# 275B Class Project

# Empirical evaluations of some benchmark Bayesian networks

Authors:
Igor Cadez, Hong Zhao, Stephen Bay, Scott Gafney, Dmitry Pavlov
and Rina Dechter

Fall Quarter, 1998

# 1 Introduction

This paper reports students' projects performed during the 275a class: "Network-based reasoning: belief networks," in the department of Information and Computer Science at UC-Irvine, taught by Rina Dechter. Students were required to select one problem from the bayesian repository benchmarks in

`http://www-nt.cs.berkeley.edu/home/nir/public_html/Repository/index.htm`

and to run a comparative study of several known algorithms and report the results. This paper include five of the students' reports. The problems used are: Pigs, Diabetes, Insurance, HailFinder and ALARM.

**Bucket Elimination and Mini Bucket Approximation on "Pigs" Domain**

Igor V. Cadez
Department of Information and Computer Science
University of California, Irvine
icadez@ics.uci.edu

# 2 Introduction

The purpose of this project is to compare two inference algorithms: "Bucket Elimination" and "Mini Bucket Elimination." The first algorithm is an exact inference algorithm for arbitrary belief network. It calculates maximum probable explanation (MPE) of observed evidence in the network by maximizing joint probability over the set of unobserved variables. To perform this task, the full joint probability is processed sequentially in such a way that the summation and maximization are performed on functions of manageable size. The mini-bucket approximation to the exact algorithm is very similar, but instead of performing full summation and maximization, functions are broken down into smaller pieces and each piece is maximized independently. In this way, the interaction between pieces is not taken into the account and the resulting algorithm is only approximate. In return, since maximization is performed only over smaller subsets, the resulting algorithm is asymptotically much faster. The motivation for this project comes from practical interest in fast any-time algorithms for inference in belief networks.

## 2.1 The domain

The domain we use to perform the tests is called "Pigs." Each node in the network represents a single specimen, and each link represents a parental relationship. The domain represents a particular breeding problem, where carriers of a specific genetic disease are to be excluded from the further breeding. To perform this task, pigs that carry the disease have to be identified. The laboratory testing approach is not cost efficient, so the inference is made based on the observed population of pigs that express the disease. Given this information, the task is to find the most probable explanation of the given network, given the observed evidence. Each node (pig) can take a value from a ternary domain: "healthy," "carrier" and "sick." With each node there is a probability associated with each label, given the labels of the parents. In other words, the conditional probability table (CPT) represents different combinations of probabilities for a child to inherit and further express the disease, given the health status of the parents and specific details of biological inheritance. The domain summary is:

- Size is 441 nodes,
- Number of edges is 592,
- Each node has either 0 or 2 parents,
- Each node can have many child nodes,
- The induced width of the network is 12.

## 2.2 The task

The task of the project is to perform MPE experiments with simulated evidence and observe the properties of the approximate algorithm on this real-life domain, as compared to the exact algorithm. In particular, the emphasis is on the two aspects:

- The quality of solution of approximate algorithm as a function of mini-bucket size (the smaller the size, the worst the approximation).
- The time it takes the approximate algorithm to reach and detect the exact solution.

The first task is relevant because it summarizes the average performance of the approximation as a function of the quality of the approximation. It is expected to give some insight into size of the mini-buckets required to obtain "good enough" solutions. However, the results obtained in that part of the project are only rough guidelines for the particular domain, and cannot be used for any other domain (some systematic evaluation with analytical background is required). The second part of the project is more interesting as it mimics a real life situation. If one was exposed to a problem of such complexity that the exact solution is intractable, the natural way to proceed would be to start iterating the approximate solution with increasing approximation quality (i.e. increasing mini-bucket size) and hope to reach the exact solution without having to run the full (intractable) algorithm. In this sense, it is not enough to actually find the true solution, but the algorithm also has to detect that the true solution has been reached. This is achieved by calculating upper and lower bounds on the possible probability of the MPE tuple. It is worth noticing that the two boundaries are discrete, hence one can expect that at a certain point they will exactly match. Once they do, it signals that the MPE tuple has been found. In practice, the true solution is typically found with certain mini-bucket size, but it is not until mini-bucket size increases by 1 or 2 before the algorithm can actually detect that the MPE tuple has been found. This is a consequence of the bounds not being tight enough.

## 2.3 Methodology

The experiment we run consists of exact bucket elimination algorithm followed by the set of approximations for mini-bucket sizes from 1 to 11 (note that with mini-bucket size of 12 (the induced width), the approximation becomes exact). The evidence consists of instantiation of 10 nodes with likely, but random evidence. After each set of models, the algorithm recreates both the evidence nodes and the evidence "contents." We record the upper and the lower bound for each of the experiments, together with time it took the algorithm to evaluate it (probability of the returned tuple is the lower bound on the exact solution). We then look at the performance of the approximation over different runs (different evidence nodes and different evidence, but the same domain) for each model separately. This yields 200 ratios of the type UpperBound/LowerBound per model. We use this measure as the quality of the solution. Note that the algorithm might have returned the exact tuple, but if the upper bound is not tight, the overall quality of the solution will be poor. This measure of quality is selected on purpose to most closely resemble a real life application where one cannot run the exact inference algorithm. There are additional measures available like: UpperBound/ExactSolution, or ExactSolution/LowerBound, but these measures use the information about the exact solution; if the exact solution were known, there would be no need to run the approximations. Hence, for a solution to qualify as the good solution, it is not enough to be close to the MPE tuple, but the approximation must be able to actually detect this.

### 2.4 Results

The main results of the first part of the project are summarized on 11 histograms that represent the distribution of the quality of the solutions for each of the approximate models. The histograms reveal the expected behavior where with increase in approximation quality (increased mini-bucket size), more and more iterations return better solutions. The histograms also reveal possible bimodality of the quality distribution, a result that has been already empirically observed on different domains.

The second part of the project is concerned with time to obtain the exact solution using mini-bucket approximation with increasing size of mini-buckets. This is much more relevant result, as it shows how one can obtain the exact solution by iterating through approximation quality. It also shows that there is typically much to be gained by using this approach. In a sense, the approach of iterating through approximation complexities is similar to $A^*$ search, where one keeps redoing what has already been done. The exponential complexity of the problems in question guarantees that the last iteration is much more time consuming than even the sum of all the previous iterations. With this in mind, it is not surprising that the iterative approach is promising. We show that the iterative approach is between one and two orders of magnitude faster than the full approach (on this particular domain), and that even the worst case scenario is just slightly worse than the full bucket elimination approach (i.e. the overhead of all the approximations is almost negligible compared to the exact algorithm). Everything said so far relies on the fact that the exact solution can be detected without explicitly evaluating it by full bucket elimination. Once again, this is a consequence of discreteness of the probabilities of tuples and the fact that the upper bound can be put on the MPE tuple probability.

### 2.5 Summary

In this project we compare the exact bucket elimination algorithm to the version of mini-bucket approximation called *iterative improvement mini-bucket elimination*, and show that significant running time improvements can be achieved (and are achieved on this domain). The achieved running time decrease is not paid for by any decrease in quality of solution; in any case we are interested in *exact solutions only.* The iterative improvement mini-bucket approximation is *not* an approximation, but rather a nice way of using exponential time complexity of bucket algorithms to achieve exact solutions in less time.

### Acknowledgments

Author would like to thank Irina Rish for the C++ code that performed the actual calculations (well, after *some* modifications at least ...). Also, thanks go to professor Rina Dechter for availability of Bucket elimination and Mini-bucket approximation algorithms.

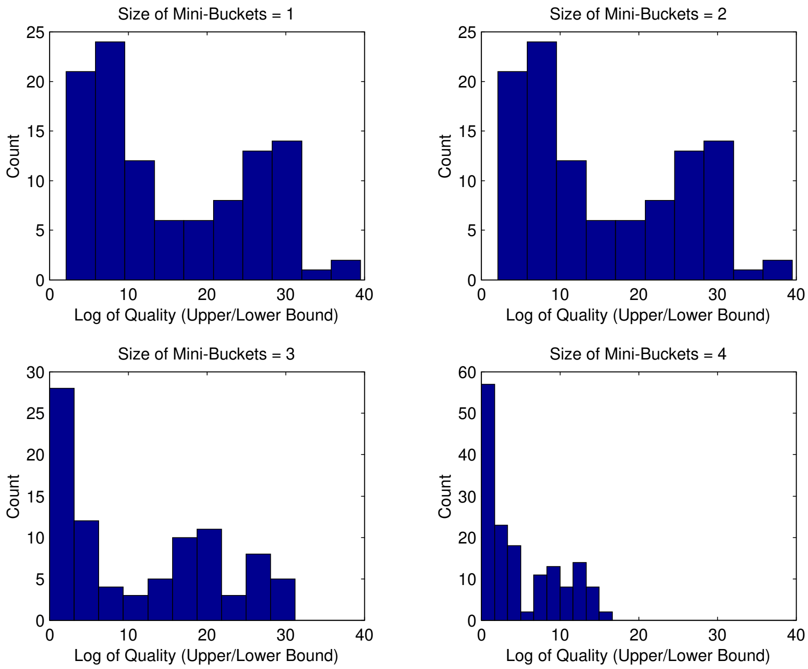


Figure 1: Histogram of fit quality for different mini-bucket sizes (Upper Bound/Lower Bound).

