Project 3

Uncertain Inference

Hao Huang (NetID: hhuang40)

April 5, 2018

1. Exact Inference

In this section, we will present the main idea of *enumeration algorithm* (`ExactInferencer.ask()`) on Bayesian networks and several key points in implementing this algorithm.

1.1 Main Idea of Enumeration Algorithm

Any conditional probability can be computed by summing terms from the full joint distribution, which is shown in Equation 1.1:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \tag{1.1}$$

Given a Bayesian network, the joint distribution on the right-hand side in Equation 1.1 can be written as products of conditional probabilities from the network using Equation 1.2:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i)) \tag{1.2}$$

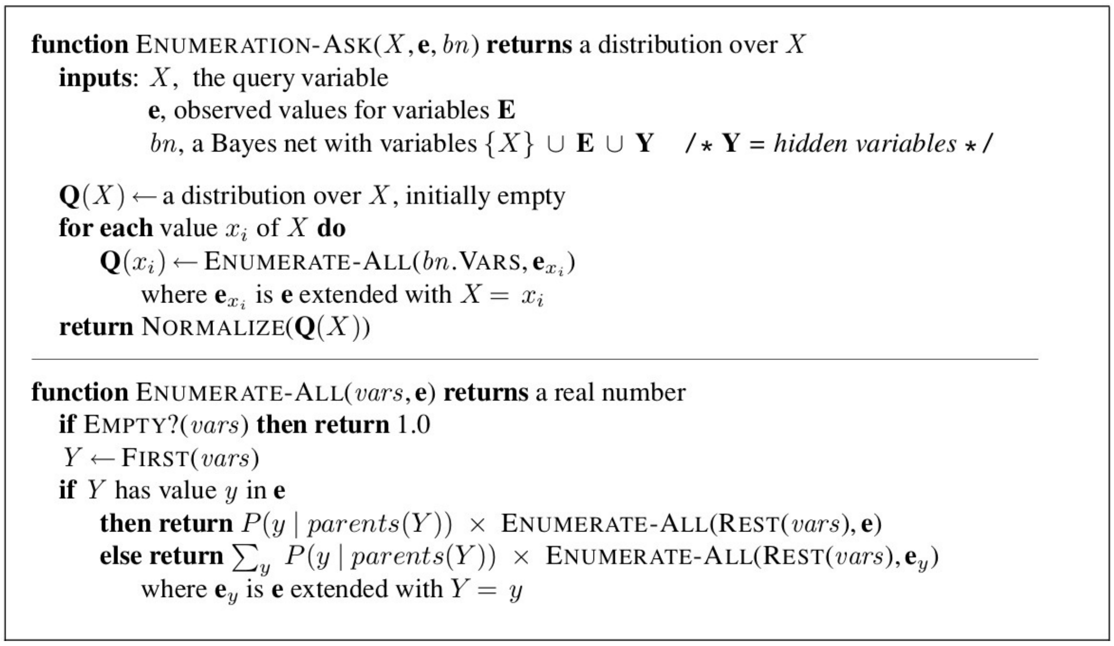
Therefore, a query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network. More specifically, if we plug Equation 1.2 into Equation 1.1, we have Equation 1.3:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \sum_{\mathbf{y}} \prod_{i=1}^{n} P(x_i | parents(X_i)) \tag{1.3}$$

where $\mathbf{y}$ are all hidden variables.

Enumeration algorithm computes the distribution of a query variable X given evidences $\mathbf{e}$ encoded in a Bayesian Network. For each variable of X, we compute products of all conditional probabilities stored in the Bayesian Network. If a variable Y is observed as an evidence, we just multiply the probability $P(y|parents(Y))$; otherwise, Y is a hidden variable, and we need to sum over all its possible values: $\sum_y P(y|parents(y))$. The multiplication process is a recursive process until all variables in the network have been touched. The pseudocode is shown in Figure 1.

The *for* loop in `ENUMERATION-ASK()` loops all possible values of the query variable X, since we hope to compute a probability distribution of X. For each possible value of X, X will be considered as an observed variable and added to evidences E. In the loop, a recursive function `ENUMERATE-ALL()` will be called. During each recursive process, the conditional probability of one variable in the network is calculated. If the variable is an evidence, e.g., having a value in e, its conditional probability is multiplied directly; otherwise, it is a hidden variable, and a summation will be computed over all its possible values. The corresponding pseudocode is the `if-then-else` block in Figure 1.



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

   $\mathbf{Q}(X) \leftarrow$ a distribution over $X$, initially empty
   for each value $x_i$ of $X$ do
      $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL($bn$.VARS, $\mathbf{e}_{x_i}$)
         where $\mathbf{e}_{x_i}$ is e extended with $X = x_i$
   return NORMALIZE($\mathbf{Q}(X)$)
```

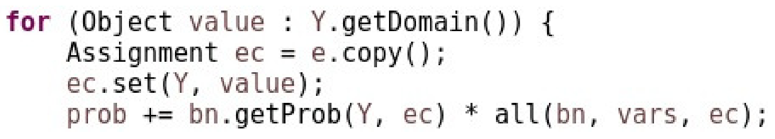
```
function ENUMERATE-ALL($vars$, e) returns a real number
   if EMPTY?($vars$) then return 1.0
   $Y \leftarrow$ FIRST($vars$)
   if $Y$ has value $y$ in e
      then return $P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), e)
      else return $\sum_y P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$),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

Figure 1: Pseudocode of enumeration algorithm

1.2 Kep Points in Implementation

To implement this algorithm, we add one class, `ExactInferencer`, which implements the interface `Inferencer` provided by the framework.

First of all, for each value of the query variable X, we add it to an `Assignment`. Then we call `all()` function to compute its probability. One thing to notice is that for each value of a hidden variable Y, we need to copy an assignment to avoid overwritten. The conditional probability, $P(y|parents(Y))$ is returned from the function `BayesianNetwork.getProb()` provided by the framework, as shown in Figure 2.



```java
for (Object value : Y.getDomain()) {
    Assignment ec = e.copy();
    ec.set(Y, value);
    prob += bn.getProb(Y, ec) * all(bn, vars, ec);
```

Figure 2: Java snippet in enumeration algorithm

2. Approximate Inference

In this section, we will present the main idea of three algorithms:`rejection sampling` algorithm, `likelihood weighting` algorithm and `gibbs sampling` algorithm, and several key points in implementing these algorithms.

The exact inference in a large, multiply connected Bayesian network is intractable, so it is necessary to find approximate inference methods.Monte Carlo algorithms are such kinds of randomized sampling algorithms. There are two families of Monte Carlo algorithms: direct sampling and Markov chain sampling. Rejection sampling and likelihood weighting belong to the first family, while gibbs sampling belongs to the second.

2.1 Rejection Sampling

The simplest random sampling process for Bayesian networks generates events without any evidence. This sample method is the basis of rejection sampling.

