

ABSTRACT

In this thesis, we provide a general background for inference and learning, using Bayesian networks and genetic algorithms. We introduce *Bayesian Networks in Java*, a Java-based Bayesian network API that we have developed. We describe our research with structure learning using a genetic algorithm to search the space of adjacency matrices for a Bayesian network. We first instantiate the population using one of several methods: pure random sampling, perturbation or refinement of a candidate network produced using the Sparse Candidate algorithm of Friedman et al., and the aggregate output of Cooper and Herskovits' K2 algorithm applied to one or more random node ordering. We evaluate the genetic algorithm using well-known networks such as *Asia* and *Alarm*, and show that it is an effective structure-learning algorithm.

A GENETIC ALGORITHM FOR LEARNING BAYESIAN NETWORK
ADJACENCY MATRICES FROM DATA

by

BENJAMIN B. PERRY

B.S., Kansas State University, 2002

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2003

Approved by:
Major Professor
William H. Hsu

Copyright © 2003 by Benjamin B. Perry

All rights reserved

TABLE OF CONTENTS

LIST OF FIGURES	iii
1 Introduction	1
1.1 Introduction to Bayesian Networks	1
1.1.1 Graphical models	1
1.1.2 Bayes' Theorem	2
1.1.3 Bayesian Networks	2
1.1.4 Bayesian Inference	3
1.2 Stochastic Algorithms	5
1.2.1 Genetic Algorithms	5
1.2.2 Simulated Annealing	6
1.3 Structure Learning	7
2 Structure Learning background	9
2.1 Introduction to Structure Learning	9
2.2 Model Selection	10
2.3 Search Methods	10
2.4 Score-based vs. Constraint-based	11
2.4.1 Score-based	11
2.4.2 Constraint-based	12
2.4.3 Similarities	13
2.5 Existing structure-learning algorithms	13
2.5.1 K2	13
2.5.2 Sparse Candidate	14

3	GASLEAK	16
3.1	Rationale	16
3.2	Design	17
3.3	Results	18
4	SLAM GA	21
4.1	Design	21
4.2	Results	27
5	Conclusion	33
5.1	Ramifications	33
5.2	Future work	36
APPENDIX		
.1	Bayesian Networks in Java	38
.1.1	Why Java?	38
.1.2	BNJ Packages	38
BIBLIOGRAPHY		40

LIST OF FIGURES

1.1	Sample Bayesian Network.	3
3.1	GASLEAK overview	17
3.2	Histogram of GASLEAK fitness values for all order permutations of the Asia network	19
4.1	The aggregate algorithm	23
4.2	Crossover applied on node C4	24
4.3	Networks used in experiments	25
4.4	Results with the Asia network	29
4.5	Results with the Poker network	30
4.6	Results with the Golf network	31
4.7	Results with the Alarm network	32
4.8	Results with the Boerlage92 network	32

CHAPTER ONE

Introduction

One of the ever-elusive questions that many psychologists ponder is that of how humans learn. This question drives some computer scientists to create artificially intelligent beings in hopes to further understand how humans learn. A common approach to machine learning is that of discovering causal connections and probabilities between events (Pearl and Verma 1991). Almost everything in life can be broken down to probabilities and dependencies. For example, we wear certain clothes based on the weather forecast - if we hear that the weather will be quite hot, we don't want to wear sweaters lest we practically kill ourselves from the combined heat. We know that certain habits lead to bad health - it is known that smoking increases the likelihood of many health risks. We can predict whether a person poses a terrorist threat by observing several traits common among terrorists.

While it is not necessarily agreed upon that conditional dependencies are all there is to knowledge, conditional dependencies are obviously a major aspect of learning. When conditional dependencies are known, we have the power to infer future events and make 'intelligent' decisions. These conditional dependencies can be shown pictorially in a graphical model.

1.1 Introduction to Bayesian Networks

1.1.1 Graphical models

A graphical model is a collection of nodes and arcs connecting nodes. Nodes represent random variables or the state of affairs. Arcs represent conditional dependency between two nodes; the lack of an arc represents conditional independence between two nodes.

Conditional independence between nodes A and B occurs when the probability

of node A is not affected by the state of node B . Simply put, if node A “causes” node B , then node B is said to be conditionally dependent on node A . Formally, we say that node A is conditionally independent of node B if $P(A|B) = P(A)$.

1.1.2 Bayes’ Theorem

Rev. Thomas Bayes, who was both a mathematician and a theologian, first published his theorem in 1763. It stated that

$$p(H | E, c) = \frac{P(H | c)P(E | H, c)}{P(E | c)} \quad (1.1)$$

In English, this says that the *posterior probability* (the probability of hypothesis H given what happens to c given E) can be determined by multiplying the prior probability of H given c ($P(H | c)$) by the overall likelihood that assumes the hypothesis H and the background information c is true ($P(E | H, c)$). The product is then divided by the likelihood $P(E | c)$, which is independent of H , in order to be normalized.

The main idea behind Bayes’ rule is the ability to explain mathematically how your beliefs should be updated in light of new evidence. This allows us to combine new information with existing knowledge. For example, consider someone who has never seen a lightning storm and has never heard thunder. His initial belief is that the probability of thunder being linked to lightning is 50%. The first time he hears thunder after seeing lightning, his belief is updated to 66%. The second time, to 75%, and so on until he is very certain that thunder is linked to lightning.

1.1.3 Bayesian Networks

A Bayesian network is a special type of graphical model. These networks are called “Bayesian Networks” because they utilize Bayes’ rule for inference. Bayesian networks represent probabilistic relationships among a set of variables. They are a directed acyclic graph of dependencies between nodes. An edge from node n_1

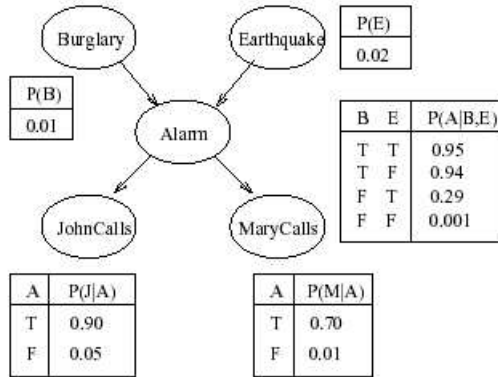


Figure 1.1. Sample Bayesian Network.

to n_2 means that node n_1 directly influences node n_2 . Note that a single edge is not a conditional probability table but is instead just one axis. A Bayesian network describes a collection of joint probabilities between several nodes. The chain rule allows us to define joint probabilities in terms of conditional probabilities. We can determine $P(X_1, X_2, \dots, X_n)$ by computing $\prod_{i=1}^n P(X_i | Pa_i)$. For nodes without parents (source nodes), we just look at the prior distribution.

A common example of a Bayesian network is figure 1.1. This network describes the following scenario: You have two neighbors - John, who lives to the west, and Mary, who lives to the east. Being good neighbors, they call your cell phone whenever your alarm goes off. If your alarm goes off, that typically means that a small earthquake just happened or you're getting robbed.

With this example, suppose that a minor earthquake just occurred. What is the probability that John is going to call? We take the product of $(.29)(.90)$ to get .261.

1.1.4 Bayesian Inference

To infer means to make a prediction based on knowledge and experience. Suppose we have a jug of a million jawbreakers that are either red or blue, but we have no idea what percentage of jawbreakers are which color. We want to know how likely

it is that we'll pull out a blue jawbreaker. In order to do that, we have to take several substantial samples of jawbreakers and count how many were red versus how many were blue. We take 100 jawbreakers out, tally the colors, and then repeat the process 19 more times. After all of the counting, we've found that 35% of the jawbreakers were red while 65% were blue.

Without having to count the other 998,000 jawbreakers, we can infer that 35% of all of the jawbreakers in the jug will be red. We can improve our confidence by calculating the standard deviation of the samples, then calculating the sample error and comparing the "z-test" value against our expectations; we might find out that we need to increase our sample size or run more experiments to improve our confidence.

With Bayesian inference, we have the luxury of the introduction of prior knowledge. This can often be a useful contribution to the evaluation process.

When we have a Bayesian network with specified CPTs (conditional probability tables) and observed values for some of the nodes in the network (evidence nodes), we want the ability to infer probabilities of values for query nodes. In the general case, this problem is NP-hard. However, it can be accomplished using one of several methods, such as exact inference (Lauritzen-Spiegelhalter algorithm) and Monte Carlo, where we simulate the network to randomly calculate approximate solutions. One of the key machine learning issues is learning this information from data; this will be discussed in detail in the next chapter.