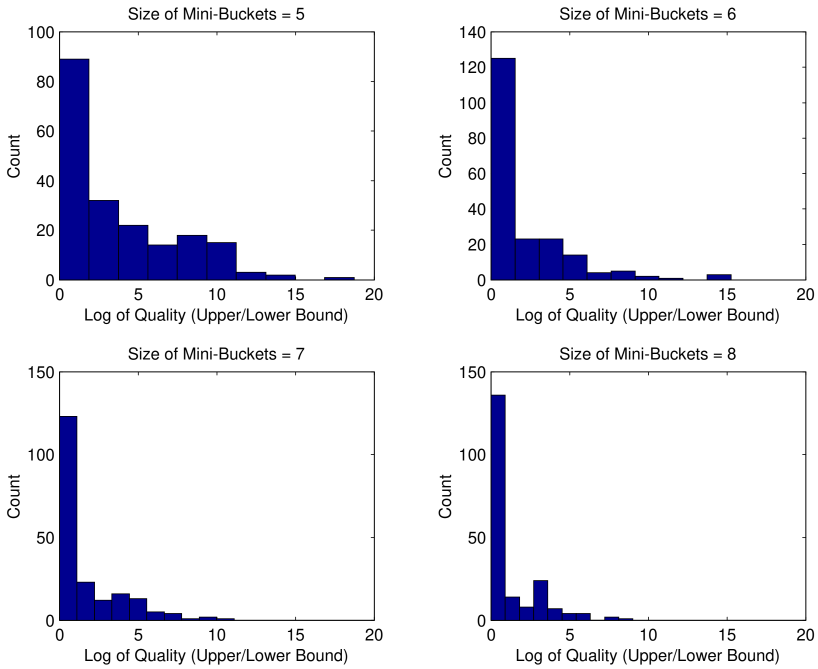


Figure 2: Histogram of fit quality for different mini-bucket sizes (Upper Bound/Lower Bound).

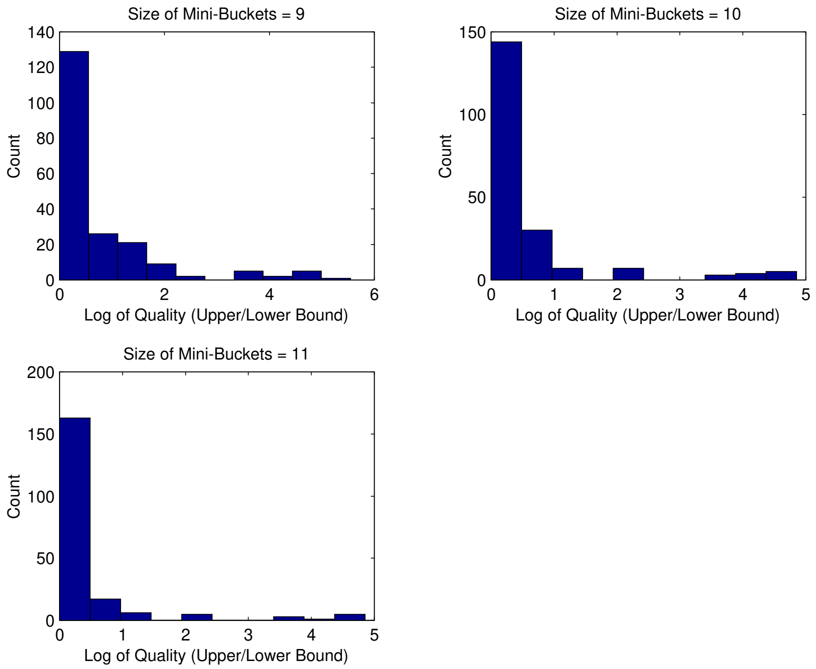


Figure 3: Histogram of fit quality for different mini-bucket sizes (Upper Bound/Lower Bound).

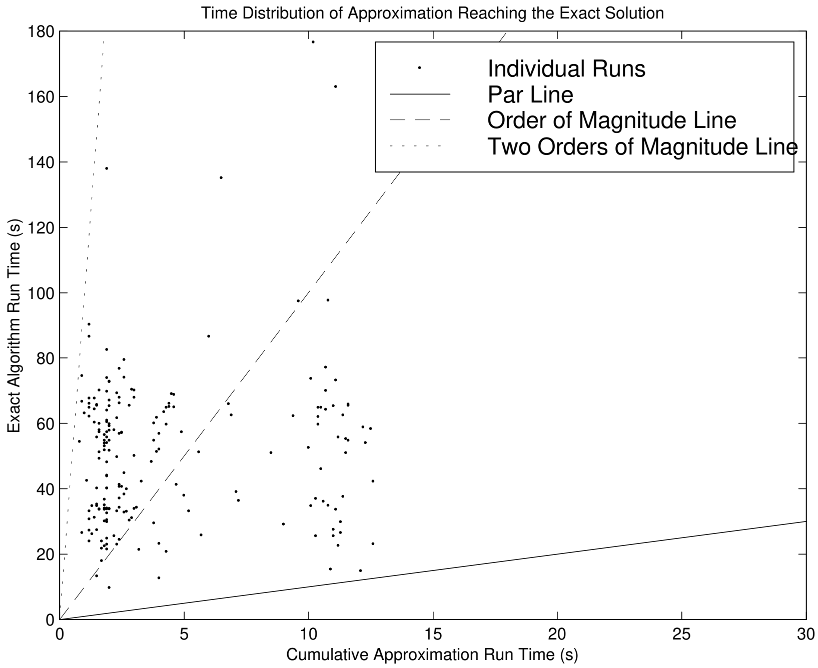


Figure 4: Time performance of the *iterative improvement mini bucket* algorithm compared to the time of the exact algorithm.

**Diabetes: A preliminary model for insulin dose adjustment**

Hong Zhao
Department of Information and Computer Science
University of California, Irvine
hzhao@ics.uci.edu

My project is to analyze the Diabetes Network. My task is compare the performances of three algorithms: Bucket-Elimination MPE, MiniBucket-Elimination approximation MPE, simple greedy MPE.

# 3 Introduction

This model is a preliminary model for insulin dose adjustment and developed by Steen Andreassen, Roman Hovorka, Jonathan Benn, Kristian G. Olesen, and Ewart R. Carson. The following is the information of the article about this model:

- Author: "Steen Andreassen and Roman Hovorka and Jonathan Benn and Kristian G. Olesen and Ewart R. Carson"
- Title: "A Model-based Approach to Insulin Adjustment"
- Booktitle: "Proceedings of the Third Conference on Artificial Intelligence in Medicine"
- Year: 1991
- Editor: "M. Stefanelli and A. Hasman and M. Fieschi and J. Talmon"
- Pages: "239–248"
- Publisher: "Springer-Verlag"

I downloaded the net work from the web site:
http://www-nt.cs.berkeley.edu/home/nir/public_html/Repository/Diabetes.htm and the contributer is Uffe Kjaerulff.

## 3.1 Network structure

### 3.1.1 Nodes

The network consists of 24 structurally identical subnetworks interconnected via temporal links. This file is .bif file. There are 413 variables in this model, which are listed as following:

- cho_init(16), ins_sens(5),
- meal_i(21), cho_i(21), bg_i(11), activ_ins_i(10), ins_abs_i(11), gut_abs_i(6), renal_cl_i(3), ins_indep_util_i(3), ins_dep_util_i(8), glu_prod_i(6), ins_indep_i(5), ins_dep_i(13), endo_bal_i(17), cho_bal_i(16), basal_bal_i(19), met_irr_i(7), tot_bal_i(15)

  for each subnetwork i, where "i" is the integer from 0 to 23, totally 17*24 = 408 variables.

- meal_24(21), cho_24(21), bg_24(11)

So totally there are 2+408+3 = 413 variables.

*Note: The integer in the parenthesis right after each variable is the total number of possible values of this variable. For example: "ins_indep_util_0(3)," the possible values of variable "ins_indep_util_0" are the following 3 values: 0_8mmol_kg_h, 0_4mmol_kg_h, 0_0mmol_kg_h.*

### 3.1.2 Edges

The relation between the variables are like this:

- For i = 1 to i = 23, there are probability matrices:
  - 1. P( cho_i — meal_i, cho_bal_i-1 ),
  - 2. P( bg_i — bg_i-1, tot_bal_i-1 ),
  - 3. P( activ_ins_i — ins_abs_i, ins_sens ),
  - 4. P( gut_abs_i — cho_i ),
  - 5. P( renal_cl_i — bg_i ),
  - 6. P( ins_indep_util_i — bg_i ),
  - 7. P( ins_dep_util_i — bg_i, activ_ins_i ),
  - 8. P( glu_prod_i — activ_ins_i, bg_i ),
  - 9. P( ins_indep_i — renal_cl_i, ins_indep_util_i ),
  - 10. P( ins_dep_i — ins_dep_util_i, glu_prod_i ),
  - 11. P( endo_bal_i — ins_indep_i, ins_dep_i ),
  - 12. P( cho_bal_i — cho_i, gut_abs_i ),
  - 13. P( basal_bal_i — gut_abs_i, endo_bal_i ),
  - 14. P( tot_bal_i — basal_bal_i, met_irr_i ).

  So there are 25 edges in each of the subnetwork 1 to 23

- For i = 0, relation 2 is canceled, and relation 1 is changed to be:
  - 1. P( cho_0 — meal_0, cho_init ),

  So there are 23 edges in subnetwork 0

- For i = 24, there are only relation 1 and 2, so there are 4 more edges. So totally there are 25*23+23+4=602 edges in the whole network.

