6.825 Project 3: Bayesian Network Learning

Edmond Lau, Lawrence Bush
December 2, 2004

Introduction

In this project, we explore how to learn Bayesian network structures to represent a given dataset. We implement a randomized, first-ascent, hill-climbing algorithm to search the structure space for good structures, and we use the BIC score as well as a Bayesian score based on Dirichlet priors to rate the structures that we find. We also present various performance analyses to evaluate the quality of the two scores, and the performance of the greedy search algorithm given two classes of initial networks. Lastly, we explore a new way to search for Bayesian network structures using a genetic algorithm.

1 Learning Network Structures with Maximum Likelihood Estimation

In this task, we assume that a network structure is provided and implement a procedure to compute the conditional probability tables (CPTs) using the maximum likelihood estimate (MLE).

1.1 Structure and File Formats

We divide up the structure and the data for a given network into two separate files and define a file format for each. A *structure file* contains a list of node descriptions, where each node description consists of the node's name followed by a comma-separated list of the node's parents in square brackets. For example, the structure file for data A looked like:

```
G [B]
B []
M [B, L]
L []
```

We defined the *data file* format to consist of a list of nodes followed by a list of data samples of assignments to those nodes. Thus, the 100 data samples in the data file for A looked like:

```
G     M     B     L
1     1     1     1
1     1     1     1
1     1     1     1
...
(97 lines omitted)
```

1.2 Implementation Detail

Our code parses a given structure file and data file to establish an initial Bayesian network. We then compute the MLE estimate of a given parameter of a given node according to the following equations:

$$\hat{\Theta}_{x|u} = \frac{M[u,x]}{M[u]}$$

$$M[u] = \sum_{X} M[u,x]$$

$\hat{\Theta}_{x|u}$ is one parameter in a given CPT where x is one domain value and u is the assignment of the parent(s). $M[u,x]$ is the counts, given the parents and value x of the node variable domain. $M[u]$ is the total counts, given the parent(s)' assignment over the entire node variable's domain. The pseudocode to compute the CPTs looks as follows:

```
1. cpt = new double[variables.cartesianProductSize()];
2. for each assignment of the variables
      (Union of the parents and this variable)
3.    get counts for this assignment.
4.    cpt[position] = (counts);
5.    cpt = normalize(cpt, node_var, parents)
      (We normalize over the total counts
       given each of the parent assignment.)
```

1.3 Analyses of Bayesian Networks A and B

To test correctness, we applied formatted the structure and data provided in dataA.txt into our own file formats and obtained Bayesian network A illustrated in Figure 1, which has a structure and CPTs that match those provided in the file.

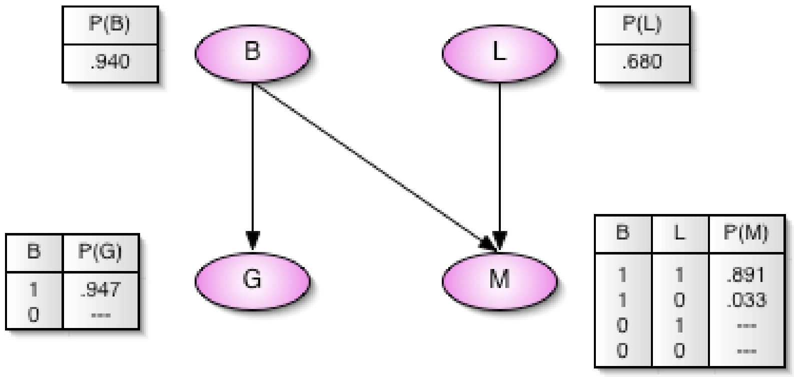


Figure 1: Bayesian network A with maximum likelihood estimates based on Data A. A "---" indicates an undefined probability value based on the data samples.

We also applied our procedure to the structure and data from dataB.txt and obtained Bayesian network B shown in Figure 2.

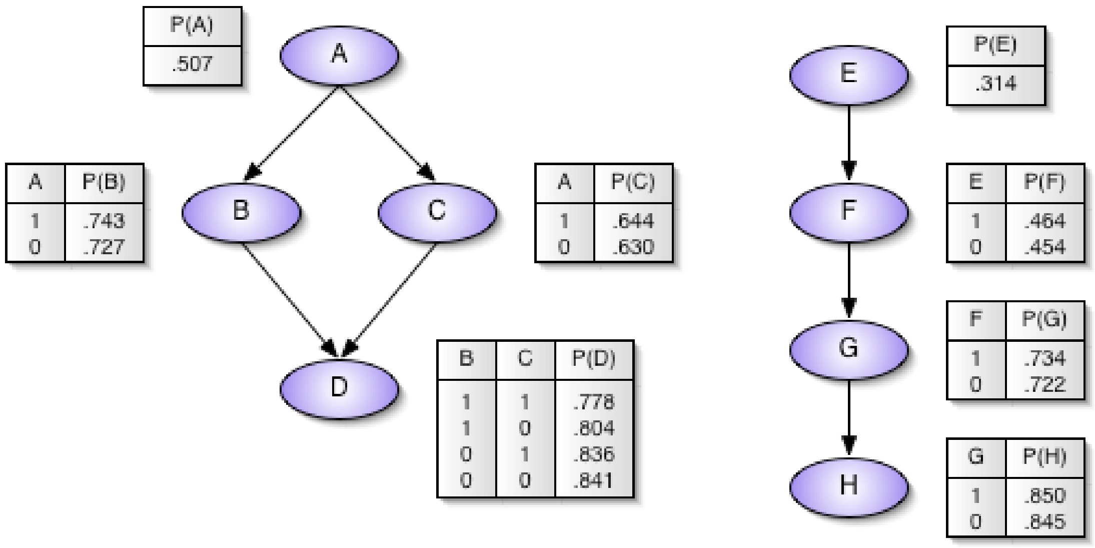


Figure 2: Bayesian network B with maximum likelihood estimates based on Data B.

2 BIC Scoring

Given a network structure and its CPTs, we determined how well the network structure represented the data using the BIC score:

$$S(G:D) = l(\hat{\theta}_G : D) - \frac{\log M}{2} Dim[G]$$

The BIC score consists of two components: the log-likelihood score and the structure penalty.

2.1 Log-likelihood Component of BIC Score

The log-likelihood score of a structure *G* and its parameters θ_G is the logarithm of the probability that the data samples were generated by the network. Mathematically, the likelihood and the log-likelihood are defined as follows:

$$\begin{aligned} &L(\hat{\theta}_G : D) \\ &\quad = \prod_m P_G(x[m] : \hat{\theta}_G) \\ &\quad = \prod_m \prod_i P_G(x_i[m] \mid pa_i[m] : \hat{\theta}_G) \\ &\quad = \prod_i \prod_m P_G(x_i[m] \mid pa_i[m] : \hat{\theta}_G) \\ &\quad = \prod_i P_G(x_i \mid pa_i : \hat{\theta}_G)^{M[x]} \end{aligned}$$

$$\begin{aligned} &l(\hat{\theta}_G : D) \\ &\quad = \log(L(\hat{\theta}_G : D)) \\ &\quad = \sum_i M[x] \log(P_G(x_i \mid pa_i : \hat{\theta}_G)) \end{aligned}$$

In the above equations, $x[m]$ denotes the vector of assignments from a random variable to its value in the m-th sample, $x_i[m]$ denotes the value of the i-th random variable in the m-th sample, and $pa_i[m]$ denotes the values of the parents of the i-th random variable in the m-th sample. In the last step of the likelihood equation, we count up the number of occurrences of each possible assignment $M[x]$ and only compute the probability of that assignment once.

Although the log-likelihood can be computed on a structure with any set of consistent parameters, in determining the quality of a particular network structure G for a given data set, assign to the networks the parameters $\hat{\theta}_G$ that would maximize the likelihood of the data, i.e. the MLE parameters.

2.2 Structure Penalty of BIC Score

The second component of the BIC score is the structure penalty, which penalizes complex structures. The term $Dim[G]$ refers to the number of independent parameters required to specify the CPTs of the Bayesian network G. In the case of binary random variables, the total number of independent variables is specified by

$$Dim[G] = \sum_i 2^{|pa_i|}$$

where $|pa_i|$ denotes the number of parents of node i.

2.3 Implementation Details

The implementation for BICScorer.java provides a method `computeScore(Vector samples, BayesNet bn)` that computes the BIC score from the specified data samples and the Bayesian network `bn`. It calculates the log-likelihood as described in the equation in Section 2.1, essentially by

- counting the number of occurrences in the data of each possible assignment to the random variables;
- computing the log-likelihood of each possible assignment; and
- tabulating a sum of the log-likelihoods weighted by the number of occurrences.

The structure penalty is trivially implemented by the previously mentioned mathematical equation in Section 2.2 by iterating over each node and determining its number of parents.

2.4 Analyses of Bayesian Networks A and B

We computed the BIC score for networks A and B and arrived at the following results:

```
Network A:
Log-likelihood = -189.563
Structure penalty = -26.575
BIC score = -216.139

Network B:
Log-likelihood: -34285.236
Structure penalty: -98.302
BIC score: -34383.537
```

The score and the individual components for Network A match those provided by the project statement. Analyzing the individual score components for Network B, we find that the contribution of the structure penalty pales in comparison to the contribution of the log-likelihood. In fact, the structure penalty accounts for only 0.29% of the total BIC score in the case of Network B, whereas it accounted for 12.30% of the total BIC score in the case of Network A.

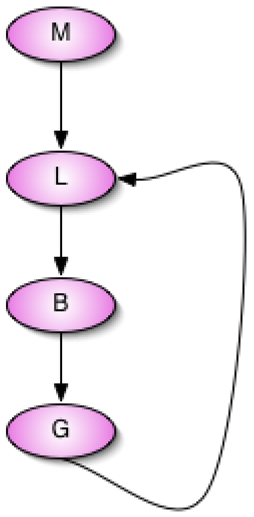
Looking back at the mathematical description of the BIC score in Section 2.1, we discover that the log-likelihood dominates such a large fraction of the BIC score in the case of Network B because the magnitude of the log-likelihood component increases linearly with the number of data samples whereas the structure penalty increases only logarithmically. Because we have 5000 data samples in dataB.txt and only 100 samples in dataA.txt, the results match our mathematical intuitions.

3 Bayesian Network Initialization and Cycle Detection

In this section we discuss generating initial Bayesian Networks. An initial network needs to be generated as a starting point for the structure search algorithms discussed in Section 4. For our implementation, we generated a random and unconnected initial network. Part of the process of generating an initial network is ensuring that it is legal. Here we discuss what constitutes a legal network along with an algorithm for determining if a network is legal. As a last point, we discuss Bayesian Correction, which is a method for handling the mathematical issues related to zero probability events.

3.1 Definition : Legal Network

In this section we define a legal network. A Bayesian Network is a graph structure consisting of nodes (representing variables in our uncertain domain) and arcs (representing dependencies between nodes). The only constraint the arcs in a Bayesian Network is that there must not be any directed cycles. A cycle is a set of edges that form a loop. In a directed cycle all of the arcs point the same way. For example, in the following diagram, once node M is removed, the remaining graph forms a directed cycle.



Likewise, a network is legal if it has no directed cycles. This is also known as a Directed Acyclic Graph (DAG).

3.2 Ensuring Structural Legality

Determining structural legality is similar to finding a topological ordering because every graph with a topological ordering is a DAG and every graph without a topological ordering contains a cycle.

To determine the topological ordering of a graph (G) we find a node in the graph with no incoming edges (a root node). We remove that node and repeat the process until every node has been removed from the graph. If the algorithm terminates, the graph was a DAG. If the graph is not empty, and each remaining node has an incoming edge, then the graph has a cycle. Essentially, this process trims away at the graph until the only part left is the cyclic part.

If we wish to determine the topological ordering, we store each node in the order in which it was removed. The following is pseudo-code for this algorithm:

```
1.      while ( ! empty (G) )
2.      find a node (n) with no incoming edges
3.      remove n from G
```

In our Java implementation, we find a node with no incoming arcs. The `getNextVariable` function does this by storing all of the nodes that have been removed, and finding a node remaining in the graph whose parents have already been removed. If a node has no parents, then it likewise can be removed. The process fails if it cannot find a new node to remove.

The code is as follows:

```
public boolean has_directed_cycles(BayesNet bn) {

      Vector structure_v = this.get_structure_vector(bn);
      HashMap fv_hm = new HashMap();
      while (!structure_v.isEmpty()) {
            Vector next_node_vector =
            (Vector) getNextVariable(structure_v, fv_hm);
            try {
            structure_v.remove(next_node_vector);
            fv_hm.put((String) next_node_vector.get(0), new Object());
            } catch (NullPointerException e) {
                  return true;
            }

      }

      return false;
}
```

3.3 Initialization Methods

For this task, we chose to implement two initialization methods for Bayesian network structures: 1) an initially unconnected network structures and 2) a randomly connected DAGs. Figures 3 and 4 illustrate the unconnected network and a sample randomly connected DAG generated by our code for dataset A; figures 5 and 6 similarly show the unconnected network and a sample random DAG for dataset B.

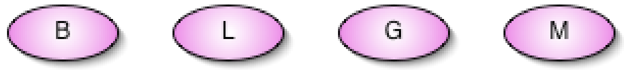


Figure 3: Initial unconnected network for Data A.