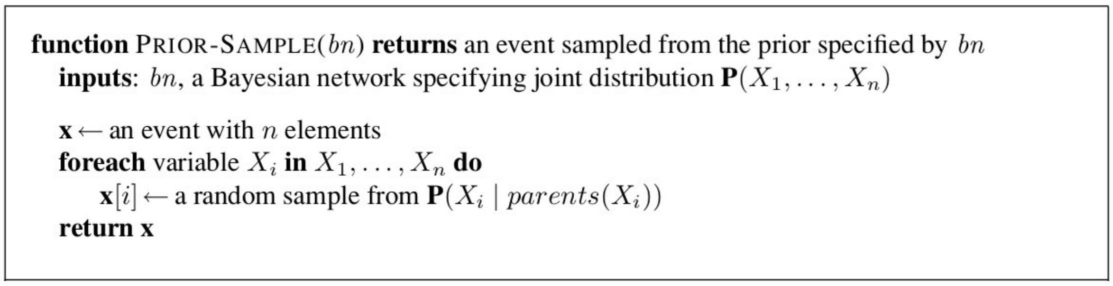
2.1.1 Main Idea of Rejection Sampling

Rejection sampling consists of three steps:

1. Sample from the prior distribution specified by the network without any evidence.
2. Reject all samples which are not consistent with the evidence.
3. Suppose there are $N_{ps}(\mathbf{e})$ remaining samples and among these samples, $N_{ps}(\mathbf{X},\mathbf{e})$ samples satisfies $X = x$, we have Equation 2.4.

$$\hat{\mathbf{P}}(X|\mathbf{e}) = \frac{N_{ps}(\mathbf{X},\mathbf{e})}{N_{ps}(\mathbf{e})} \tag{2.4}$$

The pseudocode is shown in Figure 3 and Figure 4.

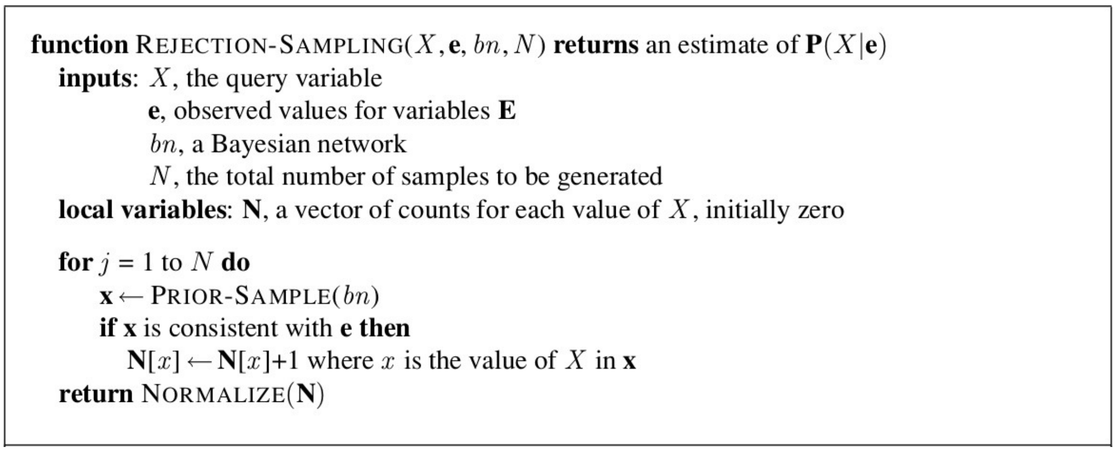


```
function PRIOR-SAMPLE($bn$) returns an event sampled from the prior specified by $bn$
  inputs: $bn$, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

  $\mathbf{x} \leftarrow$ an event with $n$ elements
  foreach variable $X_i$ in $X_1, \ldots, X_n$ do
      $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return $\mathbf{x}$
```

Figure 3: Pseudocode of prior sampling

As we can see, no evidence is provided in the function `PRIOR-SAMPLE()`. In the function `REJECTION-SAMPLING()`, only when a sample is consistent with the evidence **e**, the count number will increase by one. All samples which contradict to the evidence are ignored (or rejected).



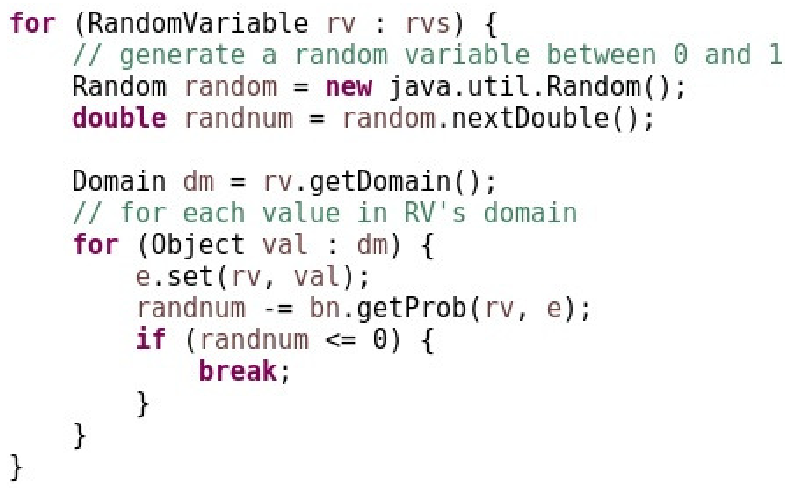
```
function REJECTION-SAMPLING($X$, $\mathbf{e}$, $bn$, $N$) returns an estimate of $\mathbf{P}(X|\mathbf{e})$
  inputs: $X$, the query variable
          $\mathbf{e}$, observed values for variables $\mathbf{E}$
          $bn$, a Bayesian network
          $N$, the total number of samples to be generated
  local variables: $\mathbf{N}$, a vector of counts for each value of $X$, initially zero

  for $j$ = 1 to $N$ do
     $\mathbf{x} \leftarrow$ PRIOR-SAMPLE($bn$)
     if $\mathbf{x}$ is consistent with $\mathbf{e}$ then
        $\mathbf{N}[x] \leftarrow \mathbf{N}[x]+1$ where $x$ is the value of $X$ in $\mathbf{x}$
  return NORMALIZE($\mathbf{N}$)
```

Figure 4: Pseudocode of rejection sampling

2.1.2 Key Points in Implementation

Once we have pseudocode of rejection sampling, it is straight-forward to turn it to Java code. The only thing we have to mention is how to draw a random sample as shown in Figure 3. Suppose a variable X has five possible values and their corresponding conditional probabilities are <0.3, 0.1, 0.2, 0.1, 0.3>. If we generate a random number between 0 and 1, for example, $R = 0.5$, we will assign the third value to X. The idea is quite intuitive. We subtract each conditional probability from the generated random number continuously until it is less then or equal to 0. Because $0.5 - 0.3 - 0.1 - 0.2 < 0$, so we choose the third value. This process is shown in Figure 5.



```java
for (RandomVariable rv : rvs) {
    // generate a random variable between 0 and 1
    Random random = new java.util.Random();
    double randnum = random.nextDouble();

    Domain dm = rv.getDomain();
    // for each value in RV's domain
    for (Object val : dm) {
        e.set(rv, val);
        randnum -= bn.getProb(rv, e);
        if (randnum <= 0) {
            break;
        }
    }
}
```

Figure 5: Java snippet of random sampling

2.2 Likelihood Weighting

The biggest problem of rejection sampling is that it rejects too many samples, especially when the number of evidence variables increase. However, likelihood weighting avoids such problem by generating only samples that are consistent with the evidence.