  There are 76 root nodes which are: cho_init, ins_sens, meal_i(i=0–24), bg_0, ins_abs_i(i=0–23), met_irr_i(i=0–23) There are 2 evidence nodes which are: cho_24, bg_24

## 3.2 Experiment

### 3.2.1 Task

My task is to compare the performances of these three different algorithms: Bucket-Elimination MPE, MiniBucket-Elimination approximation MPE, and simple greedy MPE.

### 3.2.2 Testing tool

Here I used the software developed by Irina Rish to test my network.

The file describes the network in .bif file, while the input file required by the testing software is .erg file. These two formats are quite different. So first of all, I have to convert the .bif file to .erg file. Because the file size is very large (3.1MB), so manually conversion is impossible. I wrote some little .c programs and awk script files to convert the file to the correct format such that I can get a reliable input file.

### 3.2.3 Experiment design

The experiment configuration file required by the testing tool includes the following parameters:

- EXPERIMENTS=15

  *which is the number of trials*

- RUN COMPLETE ELIMINATION? 1

  *which means I need to test both exact and approx algorithm*

- PARAMETER I: 1 6 1

  *which are min, max and step values of variables number in one mini bucket*

- PARAMETER M: 0 0 1

  *which are min, max and step values of functions number in one mini bucket*

- EVIDENCE NODES=0

  *which is number of evidence nodes. I tested several different cases with different evidence nodes number: 0, 50, 100, 200, 300, 413*

So basically, I compared the performances of exact Elim-MPE, approximation Elim-MPE, and simple greedy MPE with different i and different evidence nodes number.

### 3.2.4 Experiment results

Here are the results of my testing:

- Experiment #1:
  - number of evidence nodes = 0
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = 3.673159e-037, time = 279.7sec
  - greedy-MPE = small(*), time = 2.7sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(**) |
|---|---|---|---|---|
| i = 1 | 1.22645e-034 | 2.5 | 2 | 243 |
| i = 2 | 1.22645e-034 | 2.5 | 2 | 243 |
| i = 3 | 1.22474e-034 | 3.0 | 2 | 247 |
| i = 4 | 1.21018e-036 | 38.4 | 3 | 344 |
| i = 5 | 3.67316e-037 | 246.5 | 4 | 347 |
| i = 6 | 3.67316e-037 | 292.5 | 5 | 347 |

(*)small: means the number is so small that it is out of the limit of the machine.

(**) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

- Experiment #2:
  - number of evidence nodes = 1 (generated randomly by the program)
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = 3.360226e-040, time = 292.8sec
  - greedy-MPE = small(*), time = 2.3sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(*) |
|---|---|---|---|---|
| i = 1 | 4.89477e-037 | 2.2 | 2 | 238 |
| i = 2 | 4.89477e-037 | 2.2 | 2 | 238 |
| i = 3 | 2.45807e-037 | 2.5 | 2 | 246 |
| i = 4 | 9.76234e-040 | 78.1 | 3 | 340 |
| i = 5 | 3.36023e-040 | 211.4 | 4 | 345 |
| i = 6 | 3.36023e-040 | 300.5 | 5 | 345 |

(*)small: means the number is so small that it is out of the limit of the machine.

(*) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

- Experiment #3:
  - number of evidence nodes = 2 (generated randomly by the program)
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = 1.908847e-043 time = 176.1sec
  - greedy-MPE = small(*), time = 2.5sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(*) |
|---|---|---|---|---|
| i = 1 | 3.23031e-037 | 2.2 | 2 | 237 |
| i = 2 | 3.23031e-037 | 2.2 | 2 | 237 |
| i = 3 | 9.9691e-038 | 2.5 | 2 | 245 |
| i = 4 | 1.15471e-040 | 37.1 | 3 | 331 |
| i = 5 | 1.90885e-043 | 271.8 | 4 | 297 |
| i = 6 | 1.90885e-043 | 199.4 | 5 | 297 |

(*)small: means the number is so small that it is out of the limit of the machine.

(*) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

- Experiment #4:
  - number of evidence nodes = 3 (generated randomly by the program)
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = 3.410992e-046 time = 153.4sec
  - greedy-MPE = small(*), time = 2.3sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(*) |
|---|---|---|---|---|
| i = 1 | 6.12679e-039 | 2.4 | 2 | 239 |
| i = 2 | 6.12679e-039 | 2.2 | 2 | 239 |
| i = 3 | 2.16837e-039 | 2.4 | 2 | 247 |
| i = 4 | 1.66845e-042 | 33.6 | 3 | 326 |
| i = 5 | 3.41099e-046 | 197.9 | 4 | 256 |
| i = 6 | 3.41099e-046 | 202.0 | 4 | 256 |

(*)small: means the number is so small that it is out of the limit of the machine.

(*) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

- Experiment #5:
  - number of evidence nodes = 4 (generated randomly by the program)
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = small(*), time = 154.0sec
  - greedy-MPE = small(*), time = 2.4sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(*) |
|---|---|---|---|---|
| i = 1 | small(*) | 2.2 | 2 | 243 |
| i = 2 | small(*) | 2.2 | 2 | 243 |
| i = 3 | small(*) | 2.4 | 2 | 246 |
| i = 4 | small(*) | 31.3 | 3 | 335 |
| i = 5 | small(*) | 197.0 | 4 | 348 |
| i = 6 | small(*) | 200.5 | 4 | 348 |

(*)small: means the number is so small that it is out of the limit of the machine.

(*) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

- Experiment #6:
  - number of evidence nodes = 5 (generated randomly by the program)
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = small(*), time = 148.8sec
  - greedy-MPE = 0 (too small to be printed), time = 2.3sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(*) |
|---|---|---|---|---|
| i = 1 | small(*) | 2.2 | 2 | 235 |
| i = 2 | small(*) | 2.2 | 2 | 235 |
| i = 3 | small(*) | 2.4 | 2 | 242 |
| i = 4 | small(*) | 31.6 | 3 | 330 |
| i = 5 | small(*) | 192.0 | 4 | 310 |
| i = 6 | small(*) | 197.7 | 4 | 310 |

(*)small: means the number is so small that it is out of the limit of the machine.

(*) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

- Experiment #7:
  - number of evidence nodes = 10 (generated randomly by the program)
  - width = 4
  - induced width = 5
  - m =0
  - exact Elim-MPE = small(*), time = 145.1sec
  - greedy-MPE = 0 (too small to be printed), time = 2.3sec

| | approx Elim-MPE | Time(second) | w* | Tuple Distance(*) |
|---|---|---|---|---|
| i = 1 | small(*) | 2.2 | 2 | 240 |
| i = 2 | small(*) | 2.1 | 2 | 240 |
| i = 3 | small(*) | 2.4 | 2 | 242 |
| i = 4 | small(*) | 26.2 | 3 | 332 |
| i = 5 | small(*) | 187.2 | 4 | 343 |
| i = 6 | small(*) | 193.6 | 4 | 343 |

(*)small: means the number is so small that it is out of the limit of the machine.

(*) Tuple Distance: I define tuple distance as the total number of nodes in the approx MPE solution tuple, which has different value with that in the exact MPE solution tuple.

## 3.3 Result Analysis

- MPE accuracy: The approx elim-mpe has better results when i is bigger. When i is equal to the induced width, approx MPE has the same result with that of exact MPE. This is right, because in this case, minibucket becomes the original bucket. While the mpe accuracy of greedy approx mpe is not available because the value of mpe is too small to be printed out.

- Solution tuple accuracy: The solution tuples from the approx elim-mpe are not good, the number of nodes which has different value compared with the exact solution is about 200 ~300. That's really bad. The solution tuples of greedy-mpe is not available from this program.

- The time of simple greedy-mpe is the shortest one, the time of approx elim-mpe is also short. Both of these two approximate algorithms have much shorter time than exact elim-mpe, and they are the same order when approx MPE is close to exact MPE. The simple greedy mpe is the fast one.

## 3.4 Acknowledgment

Thanks to Uffe Kjaerulff for making this network model available on line. Professor Dechter gave me good suggestions about the project. Special thanks to Irina Rish who kindly provided me the testing software and helped me a lot when I modified the program to make it be able to test my benchmark.

## 3.5 Reference

- http://www-nt.cs.berkeley.edu/home/nir/public_html/Repository/Diabetes.htm

## A Comparison of Logic Sampling with Exact Inference on the Insurance Domain

Stephen D. Bay
Department of Information and Computer Science
University of California, Irvine
sbay@ics.uci.edu

In this project, I will compare and contrast logic sampling for belief evaluation of Bayesian Networks with the results from exact inference on the Insurance domain.

# 4 The Insurance Domain

The purpose of the insurance domain bayesian network is to evaluate the expected cost of insuring an automobile driver. The network is shown in Figure 5: there are 12 observable nodes, 12 hidden nodes, and 3 query nodes.

The observable nodes correspond to questions that a typical insurance agent would ask of any prospective client; for example, age, make of car, driving history, and so on. The hidden nodes correspond to variables that we cannot actually observe, but can reasonably infer. For example, we cannot directly observe a clients driving skill, but can infer this from other observed variables such as age, or driving history (number of accidents, traffic violations, etc). The query nodes represent the expected cost of insuring a driver divided into (1) property cost, (2) medical costs, and (3) legal costs.