There are different kinds of inference. Two examples of inference are:

- Diagnostic inference, which goes from effects to causes ($P(\textit{Burglary} \mid \textit{JohnCalls})$).
- Causal inferences, which goes from causes to effects ($P(\textit{JohnCalls} \mid \textit{Burglary})$).

Some of the uses of inference are with diagnosis (such as medical diagnosis), pattern recognition (such as speech recognition and image identification), prediction (given

evidence and by the effect of intervention), planning, explanation, and decision making.

1.2 Stochastic Algorithms

In the process of learning structures from data, various stochastic methods are used. The two we have focused on are genetic algorithms and simulated annealing.

1.2.1 Genetic Algorithms

Genetic algorithms are modeled after the natural process of evolution as it occurs through offspring. For most genetic algorithms, the main concept is that the strongest individuals survive a generation (survival of the fittest) and reproduce with other survivors, producing (hopefully) an even stronger child. The three main aspects of a genetic algorithm are:

- *Fitness function:* This function is used to evaluate how ‘fit’ an individual is.
- *Genetic representation:* Each individual is encoded in a manner that allows the algorithm to easily manipulate the information represented. This could be a series of on/off bits, arrays, trees, lists, or anything that is able to represent an individual. Each individual must represent a complete solution to the problem being optimized.
- *Genetic operators:* In nature, there are two prominent methods of passing traits down to the next generation:

crossover: This is where two parents contribute to the genetic makeup of a child. For example, half of parent *A*’s traits are copied over, comprising half of the child’s genetic makeup. Also, half of parent *B*’s traits are copied over, completing the genetic makeup of the child.

mutation: This is where a trait or a series of traits randomly change. This

does not happen all that frequently in nature. Mutation yields some unpredictable results - sometimes producing a more fit offspring, but usually producing a less fit offspring.

In most genetic algorithms, crossover and mutation are the predominant operators. Crossover simply chooses a point in both parents and copies the left half from one and the right half from the other, combining the two halves to create a new child. Mutation usually involves randomly rotating one or more traits.

The main algorithm for most genetic algorithms is as follows:

```
generate initial population
do{
    evaluate fitness for all individuals;
    select best individuals;
    generate next generation from best individuals;
    increment currentGeneration
} until (currentGeneration = maxGenerations);
```

1.2.2 *Simulated Annealing*

Simulated Annealing gets its origin from the annealing process of heated solids. This is the process in which a solid is melted and then cooled off to form a new (optimal) shape. The basic idea of simulated annealing in computer science is very similar: by allowing the 'solid' (search process) to sometimes form an unappealing shape (proceed in an unfavorable direction), we might be able to get out of a local optima and eventually reach a global optima. The main danger is over-shooting the global optima or passing right through it. In order to balance these dangers out, a sophisticated set of rules controls the occasional acceptance of ascents.

For example, suppose that an iteration of the search process produces a difference of Δ in the score (or performance value). The acceptance criterion (often called

the “Metropolis Criterion”), would be defined as:

If Δ is favorable, accept.

Otherwise, accept with a probability $\exp(-\Delta/T)$, where T is the *annealing temperature* that is gradually reduced over time k

This technique allows a simple local search algorithm to escape from a local optima and increase the chances to find a better solution with a typical local search. Obviously, T plays a vital role in the acceptance criterion. At the beginning of the search process, when T is the largest, more unfavorable results will be accepted, thus escaping the trap of local optima. As the procedure progresses, though, T decreases in size and fewer unfavorable results will be accepted while the algorithm converges on the global optima (if the parameters have been properly selected). The main algorithm is as follows:

```
k = 0
do{
  do L(k) times{
    perform search
    evaluate delta
    if delta is 'good', accept
    else if  $\exp(-\text{delta}/T(k)) > \text{rand}()$ , accept
    else reject
  }
  increment k
} until halting condition has been met
```

1.3 Structure Learning

Bayesian networks are very useful for a number of things. Medical diagnosis, equipment failure, image and speech recognition are just a few of the uses for Bayesian

networks. With the knowledge of a Bayesian network, we can make ‘intelligent’ decisions. There are times when we would like to know what the correlations are between random variables but we don’t have the luxury of a Bayesian network. All we might have is a database of events (or node instantiations). What we would like to do, then, is somehow come up with the model that generated the observed data. This is what structure learning is all about - trying to find (‘learn’) the model that matches the data. The rest of this thesis is devoted to doing just that.

CHAPTER TWO

Structure Learning background

2.1 Introduction to Structure Learning

One of the main uses of Bayesian networks is inference; specifically prediction, diagnosis, planning, decision making, explanation, etc. Once we know the structure of the Bayesian network, we can answer a lot of questions. However, we often do not know what the CPTs are. Often, all we know about the network is the collection of nodes. Suppose that all we have to deal with is a collection of samples, or a series of observable variable instantiations. We want to have the power of inference by learning the structure that most likely created the samples. Our challenge, then, is to fill in the edges. We want to know which nodes are parents of which other nodes. The set of parents and the range of each node tells us what the target conditional probabilities are.

The problem statement is:

- Given:
 - * A set of random variables X_1, X_2, \dots, X_n .
 - * A dataset of complete cases that were generated by some joint probability distribution $P(X_1, X_2, \dots, X_n)$.
- Return: the Bayesian network that most likely generated the observed data.
(expressing target CPTs)

This problem has been shown to be NP-hard (Heckerman, Geiger, and Chickering 1995).

2.2 Model Selection

In order to learn a network from data, some criterion is used that measures how well a network fits the prior knowledge and the given data. Once we have this criterion, we can utilize a search algorithm to find an equivalence class that maximizes this score (Heckerman 1996). Example criterion include computing the log of the relative posterior probability and local criteria, such as a variation of the sequential log-marginal-likelihood criterion (Spiegelhalter, Dawid, Lauritzen, and Cowell 1993). We have also tried using inferential loss (Lauritzen and Spiegelhalter 1988) as a criterion. One of the more common criterion is the BDE score, or the Bayesian Descriptor, which is equivalent to the MDL score, or the ‘Minimum Descriptor Length’ score. The main idea behind the MDL score is to maximize the final score output while minimizing the complexity of the structure. This is based off of the famous ‘Occam’s Razor’, where the simpler explanation is favored over the more complicated explanation if they both come to the same conclusion.

2.3 Search Methods

When learning a network from data, we typically limit each node to having k parents. If $k > 1$, then this problem is still NP-hard (chickering, Geiger, and Heckerman 1995). In order to reduce the search space, many heuristic approaches have been taken; greedy search, greedy search with restarts, best-first search, and Monte-Carlo methods (Heckerman 1996).

Most approaches first make a change to the adjacency matrix and then evaluate how much improvement was made with the change. For example, an edge may be added, deleted, or reversed. (Adding and reversing are tricky; cycles must be taken into consideration.)

Greedy search: The main idea with a greedy search is to add edges in an order that maximizes a score and then stop adding when no improvement to the

score is made. To start this algorithm, the graph may be empty, randomly generated, or otherwise generated by an algorithm. Perhaps the biggest issue in dealing with greedy search algorithms is that they are prone to falling into a local optima. The only hope to overcome this in a greedy algorithm is to perturb the structure some how and hope for the best.

Simulated annealing: As discussed earlier, simulated annealing is useful for a gradient-based search process. Changes to the adjacency matrix depend on the current ‘temperature’- as the algorithm progresses, this temperature decreases and stabilizes the network structure.

Best-first: Best-first search is another way for escaping local maxima (Korf 1993). The main idea here is to go in whatever direction is the best by measuring a heuristic. It has been shown that a greedy search with random restarts performs better than simulated annealing or best-first.

In learning Bayesian networks, the search space is typically huge. Some search methods try to search through only an equivalence class, or a smaller subset of the possible structures. There are pros and cons to this; the most obvious benefit is that the search space is reduced (but is still probably very large). One of the cons is sustaining an added cost in moving from one structure to the next.

2.4 Score-based vs. Constraint-based

2.4.1 Score-based

The main idea behind score-based learning is to optimize the degree of match between the generated network and the observations. It couples the construction of a network with finding the direction of causality. In other words, we are utilizing a function that evaluates how well the independence or dependence in a structure matches the actual data. We are therefore searching for the structure that maximizes

this score. This is based on statistics and information theory. We get the structure that gives us the most information from the parents. Basically, we search for the structure that maximizes the score. This approach can make compromises and takes the structure of conditional probabilities into account.