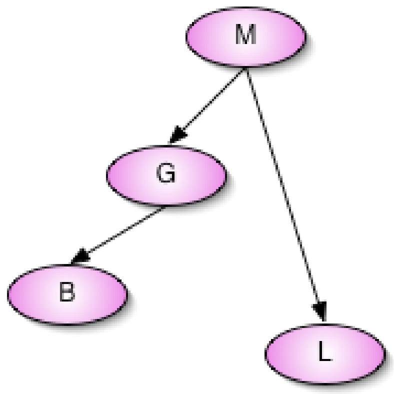


Figure 4: Initial random network for Data A.

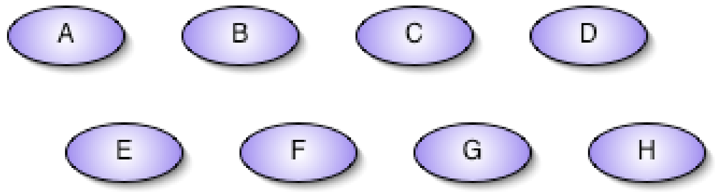


Figure 5: Initial unconnected network for Data B.

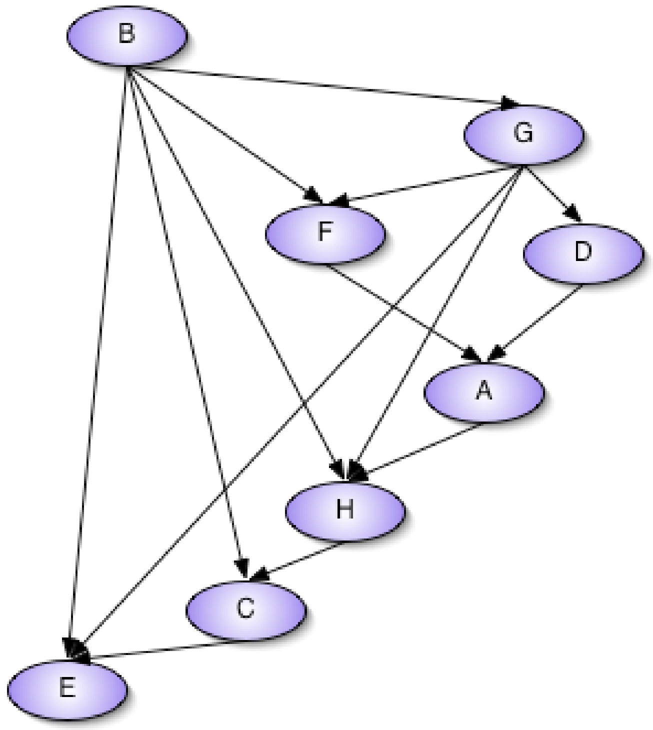


Figure 6: Initial random network for Data B.

3.4 Definition : Bayesian Correction

As a final note, we define Bayesian Correction, which is used in the in all subsequent parts of this project. Bayesian Correction is a method for handling the mathematical issues related to zero probability events. More specifically, it is a way to adjust our parameter estimation to accommodate events that do not occur in the data.

3.5 Compute CPT using Bayesian Correction:

Computing the CPTs using Bayesian Correction is similar to the way they are computed using the Maximum Likelihood Estimator (MLE) except that we assume a uniform (1,1) prior over every estimated parameter. Essentially, we initialize the counts of each possible assignment to 1 and proceed. This is referred to as the "imaginary sample" on page 46 of [2]. It is also known as the Laplace correction to the MLE. This correction is, and has the same inconsistent effect on the estimated probability as, the K2 prior.

We estimate a given parameter of a given node as follows:

$$\hat{\Theta}_{x|u} = \frac{1 + M[u,x]}{\sum_X 1 + M[u]}$$

$$M[u] = \sum_X M[u,x]$$

$\hat{\Theta}_{x|u}$ is one parameter in a given CPT where x is one domain value and u is the assignment of the parent(s). $M[u,x]$ is the counts, given the parents and value x of the node variable domain. $M[u]$ is the total counts, given the parent(s) assignment over the entire node variable's domain. $\sum_X 1$ is equal to the size of the domain of the given node variable.

Pseudo Code:

```
1. cpt = new double[variables.cartesianProductSize()];
2. for each assignment of the variables
      (Union of the parents and this variable)
3.    Get counts for this assignment.
4.    Add 1 to the counts.
5.    cpt[position] = (counts);
6.    cpt = normalize(cpt, node_var, parents)
      (We normalize over the total Bayesian Corrected
      counts given each of the parent assignment.)
```

4 Randomized First-Ascent Hill-Climbing Search for Bayesian Network Structures

To hunt down the best structures to represent datasets A and B, we implemented a randomized, first-ascent, hill-climbing algorithm that takes an initial network structure as input (either an unconnected network or a randomly connected DAG) and then greedily searches for a local maxima. The search space consists of nodes that represent all legal network representations of the data. The neighbors of a node *N* in the search space consists of all nodes that represent identical networks except for a small local operation: either 1) an edge is removed from *N*, 2) an edge is added to *N*, or 3) an edge in *N* is reversed. Each node is associated with a BIC score computed with Bayesian correction, and the goal of the search is to traverse the nodes from an initial network node to find a high scoring node in the structure space.

4.1 Implementation of Randomized Greedy Search

In our implementation, each local operation is represented by an `Operator` object, of type `AddEdgeOperator`, `RemoveEdgeOperator`, or `ReverseEdgeOperator`. Each `Operator` is associated with a particular network structure and a pair of parent and child nodes to represent the edge controlled by the `Operator`. In a graph with *n* nodes, there are *n(n-1)* possible directed edges, for a total of 3*n(n-1)* possible `Operators` at each local step, or iteration.

Not all `Operators` can be applied, however, since we cannot add an edge that already exists, remove an edge that is not in the network, or reverse an edge when either the edge does not exist or an edge in the opposite direction already exists. To handle this possibility each `Operator` has an `okToEvaluate()` method that checks for these cases. A separate `applyOperator()` method applies the local modification to the network. To check whether the modification results in a legal network, we use the procedure implemented in Section 3 to verify that the resultant network is a DAG.

Our randomized, first-ascent hill-climbing search is distinguished from other search methods by the fact that at each local step, we randomize the orderings of all 3*n(n-1)* possible `Operators`, and keep the modification from the first `Operator` that increases the BIC score. We continue until all `Operators` at a node either cannot be applied, lead to illegal network structures, or decrease the BIC score, i.e. until we have reached a local BIC score maximum.

The pseudo-code for the algorithm in GreedySearch.java looks as follows:

```
1.   GreedyHillClimbingSearch() {
2.   bestStructure <- initialNetwork
3.   bestScore <- BICScore(initialNetwork)
4.   progress <- true
5.   while (progress)
6.        progress = false
7.        currentStructure <- bestStructure
8.        operators <- Cartesian product of all Operator types on all
                        possible pairs <parent-node, child-node>
9.        randomize ordering of operators
10.       for each o in operators
11.            if not o.okToEvaluate()
12.                 continue to next o
13.            candidateStructure <- o(currentStructure)
14.            if candidateStructure is a DAG
15.                 score <- BICScore(candidateStructure)
16.                 if (score > bestScore)
17.                      bestScore <- score
```

```
                    bestStructure <- candidateStructure
                    progress = true
                    exit for loop
}
```

4.2 Sample Networks Found By Randomized Greedy Search

In this section, we present some of the network structures and associated CPTs outputted by the randomized greedy search algorithm starting with a randomly connected, legal network. Figures 7, 8, and 9 show three network structures that we found for Data A along with their CPTs generated via maximum likelihood estimation and their BIC scores as computed with Bayesian correction. For comparison, we recalculated the BIC score of the Network A presented in Section 1 with Bayesian correction and arrived at -219.811.

The first two networks shown in Figures 7 and 8 both scored higher than the structure constructed for Network A previously in Section 1 when Bayesian correction is used. Moreover, as our performance analysis in Section 4.4 will show, we are certain with high probability that these three networks are the only three local BIC score maxima for Data A, and that Figure 7 actually shows the highest possible scoring network for the data with Bayesian correction. The original Network A has a higher BIC score of -216.139 without Bayesian correction only because certain types of assignments (the ones indicated by the undefined probabilities in Figure 1) of the random variables are absent in the 100 data samples, either because they are low-probability events or zero-probability events.

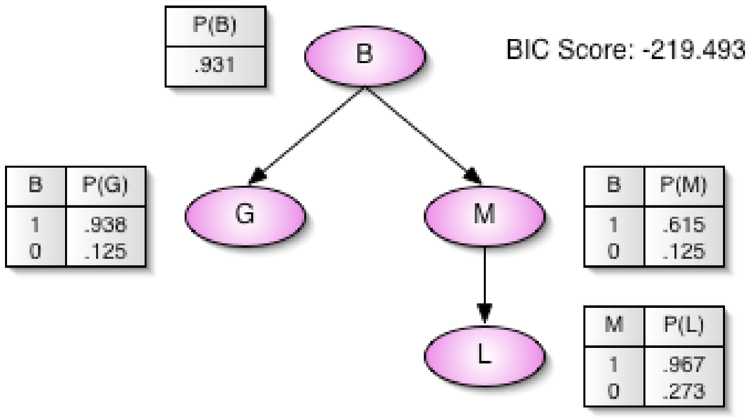


Figure 7: Output 1 of randomized greedy search on Data A.

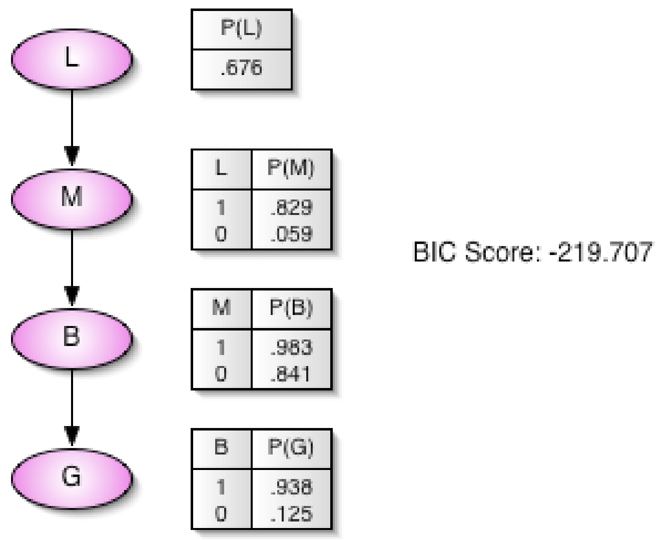


Figure 8: Output 2 of randomized greedy search on Data A.

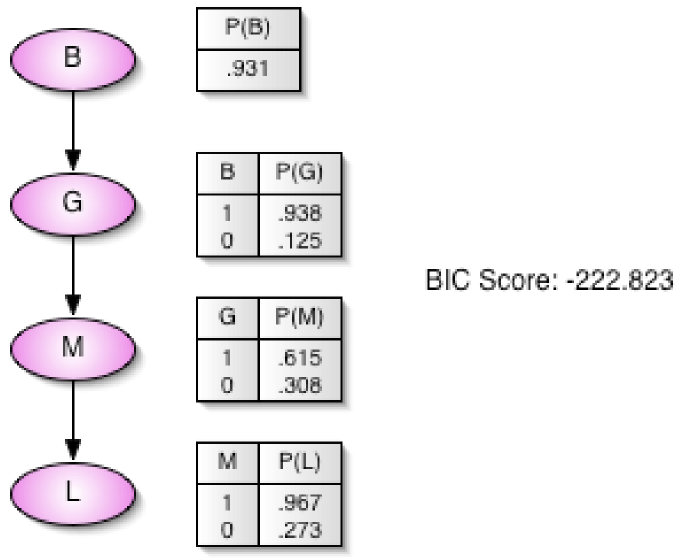
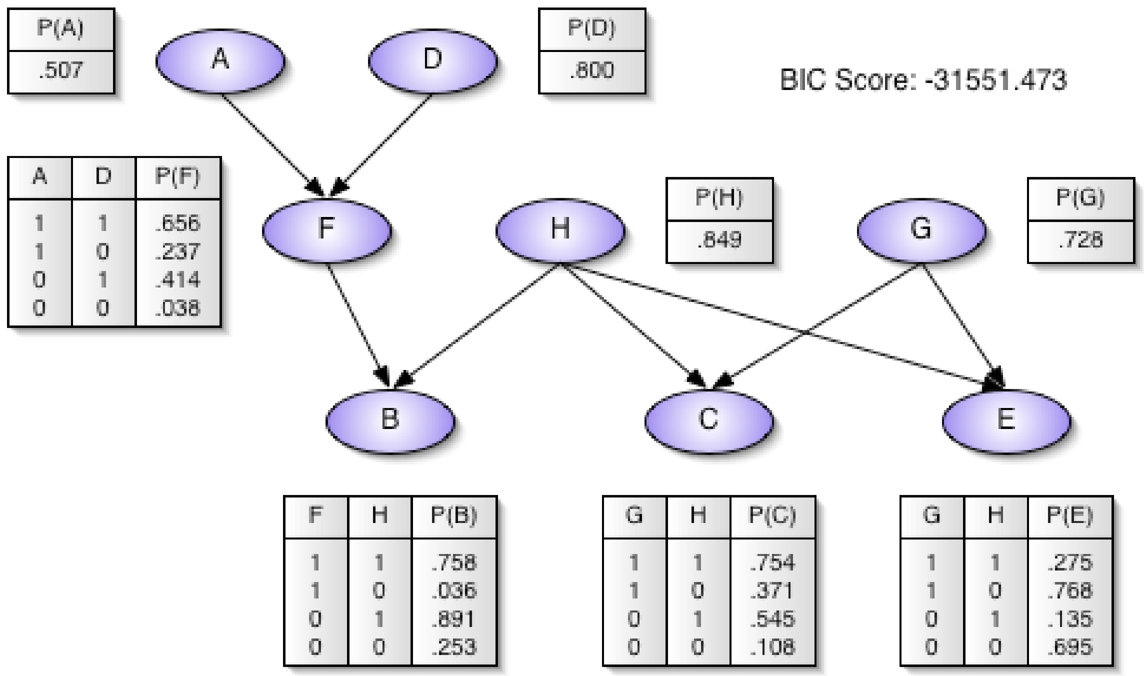


Figure 9: Output 3 of randomized greedy search on Data A.