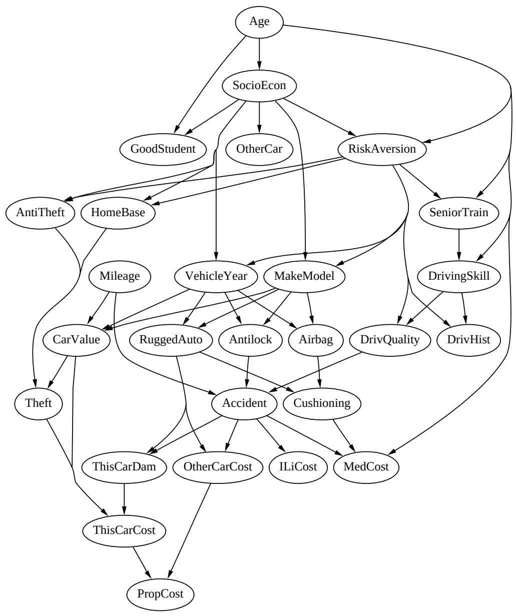


Figure 5: The Insurance Domain

Variables that are continuous have been discretized for this network. For example, cost variables are represented by the values {Thousand, TenThou, HundredThou, Million}. Similarly, mileage is discretized into {FiveThou, TwentyThou, FiftyThou, Domino}.

Note that the insurance network is not a real network actually in use by an insurance company. Many of the variable have been greatly simplified: for example, ModelYear has only the values Current and Older.

### 4.1 Algorithms

For this project, I implemented belief inference using simple logic sampling as follows:

1. generate tuples randomly according to the distribution of the Bayesian Network

2. count tuples relevant to query and from the counts, estimate probabilities of interest

Nodes were not clamped in any fashion to match the evidence. Evidence was only used to select the tuples relevant to the conditional probability query.

To get the *true* belief of a variable, I used the JavaBayes program by F. Cozman. JavaBayes uses the Bucket Elimination algorithm.

### 4.2 Experiments

I ran two experiments to evaluate the logic sampling algorithm: (1) inference with no observed variables, (2) inference with observations of a typical insurance client (myself). For both experiments the query node was PropCost, the estimated property cost. For logic sampling, I made 5 trials each with 1,000,000 tuples.

The sampling simulator can generate slightly more than 500 tuples/second on the insurance domain. Thus, each trial of 1,000,000 tuples took about 30 minutes to execute. I did not spend any effort to optimize the code, so tuples could probably be generated much faster with some effort. JavaBayes was very fast on this domain, generating responses to queries in usually less than 1 second.

#### 4.2.1 No Evidence

JavaBayes and logic sampling return the following belief vector for the node PropCost:

| | Thousand | TenThou | HundredThou | Million |
|---|---|---|---|---|
| true | 0.5629456744 | 0.3151876038 | 0.1050702190 | 0.0167965028 |
| sampling | 0.5628922000 | 0.3156036000 | 0.1047520000 | 0.0167522000 |

JavaBayes used the ordering in Table 1. The sampling results are from using all 5 million tuples from the 5 trials.

Figure 6 shows the average error in the belief vector as measured by norm-2 as the number of tuples varies. Figure 7 shows the individual trials.

#### 4.2.2 Evidence

For this experiment, I entered evidence that would be typical of an insurance client (myself). Table 2 shows the variables observed and their values.

JavaBayes and logic sampling return the following belief vector for the node PropCost:

Table 1: Variable Ordering and Bucket Size for JavaBayes

| Variable | Bucket Size |
|---|---|
| SeniorTrain | 2 |
| DrivingSkill | 2 |
| Age | 4 |
| OtherCarCost | 2 |
| Antilock | 2 |
| Theft | 2 |
| Mileage | 3 |
| SocioEcon | 5 |
| RiskAversion | 1 |
| DrivQuality | 2 |
| VehicleYear | 2 |
| MakeModel | 1 |
| ThisCarDam | 2 |
| CarValue | 2 |
| HomeBase | 1 |
| AntiTheft | 1 |
| RuggedAuto | 2 |
| Accident | 1 |
| ThisCarCost | 1 |
| PropCost | 1 |

| | Thousand | TenThou | HundredThou | Million |
|---|---|---|---|---|
| true | 0.6938398385 | 0.2984643605 | 0.0067535802 | 0.0009422208 |
| sampling | 0.7168260000 | 0.2754238000 | 0.0077501280 | 0 |

JavaBayes used the ordering in Table 3. The sampling results are from using all 5 million tuples from the 5 trials.

Figure 8 shows the average error in the belief vector as measured by norm-2 as the number of tuples varies. Figure 9 shows the individual trials.

Logic sampling with evidence was surprisingly slow because most tuples generated did not meet the conditions for the conditional probability query. On average, only about 132 tuples out of 1,000,000 matched the evidence. Thus on each trial, there were very few tuples to estimate the belief of PropCost.

### 4.3 Conclusions

After implementing logic sampling, I feel the main advantages of this technique are: (1) Logic sampling is easy to program (roughly about 200 lines of C++ code) and hence it is easy to make bug free. In contrast, F. Cozman recently found bugs in the JavaBayes inference algorithm. (2) Logic sampling is an anytime algorithm and can always give an estimate of the current belief state. Finally, (3) logic sampling is very space efficient only requiring enough memory to store the network itself.

Table 2: Evidence

| Variable | Value |
|---|---|
| Age | Adult |
| GoodStudent | False |
| Mileage | FiveThou |
| OtherCar | False |
| VehicleYear | Current |
| SeniorTrain | False |
| MakeModel | FamilySedan |
| DrivHist | Zero |
| Antilock | True |
| Airbag | True |
| HomeBase | Suburb |

The disadvantages are of logic sampling are: (1) it only provides an estimate and not exact inference. (2) sampling provides poor estimation for rare events, and (3) having more evidence slows down inference as more tuples must be discarded (i.e. the tuples do not match the probability conditions for the query).

### 4.4 Source

The latex source for this file is in:

```
/home/sbay/courses/ics275b/report
```

The code for logic sampling is in:

```
/home/sbay/courses/ics275b/bn/sbc
```

Table 3: Variable Ordering and Bucket Size for JavaBayes

| Variable | Bucket Size |
|---|---|
| GoodStudent | 0 |
| VehicleYear | 0 |
| MakeModel | 0 |
| SeniorTrain | 0 |
| HomeBase | 0 |
| OtherCar | 0 |
| DrivHist | 0 |
| SocioEcon | 8 |
| DrivingSkill | 3 |
| RiskAversion | 3 |
| DrivQuality | 2 |
| AntiTheft | 2 |
| RuggedAuto | 3 |
| Accident | 2 |
| ThisCarDam | 2 |
| Theft | 1 |
| CarValue | 2 |
| ThisCarCost | 2 |
| OtherCarCost | 1 |
| PropCost | 1 |

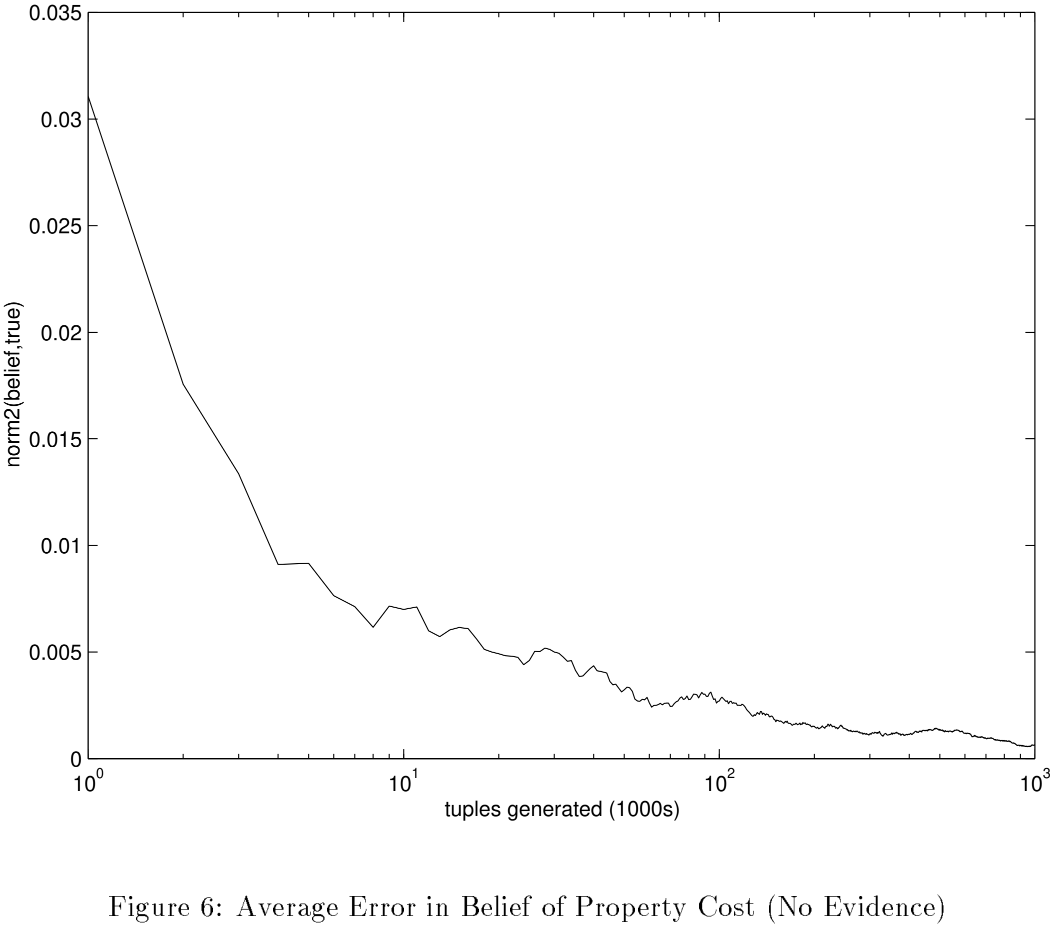


Figure 6: Average Error in Belief of Property Cost (No Evidence)