2.2.1 Main Idea of Likelihood Weighting

This algorithm keeps the values of the evidence variables fixed and only samples non-evidence variables. However, not all samples are equal given the fixed evidence. Therefore, each sample is weighted by the likelihood that the event would happen according to the evidence. The more likely an event will happen, the more weight it will be given. The pseudocode of this algorithm is shown in Figure 6.

```
function LIKELIHOOD-WEIGHTING($X$, $\mathbf{e}$, $bn$, $N$) returns an estimate of $\mathbf{P}(X|\mathbf{e})$
  inputs: $X$, the query variable
          $\mathbf{e}$, observed values for variables $\mathbf{E}$
          $bn$, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$
          $N$, the total number of samples to be generated
  local variables: $\mathbf{W}$, a vector of weighted counts for each value of $X$, initially zero

  for $j$ = 1 to $N$ do
      $\mathbf{x}, w \leftarrow$ WEIGHTED-SAMPLE($bn$, $\mathbf{e}$)
      $\mathbf{W}[x] \leftarrow \mathbf{W}[x] + w$ where $x$ is the value of $X$ in $\mathbf{x}$
  return NORMALIZE($\mathbf{W}$)

function WEIGHTED-SAMPLE($bn$, $\mathbf{e}$) returns an event and a weight
  $w \leftarrow 1$; $\mathbf{x} \leftarrow$ an event with $n$ elements initialized from $\mathbf{e}$
  foreach variable $X_i$ in $X_1, \ldots, X_n$ do
      if $X_i$ is an evidence variable with value $x_i$ in $\mathbf{e}$
          then $w \leftarrow w \times P(X_i = x_i \mid parents(X_i))$
          else $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return $\mathbf{x}$, $w$
```

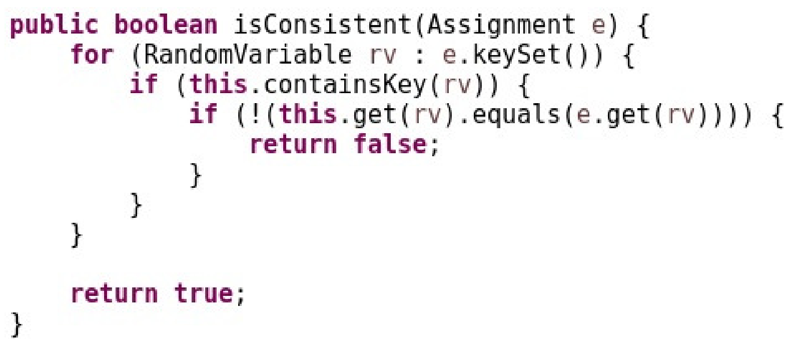
Figure 6: Pseudocode of likelihood weighting

In the function `WEIGHTED-SAMPLE()`, if a variable X is in the evidence **e**, its conditional probability is weighted multiplied directly; otherwise, it is a hidden variable and we sample it. The rest part of this algorithm is similar to rejection sampling algorithm.

2.2.2 Key Points in Implementation

As we can see in Figure 6, the function `WEIGHTED-SAMPLE()` returns two variables: $\mathbf{x}$ and w. However, Java doesn't support returning multiple variables, e.g., a tuple. So we implement a class `Pair.java`. Its constructor accepts two variables ($\mathbf{x}$ and w) and it can be passed as a whole. Please refer to the source code for detail.

Another modification to the framework is adding a member function `isConsistent()` to the class `Assignment`. It check whether the calling assignment is consistent with the passed assignment. The basic idea is to compare all common variables contained in the two assignments. If and only if all pairs of common variables have the same value in the two assignment, they are consistent. Please refer to the Figure 7



```java
public boolean isConsistent(Assignment e) {
    for (RandomVariable rv : e.keySet()) {
        if (this.containsKey(rv)) {
            if (!(this.get(rv).equals(e.get(rv)))) {
                return false;
            }
        }
    }

    return true;
}
```

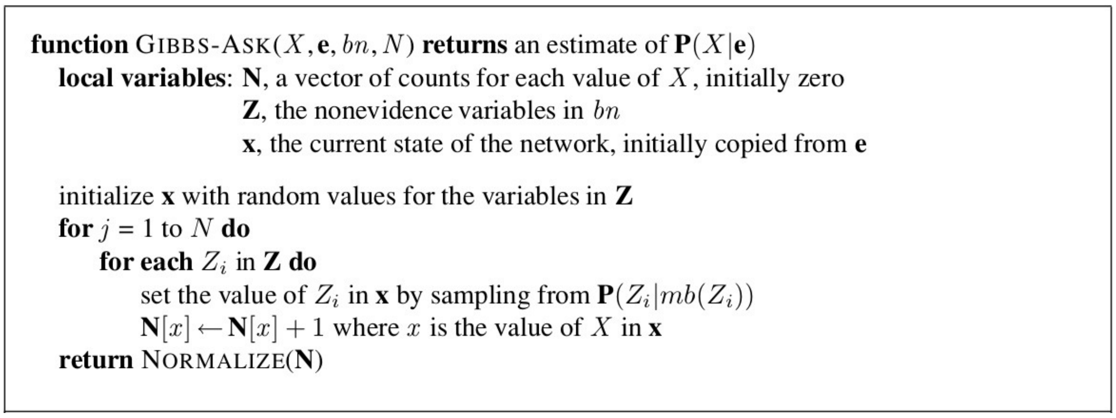
Figure 7: Java snippet of isConsistent()

2.3 Gibbs Sampling

Markov chain Monte Carlo (MCMC) algorithms generate each sample by making a random change to the preceding sample.Gibbs sampling is a particular form of MCMC algorithms which is quite suitable for Bayesian Network.