We similarly show some of the sample network structures output for Data B in Figures 10, 11, and 12. We observe that in all three cases, the networks scored at least 2780 points (around 8%) higher than the Network B presented in Section 1, indicating that the randomized greedy search is actually quite effective.



BIC Score: -31551.473

P(A)
.507

P(D)
.800

P(H)
.849

P(G)
.728

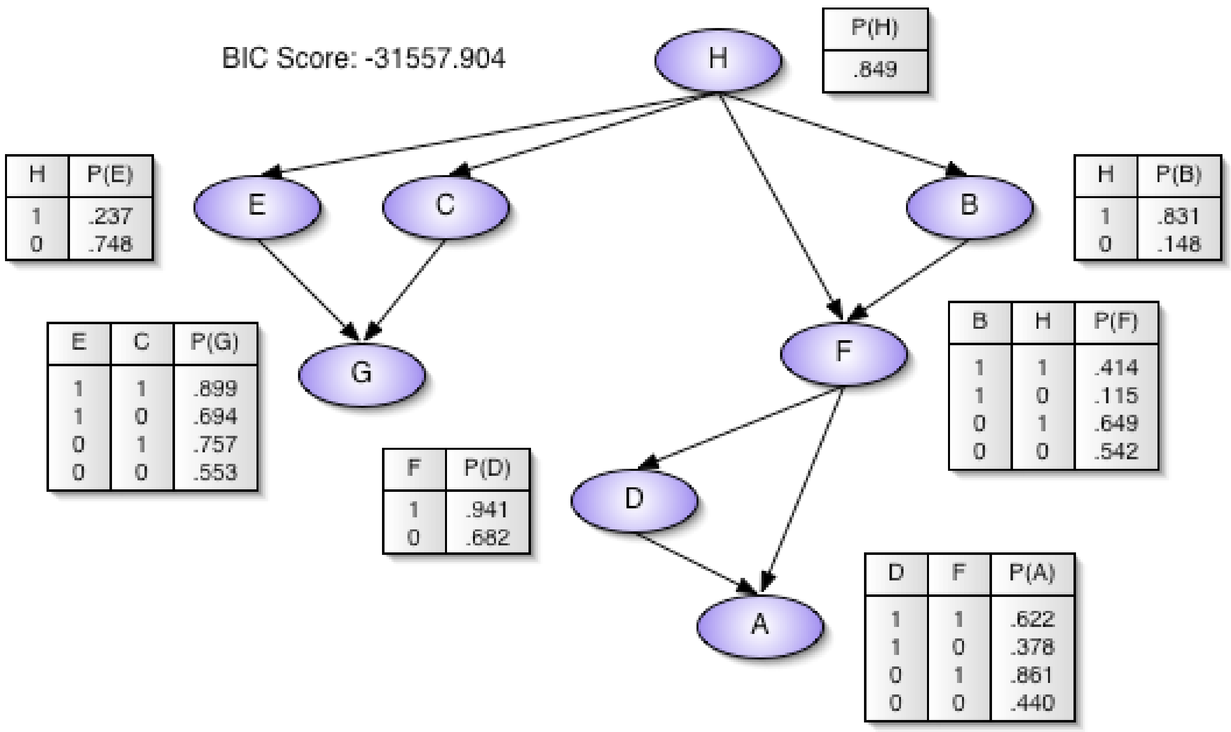
A	D	P(F)
1	1	.656
1	0	.237
0	1	.414
0	0	.038

F	H	P(B)
1	1	.758
1	0	.036
0	1	.891
0	0	.253

G	H	P(C)
1	1	.754
1	0	.371
0	1	.545
0	0	.108

G	H	P(E)
1	1	.275
1	0	.768
0	1	.135
0	0	.695

Figure 10: Output 1 of randomized greedy search on Data B.



BIC Score: -31557.904

P(H)
.849

H	P(E)
1	.237
0	.748

H	P(B)
1	.831
0	.148

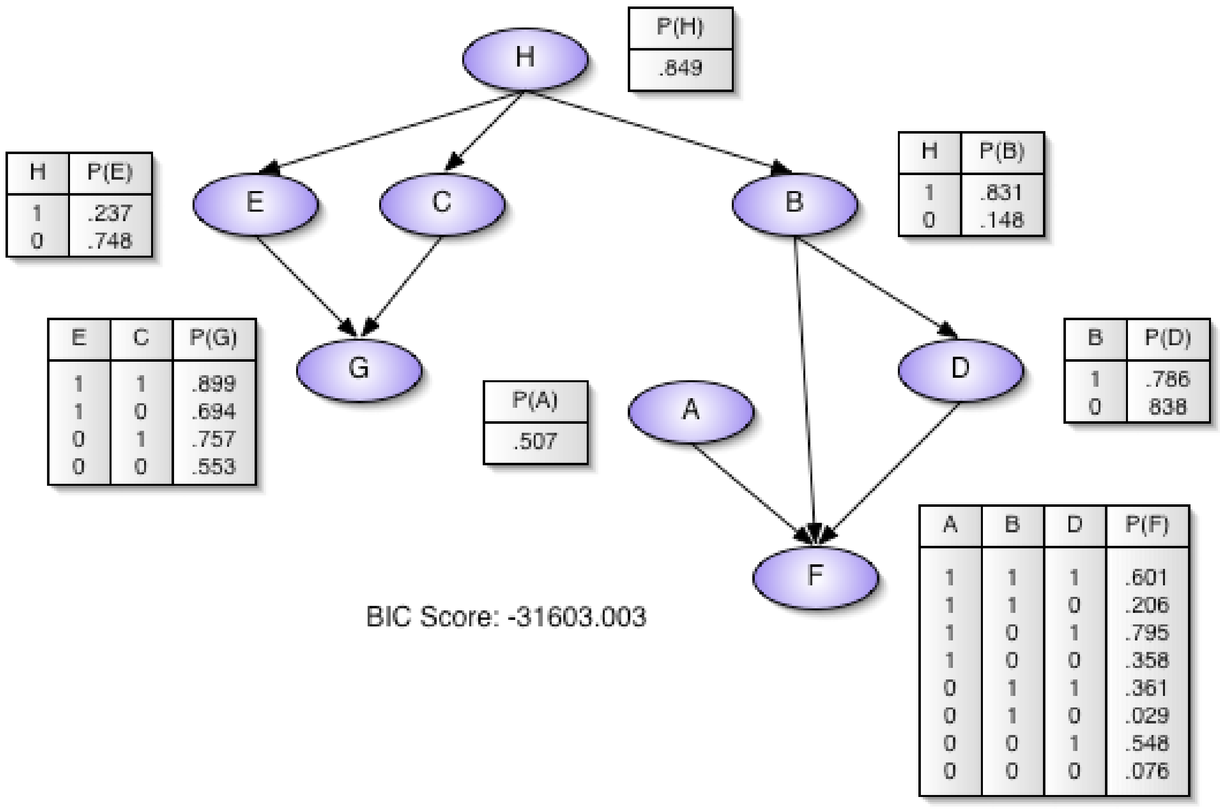
E	C	P(G)
1	1	.899
1	0	.694
0	1	.757
0	0	.553

B	H	P(F)
1	1	.414
1	0	.115
0	1	.649
0	0	.542

F	P(D)
1	.941
0	.682

D	F	P(A)
1	1	.622
1	0	.378
0	1	.861
0	0	.440

Figure 11: Output 2 of randomized greedy search on Data B.



P(H)
.849

H	P(E)
1	.237
0	.748

H	P(B)
1	.831
0	.148

E	C	P(G)
1	1	.899
1	0	.694
0	1	.757
0	0	.553

P(A)
.507

B	P(D)
1	.786
0	838

A	B	D	P(F)
1	1	1	.601
1	1	0	.206
1	0	1	.795
1	0	0	.358
0	1	1	.361
0	1	0	.029
0	0	1	.548
0	0	0	.076

Figure 12: Output 3 of randomized greedy search on Data B.

4.3 Explanation of Different Resulting Structures

Whether starting with an initial network with no dependencies or a randomly connected DAG, our experiments showed that different runs of the randomized greedy search would return different results. The variety of results means that the search space consists of various local maxima. A local maximum is a node with a BIC score that decreases for any local operation that can be legally performed on the network. In essence, the greedy algorithm is "stuck" at the hill and unable to find a transformation to a higher-scoring network, even if first performing some operations that decrease the BIC score may eventually lead to a network with a higher score.

Apart from the local maxima, the randomized, first-ascent at each local step and the randomized initial network also contribute to the differences in results. In particular, if we had implemented an exhaustive greedy search that applied the operator that would generate the highest scoring network at each local step (instead of applying the first operator that increased the score) and if we had always started with the same initial network, then each run would generate the same result each time despite the existence of local maxima.

4.4 Performance Analysis of Initialization of Unconnected Network versus Random Network

In this section, we develop four performance metrics to evaluate the performance of our randomized, first-ascent hill-climbing search algorithm with different initialization methods. For each dataset A and B, we consider two classes of initial input networks: 1) the input is an initially unconnected network and 2) the input is a randomly initialized DAG. The four performance metrics we developed to evaluate the two classes include: 1) the number of iterations performed by the greedy search; 2) the number of local operations considered by the search; 3) the quality of scores of the final networks returned by the greedy search; and 4) the quality of the scores after each iteration.

We designed experiments to collect the aforementioned performance data for the greedy algorithm over 50 runs each of (a) dataset A with an initially unconnected graph, (b) dataset A with a random initial DAG, (c) dataset B with an initially unconnected graph, and (d) dataset B with an random initial DAG, for a total of 200 runs of the algorithm. In the following four subsections, we present our analyses and the results of the experiments.

4.4.1 Quality of Returned Networks as Measured by BIC Score

We start by considering the quality of the networks returned by each initialization method on each dataset. Ultimately, we believe this to be the most important of the four performance metrics because the goal of structure learning is to find a relatively simple structure that still represents the data well. Figure 13 (a-d) shows four histograms that illustrate the frequency of the scores of various networks that were returned by the greedy algorithm.

Figure 13 (a) and (b) show that over all 100 runs, the greedy algorithm consistently returned the same 3 network structures; in fact, these 3 structures are the ones previously illustrated in Figures 7, 8, and 9. Because of the small structure space for dataset A, we are extremely confident that these 3 structures are the only local BIC score maxima for dataset A and that Figure 7 shows the optimal structure in the BIC score sense. From the histograms for dataset A, we observe that over each of the 50 runs, unconnected network initialization achieves higher scores a greater fraction of the time than the random initialization method.

In Figure 13 (c) and (d), we see that the exponentially larger search space for dataset B is also littered with substantially more local maxima. Whereas in the case of the histograms in (a) and (b), each bucket corresponded to only one structure, in the case of the histograms in (c) and (d), each bucket corresponds to multiple network structures. According to the score distributions, even though both initialization methods are able to achieve much of the same scores, the random initialization method tends to score higher for dataset B.

The obvious question to ask would be which initialization method is preferred. The score results demonstrate that neither the random initialization method nor the unconnected initialization method performs better in all cases. Because of the randomized ordering of operators, an initially unconnected network is still able to reach many different high-scoring networks via a first-ascent hill climbing approach. Had there not been a randomized ordering or had the greedy algorithm exhaustively searched the operator space at each iteration, then the initially unconnected network would deterministically reach the same local maximum, and there would be a stronger argument for the random initialization method.