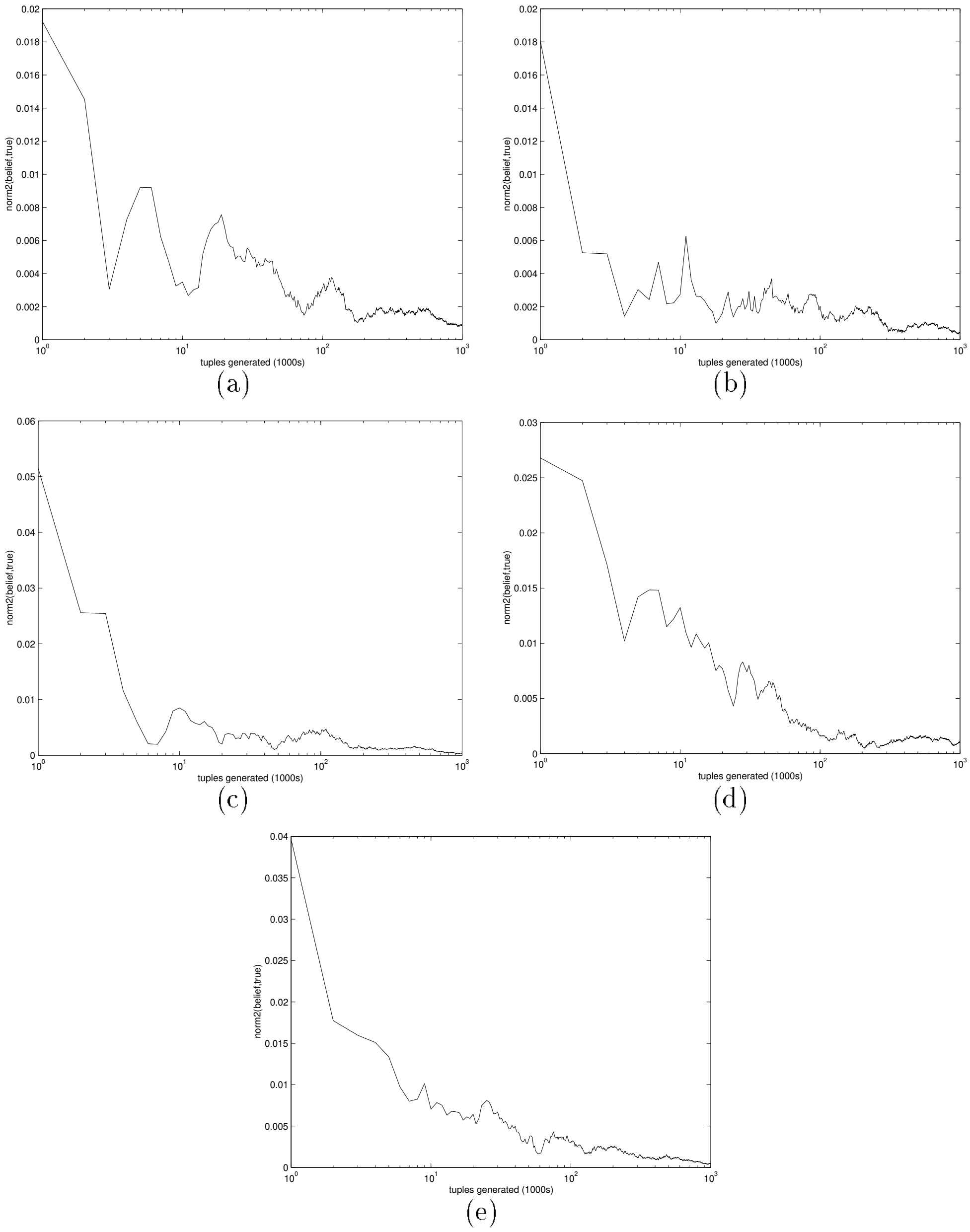


Figure 7: Error in Belief of Property Cost (No Evidence): (a) trial 1, (b) trial 2, (c) trial 3, (d) trial 4, (e) trial 5

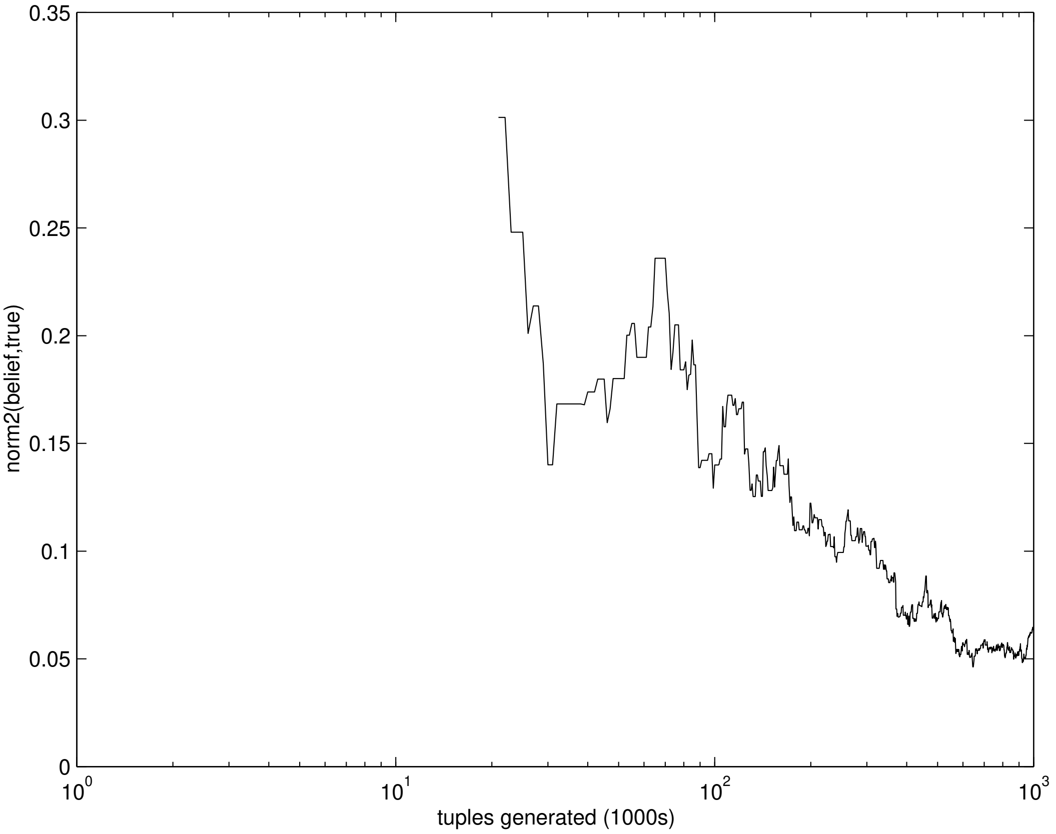


Figure 8: Average Error in Belief of Property Cost (Evidence)