2.3.1 Main Idea of Gibbs Sampling

With the evidence variables fixed at their observed values, the Gibbs sampling algorithm for Bayesian Networks starts with an arbitrary state and generates the next state by randomly sampling a value for any one of non-evidence variables. The sampling for X_i is done conditioned on the current values of the variables in the Markov blanket of X_i. The algorithm is shown in Figure 8.



```
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)
```

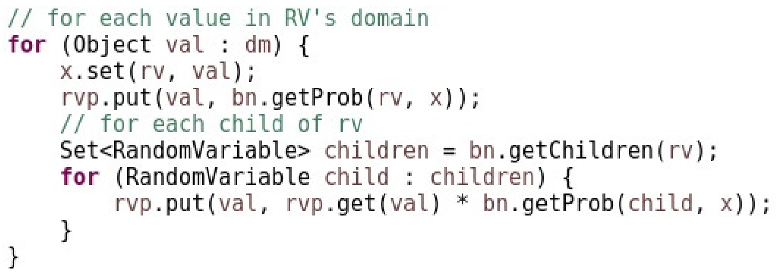
Figure 8: Pseudocode of gibbs sampling

The key part of this algorithm is how to implement "sampling from $\mathbf{P}(Z_i|mb(Z_i))$". Here $mb(X_i)$ denotes the values of the variables in X_i's Markov blanket (a variable's parents, children, and children's parents). The probability of a variable given its Markov blanket is proportional to the probability of the variable given its parents times the probability of each child given its parents, as shown in Equation 2.5.

$$P(z_i'|mb(Z_i)) = \alpha P(z_i'|parents(Z_i)) \times \prod_{Y_j \in Childrend(X_i)} P(y_j|parents(Y_j)) \tag{2.5}$$

2.3.2 Key Points in Implementation

To generate $P(z_i'|mb(Z_i))$, we have to process every possible values of the variable Z. Thanks to the provided framework, it is easy to generate all children of a given variable. We use two loops to implement it. The outer loop visit all possible values of the variable Z (in code, it is represented by "rv"), and the inner loop traverse all its children, as shown in Figure 9.



```
// for each value in RV's domain
for (Object val : dm) {
    x.set(rv, val);
    rvp.put(val, bn.getProb(rv, x));
    // for each child of rv
    Set<RandomVariable> children = bn.getChildren(rv);
    for (RandomVariable child : children) {
        rvp.put(val, rvp.get(val) * bn.getProb(child, x));
    }
}
```

Figure 9: Java snippet of part of $P(z_i'|mb(Z_i))$

3. Samples

We choose five samples as our experiment, as results are shown below. In each example, we provide an exemplar command.

3.1 Sample: aima-alarm

3.1.1 Command

In terminal: java bn.sln.ExactInferencer aima-alarm.xml B J true M true
In run.sh: java ExactInferencer aima-alarm.xml B J true M true

3.1.2 Result

Table 1: Result of aima-alarm

Enumeration	N	Rejection	Likelihood	Gibbs
{T=0.284, F=0.716}	200	{T=NaN, F=NaN}	{T=0.0, F=1.0}	{T=0.245, F=0.755}
{T=0.284, F=0.716}	1000	{T=0.667, F=0.333}	{T=0.528, F=0.472}	{T=0.259, F=0.741}
{T=0.284, F=0.716}	5000	{T=0.286, F=0.714}	{T=0.430, F=0.571}	{T=0.291, F=0.709}
{T=0.284, F=0.716}	10000	{T=0.269, F=0.731}	{T=0.251, F=0.749}	{T=0.281, F=0.719}

3.1.3 Error

The `Enumeration` algorithm will be considered as a correct result, and all three other results will be compared to `Enumeration` result. Note that if a result is "NaN" (due to too many rejection), it will be considered as "T=1, F=0".

Table 2: Error stats. of aima-alarm

N	Rejection	Likelihood	Gibbs
200	0.716	0.284	0.245
1000	0.383	0.244	0.025
5000	0.002	0.146	0.007
10000	0.015	0.033	0.003

The plot is shown in Figure 10. As we can see, the least average error is Gibbs sampling, and the largest average is Rejection sampling. In addition, as the sampling number N increases, the error is decreasing.

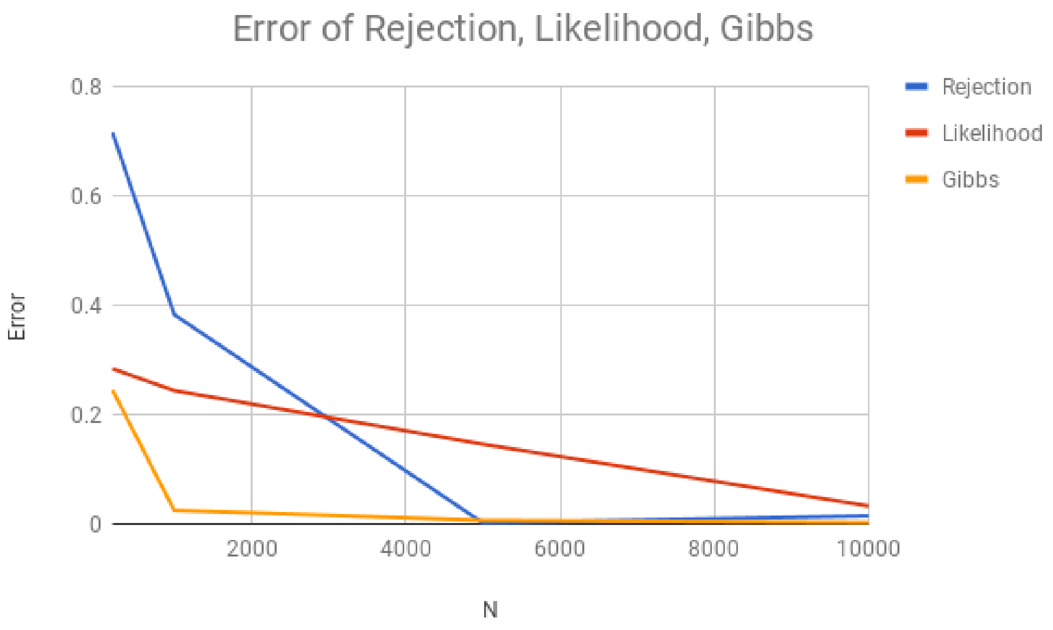


Figure 10: Error plot of aima-alarm

3.1.4 Running Time

The running time is measure in *ms*, as is shown in Table 3. The plot is shown in Figure 11.

Table 3: Running time (ms) of aima-alarm

N	**Enumeration**	**Rejection**	**Likelihood**	**Gibbs**
200	3	25	32	36
1000	3	77	83	102
5000	3	161	200	334
10000	3	324	285	590