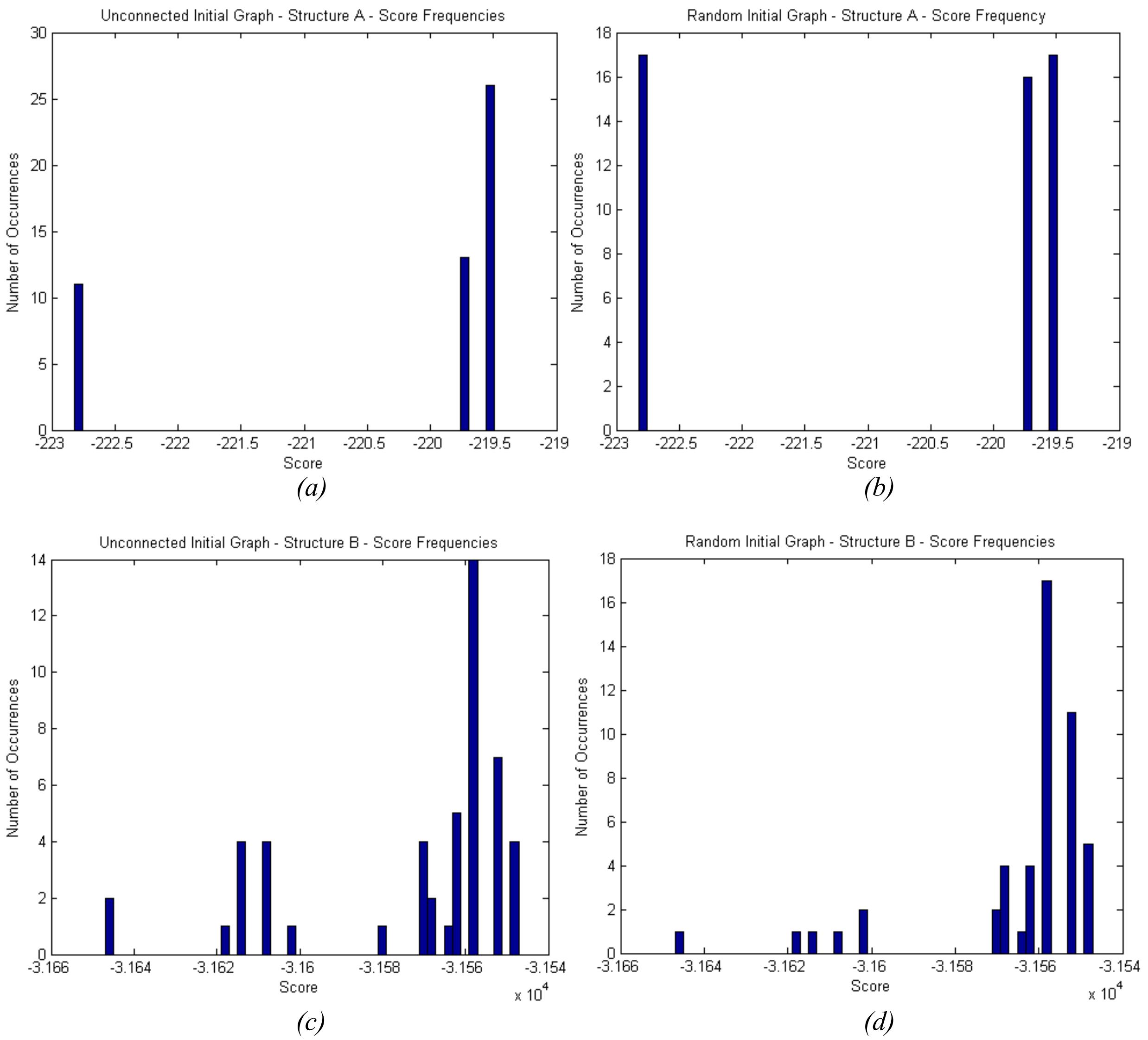


Figure 13: Frequency of scores returned by greedy search over 50 runs each of (a) dataset A starting with an initially unconnected network, (b) dataset A with a randomly connected DAG, (c) dataset B with initially unconnected network, and (d) dataset B with a randomly connected DAG.

The preferred initialization method of choice given our randomized, first-ascent hill-climbing depends on the specifics of the data and how close the better local maxima are to the unconnected network. It also depends on the realistic constraints of how many times we can perform our greedy search for the best network. Because the unconnected network is one possible DAG, the set of networks reachable by an initially unconnected network via our algorithm is a subset of those reachable by the random initialization method. If the structure search space and the number of random variables are small, then we can afford to run the search algorithm many times, and the random initialization method may be able to reach some local maxima that the initially unconnected network can not. In a large structure search space, where we cannot afford to run the search algorithm many times, we cannot tell whether an initially unconnected network or a randomly connected DAG would perform better.

4.4.2 Number of Iterations Required by Searching Algorithm

Running time is always an important consideration for any algorithm; however, the actual number of seconds used by an algorithm on different input networks can in general be affected by external factors such as CPU speed, CPU and memory load caused by other programs, cache size, etc. Thus, we decided to measure running time by the number of iterations of the **`while`** loop (shown in the line 10 of the pseudocode of Section 4.1) used by the searching algorithm before halting. This number is unaffected by the previously mentioned external factors and corresponds to the number of local modifications actually performed and kept by the greedy algorithm.

Intuitively, we would expect the running time measurements to be somewhat data-dependent. At the extreme, if the unconnected network is already a local BIC score maximum for a given dataset, then the number of iterations will always be 1; conversely, if the only maximum is a fully connected network, then a randomly connected network will typically require less iterations to reach it. Nonetheless, it would still be interesting to discover which initialization method runs faster for our particular datasets A and B.

Figure 14 (a-d) shows four histograms of the running times for the four combinations of the two datasets and the two initialization methods. With both datasets A and B, we observe from the distributions and from the different ranges on the x-axes that there is a higher variation in the number of iterations used by the greedy search under the random initialization method than under the unconnected input method. This observation follows our intuition that even though the algorithm in each case considers a random ordering of operators at each step, using a randomly initialized DAG as input has an added element of randomness allows for greater variety.

We also observe from the histograms in the cases of both datasets A and B, that the randomly initialized input network, on average, requires more iterations to reach a local BIC score maximum than that required by the initially unconnected network. Dataset A requires an average of 6.00 iterations with a randomly initialized input network as opposed to the 5.52 iterations required by an initially unconnected input network; dataset B requires 25.94 iterations in the random case and 21.04 iterations in the unconnected case. Note again, that which initialization method runs faster to completion is dependent on the particular samples that we have in our dataset.

Apart from these comparisons, we also make a few comments on the structure search space as illustrated by our experiments. Because dataset A consists of only 4 random variables, the structure space is fairly small, and we can see from Figure 14 (b) that there was even one instance out of 50 where the randomly chosen initial DAG actually turned out to be a local BIC score maximum. On the other hand, dataset B with its 8 random variables increases the structure search space exponentially, and Figures 14 (c) and (d) show that the closest input network to a local maximum required at least 14 iterations.

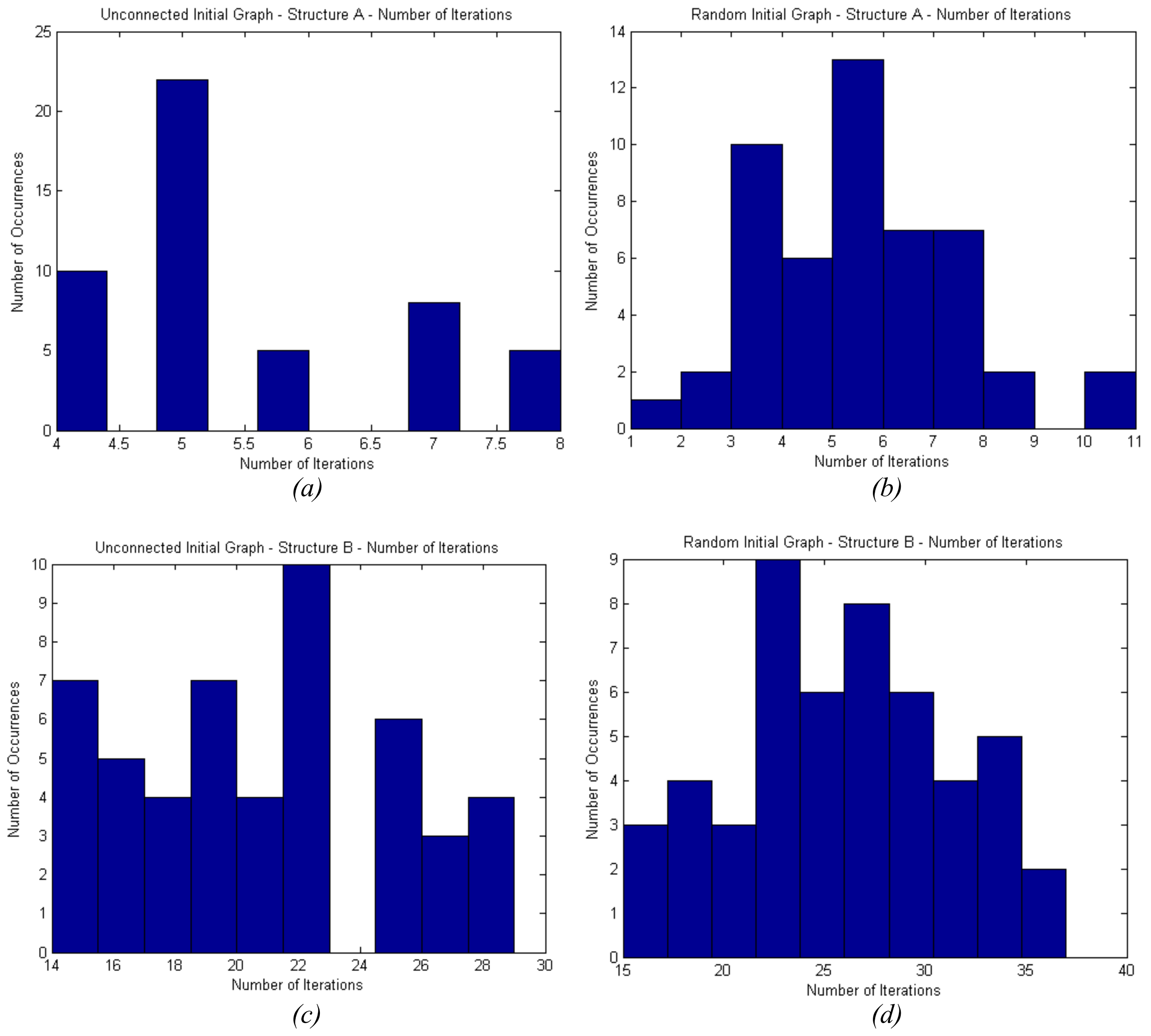


Figure 14: Number of iterations used by greedy search on 50 runs each of (a) dataset A starting with an initially unconnected network, (b) dataset A with a randomly connected DAG, (c) dataset B with initially unconnected network, and (d) dataset B with a randomly connected DAG.

4.4.3 Number of Operations Required by Searching Algorithm

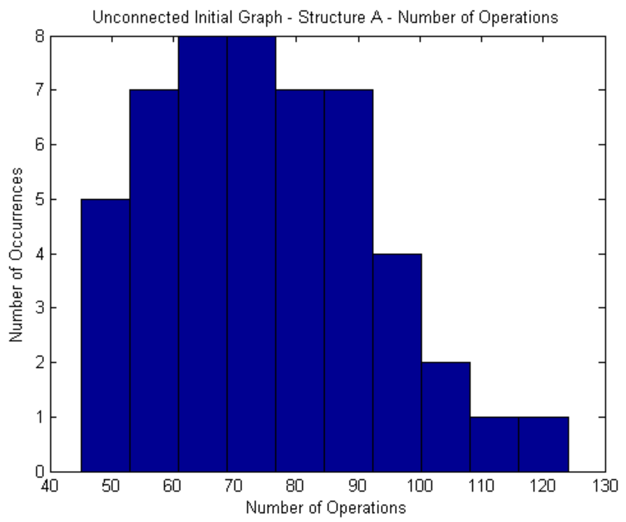
The experiments in the previous section that measured running time based on the number of iterations performed by the greedy algorithm revealed that the number of iterations were fairly similar. Even though the number of iterations used is a reasonably objective metric that does not vary based on processor speed, processor load, memory size, and memory load, it is still somewhat biased in another sense.

In particular, consider that when starting with an initial network, adding any edge would result in a legal network, and moreover, unless some of the random variables are statistically independent in the dataset, adding virtually any edge would increase the BIC score. On the other hand, when starting with a randomly initialized DAG, many operations would result in illegal network structures and not every legal operation would actually increase the score. An iteration early in the greedy search of the random initialization case would therefore require more checks before a suitable operation is found. Thus, early iterations in the unconnected initialization method intuitively seem cheaper to perform, and we would like to verify whether our intuition is in fact accurate.

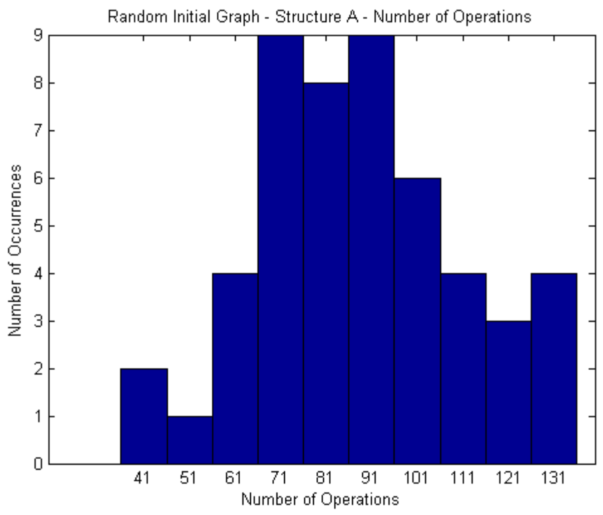
In the next series of analyses, we consider the number of operations required by the greedy search in each of the four cases. We define an operation to be an iteration of the `for` loop (shown in line 10 of the pseudocode in Section 4.1); an operation therefore represents one directed edge in the network structure that the algorithm is considering to add, remove, or reverse. We note, however, that not all operations are created equal; for example, it is much easier to check that removing an edge that does not exist is an invalid operation than it is to check that the result of adding a certain edge is both legal and results in a higher BIC score. We ignore this detail in this performance analysis.

Figure 15 (a-d) shows the results of our experiments. In fact, we do observe from the histograms that the differences in running times between the two initialization methods are accentuated when we count the more granular operations than when we counted the more coarse iterations. Dataset A requires an average of 89 operations with a randomly initialized input network as opposed to the average of 75 operations required by an initially unconnected input network; dataset B requires on average 762 operations in the random case and on average 698 iterations in the unconnected case.