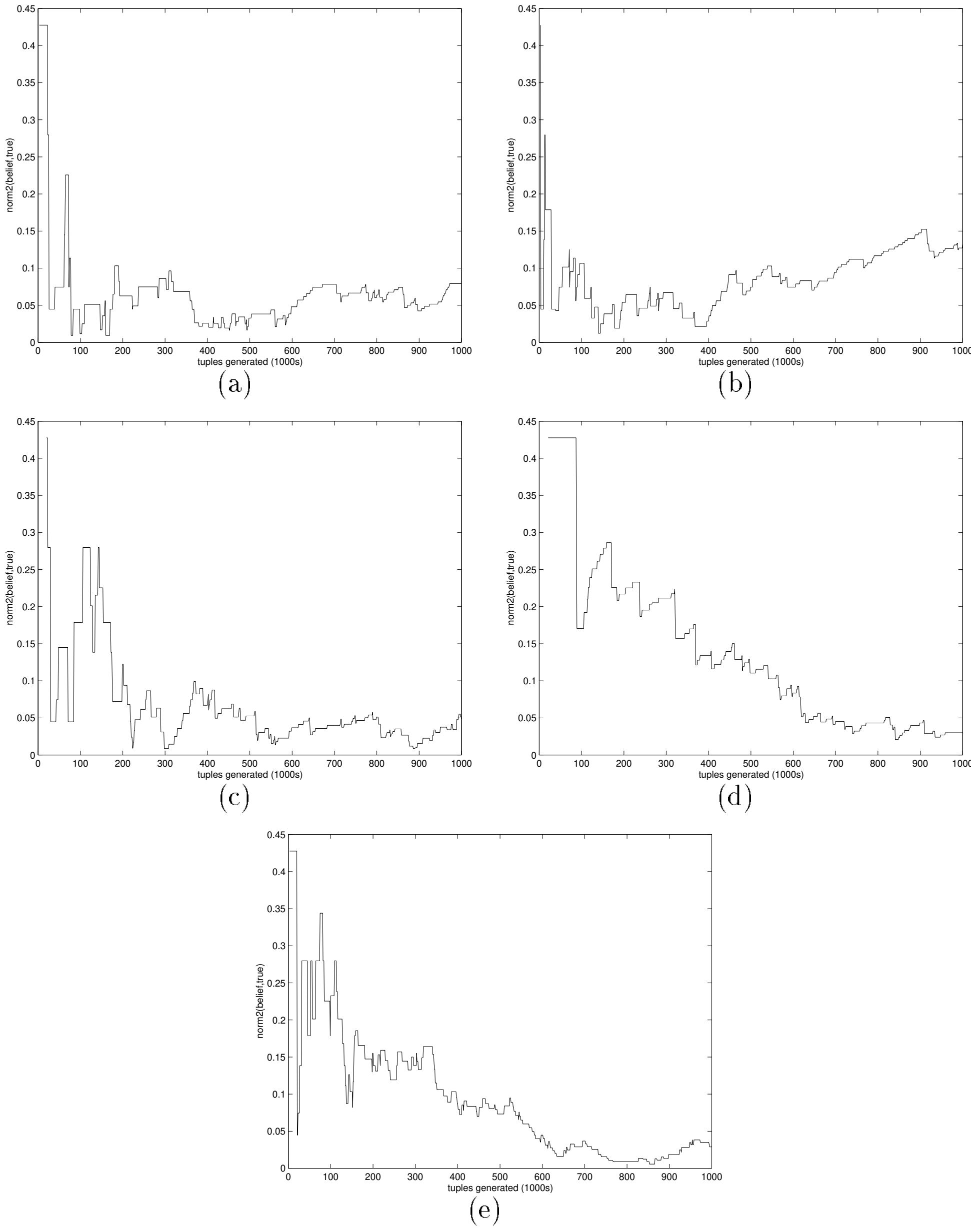


Figure 9: Error in Belief of Property Cost (Evidence): (a) trial 1, (b) trial 2, (c) trial 3, (d) trial 4, (e) trial 5

**An Empirical Study of Simulation-based Inference Methods Applied to HailFinder**

Scott Gaffney
Department of Information and Computer Science
University of California, Irvine
sgaffney@ics.uci.edu

# 5 Introduction

This report describes the project that was attempted for the ICS 275B course on belief networks. The goal of the project was to empirically evaluate the convergence properties for several different simulation-based approximate inference procedures on a bayesian network. The benchmarking network HailFinder was acquired from the Bayesian Network Repository (1998) in order to carry out experiments with simulation-based inference techniques. In the following, we will outline the four different techniques that were included in these investigations and present our empirical results obtained using HailFinder.

## 5.1 HailFinder

HailFinder is a multiply connected bayesian network used to predict severe summer hail storms in Northeastern Colorado. It is of moderate size, consisting of 56 nodes and 64 edges. During normal use, one instantiates a number of *input* nodes to values that correspond to various meteorological observations, and then reads off the effect of this evidence within the network on 3 *forecast* nodes.

The network was built and designed with the help of domain experts to accurately reflect what was thought to be a *correct* casual model for this domain. In this regard, the network may prove to be a valuable research tool in this domain. However, I suspect that many domain experts will be cautious about incorporating HailFinder into their current research, and as such, HailFinder may find that its most important roll will be to serve as a benchmarking tool for bayesian network research.

## 5.2 Simulation and Approximate Inference

It is widely known that the problem of computing exact probabilistic inference in bayesian networks is NP-hard (Cooper, 1987). As Pearl (1988) demonstrated, specific types of network topology (e.g., singly connected networks) allow efficient calculation of probabilistic inference; however, in more general networks we are still *restricted* by the results of Cooper.

To deal with this problem, many have turned to simulation-based approximate inference procedures. These methods compute probabilistic inference by estimating the probability of an event as the frequency of its occurrence in a "random" sample from the considered network. These methods are popular because their complexity scales linearly with the number of nodes in the network, thus avoiding the Cooper result mentioned above. However, we must pay a penalty for this nicety, we are only getting approximate answers instead of exact ones. Although approximate solutions aren't quite exact, some simulation methods can guarantee our result will achieve any accuracy desired by setting our sample size appropriately (however, now our sample size may explode exponentially in size).

If the network under consideration does not allow exact inferential computation, then simulation-based methods can be a valuable tool.

## 5.3 Simulation Techniques Performed on HailFinder

Four different simulation algorithms were selected to be used in our tests on HailFinder: *logic sampling*, *backward sampling*, *likelihood weighting*, and *self-importance sampling*. These methods will be outlined in the next four subsections.

### 5.3.1 Logic Sampling

Logic sampling is a very simple technique which propagates partial sampling information forward through the network until each node has sampled a value. These sampled values constitute a single event and this event is recorded. The process is repeated until we have sampled $N$ events, where $N$ is our predetermined sample size.

In this case, the partial sampling information propagated forward through the network is just the sampled values from the parents of a node. The current node then uses this information to sample a value for itself from its conditional probability table. Nodes which have no parents sample their value from their prior distribution.

The posterior probability distribution associated with any node in the network is then estimated based on the frequency that the node's values appear in the sample, constrained by the joint appearance of any evidence nodes that we wish to condition on. If we are interested in calculating the $P(x|e)$, where $x$ is some subset of the nodes in the network, and $e$ is the observed evidence, then we base our approximation on the frequency that $x$ appears in the subset of our sample where $e$ also appears.

This means that for any sample, we must *throw away* sampled events which do not agree with our evidence set. Clearly, as we add more nodes to our evidence set, the likelihood that randomly sampled events will agree with this set decreases; and thus will need to sample more and more events to get good approximations. This can be a major problem for logic sampling as we will see from our experimentations.

### 5.3.2 Backward Sampling

Backward sampling is motivated by the failings of logic sampling, namely its problem with handling evidence. Instead of ignoring our evidence set while we build up our sample from the network, and then removing those events which do not agree with our evidence, we just simply fix each evidence node before we sample, and propagate these instantiations *backward* through the network. In order to facilitate this process we can simply apply the Bayes' inversion formula to sample a parent when it has an observed child node. Otherwise, we sample in the normal way just as in logic sampling. This technique is an interesting extension to logic sampling and has been included here to judge its merit.

### 5.3.3 Likelihood Weighting

Likelihood weighting has been proposed as another extension to logic sampling in order to deal with its problems in handling evidence nodes (Fung and Chang 1990; Shachter and Peot 1990). This technique is similar to backward sampling in that it fixes evidence nodes to their observed values before it begins sampling, but it differs in the way that it *counts* frequencies when estimating posteriors.

The sampled events are generated in the same manner as logic sampling—sampling from the prior nodes and on down through the network—except that evidence nodes are never sampled but are just fixed to their values as stated above. If we denote the set of unobserved nodes as $Y$, and the set of evidence nodes as $E$, then we will refer to a particular instantiation of these nodes as $y$, and $e$, respectively. Furthermore, we will denote a single node in the above sets as $Y_i$ (or $y_i$) and $E_i$ (or $e_i$). Finally, we will refer to the $j$th sampled event as $(y^{(j)}, e)$, where the nodes in $E$ are always fixed to the observed evidence $e$.

Then for each sampled event $(y^{(j)}, e)$, we calculate the probability of the particular evidence set $e$, given the sampled values of the *state* nodes $y^{(j)}$. This can be calculated as

$$P(e|y^{(j)}) = \prod_{e_i \in e} P(e_i|\mathrm{par}(e_i)) \,, \tag{1}$$

where $\mathrm{par}(e_i)$ represents the current values for the parents of node $e_i$. Each event in the sample has associated with it a number defined by Eq. (1), and it is called its likelihood weight.

One then calculates the posterior of an event $x$ based on the sum of the likelihoods defined in Eq. (1), where $x \subseteq y$ appears, divided by the total number of samples. In other words, instead of simply counting up the number of times $x$ appears in the sampled events, as in logic, and backward sampling, we take into account the likelihood that the event which $x$ appears in, actually occurs, given the evidence set. The justification for this procedure is simple and can be found in (Fung and Chang 1990; Shachter and Peot 1990).

### 5.3.4 Self-Importance Sampling

This method described in (Shacther and Peot 1990) exactly mimics likelihood weighting except that it addresses the problem of disproportionate sampling of more probable events. It makes use of an "importance" distribution to address this problem, which essentially re-estimates the posterior for each node in between each sampled event so as to *guide* the sampling of events. This means that the distribution which determines the likelihood weights changes during the sampling procedure. This technique may only prove useful in specific circumstances, but was included in the tests here to see how it might perform in general.

