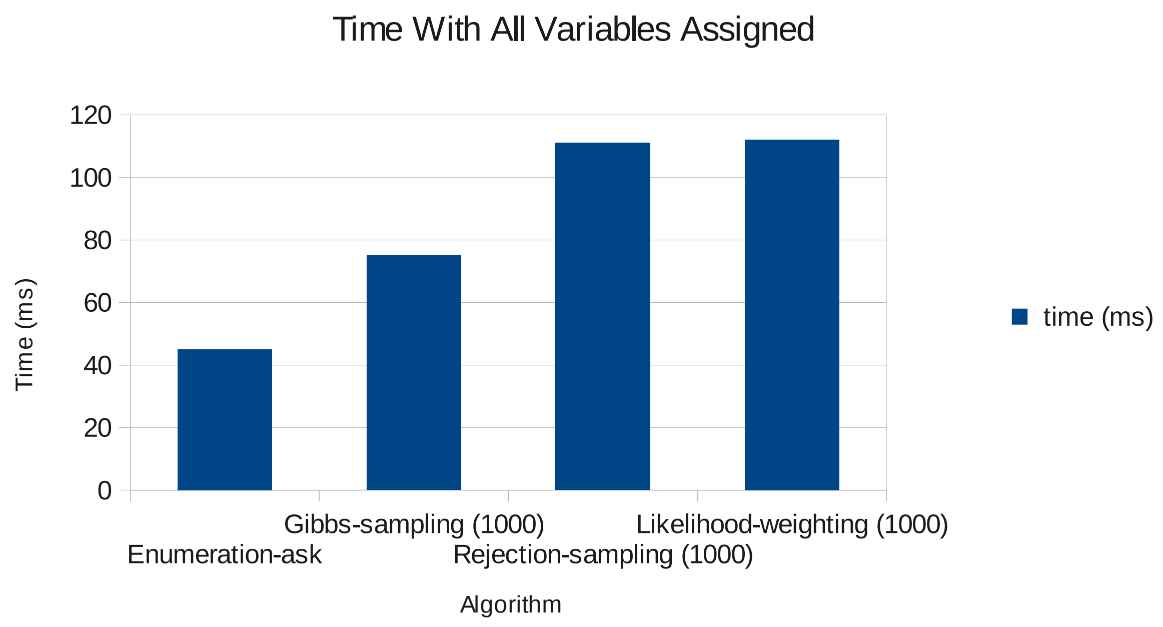


Figure 11

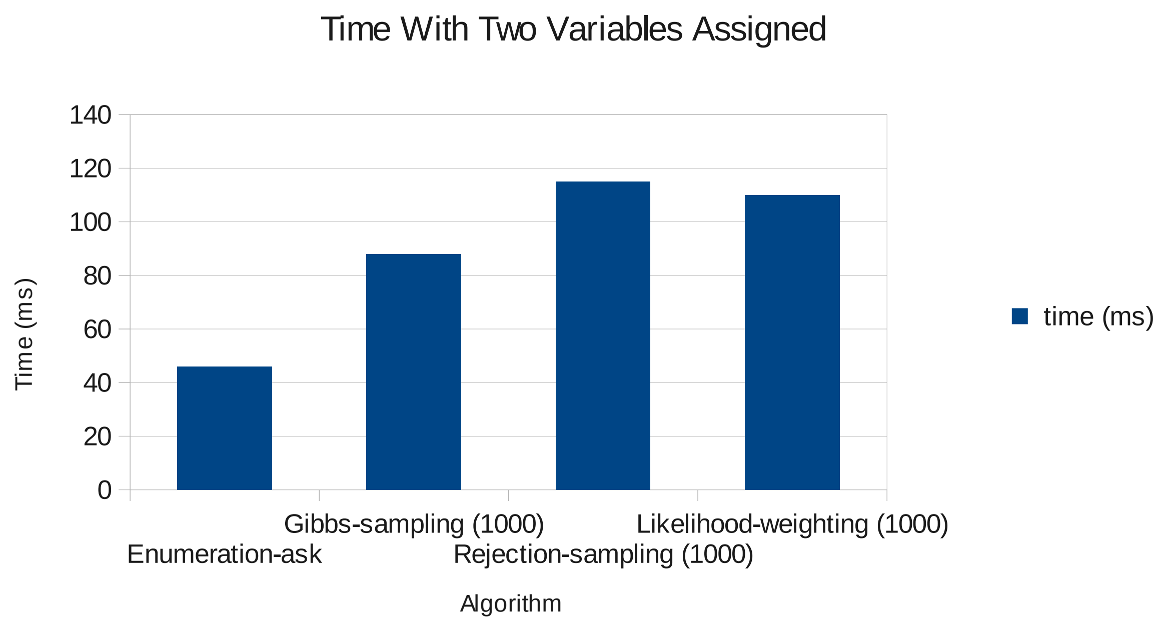


Figure 12

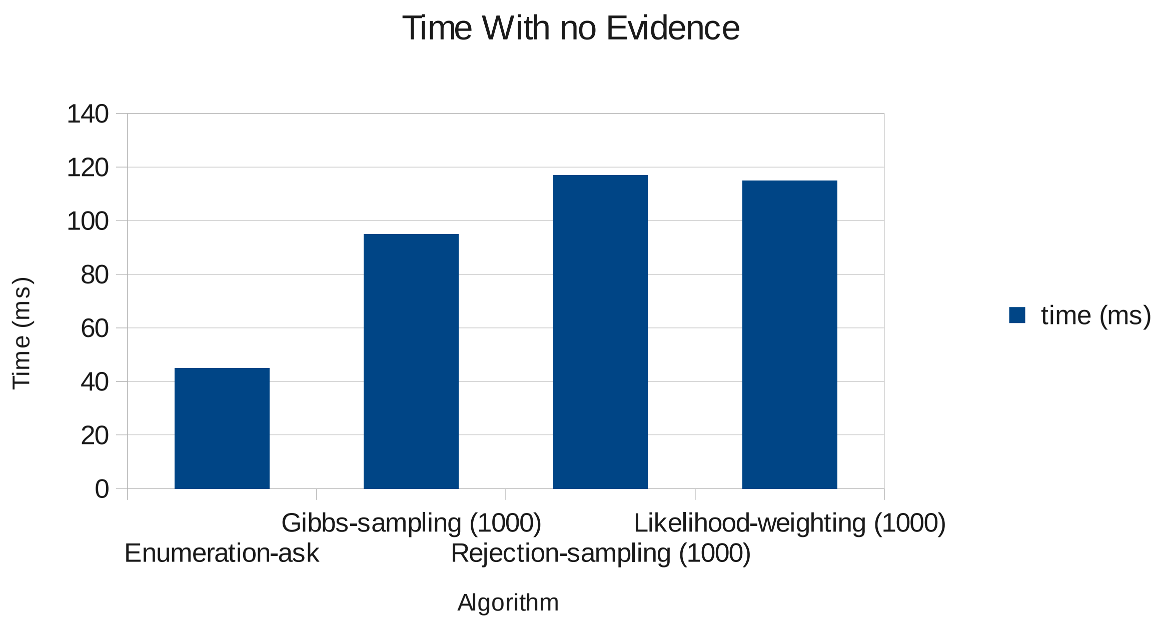


Figure 13

As us evident from above, the amount of evidence provided did not have any noticeable effect on the run-time of any of the programs, although it did seem to make likelihood-weighting and rejection-sampling take slightly longer with less evidence. Although enumeration-ask does take different actions depending on whether or not a variable is set, this does not seem to effect the run-time.

The next series of tests were done on the approximate inference algorithms to approximate the standard deviation of their responses as the increase number of iterations through their variables were increased. For this experiment, the aima-alarm.xml file was used, and the query **P**(B|M=true,J=true). The first value in the returned probability vector was returned for each test run, and ten total test runs were done. Figures 14 through 16 show the results for each algorithm.