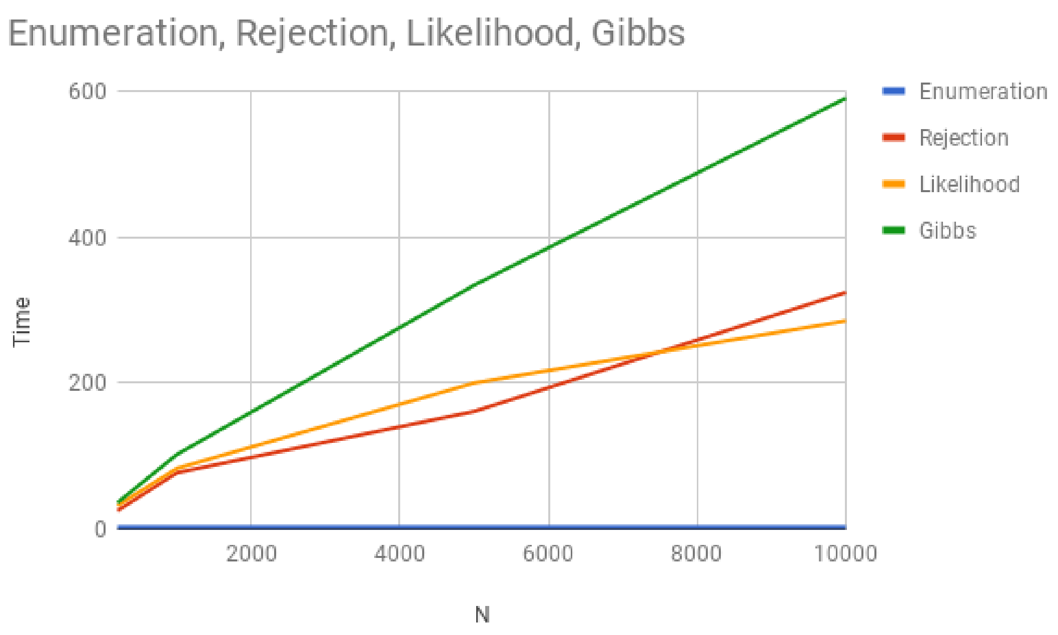


Figure 11: Running time (ms) of aima-alarm

It is obvious that the more sample N, the longer time the algorithm will take to run.

3.2 aima-wet-grass

3.2.1 Command

In terminal: java bn.sln.RSampleInferencer 1000 aima-wet-grass.xml R S true
In run.sh: java RSampleInferencer 1000 aima-wet-grass.xml R S true

3.2.2 Result

Table 4: Resulf of aima-wet-grass

Enumeration	N	Rejection	Likelihood	Gibbs
{T=0.3, F=0.7}	200	{T=0.323, F=0.677}	{T=0.298, F=0.702}	{T=0.26, F=0.74}
{T=0.3, F=0.7}	1000	{T=0.274, F=0.726}	{T=0.528, F=0.472}	{T=0.296, F=0.704}
{T=0.3, F=0.7}	5000	{T=0.305, F=0.695}	{T=0.340, F=0.660}	{T=0.300, F=0.700}
{T=0.3, F=0.7}	10000	{T=0.309, F=0.691}	{T=0.294, F=0.706}	{T=0.305, F=0.695}

3.2.3 Error

The Enumeration algorithm will be considered as a correct result, and all three other results will be compared to Enumeration result. The plot is shown in Figure 12.

Table 5: Error stats. of aima-wet-grass

N	Rejection	Likelihood	Gibbs
200	0.023	0.002	0.04
1000	0.026	0.228	0.004
5000	0.005	0.004	0
10000	0.009	0.006	0.005

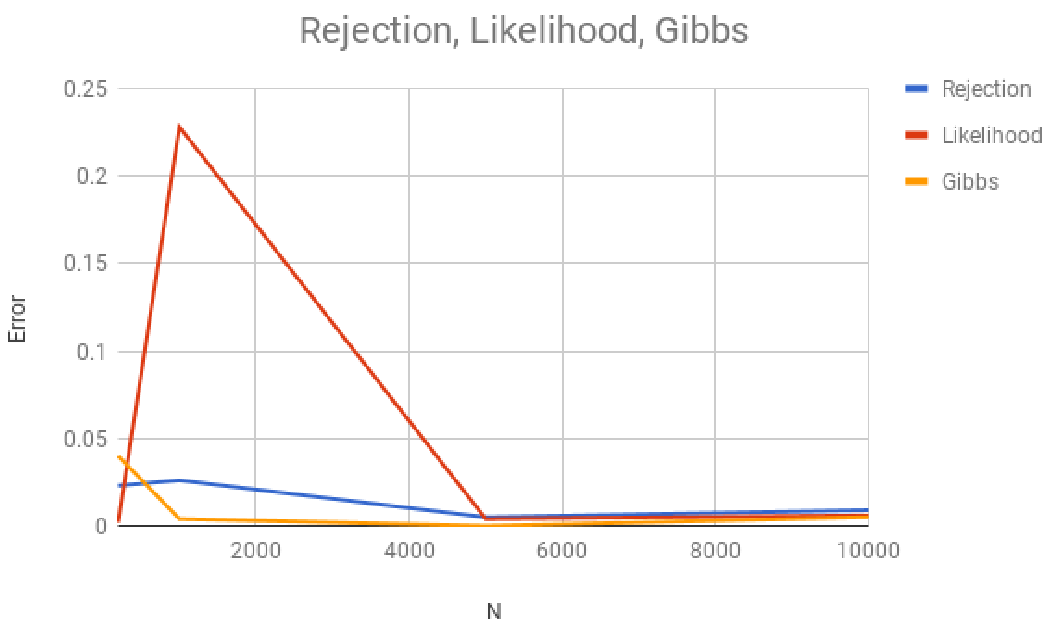


Figure 12: Error plot of aima-wet-grass

As we can see, Gibbs sampling has the best performance generally.

3.2.4 Running Time

The running time is measure in *ms*, as is shown in Table 6. The plot is shown in Figure 13.

Table 6: Running time (ms) of aima-wet-grass

N	Enumeration	Rejection	Likelihood	Gibbs
200	2	18	22	40
1000	2	53	57	92
5000	2	184	107	247
10000	2	226	170	400

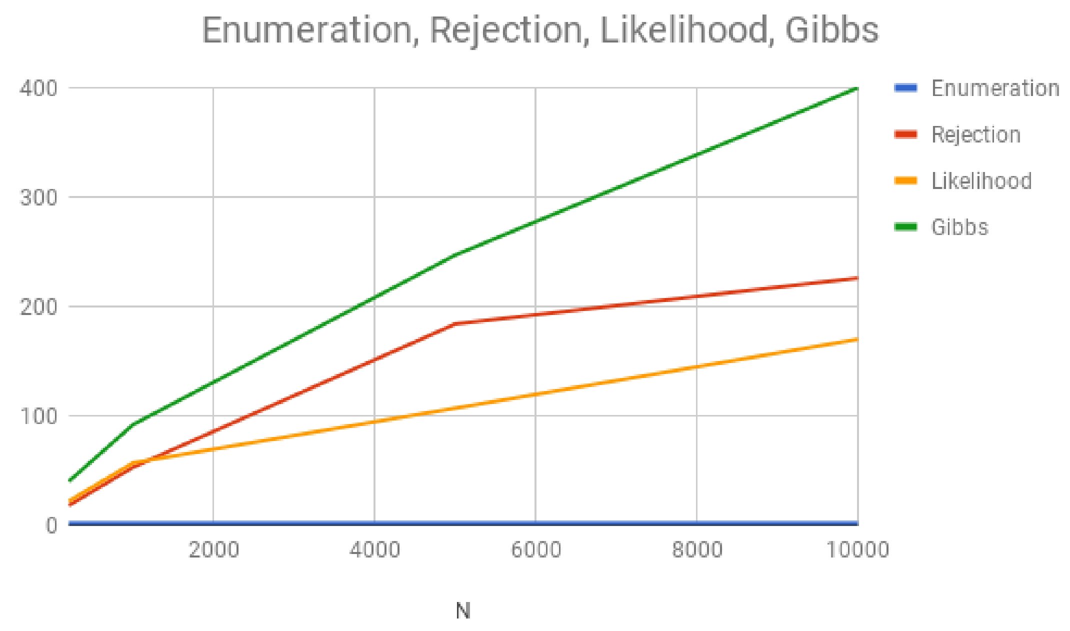


Figure 13: Running time (ms) of aima-wet-grass-time

It is obvious that the more sample N, the longer time the algorithm will take to run. The Gibbs sampling takes the longest time to run.