## 5.4 Experimental Methodology

The goal of this project was to empirically investigate the convergence properties of the selected simulation methods on the HailFinder bayesian network, under different evidence sets, as the sample size increases. The sample size determines the number of sampled events that a simulation method is allowed to base its approximations on. The larger the sample

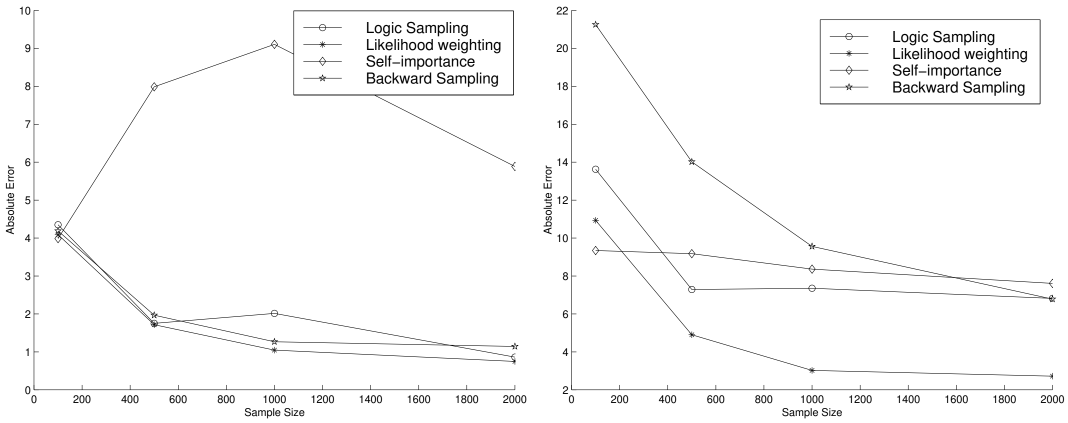


Figure 10: Absolute error given evidence set 1 (on the left), and absolute error given evidence set 2 (on the right).

size, the better chance that a method has of reporting better approximations to the exact posterior probabilities.

To facilitate the goal of this project a number of experiments were carried out using the HailFinder network. These experiments were conducted using the GeNIe bayesian network research and development software program (Decision Systems Laboratory 1998). This program has functions which run all of the above mentioned simulation methods, and more. A number of the simulation methods were also separately coded in MATLAB for this project, but the data from the test results was produced using GeNIe.

Since we want to look at the convergence properties of each of the simulation methods, we report the error that a simulation method achieves by calculating the sum of the absolute difference between its estimates for the posteriors of each of the three forecast nodes (See Section 5.1) and the exact probabilities from the join-tree algorithm. Since each run of a simulation method will return different results for exactly the same input, we then further report the mean absolute error over 15 different runs on the same input for each simulation method. We will refer to this value as the mean absolute error in one trial (i.e., a trial consists of 15 runs).

Furthermore, we carry out 4 different trials for each simulation method, each time with increasing sample size per the following schedule: 100, 500, 1000, 2000. We will refer to this as a single experiment (i.e., an experiment consists of 4 trials for each of 4 simulation methods).

Finally, we report the results on 3 different experiments. Each experiment consists of selecting a different prespecified set of evidence nodes, and performing the above trials using this evidence set. The significance of the selected evidence sets will be described below.

### 5.5 Results

In the first experiment we instantiated a small set of evidence nodes (6 nodes) that have no ancestors. This is the simplest case among the experiments, and should allow us to see how each technique performs under only the simplest of circumstances. In the left chart of Figure 10, we see the mean absolute error reported for each of the four simulation methods,

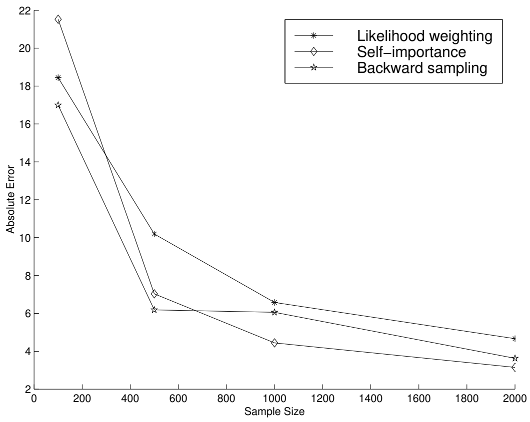


Figure 11: Absolute error given evidence set 3.

as the sample size increases. We see that logic, and backward sampling perform almost identical to likelihood weighting, but that self-importance sampling behaves very strangely. In fact, for this evidence set, self-importance sampling behaved very erratically over many different attempts of this experiment. It appears that it misjudges its "importance" distribution on this evidence set, and may not recover until its swamped with a much larger sample size.

The right side of Figure 10 reports the mean absolute error obtained for experiment two. In this experiment, a small set of evidence nodes were instantiated (5 nodes), but they lived in the middle of the bayesian network, where they might cause some simulation methods to break down, and others to excel. In the figure we see that likelihood weighting pulls away from the crowd as soon as the sample size reaches 500 or so; logic sampling begins to feel its problems with evidence handling, while backward sampling starts off really bad and looks as if it might meet likelihood weighting in the not too distant future. As for self-importance sampling, it looks like its strategy doesn't perform too good in this case either. It is interesting to note that plain old logic sampling performs better than backward sampling early on, even though backward sampling handles the evidence nodes specially.

Figure 11 shows the mean absolute error reported for the final experiment. The evidence set for this experiment consists of 28 nodes, of which 15 are simple prior nodes (i.e., they don't have any parents). The first thing that one notices from the figure is that logic sampling is missing. This is because, for the most part, it did not return any estimate at all. That is to say, the specified evidence set was rarely ever generated by logic sampling, and thus it could not make an approximation. This firmly underscores its failings with evidence handling.

The other three simulation methods all performed admirably in this experiment. Self-importance sampling seemed to excel under this extensive evidence set; it definitely achieved its best performance under this experiment. Again, we see that the interesting approach of backward sampling pays off when there are many evidence nodes in the network, whereas logic sampling doesn't even return any answer at all. Lastly, likelihood weighting doesn't seem to miss a beat, although it does appear that self-importance sampling improved upon it.

### 5.6 Final Remarks

We have attempted to look at the convergence properties of four different simulation techniques on the HailFinder bayesian network. By employing three different sets of evidence nodes, we were able to glimpse the relative effectiveness of each technique when presented with different size sets of evidence nodes, located in different areas of the network (e.g., prior nodes, *output* nodes, etc.).

The results suggest that likelihood weighting gives the most consistent results over all of the evidence sets, never veering to far away from the "top" spot. Backward sampling, an interesting extension to logic sampling, seemed to perform as advertised when presented with large amounts of evidence nodes, which was its initial motivation. However, it might not perform as well as logic sampling if the network contains very few instantiated evidence nodes. Logic sampling was revealed to be unable to return estimates if the evidence set is large, which is its biggest failing. And the final simulation technique, self-importance sampling, gave erratic results most of the time, but seemed to perform really well when the evidence set was expanded to large amounts of nodes. In fact, it out-performed likelihood weighting in this case.

## References


Bayesian Network Repository (1998). "http://www-nt.cs.berkeley.edu/home/nir/public_html /Repository/index.htm."

Cooper, G. (1987). Probabilistic inference using belief networks is NP-hard. *Report KSL-87-2*, Medical Computer Science Group, Stanford University.

Decision Systems Laboratory (1998). *GeNIe: A graphical network interface.* University of Pittsburgh. "http://www2.sis.pit.edu/~genie/."

Fung, R., & Chang, K. (1990). Weighting and integrating evidence for stochastic simulation in bayesian networks. In M. Henrion, R.D. Shachter, L.N. Kanal, & J.F. Lemmer (Eds.) *Uncertainty in artificial intelligence 5.* North Holland: Elsevier Science Publishers.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference.* San Mateo, CA: Morgan Kaufmann Publishers.

Shachter, R.D., & Peot, M.A. (1990). Simulation approaches to general probabilistic inference on belief networks. In M. Henrion, R.D. Shachter, L.N. Kanal, & J.F. Lemmer (Eds.) *Uncertainty in artificial intelligence 5.* North Holland: Elsevier Science Publishers.

# Report on the Project Assignment "Numerical Investigation of the Maximum Probable Explanation (MPE) Computing Algorithms for Belief Networks on the ALARM domain"

Dmitry Pavlov *

April 7, 1999

## 6 Task Description and Network Parameters

I ran experiments on the ALARM (A Logical Alarm Reduction Mechanism) belief network which is described in detail in [1]. ALARM implements an alarm message system for patient monitoring; the medical knowledge is encoded in a graphical structure connecting 8 diagnoses, 16 findings and 13 intermediate variables. The belief network at hand is an acyclic directed graph (Figure 1), parameters of the network are the following:

1. The number of nodes - 37
2. The number of edges - 46
3. The number of values per node - from 2 to 4
4. The width of the graph - 4
5. Induced width of the graph - 4

As it was mentioned above all nodes in the ALARM belief network can be divided into 3 mutually exclusive and exhaustive categories: diagnostic, intermidiate and measurement nodes. Measurement nodes are for entering evidence, they can be directly measured. Intermediate nodes cannot be directly measured and in general are of little interest, while updating the belief in certain values of diagnostic variables (nodes) is the task at hand. So, the task is to infer diagnostic nodes that will reflect the way our belief in causes of certain manifestations changes given the evidence. One of the possible ways to address this problem is to find the Maximum Probable Explantion (MPE) of the evidence, i.e. the instantiation of all the rest variables that gives the maximum value of the density function encoded by a belief network. (MAP - most probable aposteriori hypothesis or simple belief updating task might be more relavant though).

*Information and Computer Science, University of California at Irvine.

# 7 Algorithms

In experiments the following algorithms were used:

1. *elim-mpe* (bucket elimination for MPE computing, a complete strategy)

2. *mini-buckets approximation* [3] that is dependent on the parameter $i$ that limits the number of variables allowed in the bucket (the number of functions allowed in the bucket was chosen to be arbitrary non-negative). Since induced width of the graph equals 4 this means that in any ordering each variable is connected to no more than 4 parents which in turn means that it only makes sense to vary $i$ between 1 and 4 and assignment $i$ equals 4 corresponds to a complete strategy.