Some of the score-based search algorithms, such as Sparse Candidate (discussed below), develop some criterion for including or excluding an edge in the adjacency matrix. Others use some heuristic-based approach to search over the space of adding, deleting, or reversing an edge. Whatever decision that maximizes the score is chosen.

2.4.2 *Constraint-based*

Constraint-based techniques actually look at the network and see if the cause is going in the right direction. Determining this is somewhat difficult. We can use the common-sense idea - observing two things happening simultaneously implies a correlation. That's a necessary condition for inferring that one is causing the other, but it is not a sufficient condition for determining in which direction. We get around this by forcing a consistency. Suppose we know that three nodes, a , b , and c , are all related somehow. We arbitrarily pick the direction and then consider evidence about b and c to see how accurately the model predicts the data, thus inferring the direction of causality. We want to search for a network that is consistent with the observed dependencies. This is very intuitive with Bayesian networks because it follows the definition of Bayesian networks in terms of the chain rule property of inference. It also separates structure learning from the form of the independence tests. However, it is sensitive to errors in independence tests. Unlike score-based approaches, this is not based on statistics.

2.4.3 Similarities

Constraint-based and score-based share some common properties. Given sufficient data and computation, both approaches are sound. They can learn the correct structure. They both learn from observations and can incorporate knowledge. Constraint-based learning incorporates knowledge in the form of hard-constraints, whereas score-based learning can build the network in a greedy form and use prior knowledge to skew the prior values.

2.5 Existing structure-learning algorithms

Several algorithms exist today that attempt to learn Bayesian networks from data. The BNJ toolkit has two structure-learning implementations: K2 and Sparse Candidate.

2.5.1 K2

K2 is a score-based greedy search algorithm for learning Bayesian networks from data (Cooper and Herskovits 1992). It is a Bayesian method for the induction of probabilistic networks from data. K2 uses a Bayesian score, $P(Bs, D)$, to rank different structures and uses a greedy search algorithm to maximize $P(Bs, D)$. The algorithm assumes an ordering of the nodes such that if X_i comes before X_j , then X_j cannot be a parent of X_i . The candidate parents PA_i for node X_i is initially set to nothing. The algorithm visits each node X_i and ‘greedily’ adds to PA_i the node that maximizes $score_B(d, X_i, PA)$ until the maximum number of parents has been reached, no more parents are legal to add, or no parent addition improves $score_B(d, X_i, PA)$.

There are four primary assumptions with the K2 algorithm:

- (1) The database variables are discrete.
- (2) Cases occur independently, given a Bayes network model.
- (3) There are no missing values in the database.

- (4) The density function $f(B_p \mid B_s)$ is uniform. B_p is a vector whose values denote the conditional-probability assignment associated with structure B_s .

K2 uses a scoring heuristic g to build the network.

$$g(i, \pi_i) = \prod_{j=1}^{q_i} \frac{(r_i-1)!}{(N_{ij}+r_i-1)!} \prod_{k=1}^{r_i} N_{ijk}!$$

N_{ij} and N_{ijk} are frequency counts of how many times a certain node configuration occurs in data. π_i is the set of parents for node i .

There are a few problems with K2. It is a greedy search algorithm, thus it may fall into a local maxima. The algorithm is highly dependent on the node ordering. To get an accurate network, an optimal ordering must be given. The ordering is usually not known except for rare situations. For example, if we have extensive domain knowledge and the ordering might be related to a sequence of events, then we might be able to provide a node ordering. In most cases, if the ordering is already known, it is likely that an expert could already construct the network. The number of orderings consistent with directed acyclic graphs is exponential. It is unfeasible to iterate through every single possible ordering.

2.5.2 Sparse Candidate

Sparse Candidate is an iterative algorithm that attempts to restrict the search space of Bayesian networks. It achieves this by selecting for each node a small set of candidate parents and then searching for a network that satisfies the constraints. The majority of structure-learning algorithms do not use any knowledge about the expected structure of the network; however, the Sparse Candidate algorithm measures the ‘distance’ between each node to select the candidate parents.

The algorithm flows like this:

```

Loop until no improvements or iteration limit exceeds:
  For each node, select top k parent candidates
    (mutual information or m_disc) [restrict phase]

  Build a network by manipulating parents (add, remove,

```

reverse edges from node to parent) [maximize phase]

During the maximize phase, cycles must be taken into consideration. Adding an edge or reversing an edge may introduce a cycle in the network. This is fairly expensive to find out. K2 gives us this for free; since the algorithm imposes a node ordering and only allows parents whose index comes before the child's index, cycles cannot be introduced in K2.

The authors of the Sparse Candidate algorithm have described a divide-and-conquer approach to the maximize phase, but we have not been able to implement that at the time of this writing.

Our initial experiments using a hill-climbing approach in the maximize phase produced unfavorable results. This is either due to a faulty implementation or the poor performance of the maximize phase.

CHAPTER THREE

GASLEAK

3.1 Rationale

The main problem with K2 is that it is a greedy algorithm; K2 must be given a good node ordering in order to perform well. K2 cannot backtrack bad decisions and it cannot consider any joint effects of adding multiple parents. K2 adds nodes “upstream”, meaning that only nodes of higher index can be considered for parents. If a true parent is not upstream of its child, the parent will never be added; furthermore, many false parents will be added, causing unexplained correlations. Unfortunately, when we know very little about the domain of the network we’re trying to learn, the true node ordering is not known. Thus, although K2 can be a good learning algorithm at times, it is typically useless when we know nothing about the domain. Both the optimal network and optimal ordering are intractable (Heckerman, Geiger, and Chickering 1995).

Enter GASLEAK: *Genetic Algorithm for Structure Learning with Evidence using Adaptive-importance-sampling and K2*. The main idea is to search through the different permutations of node orderings in hopes of finding the right one (or one close to it). GASLEAK wraps around the K2 (Kohavi and John 1997) (Raymer, Punch, Goodman, Sanschagrín, and Kuhn 1997) and manipulates the hyperparameters for K2 (the node ordering). See figure 3.1 for an overview.

The looming statistic that GASLEAK faces is the exponential nature of the search space- $n!$; however, this is much better than the search space for all possible adjacency matrices of a network, which is somewhere along the lines of $2^{(n^2)}$. To further help our search space, many orderings may make K2 produce the exact same network, so the optimal network might be generated by more than one node ordering.

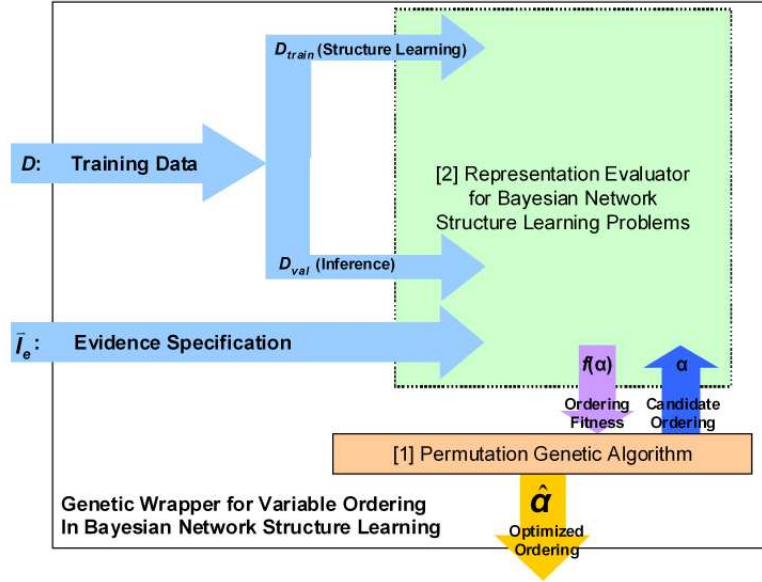


Figure 3.1. GASLEAK overview

3.2 Design

For the genetic algorithm backbone, we extended the GAJIT framework (Faupel 2000). Each chromosome is a node ordering. If there are three nodes in a network, one sample chromosome might look like [2 1 3]. The initial population for GASLEAK is generated by shuffling the node ordering for each chromosome.