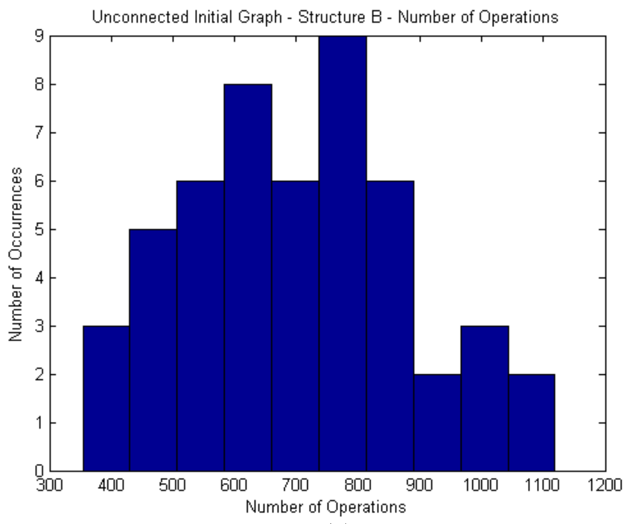
One counterintuitive result that we found occurred when we also tabulated the average number of operations per iteration (OPI) across all 50 runs of a particular initialization method and particular dataset. For dataset A, the average OPI was 13.57 in the unconnected case and 14.83 in the random case, which aligns with our intuitions and previous results. For dataset B, however, we found that the average OPI was 33.19 in the unconnected case but only 29.38 in the random case. This interesting mathematical discrepancy arises because in the random case, the longer runs (as measured by the number of iterations) used fewer operations per iteration while the shorter runs used more operations per iteration.



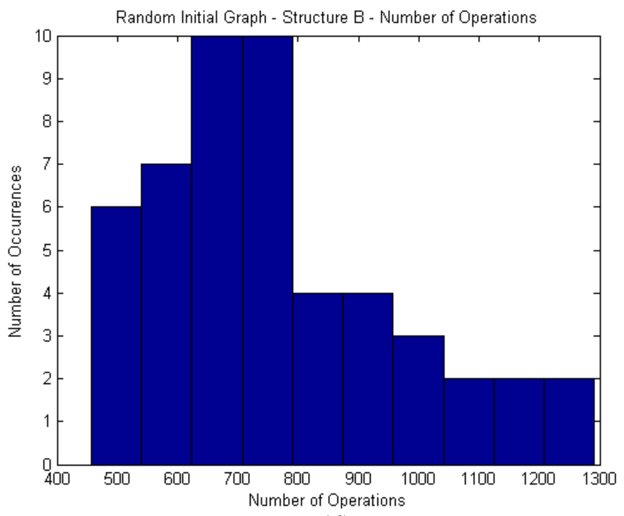
(a)



(b)



(c)



(d)

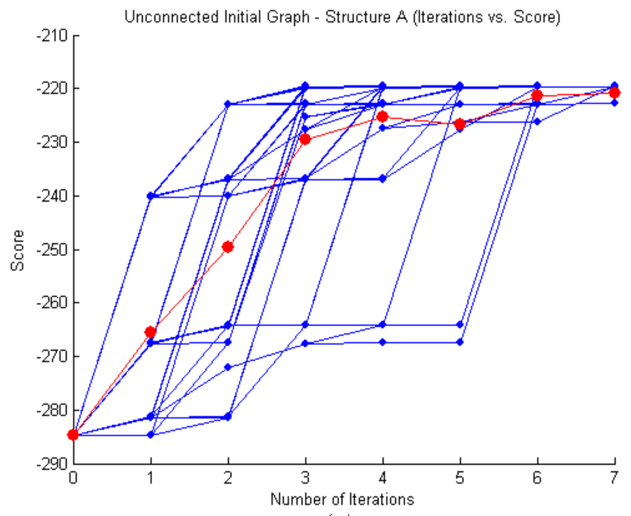
Figure 15: Number of operations used by greedy search on 50 runs each of (a) dataset A starting with an initially unconnected network, (b) dataset A with a randomly connected DAG, (c) dataset B with initially unconnected network, and (d) dataset B with a randomly connected DAG.

4.4.4 Quality of Returned Networks After Each Iteration

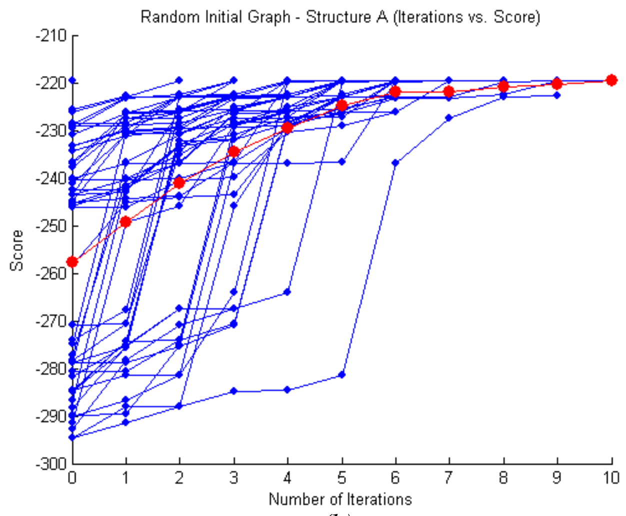
More out of intellectual curiosity than anything else, we decided that an interesting performance analysis would be to track the improvement of the BIC score over each stage of the greedy algorithm. In Figure 16 (a-d), we plot the BIC score of the intermediate network in the greedy algorithm after each number of iterations. The blue lines track individual runs, and the red line tracks the average score at each iteration over the 50 runs.

For both datasets A and B, the BIC score of the initially unconnected network was lower than the average score of a randomly initialized DAG. In fact, in all 50 runs of the random initialization method, the random DAG scored higher than the unconnected network. These properties fit with our intuition that since the random variables are not independent in the data, any possible dependency that can be expressed by the Bayesian network is probably better than having no dependencies at all.

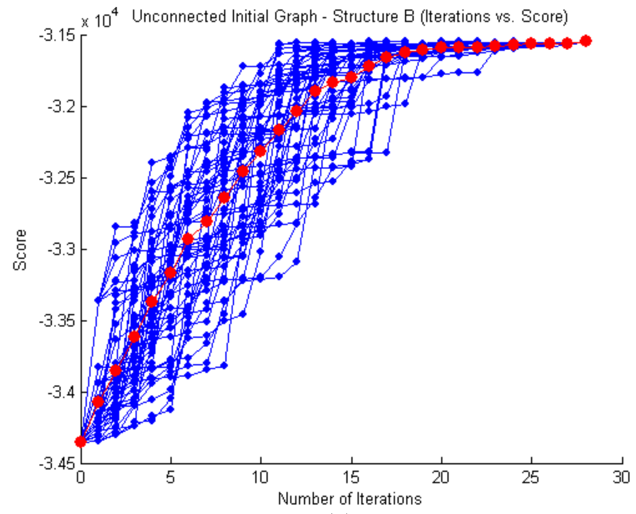
Another result that aligns with the previous observations is that the BIC score of the unconnected network increases more rapidly than the random DAG especially in the first few iterations. For dataset A, the average BIC score of the intermediate network in the unconnected initialization case catches up with the average BIC score in the random initialization case after 3 iterations; for dataset B, that number is 11. Thus, the initialization method using an unconnected network makes up for its starting handicap of a lower score quite quickly.



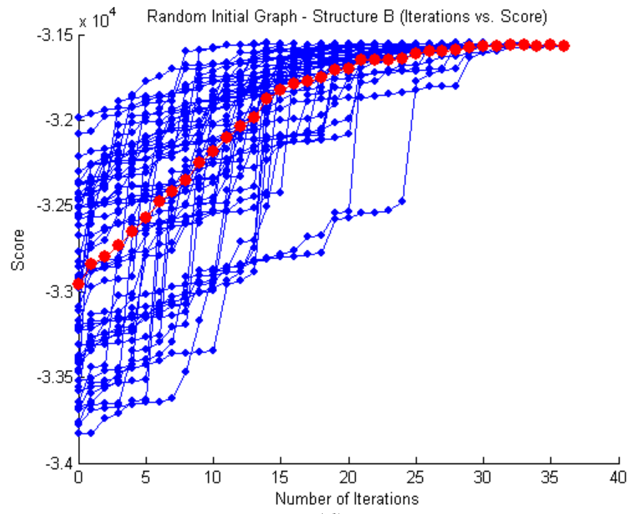
(a)



(b)



(c)



(d)

Figure 16: Plots of intermediate BIC score vs. number of iterations elapsed on 50 runs each of (a) dataset A starting with an initially unconnected network, (b) dataset A with a randomly connected DAG, (c) dataset B with initially unconnected network, and (d) dataset B with a randomly connected DAG.

5 Alternative Scoring Metric – Bayesian Score with Dirichlet Priors

Discussed in this section is an implementation of an alternative scoring metric, specifically, the Bayesian Score with Dirichlet Priors. The formula for this metric is represented on page 513 of [2]. In order to be properly implemented, we made some corrections to the formula, in line with Equations 14.9 page 511 of [2].

Equation 14.9 pertains to the multinomial case. In our networks, the variable domains are all binomial, which is just a particular case of the multinomial equation. Equation 14.9 applies to a single variable or node. In order to use this formula as a metric for the entire network, it is applied to each node, conditioned on each possible assignment of its parents. The combined equation is as follows:

$$P(D:G)=\prod_{i}\prod_{u_i \in Val\left(Pa_{X_i}^{G}\right)}\left[\frac{\Gamma\left(\alpha_{X_i|u_i}^{G}\right)}{\Gamma\left(\alpha_{X_i|u_i}^{G}+M[u_i]\right)}\right]\prod_{x_i^j \in Val(X_i)}\left[\frac{\Gamma\left(\alpha_{x_i^j|u_i}^{G}+M\left[x_i^j,u_i\right]\right)}{\Gamma\left(\alpha_{x_i^j|u_i}^{G}\right)}\right]$$

Notation:

In this equation, *i* represents a node in the network, *G* represents the given network structure, u_i represents a particular assignment of the parents, X_i is the domain of node *i*, and x_i^j represents a particular domain value of node *i*.

Algorithm:

Essentially, the equation acts over each possible domain value of a given node, for each possible parent assignment of that node, for each possible node in the network. This metric implicitly trades off between model complexity and fit. The BIC score is an approximation of this score.

Pseudo code to compute this metric is as follows:

```
1. Compute Bayesian Score Given (samples, G)

2. assignment_counts <- compute_assignment_counts(samples, G);
      (The function "compute_assignment_counts" makes one pass through the
       data and add up the counts for each possible assignment of the entire
       variable set of G)
3. bayesian_score <- 0 (Set the initial score to 0.)
4. for each node in G
5.    for each possible parent(s) assignment.
6.          Counts <- counts given the parent assignment
7.          bayesian_score <- Bayesian_score +
            logn_gamma(2) - logn_gamma(2 + Counts)
8.          for each domain value for this node (variable)
9.                Combine the parent assignment with this domain value
10.               Counts <- counts given the combined assignment
11.               bayesian_score <- Bayesian_score +
                        logn_gamma(1 + Counts) - logn_gamma(1)
```

5.1 Implementation Details

In our implementation, we use the *log* of the above formula, because it is more manageable. The equation is transformed as follows. The gamma function is computed as the sum of the log of the factorial elements. These sub-calculations can then be added and subtracted, rather than multiplied and divided.

We used uniform Dirichlet priors of (1,1) for each parameter to be estimated. We chose uniform priors because we have no knowledge of the prior distributions. We used very small (1,1) priors because we did not want the priors to have much effect on the scoring metric. Essentially, we have very little confidence in how accurately our priors will estimate the respective parameters.

We used the same (1,1) prior for each component (parameter) of the BD metric. This approach, known as the K2 prior, is efficient to implement. This method worked well for us, however, the K2 prior has been criticized as inconsistent. For example, due to the way that the BD components combine, we are effectively applying a different prior depending on the number of possible values of the parents. While the K2 prior usually works well for comparing the merits of 2 structures, it is not score equivalent. In other words, it can cause 2 equivalent graphs to be given different scores. This property effect can be avoided by using a BDe prior.

5.2 Analysis of Bayesian Networks A and B

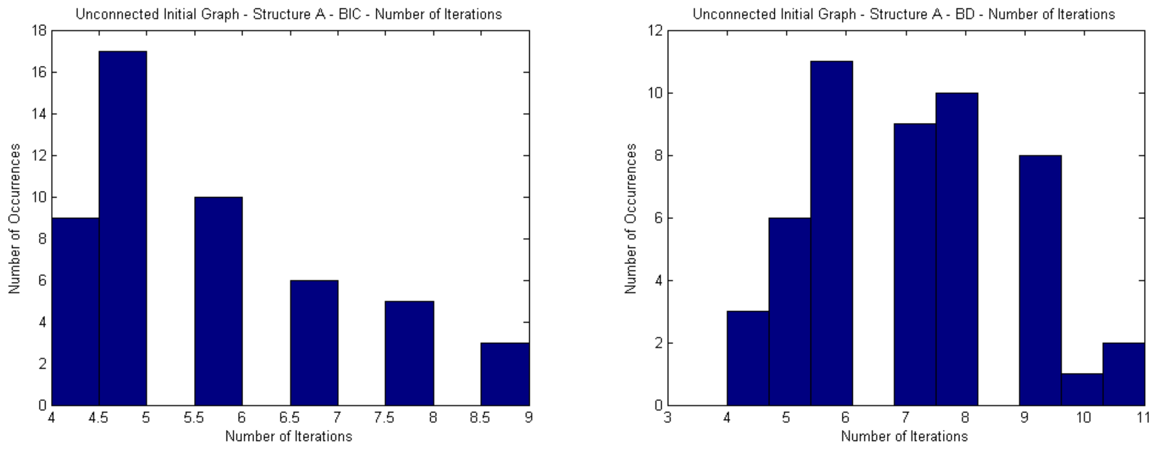
In order to analyze the performance of the Bayesian Score with Dirichlet Priors (BD), we compared it to the BIC scoring metric using the Randomized First-Ascent Hill-Climbing Search (RFAHC) described in Section 4. We compared the efficiency of the metrics in terms of the number of iterations required to reach a result. We also compared them in terms of the complexity of the resulting structure.