3.3 alarm

3.3.1 Command

In terminal: java bn.sln.LSampleInferencer 1000 alarm.bif LVEDVOLUME HYPOVOLEMIA FALSE LVFAILURE TRUE
In run.sh: java LSampleInferencer 1000 alarm.bif LVEDVOLUME HYPOVOLEMIA FALSE LVFAILURE TRUE

3.3.2 Result

The `Enumeration` algorithm takes to much time to run, so we don't have its result. However, we can use the result of Gibbs sampling as a correct result.

Table 7: Result of alarm

Enumeration	**N**	**Rejection**	**Likelihood**	**Gibbs**
-	200	{L=1.0, N=0.0, H=0.0}	{L=0.980, N=0.015, H=0.005}	{L=0.955, N=0.02, H=0.025}
-	1000	{L=1.0, N=0.0, H=0.0}	{L=0.984, N=0.007, H=0.009}	{L=0.941, N=0.007, H=0.052}
-	5000	{L=0.989, N=0.005, H=0.005}	{L=0.984, N=0.008, H=0.009}	{L=0.985, N=0.008, H=0.006}
-	10000	{L=0.988, N=0.012, H=0.0}	{L=0.979, N=0.011, H=0.010}	{L=0.983, N=0.014, H=0.004}

3.3.3 Error

The `Enumeration` algorithm takes to much time to run, so we don't have its result. However, we can use the result of Gibbs sampling as a correct result. and all two other results will be compared to `Gibbs` result. The plot is shown in Figure 14.

Table 8: Error stats. of alarm

N	**Rejection**	**Likelihood**	**Gibbs**
200	0.045	0.025	0
1000	0.059	0.043	0
5000	0.004	0.001	0
10000	0.005	0.004	0

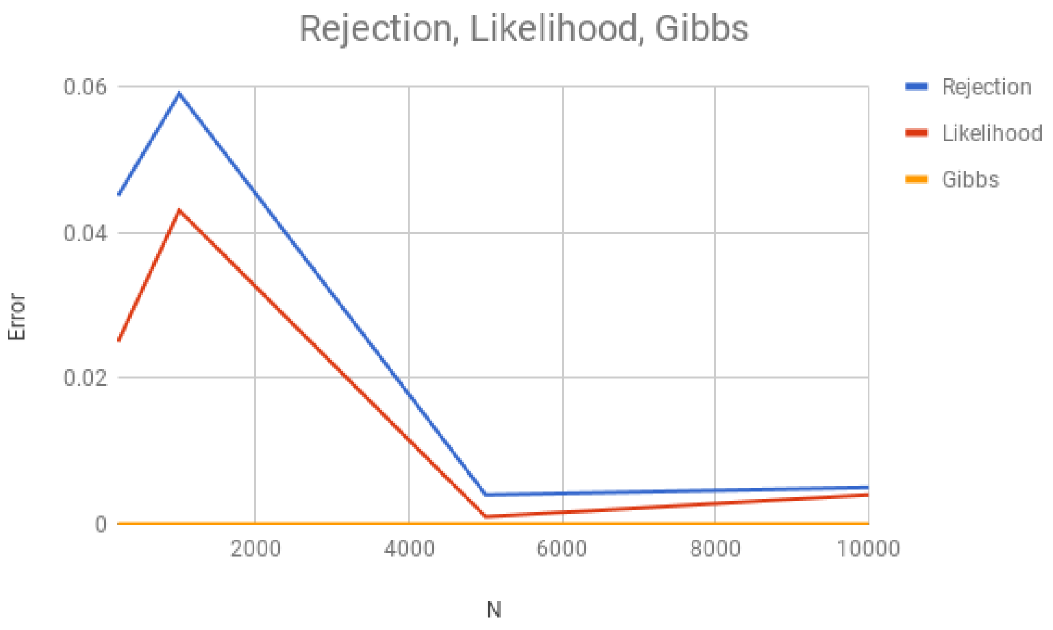


Figure 14: Error plot of alarm

3.3.4 Running Time

The running time is measure in *ms*, as is shown in Table 9. The plot is shown in Figure 15.

Table 9: Running time (ms) of alarm

N	**Rejection**	**Likelihood**	**Gibbs**
200	178	208	812
1000	768	726	4016
5000	3380	3191	17976
10000	6642	6808	35761

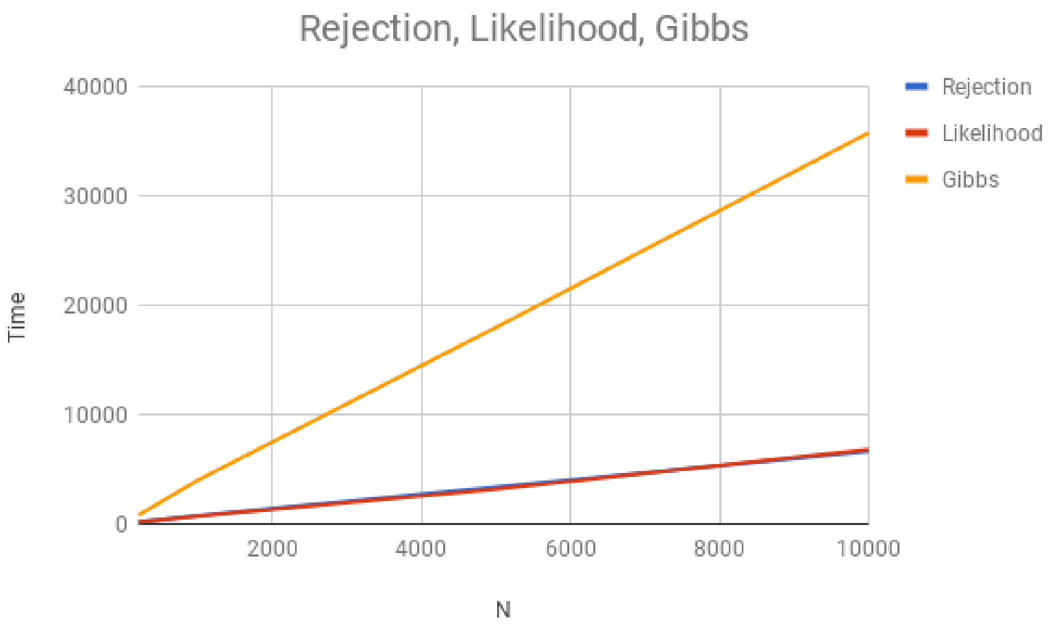


Figure 15: Running time (ms) plot of alarm

Note that the lines of Rejection sampling and Likelihood weighting are overlap. The Gibbs sampling is the most time-consuming.