The crossover technique for GASLEAK is order crossover (Haupt and Haupt 1998). Order crossover exchanges subsequences of two permutations, displacing duplicate indices with holes. It then shifts the holes to one side, possibly displacing some indices, and replaces the original subsequence in these holes. If two parents $p_1 = [3\ 4\ \underline{6\ 2}\ 1\ 5]$ and $p_2 = [4\ 1\ \underline{5\ 3}\ 2\ 6]$ are recombined using order crossover, with the crossover mask underlined, the resulting intermediate representation is $i_1 = [x\ x\ \underline{5\ 3}\ 1\ 4]$ and $i_2 = [x\ x\ \underline{6\ 2}\ 4\ 1]$, and the offspring are $o_1 = [6\ 2\ \underline{5\ 3}\ 1\ 4]$ and $o_2 = [5\ 3\ \underline{6\ 2}\ 4\ 1]$. Mutation is implemented by swapping uniformly selected indices. Catastrophic mutation can easily be implemented using a shuffle operator, but we did not find this necessary.

The fitness function used for GASLEAK is measured by inferential loss (Lauritzen and Spiegelhalter 1988; Neapolitan 1990). We determine how well the learned network performs by using a holdout validation set.

The GASLEAK algorithm reads in a training file and instantiates the population. Each chromosome is then fed to K2 and the resulting network is scored using inferential loss. After all of the chromosomes have been evaluated, the top 20% of the chromosomes are then automatically copied to the next generation. The rest of the population is filled in with chromosomes that have been processed by either the order crossover operator or the mutator operator. This process is repeated until the maximum number of generations has been reached. The best network of the final generation is chosen as the learned network.

3.3 Results

The results were not entirely favorable. The mechanics of the genetic algorithm itself actually work fine. The initial population of shuffled orderings is usually improved throughout the course of the algorithm, if even just by a little bit. The main result that we found, albeit negative, is that the fitness function of comparing the inferential loss against holdout data is not an effective fitness function and is instead very misleading at times.

We first tested the effectiveness of our genetic algorithm by trying to learn the small Asia network. The Asia network is comprised of only eight nodes, which means that there are 40,320 possible permutations of node orderings. We decided to calculate every possible permutation's fitness value for Asia network in order to keep an eye on the genetic algorithm's progress. Figure 3.2 shows the histogram of the number of permutations that share a certain score range.

We discovered that over 40% of the permutations produced a fitness value that differed very little from the gold standard network. This probably means that many

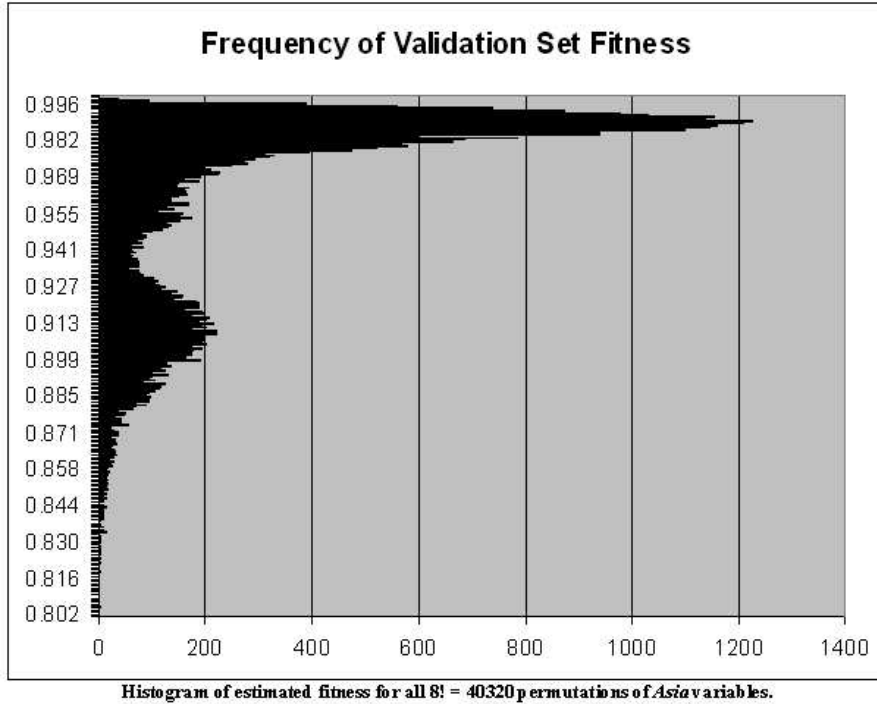


Figure 3.2: Histogram of GASLEAK fitness values for all order permutations of the Asia network

networks have the same inferential capability as the gold standard, even though many correlations are superfluous. However, this may also mean that we are using a ‘bad’ evidence bit vector. Part of the future work for the GASLEAK project is to try modulating the evidence bit vector (or simply discard the evidence bit vector) and to try using different fitness functions.

The best networks from GASLEAK usually had several graph errors; in several trial runs through GASLEAK, the small ‘Asia’ network was never successfully learned. The accuracy of the learned networks is very problematic; there are many superfluous edges, reversed edges, and deleted edges. All three graph error types are common in GASLEAK, which is very discouraging.

We found that GASLEAK is not very scalable as of yet. It is *very* computationally expensive- K2 is executed at most $g*n$, where g is the number of generations and n is the number of individuals in the population. In order to alleviate some of

the computational burden, we have designed a job farm that hands out orderings to slaves for them to run the K2 algorithm and return a final fitness to the slave. While this helps reduce the amount of time the genetic algorithm takes to execute, it is not feasible to keep adding slaves as networks become more complicated and the number of individuals in a population grows in size.

As networks become larger, the search space blows up, making it even harder to find the optimal ordering. Since the search space mushrooms to unimaginable sizes, the genetic algorithm parameters will have to be tweaked to help diversify what is being searched. For example, the population size might have to grow at a fraction of the exponential pace, or the number of generations might need to be considerably lengthened. For networks of thousands of nodes, this is simply not a feasible algorithm as of yet.

CHAPTER FOUR

SLAM GA

4.1 Design

After having limited success with GASLEAK, we considered improvements to the system. Many of the improvements discussed were related to the parameters and behaviors of the genetic algorithm itself. It was during these discussions when the idea of using a genetic algorithm to manipulate the adjacency matrix was considered.

For my Master’s thesis, we designed the SLAM GA. SLAM GA stands for *Structure Learning Adjacency Matrix Genetic Algorithm*. The main idea is to manipulate the adjacency matrix via a genetic algorithm in order to maximize the structure score. The search space is astronomically huge; it is at most $2^{(n^2)}$. A genetic algorithm comes to the rescue. Networks that score well in previous generations survive to produce better networks in future generations.

A few people have already designed genetic algorithms to learn Bayesian networks (Larranaga, Murga, Poza, and Kuijpers 1995; Habrant 1999) with varying results. The differences with our genetic algorithm are how the initial population is built and how the crossover and mutation operators are designed.

During the lifetime of this experiment, we have tried several approaches to generate the initial population. The first approach was to generate purely random edge matrices. This would create numerous illegal structures, so we had to write an algorithm that fixes these cyclic structures. This algorithm locates a cycle and knocks out a random edge in the cycle until no more cycles are found. The number of cycles depended largely on the complexity allowed for the graph; for a moderately-complex graph, approximately 22% of the networks generated with this random adjacency matrix generator were cyclic.

We implemented the Box-Muller random number generator to select how many

parents would be chosen for each individual node. The Box-Muller algorithm takes as parameters the desired average and standard deviation and then generates a number that fits the distribution. The ‘average’ for this specific algorithm is the average number of parents. After much tweaking, we found that the best average was 1.0 with a standard deviation of 0.7. The typical network that was randomly generated scored very low.

The next approach for building the initial population was to limit the selection of candidates each node could have for a parent. We used the ‘restrict’ phase described by the Sparse Candidate paper to select which nodes could be parents of certain children. We then used the Box-Muller algorithm to select how many parents would be chosen, and randomly added a candidate parent. Cycles had to be fixed by this approach as well. Networks generated this way scored a little better than the purely-random networks, but still scored low.

The third approach was modeled after the second approach. Instead of choosing a random number of parents, we added parents using the g function from K2 until no improvements were made to the g score. Networks generated in this manner scored slightly better than the previous attempts, but still scored much lower than the gold standard network.

The last approach was to supply random permutations of node orderings to the K2 algorithm one or more times and create an individual chromosome from the aggregate of the learned networks. Cycles had to be checked for and eliminated. See figure 4.1 for a pictorial explanation.