5.2.1 Efficiency

In order to test the efficiency of the BD metric to the BIC metric, we ran 50 trials of the RFAHC Search on Structure A and B, using Unconnected and Random initial graphs. The charts below show a histogram of the number of iterations required to find a structure.

Structure A:

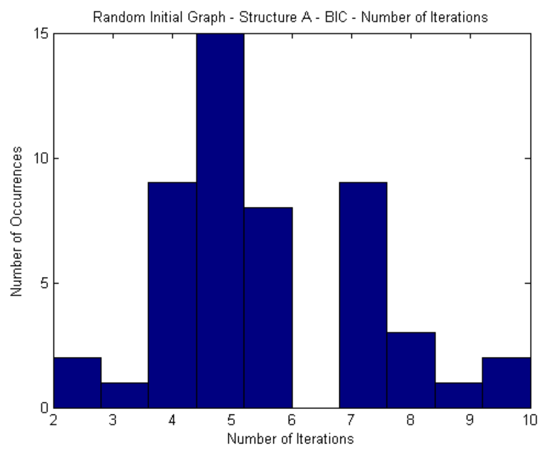
The results using Structure A and an unconnected initial starting graph, show that the BIC metric required approximately 5 iterations to find a structure while the BD metric required approximately 7 iterations.



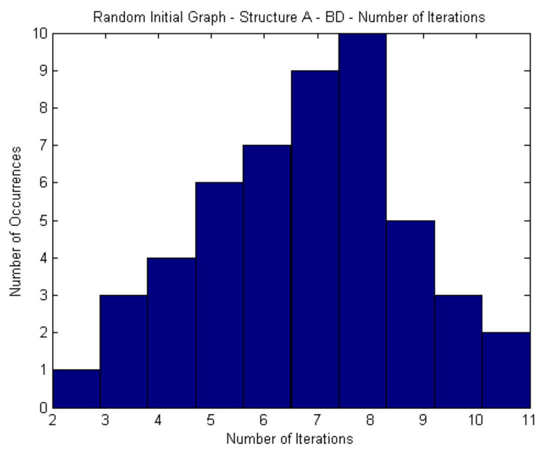
(a) (b)

Figure 17: Number of iterations used by greedy search on 50 runs each of dataset A starting with an initially unconnected network, (a) BIC metric (b) BD metric.

The results using Structure A and a random initial starting graph, show that the BIC metric required approximately 5 iterations to find a structure while the BD metric required approximately 7.5.



(a)

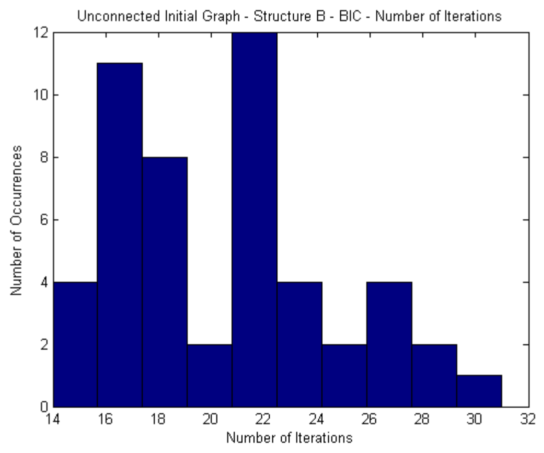


(b)

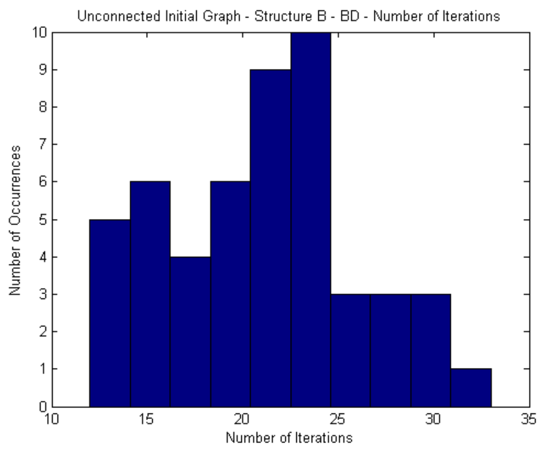
Figure 18: Number of iterations used by greedy search on 50 runs each of dataset A starting with a randomly connected network, (a) BIC metric (b) BD metric.

Structure B:

The results using Structure B and an unconnected initial starting graph show that both the BIC metric and the BD metric required approximately 21 iterations to find a structure.



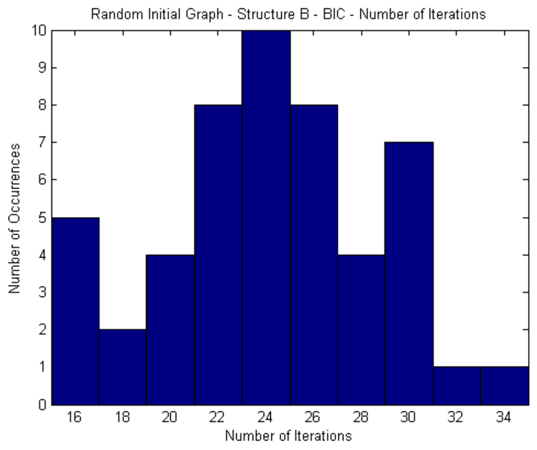
(a)

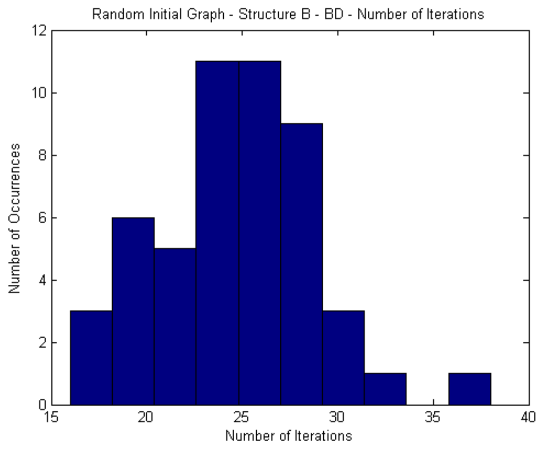


(b)

Figure 19: Number of iterations used by greedy search on 50 runs each of dataset A starting with a randomly connected network, (a) BIC metric (b) BD metric.

The results using Structure B and a random initial starting graph show that the BIC metric required approximately 24 iterations to find a structure while the BD metric required approximately 25.





(a) *(b)*

Figure 20: Number of iterations used by greedy search on 50 runs each on dataset B starting with a randomly connected network, (a) BIC metric (b) BD metric.

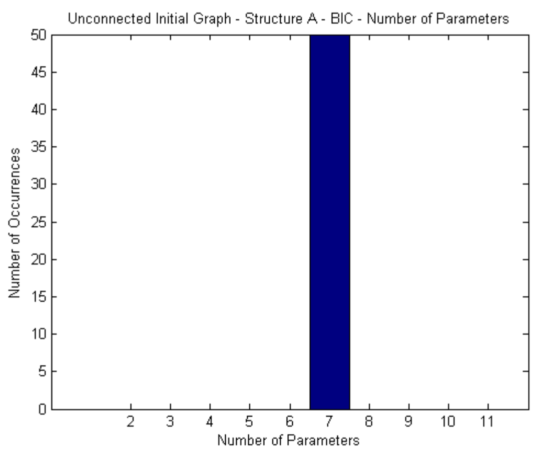
Essentially, the BD and BIC metrics behave similarly, overall, the BIC score found the respective structures, using the Randomized First-Ascent Hill-Climbing Search, faster than the BD Metric did.

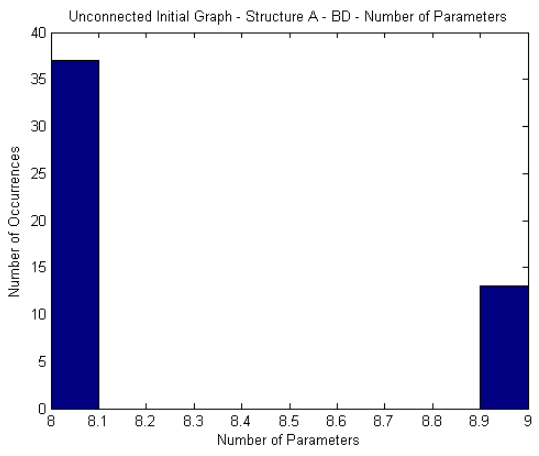
5.2.2 Complexity (Number of Parameters)

In this section we compare the two structure metrics in terms of the complexity of the resulting structure. As above, we ran 50 trials of the Randomized First-Ascent Hill-Climbing Search on Structure A and B, using Unconnected and Random initial graphs. The charts below show a histogram of the number of parameters in the structures generated using the respective methods.

Structure A:

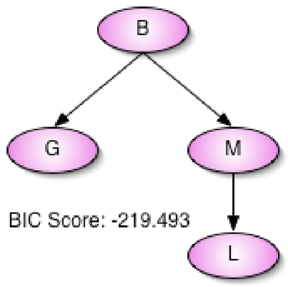
The results using Structure A and an unconnected initial starting graph, show that the BIC metric generated structures with 7 parameters while the BD metric generated structures with 8 or 9 parameters. The most commonly generated structures for each parameter number are also shown.

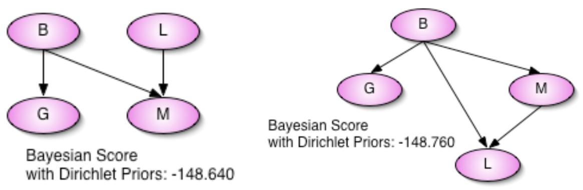




(a) *(b)*

Figure 21: Number of parameters of the structures produced by the 50 runs each, with an unconnected initial network, data set A (a) BIC metric, (b) BD metric.

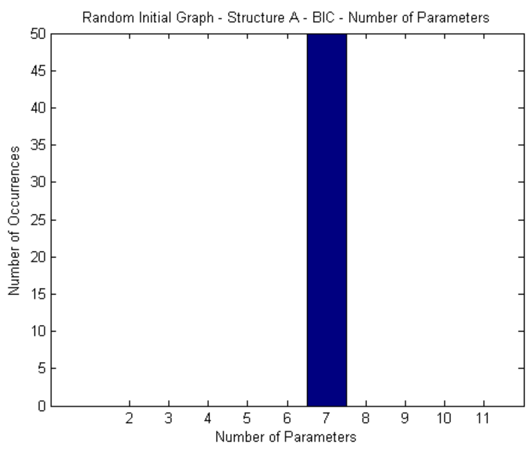



B
L
G
M
Bayesian Score
with Dirichlet Priors: -148.640
B
G
M
Bayesian Score
with Dirichlet Priors: -148.760
L

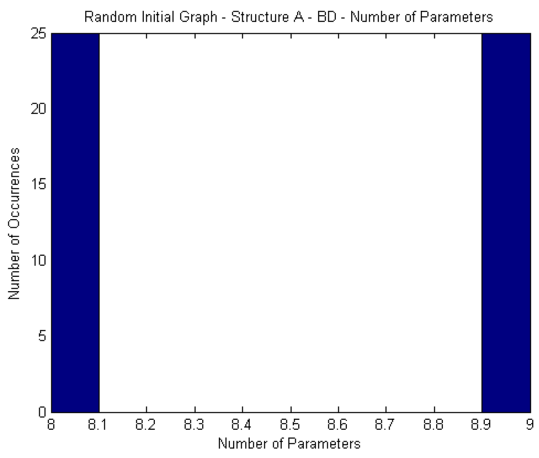
(a) *(b)*

Figure 22: The most common resulting graphs from the greedy search, 50 runs each, with an unconnected initial network, data set A (a) BIC metric, (b) BD metric.

The results using Structure A and a random initial starting graph, show that the BIC metric again generated structures with 7 parameters while the BD metric generated structures with 8 or 9 parameters. The associated graph structures are shown below. The most commonly generated structures for each parameter number are also shown.

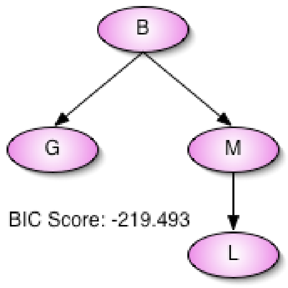


(a)

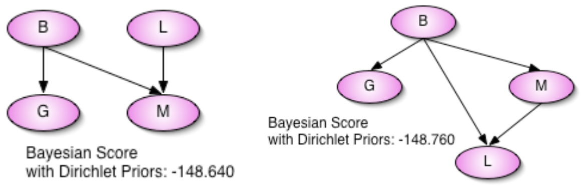


(b)

Figure 23: Number of parameters of the structures produced by the 50 runs each, with a random initial network, data set A (a) BIC metric, (b) BD metric.