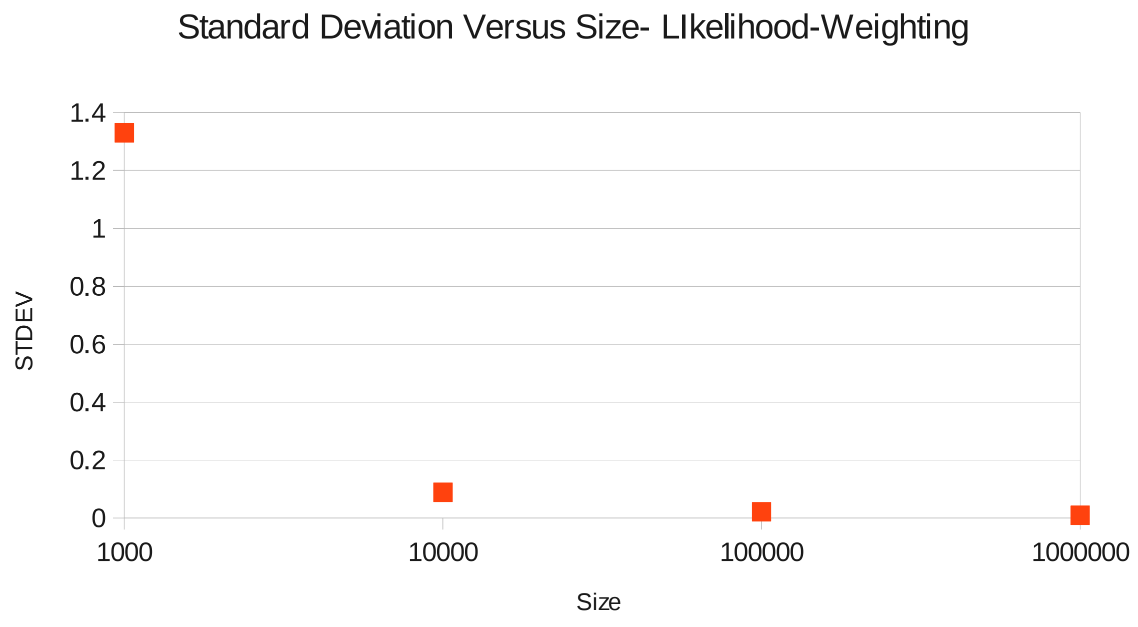


Figure 14

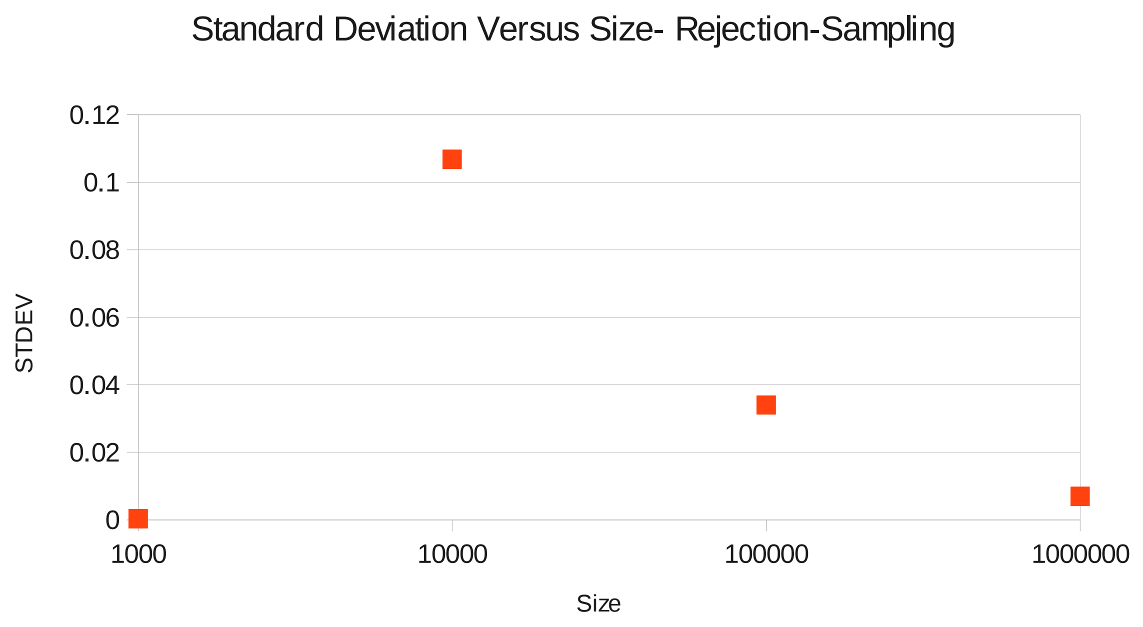


Figure 15

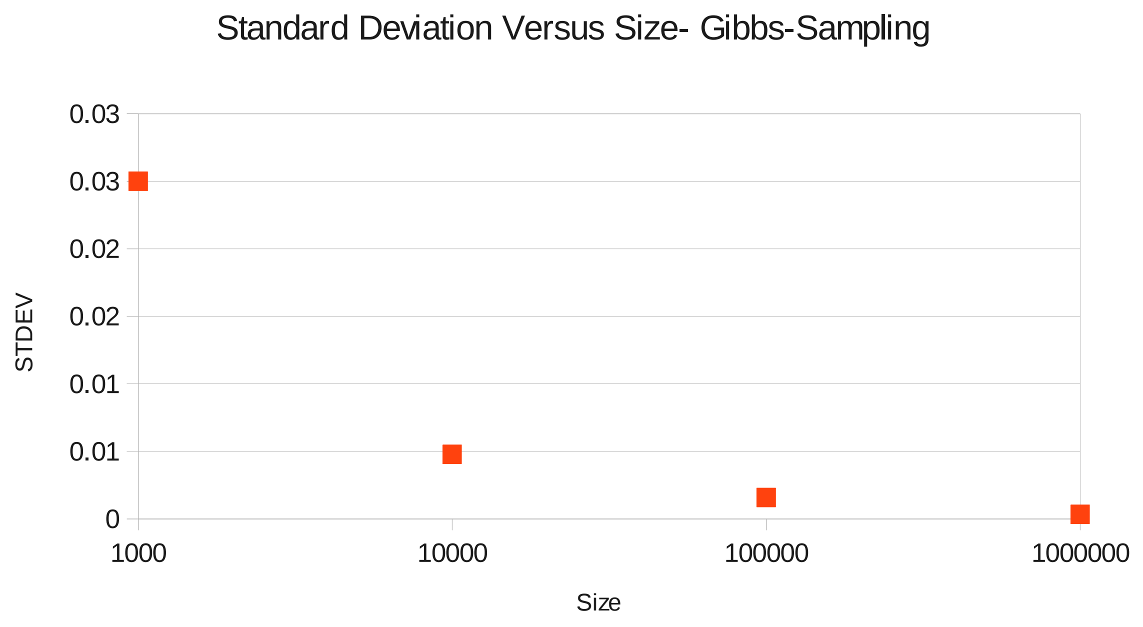


Figure 16

The graphs above provide interesting data. It would seem that the standard deviation of each reaches zero as the number of iterations becomes large. Further, it would seem that likelihood-weighting outperforms rejection-sampling, and that gibbs-sampling outperforms both. This is expected. The only bit of data that stands out is the first measurement of rejection-sampling. It shows a very low standard deviation at size = 1000. This is, however, most likely due to a measurement anomaly. Rejection sampling rejected so man samples at this level that the vector returned consisted of zero to one variables, and as such consistently (and erroneously) produced a probability of 0.0.

The last series of tests performed were on the run-time of the approximate inference algorithms. Their run-time was measured against the number of iterations through each variable. Figures 17 through 19 show the results. The same problem and query were used as in the other experiments.