We found that the optimal number of networks to aggregate per individual was 1 for small networks and 2 for larger networks. Small networks (less than 10 nodes) comprised of more than one subnetwork performed no better at the conclusion of the GA than networks comprised of only one subnetwork. The initial generation, however, does tend to score better for multiple subnetworks, regardless of network

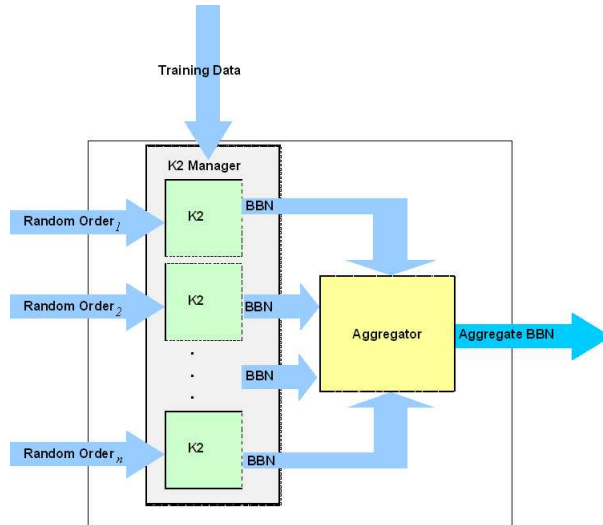


Figure 4.1. The aggregate algorithm

size.

For the crossover operator, two chromosomes are chosen. From those two chromosomes, a node index i is randomly chosen. The i^{th} node is selected from both chromosomes - we'll call them node A and node B . A and B represent the same attribute, but the parents and children of the chosen nodes may differ drastically. Nodes A and B swap parents. Specifically, node A removes all of its parents and adds to its parent vector all of the parents of node B . Likewise, node B removes all of its parents and adds the original parents of node A . Because this process may introduce cycles, we check for and eliminate cycles. See figure 4.2 for an example.

The mutator operator randomly picks a node and either adds, removes, or reverses an edge. For additions or reversals, cycles must be fixed.

To determine how 'fit' a chromosome is, the network is scored using the BDE score (Bayesian Descriptor). For some larger networks, we have found that the gold standard network scores lower than some of the candidate networks. However, the networks that do score better than the gold standard network generally differ by a small number of graph errors. Graph errors are defined to be an addition, a deletion, or a reversal of an edge.

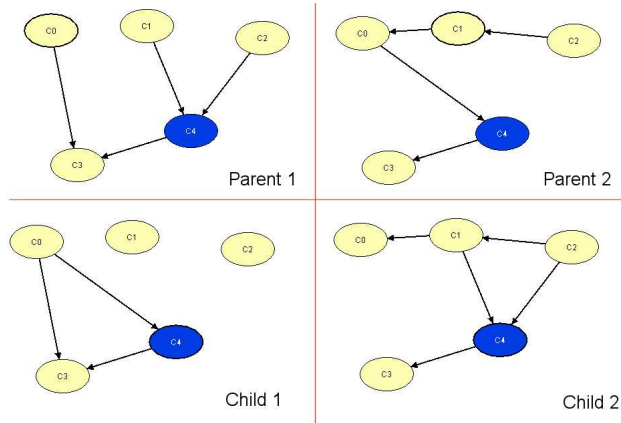


Figure 4.2. Crossover applied on node C4

For the training data, we used synthetic data generated by a data generator sampling from known networks of varying sizes and complexities. See figure 4.3 for the networks used. We used known networks in order to evaluate how well the algorithm might perform when the actual network is not known. The data generator generates random samples based off a known Bayesian network’s joint probability distribution table. The data generator sorts nodes topologically, picks a value for each root node using the probability distribution, and then generates values for each child node according to its parents’ values in the joint probability table. The root mean square error (*RMSE*) of the data generated compared to the network it was generated from is approximately 0. This means that the data is being generated correctly.

When the algorithm is launched, the SLAM GA parses an ‘options’ file to determine the method of instantiation, the samples file to load, and other GA parameters. Once all of the parameters are set, the training samples are loaded into memory. This might not be possible depending on how many samples are used and how complex the network is; if this is the case, then the samples must be continuously read from a disk, which considerably slows down the execution time. We experimented with the number of samples and found that 1,000 samples were sufficient for most networks.

For our experiments, we used 100 generations and 100 population size. Smaller networks typically only needed a population size of 10 to correctly learn the network; having a population size of 100 made the smaller networks learn a little faster. Larger and more complex networks typically require larger population sizes. To keep our experiments standardized, we decided to use 100 for all networks.

The initial population is generated by the method specified in the parameter file. The instantiator is created by a class factory, making the code modular in nature. For our experiments, we used the random instantiator, the K2-Aggregator with $n = 1$, and the K2-Aggregator with $n = 2$. Here, n means the number of networks that are aggregated together.

For the genetic algorithm backbone, we again extended the GAJIT framework (Faupel 2000). In the SLAM GA, each chromosome is a bit-string that represents the adjacency matrix. The bit-string has n^2 bits, where n is the number of nodes in the Bayesian network. Each bit represents an edge between two nodes. The bit-string can be thought of as a flattened two-dimensional array. The first n bits represent edges to the first node in the network; specifically, the first bit represents an edge from node 1 to node 1 (obviously, this bit would never be set - a node can't be a child of itself). The second bit represents an edge from node 2 to node 1. In general, bits $n * i$ through $n * (i + 1) - 1$ represent edges to node i . The conversion to and from a bit-string and a Bayesian network is trivial.

After the initialization process, the genetic algorithm simply runs its course. Each individual is scored and sorted according to fitness. The top 20% of the individuals are automatically copied to the next generation. The rest of the generation is populated by either the crossover operation or the mutation operation. After a lot of tweaking, we found that the best ratio of crossover operations to mutation operations is 1:1. Thus, approximately $\frac{2}{3}$ of the leftover population are generated by crossover since crossover is a binary operation, and the last $\frac{1}{3}$ is generated by mutation. This

process continues for each new generation until 100 generations have passed. At the completion of the 100th generation, the chromosome with the highest fitness is chosen as the learned network.

Our experiments were set up as follows:

For each network, do:

```
10 runs of 100pop/100gen with RandomInstantiator
10 runs of 100 pop/100gen with K2Instantiator with agg=1
10 runs of 100pop/100gen with K2Instantiator with agg=2
done
```

pop = population, gen = generation, agg = number of networks to aggregate

The best networks of both the initial generation and the last generation were recorded. The average fitness and the best fitness for each generation were recorded.

4.2 Results

Results from the SLAM GA were very encouraging. We have shown significant improvement over the GASLEAK algorithm and have overcome some of K2's weaknesses. Over the course of the genetic algorithm, the best fitness for each generation improved by as much as 600% over the initial population, which is an immediate indicator that the genetic algorithm is making excellent progress. This also suggests that the search space for the best structures is not deceptive.

With GASLEAK, the gold standard was rarely recovered after 100 generations with a population size of 100. However, with the SLAM GA, smaller networks have been successfully learned using the same population size and number of generations, even when the initial population is randomly generated. Figures 4.4, 4.5, and 4.6 show how the SLAM GA performs very well with smaller networks. These figures were randomly picked from the various experiment trials. Some trials actually learned the correct network.

Figures 4.7 and 4.8 show how the fitness curve improves for larger networks. The most interesting result is the improvement made by the 2-aggregate-instantiated

	Asia	Poker	Golf	Boerlage92	Alarm
K2x1	3722.084	1999.395	3081.16	1228.621	5006.827
K2x2	3720.6069	2011.54	3220.985	1429.355	7095.658
Random	3722.249	2001.884	3214.614	1459.587	6861.285

Table 4.1. Average best fitness of all networks with all instantiation methods

	Asia	Poker	Golf	Boerlage92	Alarm
K2x1	4.782025	37.1443	30.20314	147.7253164	468.7004
K2x2	0.675855	6.676625	8.591809	43.67486593	274.143
Random	4.396101	69.92143	20.65524	109.5263253	685.3574

Table 4.2. Variance of all ten runs for each network with each instantiation method populations over their respective random or single-K2 populations. The final graph for the Boerlage92 network had 22 graph errors whereas the final graph for Alarm had 19. These numbers are very encouraging, even though they appear high. Considering the size of the search space, relatively few networks in the search space will perform this well. There is room for improvement, however. We found that the larger networks such as Alarm and Boerlage92 gave scores that were better than the gold standard to structures that were more complex than the gold standard. This indicates that our fitness function needs to be improved even though it produces good results for smaller networks like Asia.