(a)

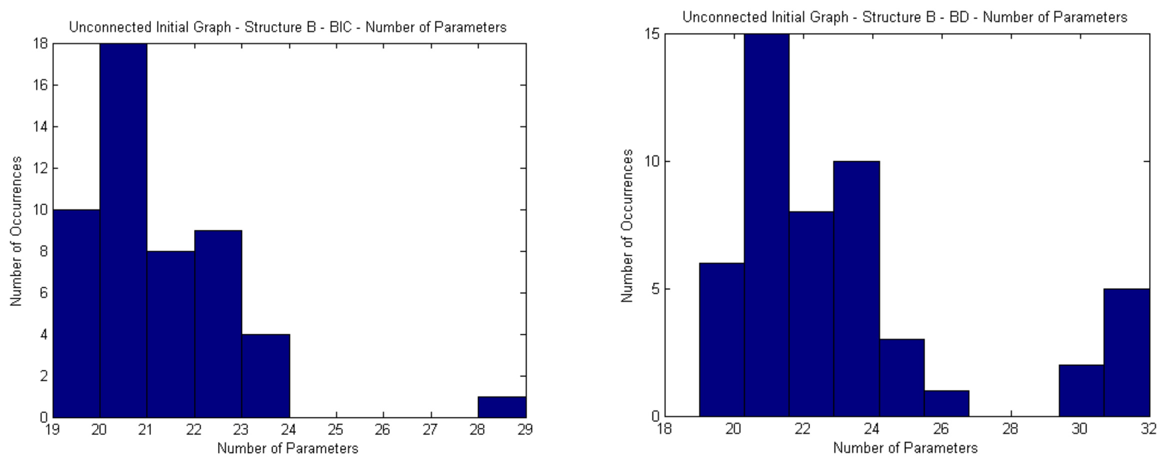

B
L
G
M
Bayesian Score
with Dirichlet Priors: -148.640
B
G
M
L
Bayesian Score
with Dirichlet Priors: -148.760

(b)

Figure 24: The most common resulting graphs from the greedy search, 50 runs each, with a random initial network, data set A (a) BIC metric, (b) BD metric.

Structure B:

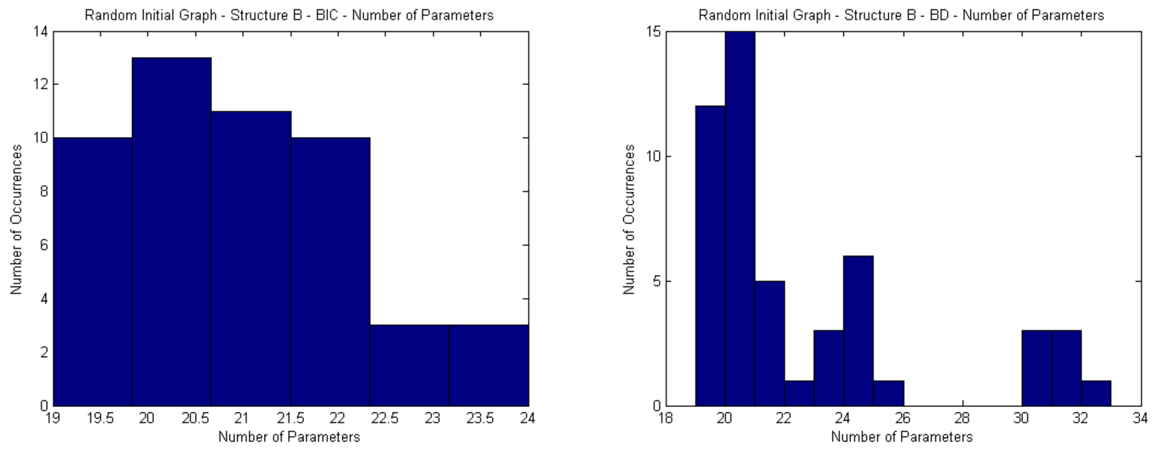
The results using Structure B and an unconnected initial starting graph show that the BIC metric generated structures with a median of 20 parameters while the BD metric generated structures with a median of 22 parameters.



(a) *(b)*

Figure 25: Number of parameters of the structures produced by the 50 runs each, with an unconnected initial network, data set B (a) BIC metric, (b) BD metric.

The results using Structure B and a random initial starting graph show that both the BIC metric and the BD metric generated structures with a median of 21 parameters, although the distribution was different.



(a) *(b)*

Figure 26: Number of parameters of the structures produced by the greedy search on 50 runs each, with a random initial network, data set B (a) BIC metric, (b) BD metric.

Together, these results show that the BD metric is less biased toward simpler structures than the BIC score, however, this difference decreased as the number of data samples increased.

The representative structures for data set A are an interesting visualization of this phenomenon. The results from data set B are wide ranging, yet the median number of parameters are similar. Due to the large number of representative structures and the closeness in the number of parameters, it is not as practical or meaningful to show these.

The explanation of the observed behavior is that as the sample size increases, the log-likelihood component of the BIC score becomes increasingly important, which biases the metric toward more complex structures with a better fit. This happens because the log-likelihood component of the BIC score grows linearly while the complexity penalty grows logarithmically. When searching for an appropriate structure for data set B, with 5000 data samples, the penalty for the number of parameters would be dwarfed by the log-likelihood component in the BIC score.

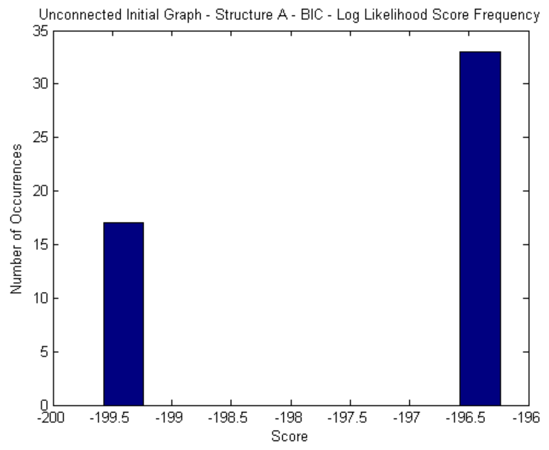
That relationship, by itself is not the whole picture, since the BD metric should behave similarly. Both metrics should have a bias toward simpler structures, which gradually favor a more complex structure as the evidence for it (number of samples) increases. The difference must be the implicit weighting of the terms.

5.2.3 Accuracy (Log-likelihood Score)

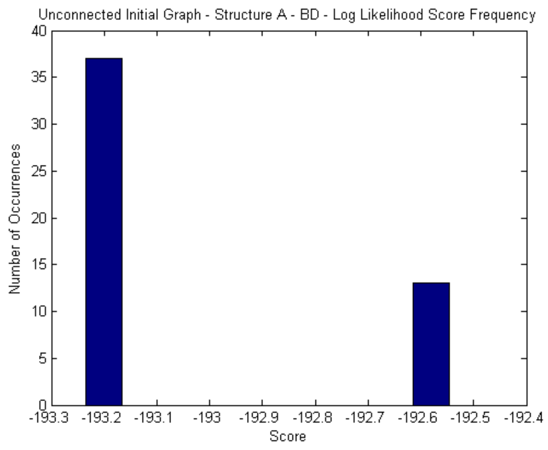
In this section, we compare the structures generated using the 2 metrics in terms of fit. In order to test how accurately the network model fits the data, we computed the log-likelihood score of the structure generated in each of the 50 trials of the Randomized First-Ascent Hill-Climbing Search on Structure A and B, using Unconnected and Random initial graphs mentioned above. The charts below show a histogram of the number of the log-likelihood scores. Using this information, we can see how well the BIC and BD metrics trade off between fit and complexity.

Structure A:

The results using Structure A and an unconnected initial starting graph show that the BD metric generated structures with a higher log-likelihood score than the BIC metric.



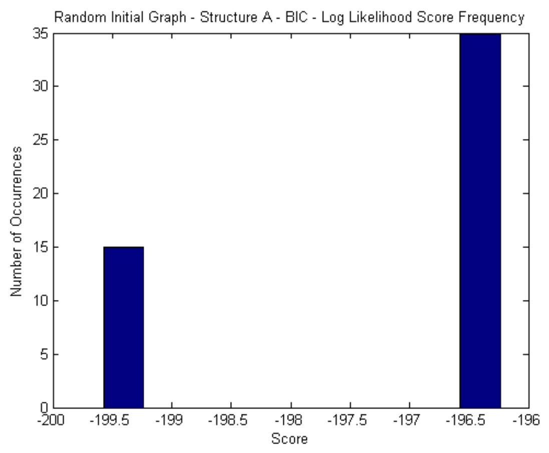
(a)

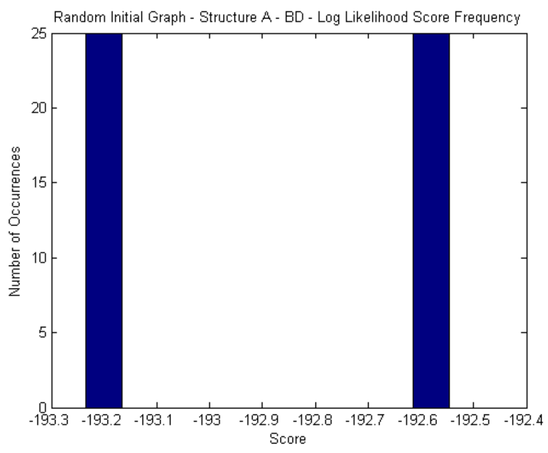


(b)

Figure 27: Log-likelihood Score produced by greedy search on 50 runs each, with an unconnected initial network, data set A (a) BIC metric, (b) BD metric.

The results using Structure A and a random initial starting graph also show that the BD metric generated structures with a higher log-likelihood score than the BIC metric.



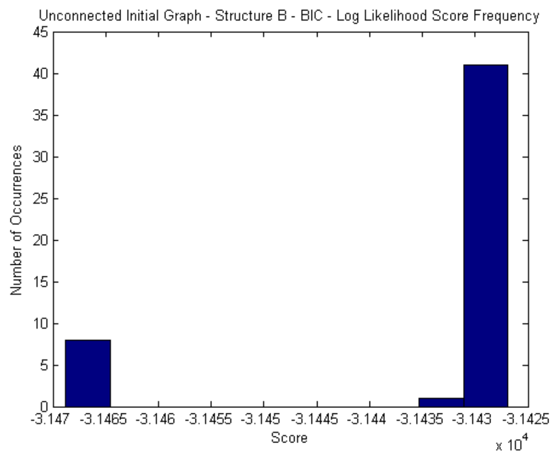


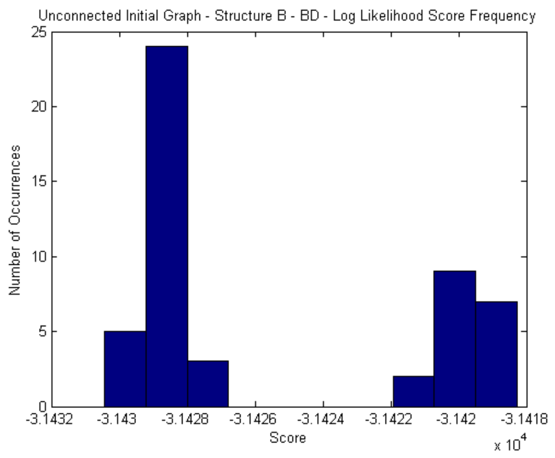
(a) *(b)*

Figure 28: Log-likelihood Score produced by greedy search on 50 runs each, with a random initial network, data set A (a) BIC metric, (b) BD metric.

Structure B:

The results using Structure B and an unconnected initial starting graph show that the BD metric generated structures with a higher log-likelihood score than the BIC metric, most of the time.

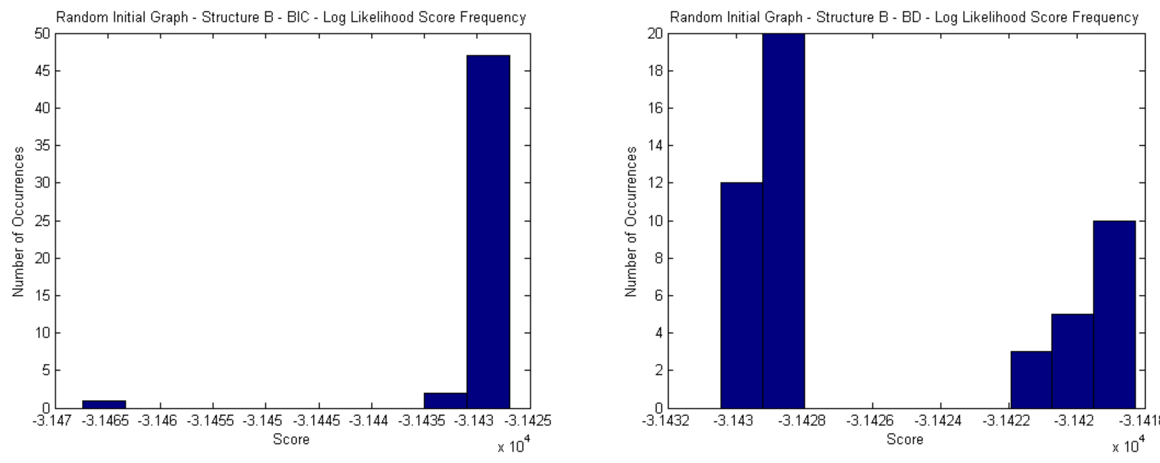




(a) *(b)*

Figure 29: Log-likelihood Score produced by greedy search on 50 runs each, with an unconnected initial network, data set B (a) BIC metric, (b) BD metric.

The results using Structure B and a random initial starting graph show that the BD metric usually generated structures with a higher log-likelihood score than the BIC metric.



(a) (b)

Figure 30: Log-likelihood Score produced by greedy search on 50 runs each, with a random initial network, data set B (a) BIC metric, (b) BD metric.

From these graphs, we can see that the BD metric finds a structure that fits the data better than the BIC score does. This observation, combined with the complexity observation from the previous sub-section indicate that the BD metric is less biased toward simple structures (than the BIC metric) and puts more emphasis on fit.

