Implementing an Approximation Scheme for All Terminal Network Reliability

by

Ray P. Tai

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author ....
Department of Electrical Engineering and Computer Science
June 3, 1996

Certified by ....
David R. Karger
Assistant Professor
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Implementing an Approximation Scheme for All Terminal Network Reliability

by

Ray P. Tai

Submitted to the Department of Electrical Engineering and Computer Science
on June 3, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I implemented a randomized fully polynomial time approximation scheme for the All Terminal Network Reliability Problem. The scheme consists of a combination of a naive Monte Carlo algorithm for computing the reliability of networks with a high failure probability, and a method of enumerating small cuts to compute the the reliability of networks with a small failure probability. The implementation was tested on several different network topologies. The implementation demonstrated that the algorithm performed better than its theoretical $O(n^4)$ time bound in practice, and it also provided insight into the causes of unreliability in networks.

Thesis Supervisor: David R. Karger
Title: Assistant Professor

Acknowledgments

Thanks to Dr. Miro Kraetzl, who provided a network reliability package that was invaluable during the initial testing of the implementation. Thanks also to Milena Mihail, who suggested graph topologies for testing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

One of the classic problems in reliability theory is the All Terminal Network Reliability problem. In this problem, we are given a network with n vertices whose edges fail independently with some probability. Given this input, we wish to determine the probability that the network is disconnected at any particular time. By disconnected, we mean that there exist two vertices that do not have a path between them formed by edges that have not failed.

There are several practical applications of this problem in the analysis of communications networks and other networks, which has caused the problem to undergo a large amount of scrutiny [Col87]. In this thesis, we implement an approximation scheme that we hope will provide a way to analyze network reliability quickly, and thus aid engineers in designing reliable networks.

1.2 Previous Work

The All Terminal Network Reliability problem has been shown to be $\#P$-complete, a complexity class that is at least as intractable as NP[Val79, PB83]. Therefore, it is very unlikely that the problem has a polynomial time solution. Given the difficulty of the problem, work has been done to try to approximate solutions. Initially, researchers believed that even approximating a solution to this reliability problem was $\#P$-complete[PB83]. This belief, however, was based on the assumption that the approximation parameter ϵ was part of the input, and that one could encode an exponentially small ϵ.

Currently, research is being performed on fully polynomial time approximation schemes (FPTAS). An approximation scheme is deemed a FPTAS if it has running times polynomial in n and also polynomial in $\frac{1}{\epsilon}$. Another interpretation is that an FPTAS runs in polynomial time with respect to the input size if ϵ is given in unary as opposed to binary notation. Using the FPTAS model, it is possible to create an approximation scheme for the All Terminal Network Reliability Problem that runs in polynomial time. This fact was proven in Professor David Karger's paper on this problem [Kar95], as he demonstrated that this reliability problem can be solved by reducing it to the problem of DNF counting, which has a FPTAS [KLM89]; reliability can therefore be solved in $O(\frac{cn^4}{\epsilon^3})$ time.

Although a solution to the reliability problem has been conceived in theory, it remains to be seen whether or not this solution is useful when applied to real networks. Thus, this thesis implements Karger's approximation scheme and explores its practicality. In addition, the algorithm was modified to make it more efficient for several different types of test networks, which hopefully translates into better performance for real networks.

Chapter 2

Implementation

2.1 Overview

Karger's approximation scheme has two paths which it can take. For graphs with a relatively large failure probability, a naive Monte-Carlo algorithm (MCA) is used to determine the network reliability. For graphs with relatively small failure probability, a two-step procedure is used to determine reliability.

The MCA, discussed in section 2.2, works by running several trials, where each trial consists of simulating edge failures and then checking to see if the graph is connected. The graph failure probability is then estimated by taking the percentage of all trials that caused the graph to become disconnected. The total number of trials needed for the MCA to produce a close estimate is inversely proportional to the graph failure probability. Thus, in graphs with a high failure probability, the MCA provides an accurate estimate of reliability using a reasonable number of trials. In graphs with a small failure probability, however, too many trials are needed to find enough disconnected graphs to give an accurate estimate with high probability. In this case, a two step procedure is used.