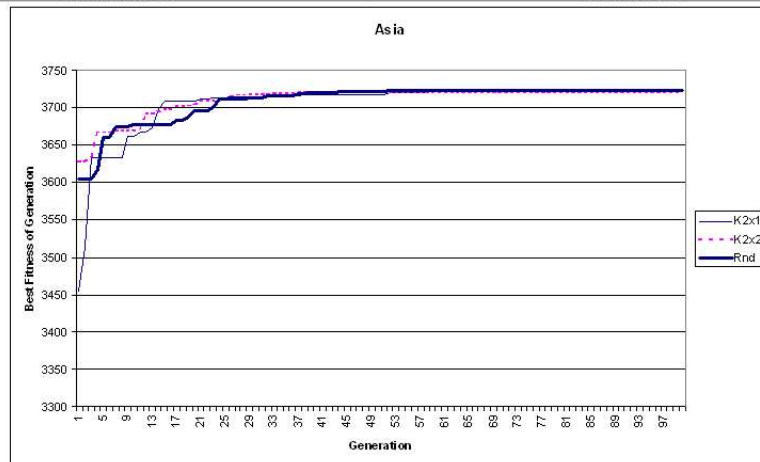
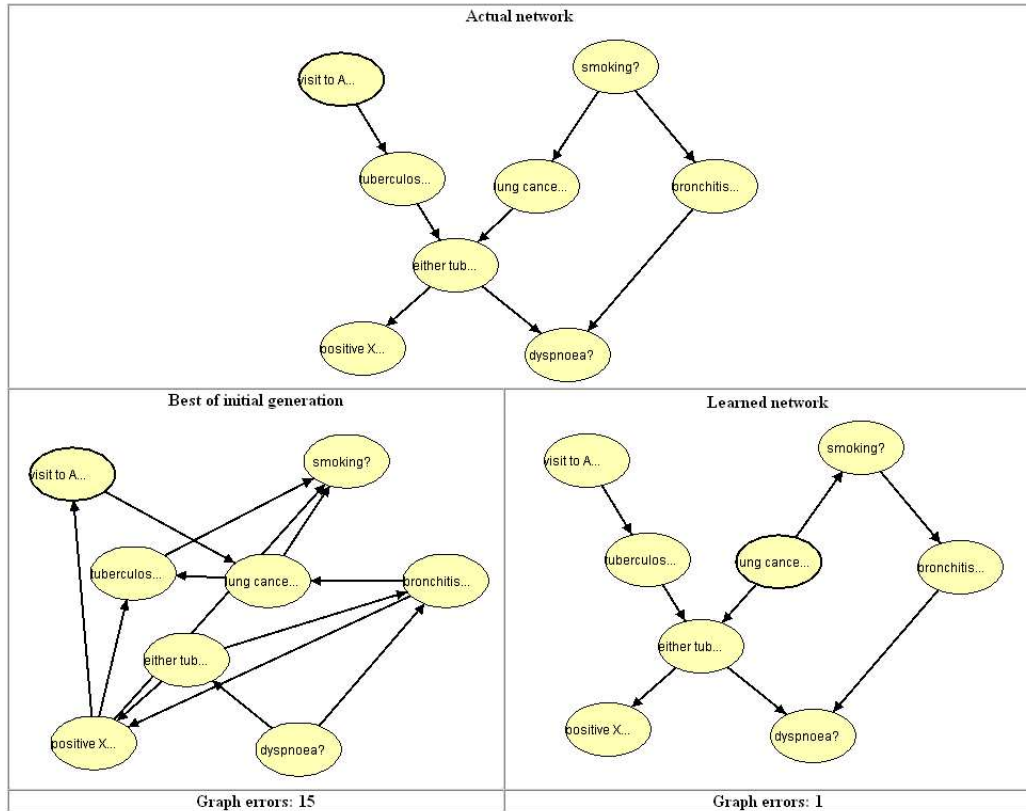