3. *greedy strategy* in which there is no bucket elimination, but only bucket creation and greedy assignment computing. This strategy is incomplete but as expected might give a good initial approximation to a solution to be then used by stochastic GSAT-type techniques.

4. *naive stochastic method* that uses greedy solution as an initial point. Method has one parameter that controls the maximum allowed size of a set of variables to be flipped at a time. The algorithm starts off from the greedy solution (as simulations show timewise it is very cheap) and then in a cycle flips values of $m$ variables. If the assignment after a flip gives higher value of a density it is accepted else the previous assignment is retained. Thus, the worst guranteed accuracy for the method is the accuracy of the greedy solution.

The code used in experiments was written by Irina Rish and slightly modified by me to include naive stochastic simulation.

# 8 Experiments

The purpose of experiments was to compare algorithms described above time- and accuracywise. In order to properly assess behaviour of the algorithms I chose to vary the number of evidence nodes from 0 to 15 (16 is the maximum number of findings for the domain, values used were $0, 2, 5, 10, 15$). For each fixed number $q$ of evidence nodes 100 experiments were run: in each experiment $q$ nodes were randomly selected and random values from the domains were assigned to them. One complete strategy was used *elim-mpe* so that actual mpe value was always known. Then 3 approximate strategies could be used; in *mini-buckets* I also looked at different values of the parameter that controls the number of variables in the bucket (from 1 to 4) and in the naive stochastic method a fixed number of iterations (100) was used in attempt to improve greedy solution. Parameter $m$ - the size of the flip set - was varied for the naive stochastic method from 1 to 15 (values were $1, 2, 5, 10, 15$).

Accuracy was assessed based on the ratio of the mpe returned by the algorithm to the mpe value returned by *elim-mpe* (which is 1 for *elim-mpe*), so that the lesser the ratio is the worse algorithm is. It made no sense to compare variable assignments since in principal solution in terms of assignment could be non-unique.

All timing information was computed in the similar fashion as the ratio of time in seconds taken by a specific algorithm to the time taken by *elim-mpe*. Thus, if this ratio is less than 1 then algorithm is faster than *elim-mpe*.

# 9 Results

First of all, it is worth mentioning that the problem at hand is really simple, with graph having 37 nodes and low connectivity (induced width of only 4) it takes almost no time to get an exact solution by *elim-mpe*.

Below is 3 tables that give a complete information on times taken by different algorthms. Table 1 gives the average running times in seconds taken by the complete algorithm *elim-mpe* to compute the mpe for different sizes of the evidence set. Tables 2 and 3 give the ratios of the time taken by the greedy algorithm and 100 iterations of the naive stochastic method correspondingly to time taken by *elim-mpe*. Finally, Table 4 gives ratio result for *elim-mpe* and *mini-buckets* algorithms for different values of parameter $i$.

Table 4: Average *elim-mpe* times for different sizes of the evidence set.

| Size of the evidence set | 0 | 2 | 5 | 10 | 15 |
|---|---|---|---|---|---|
| Aver. *elim-mpe* Time, s | 0.0838 | 0.101 | 0.054 | 0.0516 | 0.0478 |

Table 5: Average greedy algorithm times for different sizes of the evidence set.

| Size of the evidence set | 0 | 2 | 5 | 10 | 15 |
|---|---|---|---|---|---|
| Aver. *elim-mpe* Time, s | 0.0076 | 0.008 | 0.0074 | 0.002 | 0.0078 |

Table 6: Average naive stochastic algorithm times (for 100 cycles) for different sizes of the evidence and flip sets.

| Size of the evidence set | 0 | 2 | 5 | 10 | 15 |
|---|---|---|---|---|---|
| (1) | 0.804 | 1.542 | 0.52 | 0.911 | 0.558 |
| (2) | 4.114 | 5.409 | 2.931 | 2.564 | 2.5 |
| (3) | 17.823 | 13.8 | 6.579 | 7.365 | 6.442 |

(1), (2), (3) - Average Time of 100 Iterations of Naive Stochastic Algorithm, Flip Set Size is repectively 2, 5, 10.

Table 7: Average *mini-buckets* algorithm times ratios to the time of *elim-mpe* for different sizes of the evidence set and parameter $i$.

| Size of the evidence set | 0 | 2 | 5 | 10 | 15 |
|---|---|---|---|---|---|
| (1) | 0.93 | 1.36 | 1.78 | 2.06 | 1.3 |
| (2) | 1.1 | 1.26 | 2.12 | 1.55 | 1.37 |
| (3) | 1.29 | 1.45 | 2.06 | 1.63 | 1.69 |
| (4) | 1.17 | 1.29 | 1.93 | 2.2 | 1.54 |

(1), (2), (3), (4) - Average Time Mini-Buckets, s, for respectively $i = 1$, $i = 2$, $i = 3$, $i = 4$.

Tables $1 - 4$ prove that the best timing results for ALARM network are given by the greedy strategy, then complete technique, with *mini-buckets* strategy being the third, although the difference in time between the three is negligible (my guess would be that, for instance, roughly 1.5 times worse performance of *mini-buckets* compared to the complete strategy is the result of the overhead which for larger problems should cease to play any role). Naive stochastic algorithm is the worst, with decaying performance when the flip set size increases. In view of results above, an interesting comparison would be how the algorithms perform accuracy-wise, this could help pick up the best for the domain among the ones considered. (Note that although the time is crucial for medical diagnosis problems like ALARM, it is also the case that solution should be as accurate as possible, revealing the true reason for the finding observed. Thus, the combination of high accuracy and low diagnostic time are imperative).

Empirical results on accuracy of different algorithms prove that among all approximating techniques considered *mini-buckets* scheme is the undoubtful leader, moreover its performance on accuracy for the fastest $i = 2$ is only $1 - 2$ orders of magnitude worse than true mpe value (with all other algorithms being roughly ten orders worse! Naive stochastic algorithm is only able to slightly improve accuracy of the greedy algorithm, but it spend a lot of time and the resulting solution is still far from the desired one!). As expected $i = 4$ gives an exact solution. Unfortunately, ratios of upper and lower bounds to mpe are not monotone (see column evidence set size 15), which again [3] proves that it is difficult to automate the process of selecting $i$ a priori and give the guaranteed bounds for mpe. It is

Table 8: Average accuracy ratios (Upper/MPE and Lower/MPE) for *mini-buckets* algorithm for different sizes of the evidence set.

| Size | 0 | | 2 | | 5 | | 10 | | 15 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Parameter | U/M | M/L | U/M | M/L | U/M | M/L | U/M | M/L | U/M | M/L |
| $i = 1$ | 2.48 | 9 | 2.41 | 9 | 31 | 49.8 | 5.69 | 28.7 | 582 | 32.3 |
| $i = 2$ | 2.48 | 9 | 2.41 | 9 | 31 | 49.8 | 5.69 | 28.7 | 582 | 32.3 |
| $i = 3$ | 1 | 9 | 1 | 9 | 4.1 | 13.6 | 2.64 | 28.7 | 1208 | 36.8 |
| $i = 4$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 9: Average accuracy ratios of the greedy and naive stochastic algorithms to accuracy of the complete *elim-mpe* algorithm.

| Size of the evidence set | 0,2,5,10,15- difference negligible |
|---|---|
| Greedy | $1.0E - 11$ |
| Stochastic | $1.0E - 08$ |

interesting that for small sizes of the evidence set upper bound is closer to the true mpe, and for the problems in which a lot of evidence is available lower bound is closer to the true mpe (which is good since we also know solution that gives this bound).

## 10 Discussion

Unfortunately, paper [1] doesn't describe any results on time or accuracy that were obtained, so it is impossible to do systematic comparison. Two algorithms were applied to the network: clustering (join tree) and conditioning along with Pearl's propagation [2]. There is no a word about the way network was coded, what language, processor, amount of memory were used. The only known fact is that "it took 8 minutes to update all 8 diagnostic nodes for each set of measurements" on Macintosh II using Pearls' algorithm and conditioning, which, in view of results I get, I believe is too much, although again it is difficult to judge since no details were provided. An interesting results obtained in [1] is that clustering technique allowed to obtain solution orders of magnitude faster then Pearl's propagation algorithm, with even more superior performance when there were a lot of findings (evidence) available.

While in [1] authors did only belief updating on diagnostic nodes while I considered the task of maximum probable explanation computing. In view of the simplicity of the problem, results on accuracy and time of *elim-mpe* (complete strategy) and *mini-buckets* approximation were almost the same, I would expect that for larger problems *mini-buckets* will retain its good performance on the quality of the solution and give superior timing results, moreover it is in principal possible to make up an anytime algorithm by cycling over its parameter $i$. Algorithms of the GSAT type could give satisfactory approximating results only after a lot of trials or when special heuristics are used (that were not considered here), naive techniques would fail to compete adequately with *mini-buckets* algorithm.

## References

[1] I. Beinlich, G. Suermondt, G. Cooper *The ALARM monitring system: A case study with two probabilistic inference techniques for belief networks.*

[2] J. Pearl *Fusion propagation and structuring in belief networks.* Artificial Intelligence. 1986. 29. Pages 241-288.

[3] R. Dechter, I. Rish *Mini-buckets: a general scheme for approximating inference.* Artificial Intelligence. 1998.