5.2.4 Conclusion:

Essentially, BD and BIC behave similarly. Our primary findings are that the BIC score is more toward simpler structures than the BD metric is, and BD finds a structure with a better fit. An additional point is that, overall, the BIC score found the respective structures, using the Randomized First-Ascent Hill-Climbing Search, faster than the BD Metric did. Armed with this information, if efficiency is paramount, then the BIC Metric is better than the BD Metric. Not only did it find the structure faster, but the simpler structure will enable more efficient inference.

6 A Genetic Algorithm for Learning Bayesian Networks

6.1 Introduction

In this section, we discuss an interesting change what we made to our search algorithm. We implemented a genetic algorithm to search the structure space. We considered several algorithm variations. We initially considered a classical genetic algorithm. We then tailored it to the structure search problem. For example, certain arcs in a graph are interrelated; therefore, it is beneficial if these related arcs are swapped as a group. This model is implemented in our "Genetic Search" class. This was a very robust search, but it was not fast. We then made modifications to the termination rules, which are based on score convergence. We also modified the mating procedure and mutation selection in order to speed up score convergence. The final algorithm is implemented in our "Genetic Greedy" class.

Our algorithm uses the Bayesian Net Node Set to store the chromosome, the BIC Score as its objective function and a population of 16 structures (initially 15 random structures and 1 unconnected structure). The mating stage of the algorithm is performed by swapping nodes, which is a way to swap sets of related arcs. To perform mutations, our algorithm intelligently picks the first beneficial mutation, considered in random order. These characteristics are discussed in the subsequent section.

One benefit of using a genetic algorithm is a broader search of the solution space. Our implementation was able to search the solution space more thoroughly than the Randomized First-Assent Hill Climbing algorithm discussed earlier. The genetic algorithm always found a structure with the highest score (Figures 31 and 32). The genetic algorithm however was slower in speed. Therefore, the genetic algorithm is preferred when inference precision is valued over structure generation speed.

6.2 Algorithm Overview

The following are certain specifics of our genetic algorithm definition:

Chromosome:	Bayesian Net Node Set (including parents)
Population:	16 structures
Initial population of structures:	15 random structures, 1 unconnected structure
Objective Function:	BIC Score

6.3 Algorithm Stages

The algorithm progresses in the following 4 stages:

1. Initialize a population of structures.
2. Mating
3. Mutations
4. Termination

The algorithm starts by initializing a population of chromosomes.

It then recombines the chromosomes in a mating procedure which generates the successive generation. Mutations are performed on the new generation, in the form of arc operations. The termination stage, simply decides, based upon the algorithm's progress, whether it should continue or stop. The mating stage and mutation stage are repeated until termination is warranted.

6.3.1 Initialization Stage

This stage involves the creation of 16 structures. 15 of them are generated randomly, and 1 of them are a totally unconnected graph.

6.3.2 Mating Stage

Selection

Generally, in the mating stage of a genetic algorithm, the best chromosomes in the population are more likely. Our initial strategy was to promote the best structure to the next generation, and mate each of the remaining structures. A given structure would choose a mate randomly, where each potential mate is chosen based on its BIC score. In other words, if a structure has a higher BIC score, then it is more likely that another structure will select it to mate with. This is the traditional strategy; however, we chose the following strategy, which is simpler, faster, and has a similar outcome.

1. Remove the worst 3 chromosomes from the current population.
2. Promote the best 3 chromosomes to the next generation.
3. Mate the remaining 12 chromosomes with the one of the top 3 chromosomes chosen at random.

We want bad structures to be weeded out of the gene pool. Therefore, we directly removed the worst 3 structures. We want the best structures (chromosomes) to be maintained as is; therefore, we promoted the top 3 structures to the next generation. We want these top structures to be mixed with some other ideas (structures with some good qualities) however; we also want to maintain a breadth of structures in order to search the solution space thoroughly. Therefore, we mated all of the remaining structures with one (chosen randomly) of the top 3 structures. An additional ramification of this strategy is that we chose who gets to mate, by digging deeper into the population. This maintains a broad search of the solution space, however, it hones down the search more than the traditional method. Our obective for doing this is to get faster convergence to a "best structure."

Reproduction:

Reproduction is accomplished by swapping portions of the chromosome. For example, in the CaMML genetic algorithm [1], which searches the DAG space of a Bayesian network, the offspring are produced by swapping a sub-graph. An alternate approach is to make repairs to the offspring. For example, we could swap skeletal structures (undirected graphs) and then randomly choose a root node and direct all arcs away form the root as was done in [1]. This however, would change the structure in a way that does not seem beneficial.

In our algorithm, we randomly chose how many nodes to exchange. We then randomly chose which nodes will be exchanged. We do not attempt to have all of these nodes correlate to a subgraph as was done in the algorithm described in [1], however, we did chose to swap entire nodes, which carry information about multiple related arcs.

In using this method, it still seems likely that illegal structures (with loops) will be produced. To overcome this problem, we make the swap multiple times (up to a set maximum) until we found an acceptable swap. In our testing of the algorithm, we found that illegal structures did not occur very often. It took very few (usually 1) tries to get a legal structure. Therefore, this random approach worked well in that respect.

We also noticed an interesting property of the mating process. We observed the structure score before and after each mating round. The mating round, by itself, never improved upon the best score in the population However, we tracked the score to the genetic algorithm with the Randomized First-Ascent Hill-Climbing Search (RFAHC). We initialized the RFAHC to the best score in the population. In very few rounds the genetic algorithm surpassed the RFAHC algorithm in terms of best score. This shows that the mating, combined with mutations add value to the algorithm.

Parameters:

We chose to swap N nodes into the chromosome that we are evolving.
N is a uniform random number from 1 to the length of the genome (the number of nodes). Therefore, on the average, we will swap 50% of the nodes with the mating structure.

Chromosome Selection:

For our chromosome, we used nodes, rather than the arcs. We considered representing the chromosome as an array of N(N-1)/2 arcs that can take on 1 of 3 values ([-1, 0 or 1] : [incoming arc, no arc, outgoing arc]).

6.3.3 Mutation Stage:

If the chromosome were represented as an array of length N(N-1)/2, where each element represents an arc (incoming arc, outgoing arc, no arc), a mutation would simply be changing an arc to a different one (add, remove, reverse). The number of mutation performed in a population can vary. That parameter is typically set low because many mutations hurt performance rather than help. If the number of mutations is too high, the procedure devolves to a random sequence of guesses. However, we already have in place a systematic method of choosing a beneficial arc change (the greedy search). Therefore, we took advantage of this by mutating every genome, in each iteration. This idea deviates from the traditional notion of a genetic algorithm, however, if a traditional genetic algorithm had an efficient means to choose a beneficial mutation, it would be wise to use it.

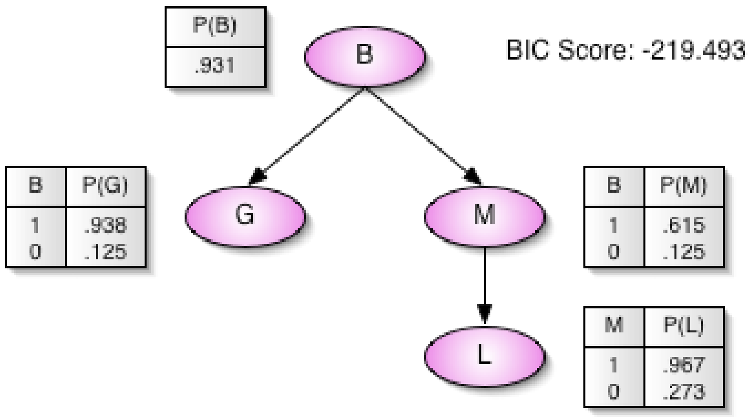
This mutation scheme used in our genetic algorithm amounts to a localized search; whereas the mating phase amounts to a more global search. The mating aspect of the genetic algorithm helps to search the entire space more fully, as does the existence of a suitably large population size. It should be noted that neither of these aspects of the genetic algorithm guarantee that it will not be caught in a local minima. Indeed, a genetic algorithm can be susceptible to this, as are many search techniques (i.e. simulated annealing). However, it is a big improvement over the Randomized First-Assent Hill Climbing algorithm.

6.3.4 Termination:

The algorithm was terminated when no progress was made in 5 successive iterations. It did not seem appropriate to stop after seeing only 1 iteration of no progress because the reproduction stage of the algorithm is designed to move the search to other areas of the search space. For example, some of the lower scoring chromosomes could be in another promising area of the search space, but have simply not found the minima of that search space yet. We tried running the algorithm with 1, 5 and 10 for this parameter. 5 seemed to be the best compromise between quality of results and running time.

6.4 Performance (Speed & Quality):

The performance of the genetic algorithm is both good and bad. The quality of the resulting structure measured in terms of the BIC score was high. The algorithm performed a thorough search of the solution space. This is evident in the consistently high BIC score that was found (See figure 32). Figure 13c and 13d show that the RFAHC algorithm found the optimal Structure B 10% of the time (for both of the initialization strategies). However, our genetic algorithm found the optimal structure (Figure 32) 100% of the time. The results are similar for Structure A (Figure 13a and 13c and 31) however the problem is much easier.



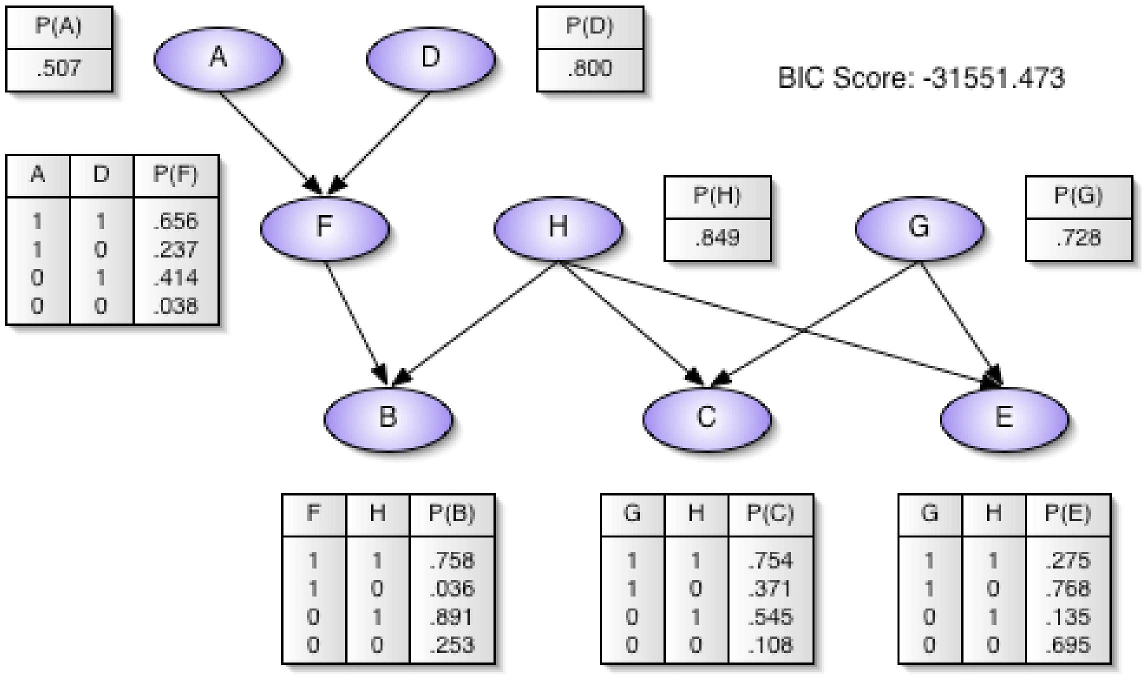
P(B)
.931

B	P(G)
1	.938
0	.125

B	P(M)
1	.615
0	.125

M	P(L)
1	.967
0	.273

Figure 31: Genetic Algorithm Output on Data A.



P(A)
.507

P(D)
.800

A	D	P(F)
1	1	.656
1	0	.237
0	1	.414
0	0	.038

P(H)
.849

P(G)
.728

F	H	P(B)
1	1	.758
1	0	.036
0	1	.891
0	0	.253

G	H	P(C)
1	1	.754
1	0	.371
0	1	.545
0	0	.108

G	H	P(E)
1	1	.275
1	0	.768
0	1	.135
0	0	.695

Figure 32: Genetic Algorithm Output on Data B.

The down side of our genetic algorithm is that it took much longer to run than the RFAHC algorithm. This is primarily due to setting up and maintaining a population of 16 structures.

6.5 Observations:

We see from the previous discussion that the genetic algorithm is a more robust search than the RFAHC algorithm. Consequently, it finds a better structure, particularly for difficult problems. Consequently, if the Bayesian Network Structure that we are searching for will be static or change infrequently then the genetic algorithm is better. Also, if a high degree of representational accuracy is needed, then the genetic algorithm is the preferred choice. However, if the structure will be subject to frequent updates due to new data, then the RFAHC is the better algorithm. Also, if the precision or representational efficiency is not paramount, whereas, a close representation is sufficient, then the greedy algorithm would be a better choice.

Another interesting algorithm alternative is simply to run the RFAHC algorithm with 16 initial structures, to see how it compares to the genetic algorithm. This is the same as running RFAHC 16 times and taking the best result. Our observations of Structure B show that it finds the optimal structure 10% of the time. Therefore, this algorithm would have a high probability of achieving optimal results with 16 starting structures.

References:

1. K. Korb, A. Nicholson, “Bayesian Artificial Intelligence,” Chapman & Hall/CRC, 2003.

2. Daphne Koller and Nir Friedman, “Bayesian Networks and Beyond,” Unpublished.