Figure 4.4. Results with the Asia network

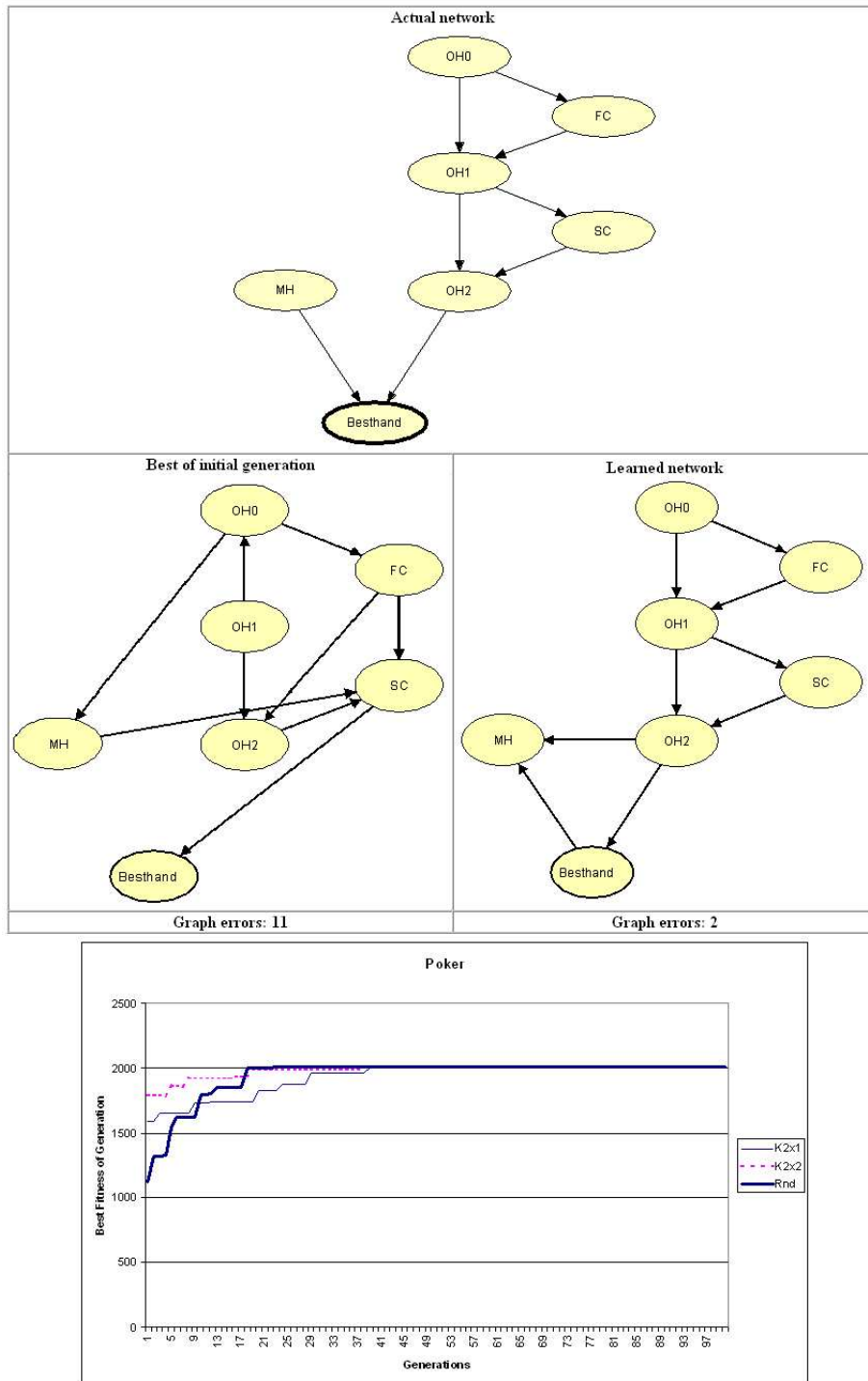


Figure 4.5. Results with the Poker network

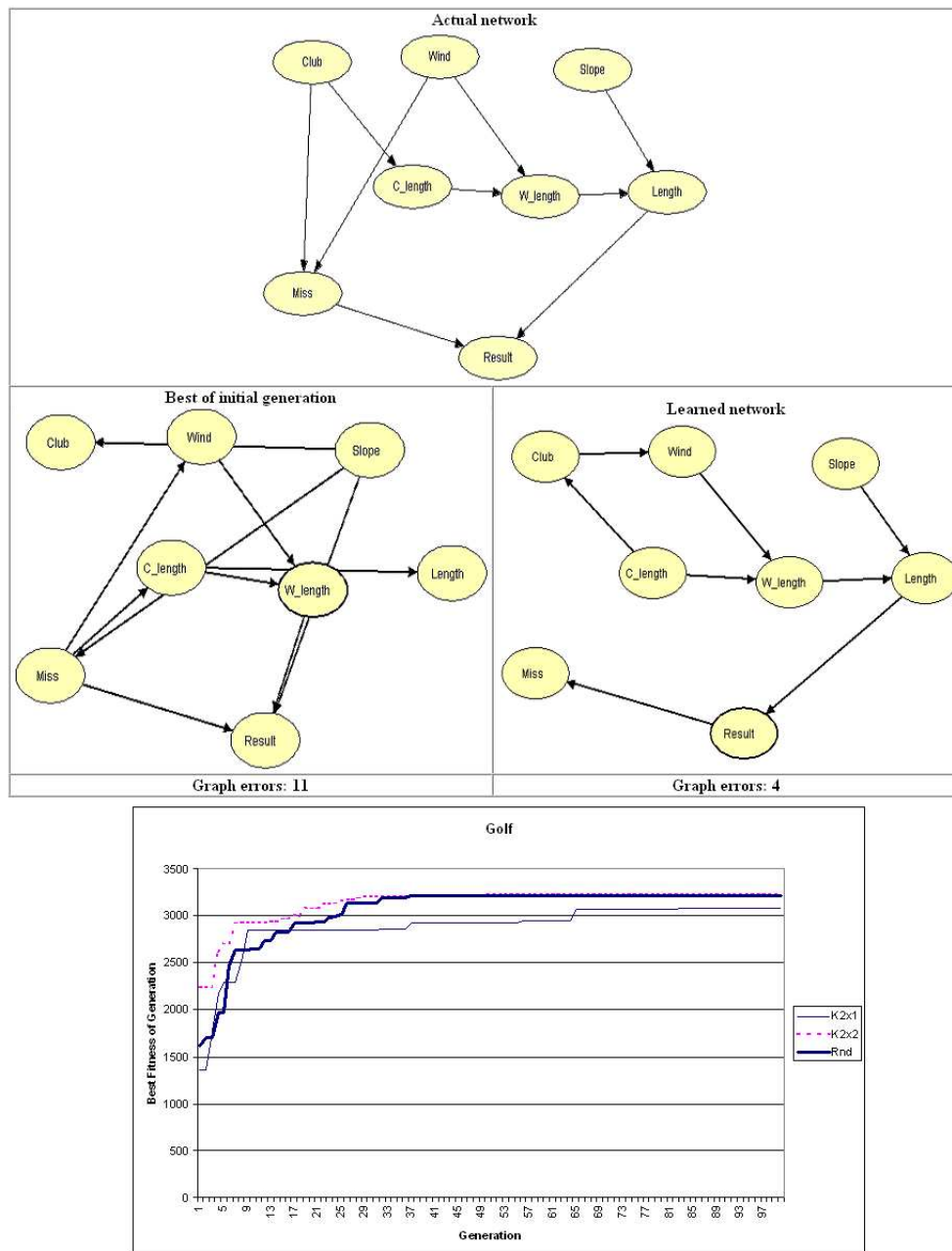


Figure 4.6. Results with the Golf network

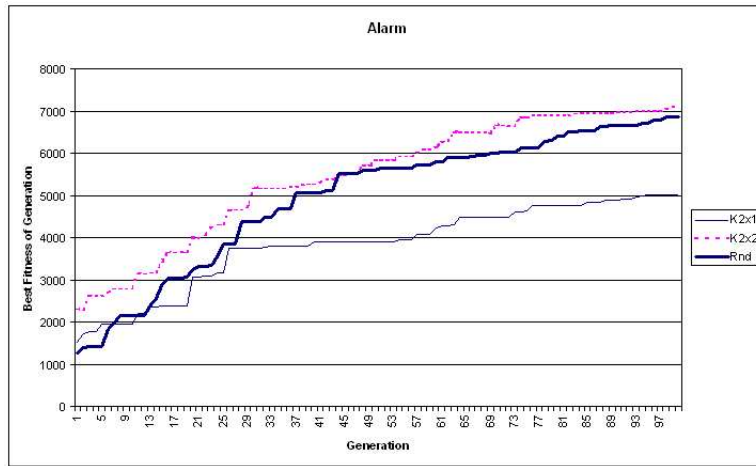


Figure 4.7. Results with the Alarm network

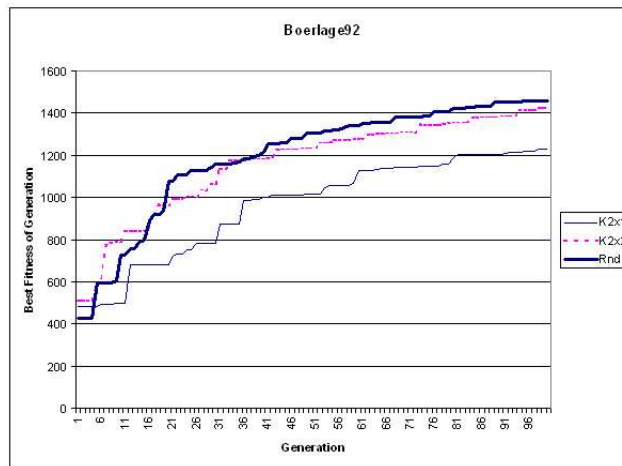


Figure 4.8. Results with the Boerlage92 network

CHAPTER FIVE

Conclusion

5.1 *Ramifications*

In creating the SLAM GA, we have contributed another effective structure-learning algorithm. Considering how large the search space is ($2^{(n^2)}$), the final network generated by the SLAM GA is outstanding, particularly for smaller networks. For example, a network with just 8 nodes has at most 2^{64} possible structures!! Obviously, a significant percentage of the possible structures is illegal (cyclic); even after weeding out the illegal structures, the search space is still astronomically huge. To find the correct network out of all of the possible structures is incredibly good, especially given the relatively few computations. Once the initial population is generated, the only process that is computationally expensive is the actual scoring of each network. When running 100 generations of 100 individuals per generation, at most 10,000 unique individual networks are evaluated. 10,000 is minuscule compared to size of the search space.

Since the SLAM GA is successfully learning the correct structure of smaller networks (overcoming the astronomical search space), we can say with some degree of certainty that the SLAM GA works, at least for smaller networks. The ability to start from completely random networks and arrive at the gold standard network in less than 10,000 evaluations is incredibly good and is a clear indicator that the algorithm works.

There are four main reasons why we believe the SLAM GA performs so well for these smaller networks.

- (1) The search space for structures is not overwhelmingly deceptive.

By this, we mean that changing a few edges is not going to catastrophically

alter the fitness score. Notice in figures 4.4 through 4.7 that there are very few generations where the fitness value jumps up dramatically. Furthermore, changes that improve the score tend to stick with future generations and eventually contribute to the final network.

- (2) The BDE score is very good for smaller networks.
- (3) The actual search space that we deal with is a tiny fraction of the entire search space.

Since K2 limits the number of parents to some constant K (which is 5 in our implementation), and the random instantiation method has an average of 1 parent with a standard deviation of 0.7, the search space is immediately reduced in size. Furthermore, even though the operators we use do not limit the number of parents a node can have, a very complicated network is rarely seen throughout the life of the genetic algorithm.

- (4) The parent-swapping crossover technique is very effective in a couple aspects. By swapping parents with a parallel node, networks maintain most of their past knowledge. Only one node is affected directly (which then affects its children) while the rest of the network is kept intact. The parent-swapping crossover technique is also manages to explore the search space diversely.

We believe that the SLAM GA is generalizable. We have shown with at least three very different networks (with nodes that differ greatly in arity, number of parents, etc) that the SLAM GA is able to recover the gold standard. As long as the fitness function is generalizable, the SLAM GA will be generalizable.

The SLAM GA does moderately well with larger networks. The main issue at hand is that the scoring function we are using scores many non-gold standard networks better than the gold standard network. However, the final network produced

is still relatively close to the gold standard. We believe that with an improved fitness function, the SLAM GA will perform well with larger networks as well.