3.4 dog-problem

3.4.1 Command

In terminal: java bn.sln.LSampleInferencer 200 dog-problem.xml bowel-problem light-on false hear-bark true
In run.sh: java LSampleInferencer 200 dog-problem.xml bowel-problem light-on false hear-bark true

3.4.2 Result

Table 10: Result of dog-problem

Enumeration	N	Rejection	Likelihood	Gibbs
{T=0.027, F=0.973}	200	{T=0.025, F=0.975}	{T=0.016, F=0.984}	{T=0.03, F=0.97}
{T=0.027, F=0.973}	1000	{T=0.045, F=0.955}	{T=0.027, F=0.973}	{T=0.034, F=0.966}
{T=0.027, F=0.973}	5000	{T=0.026, F=0.974}	{T=0.024, F=0.976}	{T=0.026, F=0.974}
{T=0.027, F=0.973}	10000	{T=0.026, F=0.974}	{T=0.030, F=0.970}	{T=0.027, F=0.973}

3.4.3 Error

The `Enumeration` algorithm will be considered as a correct result, and all three other results will be compared to `Enumeration` result. The plot is shown in Figure 16.

Table 11: Error stats. of dog-problem

N	Rejection	Likelihood	Gibbs
200	0.002	0.011	0.003
1000	0.018	0	0.007
5000	0.001	0.003	0.001
10000	0.001	0.003	0

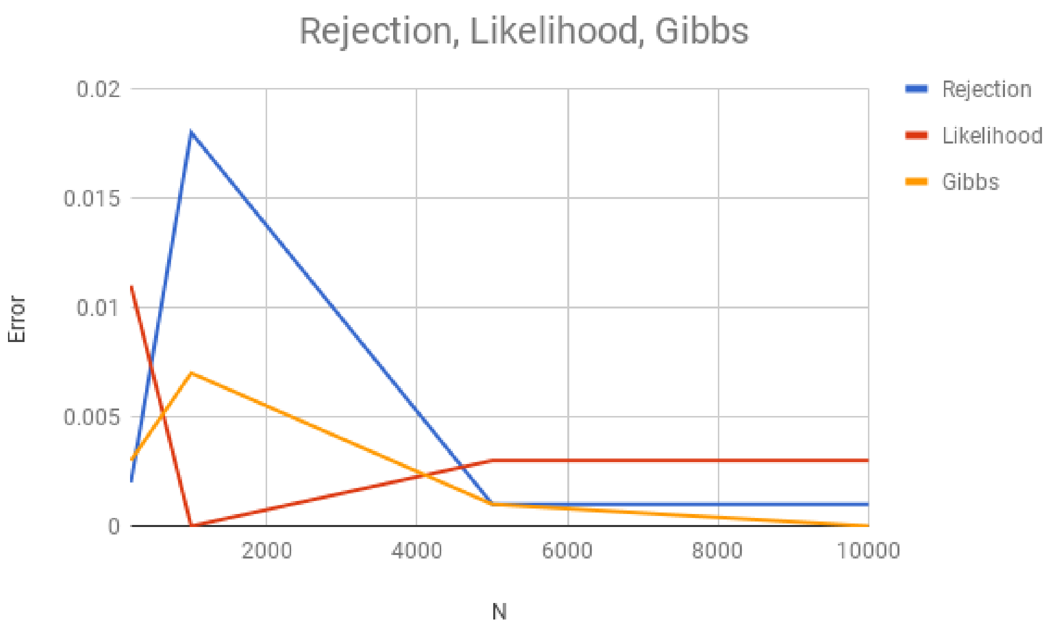


Figure 16: Error plot of dog-problem

3.4.4 Running Time

The running time is measure in *ms*, as is shown in Table 12. The plot is shown in Figure 17.

Table 12: Running time (ms) of dog-problem

N	Enumeration	Rejection	Likelihood	Gibbs
200	3	26	34	66
1000	3	52	64	116
5000	3	199	202	297
10000	3	278	303	559

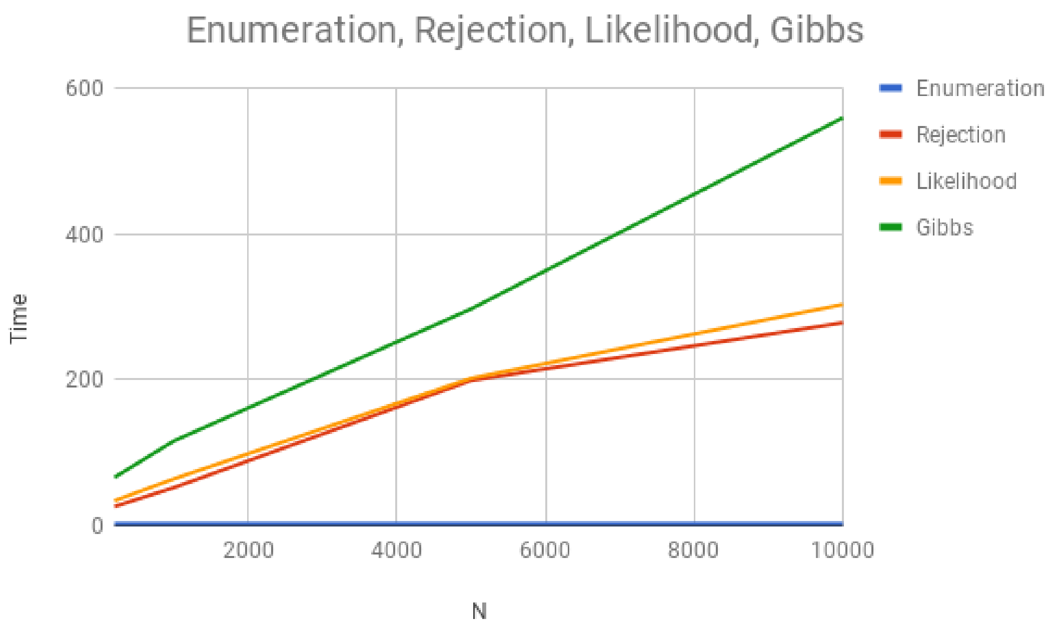


Figure 17: Running time (ms) plot of dog-problem

The Gibbs sampling still needs the longest time, while the enumeration inference uses the least time.

3.5 insurance

3.5.1 Command

In terminal: java bn.sln.GSampleInferencer 5000 insurance.bif Accident Age Adult Mileage TwentyThou MakeModel SportsCar
In run.sh java GSampleInferencer 5000 insurance.bif Accident Age Adult Mileage TwentyThou MakeModel SportsCar

3.5.2 Result

The `Enumeration` algorithm takes to much time to run, so we don't have its result.