The two step procedure is as follows. We first find the small cuts, where a cut is a set of edges whose removal would cause the graph to become disconnected. A cut with the least number of edges possible is called the minimum cut, and the total weight of the edges in the minimum cut is called . If the input graph is not weighted, then c is simply the number of edges in the minimum cut. We define an α-minimum cut as one whose weight is less than or equal to αc. α-minimum cuts for small α are the sets of edges where the network is most likely to fail, since the larger a cut gets the less likely all of its edges are to fail simultaneously and cause the graph to become disconnected. In his paper [Kar95], Karger demonstrates that in graphs with a small chance of failure, the probability that one of these α-minimum cuts fails comprises most of the graph's failure probability, so we do not have to examine all cuts.

After finding these small cuts, we approximate the overall probability of network failure by representing these cuts as a formula in disjunctive normal form and finding the truth probability of this formula. If this formula is true, it means that at least one of the small cuts has failed. The Recursive Contraction Algorithm (RCA), discussed in section 2.4, was used to find the small cuts [KS93], and the Self-Adjusting Coverage Algorithm (SACA), discussed in section 2.5, was used to determine the truth probability of the formula in disjunctive normal form [KLM89].

We first attempted a faithful implemention of the algorithm specified in the theory paper. This algorithm runs the MCA on graphs using the hypothesis that $FAIL(p)$, the probability the graph is disconnected given that an edge fails with probability p, is larger than $\frac{1}{n^4}$. If the result from the MCA suggests that $FAIL(p)$ is less than $\frac{1}{n^4}$, the RCA/SACA is run to get a better estimate of the reliability of the graph. Otherwise, the result from the MCA is considered a close enough estimate. According to the theory paper on this scheme, this split between the MCA and the RCA/SACA results in a running time of $O(n^4)$. In practice, however, we discovered that certain modifications could be made to reduce running time.

2.2 Naive Monte Carlo Algorithm

Graphs with a relatively high likelihood of failure can use a naive Monte Carlo method to calculate their reliability. We used the naive Monte Carlo algorithm described in Karp, Luby, and Madras's paper and implemented it as follows [KLM89]:

N trials are run, where each trial consists of the following.

(1) The on/off state of each edge in the graph is set according to its failure probability.

(2) The graph is checked to see if it is still connected. A variable, Y, is incremented if the graph is not connected.

After N trials, $FAIL(p)$ is estimated by $FAIL(p) = \frac{Y}{N}$. The number of trials, N, is obtained from the Zero-One Estimator Theorem [KLM 89].

The following definitions are needed for the theorem. Let U be the complete set of possibilities. In our case, it consists of the entire probability space. Let G be the event in U whose probability we wish to estimate using a Monte-Carlo algorithm. In our case, $\Pr[G]$ is $FAIL(p)$, the sum of the probabilities of all edge configurations that result in the graph being disconnected. Finally, an (ϵ, δ)-approximation algorithm is one that produces an estimate that is between $1-\epsilon$ and $1+\epsilon$ times the actual value with probability at least $1-\delta$.

Theorem 2.2.1 (Zero-One Estimator Theorem) *Let $u = \Pr[G]$. Let $\epsilon \leq 2$. If $N \geq \frac{1}{u} \cdot \frac{4ln(\frac{2}{\delta})}{\epsilon^2}$, then the Monte-Carlo algorithm described above is an (ϵ, δ)-approximation algorithm.*

For the faithful implementation of the theory paper, we use the MCA whenever $FAIL(p) = \Pr[G]$ is larger than $\frac{1}{n^4}$. Since $u = \Pr[G]$ and we arbitrarily choose ϵ and δ, we can solve for a bound on N easily. Substituting $\frac{1}{n^4}$ for u, we deduce that running the MCA for $N = n^4 \cdot \frac{4ln(\frac{2}{\delta})}{\epsilon^2}$ trials will produce an estimate that is between $1-\epsilon$ and $1+\epsilon$ times the actual value with probability at least $1-\delta$.

For this part of the algorithm, the graph was represented as an adjacency list, with additional storage provided for the failure probability of each edge. This representation enabled fast connectivity checking, which was implemented as a depth-first search of the graph starting from an arbitrary node with marking of all nodes encountered. If all nodes became marked, the graph was connected.