Since the number of parents are limited and the only computationally expensive parts of the algorithm are the initial population creation (which is very fast with the random generator, but slow with K2) and the scoring function, we project that this is reasonably scalable. As networks grow to hundreds or thousands of nodes, the parameters of the genetic algorithm will likely need to be tweaked. As long as the scoring function is not unreasonably computationally expensive, this should not be a major issue.

The SLAM GA performs much better than GASLEAK in several aspects, even though both genetic algorithms use the same parameters, such as population size and number of generations. The most notable aspect is in the final network. GASLEAK produces a network that has many graph errors compared to the SLAM GA, which often produces the gold standard network for small networks. GASLEAK takes much longer to execute; GASLEAK must run the K2 algorithm on every individual in each generation and then score each individual. For the SLAM GA, the most computationally-expensive aspect is the creation of the initial population. This takes about the same amount of time as a single generation evaluation in GASLEAK. The actual genetic operators require very little time for both GASLEAK and SLAM. In general, the SLAM GA performs significantly faster and more accurately than GASLEAK.

Using the aggregate of multiple K2 networks produces a better initial population, which tends to reach the best score in fewer generations. The cause for this might simply be due to the network having more edges than otherwise generated networks have. One aspect of the future work would be to investigate this matter by increasing the average number of parents in the random generation instantiation method and comparing the outputs.

The SLAM GA is an improvement on K2. K2 relies on a good ordering to produce a good network. However, an optimal ordering is often not known. Without the known ordering, K2 is rendered useless since it is infeasible to explore the entire search space for the best ordering without the help of a wrapper like GASLEAK. If K2 is given a bad ordering, the final network is usually hopeless. However, the SLAM GA has shown that with a few edge manipulations on bad networks, a good network can be produced. Furthermore, even a bad network produced by K2 is often better than a randomly generated network. This gives us a better initial population than pure random adjacency matrices.

5.2 *Future work*

At the time of this writing, the SLAM GA is in a good state to be continued. The SLAM GA is a completely functioning genetic algorithm that is designed in a modular fashion. Anyone who would decide to continue with the SLAM GA would easily be able to insert their own instantiator or fitness function and easily tweak the genetic parameters – all without even touching a line of code in the SLAM GA project. A simple modification to the variables file (or supplying a new one at the command line) gives the researcher complete control over the behavior of the genetic algorithm. We have provided a simple interface to the instantiator and score classes so that anyone can write their own and plug it in the variables file. The framework and supporting code is already there so that the researcher can focus just on research-oriented work. (See the appendix for specifics.)

The main immediate area of future research with the SLAM GA is in the scoring function. As stated earlier, some structures score better than the gold standard network even though they have many graph errors compared to the gold standard. The SLAM GA will happily select the better-scoring network even though the selected structure is worse than the gold standard. A better fitness function will alleviate this

and learn more complex networks with better accuracy, making the SLAM GA be able to take on hundreds or thousands of nodes on a larger scheme.

The mutation operator could be designed a little more intelligently. Currently, the edge manipulations are just random according to a preset distribution that favors deleting an edge over adding and reversing. The 'add' option of the mutation operator has the most room for improvement. Right now, it just adds an edge randomly from the network. However, it could be designed so that it only adds an edge that is preselected in the 'restrict' phase of the Sparse Candidate algorithm rather than randomly adding any edge.

The SLAM GA could be tested with several more instantiation methods. Perhaps one new instantiation method could utilize GASLEAK. After GASLEAK runs its course, its final generation would be used as the first generation for the SLAM GA. Other instantiation methods might utilize other structure learning algorithms. We envision the SLAM GA as being a useful tool for assisting other learning algorithms at the completion of their executions.

The scalability could be improved by parallelizing the genetic algorithm. There are only two computationally-expensive aspects of the SLAM GA: initializing the first generation and evaluating each chromosome's fitness. The fitness evaluations could be farmed out to slaves in parallel, which will shave a lot of time off the final running time.

APPENDIX

.1 Bayesian Networks in Java

In order to experiment with Bayesian networks, the KDD group at Kansas State University has created Bayesian Networks in Java. BNJ is an open-source collection of software tools for research and development using Bayesian networks. This package has undergone at least three rewrites; most of my effort was spent in the first two rewrites. The latest version of the BNJ toolkit can be found at bndev.sourceforge.net and is freely available to the public under the GNU license.

.1.1 Why Java?

We chose Java as the programming language primarily because it is a platform-independent programming language. Java is also the programming language of choice for most undergraduate programming courses at Kansas State University. The most common complaint about Java is that it is too slow. However, with processors increasing in speed so rapidly, speed is becoming less of an issue. In fact, the average processor speed has increased more than six-fold since the time BNJ was in its early stages of development.

.1.2 BNJ Packages

The main BNJ modules are:

- Exact Inference:

ls: Lauritzen-Spiegelhalter inference algorithm.

elimbel: Elimbel, a variable elimination inference algorithm.

pearl: Pearl's inference algorithm for singly connected networks

- Approximate Inference:

ss: Simple sampling only for inference.

logicsampling: Logic sampling for both inference and data generation.

lw: Likelihood weighting for both inference and data generation.

sis: Self importance sampling for both inference and data generation.

pearlmcmc: Pearl MCMC method for both inference and data generation.

chavez: Chavez MCMC for both inference and data generation.

- Structure Learning:

k2: K2 structure learning.

gawk: Genetic Algorithm Wrapper for K2 [EXPERIMENTAL].

- Data Generator:

logicsampling = Logic sampling for both inference and data generation.

lw: Likelihood weighting for both inference and data generation.

sis: Self importance sampling for both inference and data generation.

pearlmcmc: Pearl MCMC method for both inference and data generation.

chavez: Chavez MCMC for both inference and data generation.

- Analysis:

robustness: Robustness analysis for structure learning.

- Utilities:

converter: Bayesian Network file format converter.

dataconverter: Data file format converter.

BIBLIOGRAPHY

- chickering, D., D. Geiger, and D. Heckerman (1995). Learning bayesian networks: Search methods and experimental results. In *Fifth conference on Artificial Intelligence and Statistics*, Ft. Lauderdale, FL, pp. 112–128.
- Cooper, G. F. and E. Herskovits (1992). A bayesian method for the induction of probabilistic networks from data. *Machine Learning* 9(4), 309–347.
- Faupel, M. (2000). Gajit. GAJIT is a Genetic Algorithm Java Interface Toolkit created by Faupel.
- Habrant, J. (1999). Structure learning of bayesian networks from databases by genetic algorithms-application to time series prediction in finance. In *ICEIS*, pp. 225–231.
- Haupt, R. L. and S. E. Haupt (1998). *Practical Genetic Algorithms*. Wiley-Interscience.
- Heckerman, D., D. Geiger, and D. Chickering (1995). Learning bayesian networks: The combination of knowledge and statistical data. In *Machine Learning*.
- Heckerman, D. A. (1996). A tutorial on learning with bayesian networks. Technical Report 95–06, Microsoft Research.
- Kohavi, R. and G. John (1997). Wrappers for feature subset selection. *Artificial Intelligence journal, special issue on relevance* 97(1-2), 273–324.
- Korf, R. (1993). Bagging predictors. *Artificial Intelligence* 62, 339–409.
- Larranaga, P., R. Murga, M. Poza, and C. Kuijpers (1995). Structure learning of bayesian networks by hybrid genetic algorithms.
- Lauritzen, S. L. and D. J. Spiegelhalter (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Royal Statistic Society* 50, 154–227.
- Neapolitan, R. E. (1990). *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. John Wiley and Sons, New York.
- Pearl, J. and T. S. Verma (1991). A theory of inferred causation. In J. F. Allen, R. Fikes, and E. Sandewall (Eds.), *KR'91: Principles of Knowledge Representation and Reasoning*, San Mateo, California, pp. 441–452. Morgan Kaufmann.
- Raymer, M., W. Punch, E. Goodman, P. Sanschagrín, and L. Kuhn (1997). Simultaneous feature extraction and selection using a masking genetic algorithm. In *Proceedings of the 7th International Conference on Genetic Algorithms*, San Francisco, CA, pp. 561–567. Morgan Kaufmann.

Spiegelhalter, D., A. Dawid, S. Lauritzen, and R. Cowell (1993). Bayesian analysis in expert systems. In *Statistical Science*, pp. 8:219–282.