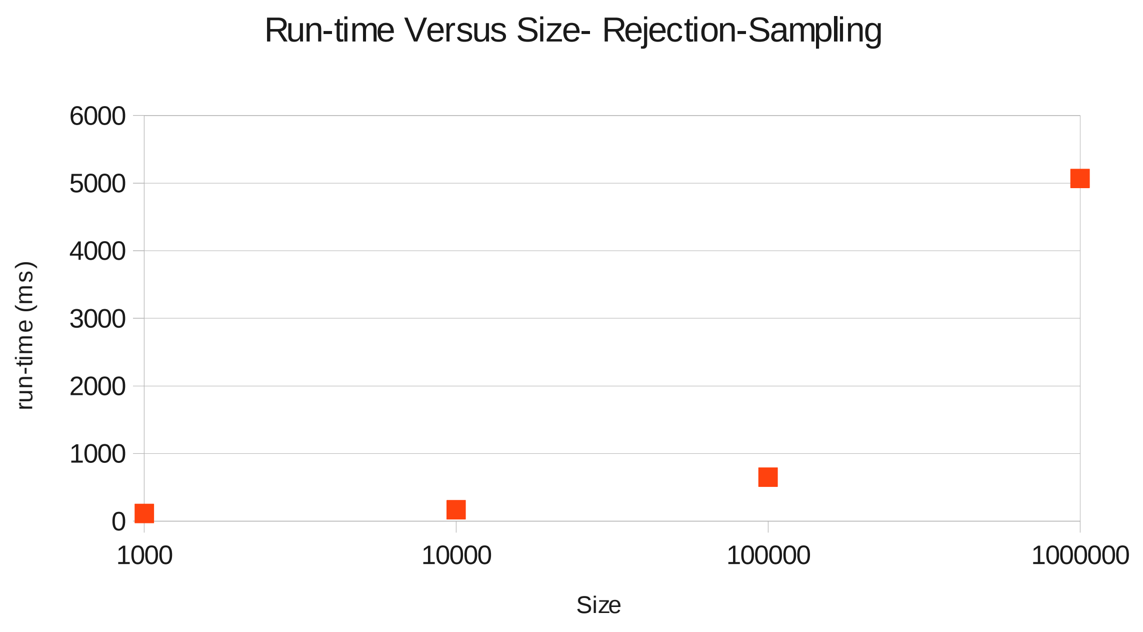


Figure 17

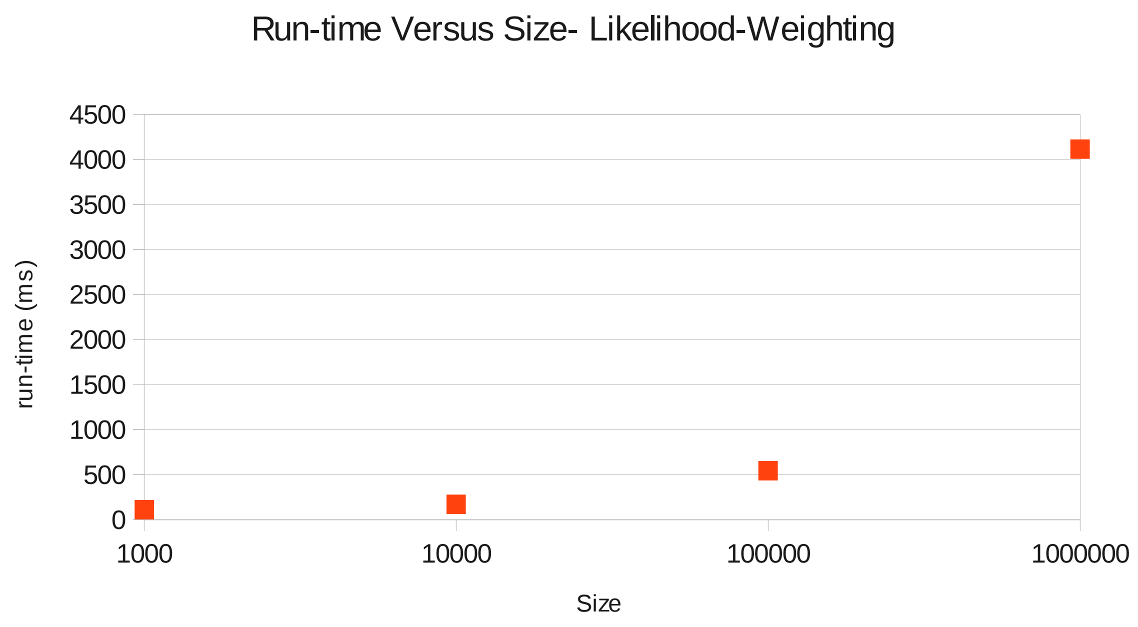


Figure 18

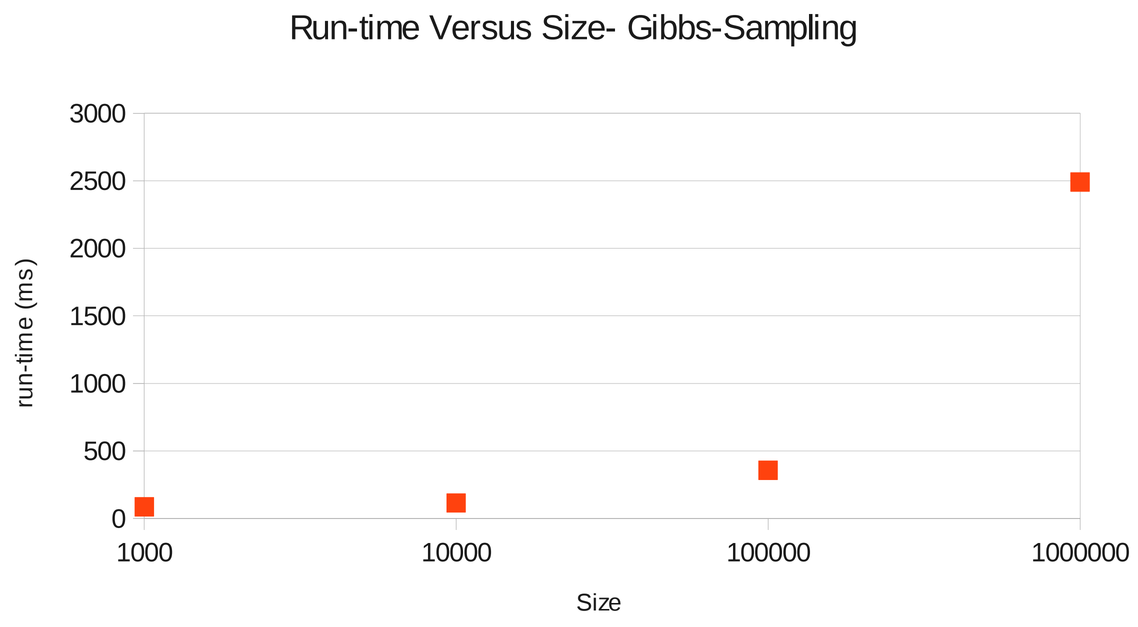


Figure 19

The trends shown in the above graphs reflect the trend demonstrated in the measurements of standard deviation- namely that likelihood-weighting is an improvement on rejection-sampling, and that gibbs-sampling is an improvement on both.

Conclusions and Discussion

This experiment tested a number of different algorithms for solving inference in Bayesian networks. The results shown above gave great insight into the efficiency of various methods of computation. From the above, it can be concluded that gibbs-sampling is by far the best choice for approximate inference, with likelihood-weighting coming second and rejection-sampling being the least preferable (as expected). One of the great drawbacks of the code used was that it was only able to parse .xml representations networks. Because of this, it was not able to test BIF files that contained much larger networks. Computational experiments on large chunks of information was therefore not done, and as such there was not much to test on the exact inference algorithm, enumeration-ask (since it does not run through multiple iterations the impact of increased iterations could be tested on neither run-time nor standard deviation).Any future experiments on these algorithms should probably begin by testing of enumeration-ask with large data sets. There should also be some further research done on why rejection-sampling and likelihood-weighting apparently take slightly more time with less evidence.

Further, there was not enough time to develop and run other exact-inference algorithms. Despite the fact that the text book names other algorithms (such as elimination-ask) non besides enumeration-ask were implemented. Future experiments should also note and compare these other algorithms with respect to the results of enumeration-ask demonstrated here.