2.3 Restriction to Small Cuts

When the failure probability of a graph is small, the MCA does not work because too many trials are needed to obtain an estimate with good accuracy. In this case, another method is needed to estimate the reliability.

One way of computing the exact reliability of a graph is to enumerate all cuts and then find the probability that at least one of the cuts fails. Enumerating all cuts of a graph, however, can be very time consuming as there can be 2^n cuts. Finding the probability that at least one cut fails is also non-trivial when the number of cuts is large.

Since enumerating all cuts usually proves too time consuming, we wish to take some subset of the cuts and find the probability that the subset fails. Karger proved in [Kar95] that the probability that a cut of value exceeding αc is exponentially small in α. Thus, if we wish to to obtain an estimate of the reliability that is between $1+\epsilon$ and $1-\epsilon$ times its actual value, we need only to find all α-minimum cuts such that the probability that a cut of value greater than αc fails is less than ϵ times the actual value. Since we do not know the actual failure probability, the theory paper uses the probability that a minimum cut fails as a pessimistic estimate of graph failure.

For graphs with a relatively low chance of failure, the value of α necessary to perform this type of estimation is low enough so that the α-minimum cuts can be found quickly. The Recursive Contraction Algorithm is used to find these near minimum cuts.

2.4 Recursive Contraction Algorithm

Originally, the RCA was designed to find only minimum cuts. We will discuss the theoretical RCA in the following section. In reality, we were given an implementation of the RCA done by Karger and Stein which contained a few modifications on the basic contraction algorithm. These modifications will be discussed in section 2.4.2. Finally, the RCA had to be modified to find α-minimum cuts for this thesis. These changes will be discussed at length in section 2.4.3.

2.4.1 Theoretical Recursive Contraction Algorithm

The RCA uses the Contraction Algorithm (CA), which consists of the following [KS93]:

(1) Choose an edge uniformly at random from the input graph G.

(2) Contract this edge, where contraction consists of replacing the endpoints of the edge with a single vertex. The set of edges incident to this new vertex is the union of the sets of edges incident on the original endpoints.

(3) Replace G with the partially contracted graph, which has one less vertex. Repeat steps 1-3 until a specified number of vertices remain.

The intuition for the CA is that after each iteration, the number of vertices is reduced by one. When the graph has only two vertices left, the set of edges connecting the two vertices defines a cut. This cut is the minimum cut if no contraction has contracted an edge from the minimum cut in previous iterations. If we run the CA enough times, we are assured that with high probability the minimum cut will survive contraction to two vertices in at least one of the trials.

The original Recursive Contraction Algorithm, which only finds the minimum cut, consists of the following:

Given a graph G with n vertices,

(1) If G has fewer than 6 vertices perform the Contraction Algorithm on it until 2 vertices remain. Return the weight of the cut defined by the the two remaining vertices.

(2) Otherwise, run the Contraction Algorithm on G until $\lceil \frac{n}{\sqrt{2}} + 1 \rceil$ vertices remain, and run the Recursive Contraction Algorithm recursively on the resulting graph. Do this procedure twice and return the smaller of the two values.

The RCA is designed so that each call to the CA has at most a $\frac{1}{2}$ chance of contracting an edge in the minimum cut. The RCA runs two trials of the CA so that on the average, one of the two recursions will preserve the minimum cut. The RCA is also run for several iterations to produce the minimum cut with even higher probability.

We now wish to determine how many iterations of the RCA need to be run to ensure that the success probability for finding the minimum cut is greater than some desired value. To answer this question, we need to find the success probability of one iteration of the algorithm. Karger and Stein solved this problem using the following recurrence, where $P(t)$ is the probability that the minimum cut is found at the t^{th} level of the tree above the leaves.