Table 13: Resulf of insurance

Enumeration	N	Rejection
-	200	{None=0.875, Mild=0.0, Moderate=0.0, Severe=0.125}
-	1000	{None=0.793, Mild=0.103, Moderate=0.034 Severe=0.069}
-	5000	{None=0.722, Mild=0.127, Moderate=0.070, Severe=0.082}
-	10000	{None=0.729, Mild=0.101, Moderate=0.104, Severe=0.065}

Table 14: Result of insurance (cont.)

N	Likelihood	Gibbs
200	{None=0.761, Mild=0.065, Moderate=0.056, Severe=0.119}	{None=0.985, Mild=0.01, Moderate=0.005, Severe=0.0}
1000	{None=0.746, Mild=0.091, Moderate=0.080, Severe=0.082}	{None=0.530, Mild=0.242, Moderate=0.081, Severe=0.147}
5000	{None=0.747, Mild=0.102, Moderate=0.070, Severe=0.081}	{None=0.7, Mild=0.153, Moderate=0.058, Severe=0.090}
10000	{None=0.761, Mild=0.0945, Moderate=0.073, Severe=0.071}	{None=0.771, Mild=0.058, Moderate=0.083, Severe=0.089}

Note that there are some abbreviations used in the table: N: None, Mi: Mild, Mo: Moderate, S: Severe.

3.5.3 Error

The `Enumeration` algorithm takes to much time to run, so we don't have its result. However, we can use the result of Gibbs sampling as a correct result. and all two other results will be compared to `Gibbs` result. The plot is shown in Figure 18.

Table 15: Error stats. of insurance

N	Rejection	Likelihood	Gibbs
200	0.11	0.224	0
1000	0.263	0.216	0
5000	0.022	0.047	0
10000	0.042	0.011	0

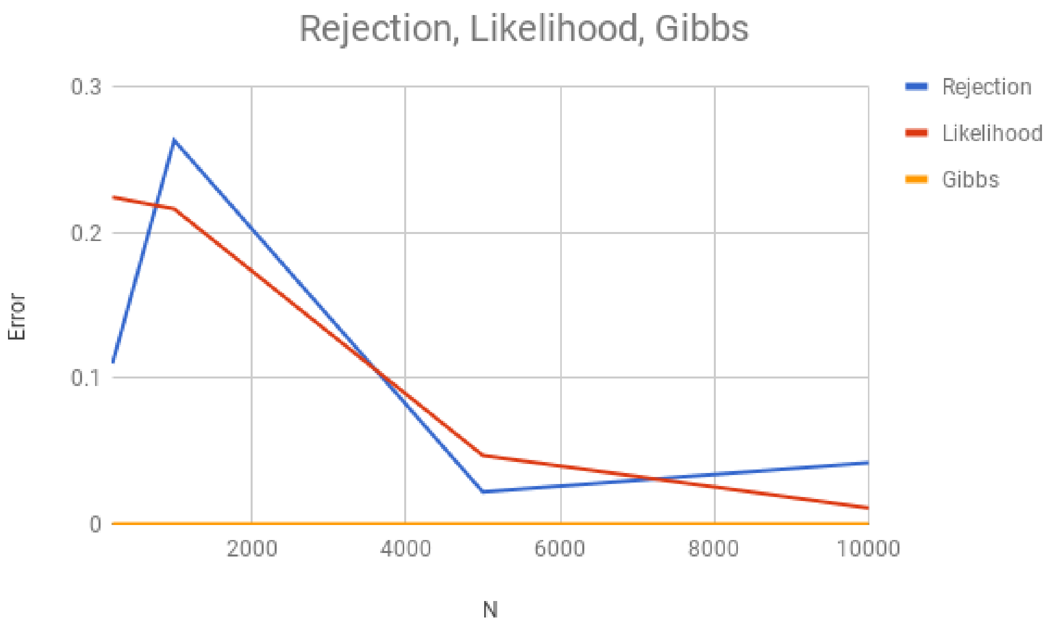


Figure 18: Error plot of insurance

3.5.4 Running Time

The running time is measure in *ms*, as is shown in Table 16. The plot is shown in Figure 19.

Table 16: Running time (ms) of insurance

N	Rejection	Likelihood	Gibbs
200	108	155	664
1000	454	475	2783
5000	1879	1682	14424
10000	3203	3064	25890

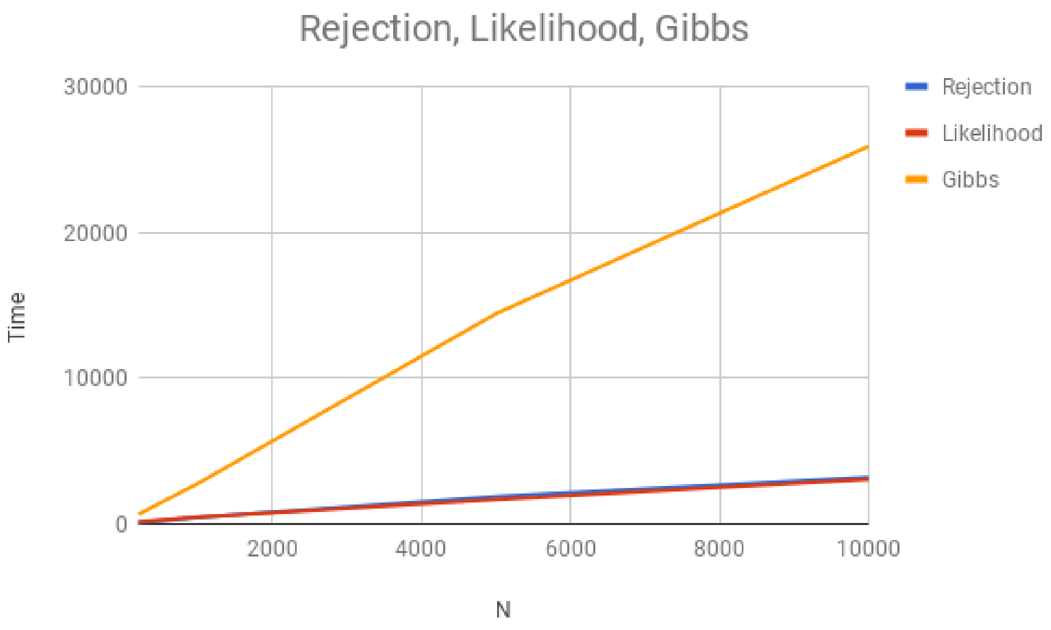


Figure 19: Running time (ms) of insurance

Note that the lines of Rejection sampling and Likelihood weighting are overlap. The Gibbs sampling is the most time-consuming.

References

[1] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., & Edwards, D. D. (2003). *Artificial intelligence: a modern approach* (Vol. 2, No. 9). Upper Saddle River: Prentice hall.