$$P(t) = P(t-1) - \frac{1}{4}P(t-1)^2$$

The recurrence is derived from the fact that each time the RCA is called, it splits into two branches. In each branch, the RCA runs the CA and then recursively calls itself using the result from the CA as the input. A branch finds the minimum cut if and only if the CA preserves the minimum cut, and its recursive call to the RCA succeeds in finding the minimum cut of its input. Let $P(t)$ be the probability that the parent finds the minimum cut of its input graph and let $P(t-1)$ be the probability that the branch finds the minimum cut of its input graph. Since the CA succeeds with probability $\frac{1}{2}$, the probability that a branch finds the minimum cut is $\frac{1}{2}P(t-1)$. The algorithm succeeds if at least one branch succeeds. Using the fact that for two events A and B, $P(A \cup B) = P(A) + P(B) - P(A)P(B)$, and that $P(A)$ and $P(B)$ in this case are $\frac{1}{2}P(t-1)$, we arrive at the recurrence above. [KS93] showed that $P(t) = \theta(\frac{1}{t})$. Thus, to find the success probability of one tree, we only need to know the tree's height. Since a set fraction of the vertices are contracted after every call to the RCA, it is easy to find this height.

The running time of the RCA as listed in [KS93] is $O(n^2 \log^3 n)$. A rigorous analysis of the two algorithms listed in this section is given in the same paper.

2.4.2 Karger and Stein's Implementation of the RCA

In their implementation of the RCA, Karger and Stein made a few modifications that decreased the running time of the algorithm for some cases. Instead of contracting to $\lceil \frac{n}{\sqrt{2}} + 1 \rceil$ vertices one vertex at a time, each edge is given a contraction probability which governs how likely a particular edge is contracted in one branch of the RCA. This probability is chosen so that the the probability the minimum cut is lost is less than $\frac{1}{2}$ for one iteration. Since this bound is the only assumption used to prove the correctness of the RCA in [KS93], the overall algorithm is still correct. While the implementation is still correct, it also allows many graphs to be contracted more quickly than the theoretical RCA. Consider a graph with a minimum cut of 1 edge. If the number of edges is large, than the likelihood of contracting the minimum cut edge during one level of the theoretical RCA tree is much less than one half since it contracts until a set number of vertices remain. Using edge contraction probabilities, we ensure that enough edges are contracted so that the probability that the minimum cut survives is exactly one half. We now demonstrate how this edge contraction probability was derived and explain the new calculation of recursion depth necessary for this modification.

Finding the edge contraction probability

We wish to determine the correct probability of contracting an edge which assures that the probability of contracting an edge in the minimum cut is at most $\frac{1}{2}$. Assume we are given a weighted graph containing a minimum cut with k edges whose weights are $w_1 \dots w_k$, and let c be the total weight of the cut, such that $c = \sum_1^k w_i$.

Statement 1:

We decide that the probability a particular edge in the minimum cut is not contracted by one iteration of the contraction algorithm is $e^{-\beta w}$ where β is a constant to be determined and w is the weight of the edge.

Statement 2:

The probability that no edges in the minimum cut are contracted by the contraction algorithm is $e^{-\beta c}$.

Statement 2 follows from statement 1, since no edges in the minimum cut are contracted if and only if each particular edge survives contraction. The probability that each particular edge avoids contraction is the product of the likelihoods, which simplifies to $e^{-\beta c}$.

We wish for this probability to be at least $\frac{1}{2}$, so we set:

$$e^{-\beta c} = \frac{1}{2}.$$

Solving for β, we get:

$$\beta = \frac{\ln 2}{c}.$$

Substituting β in for the probability in statement 1, we find that the probability that an edge should survive contraction is $e^{-(\frac{\ln 2}{c})w}$, which can be simplified to $2^{-\frac{w}{c}}$. Thus, if the RCA is modified so that the Contraction Algorithm tries to contract each edge with probability $2^{-\frac{w}{c}}$, the minimum cut still survives with probability at least $\frac{1}{2}$ after each iteration of the algorithm.

Finding depth

Since Karger and Stein's implementation did not contract a set fraction of the vertices during each call to the RCA, the depth of the tree was slightly more difficult to find. The depth of the RCA's tree is calculated in the following manner. A vertex is contracted away if any of its incident edges are contracted during the algorithm. The vertex most likely to survive would be one whose edges formed a minimum cut, since the probability of an edge being contracted is proportional to its weight. Given such a vertex exists, we know that the probability that it survives is $\frac{1}{2}$. This is true since an edge is contracted with probability $2^{-\frac{w}{c}}$, and the sum of edges' weights for such a vertex would be c. Thus, the probability that none of the edges would be contracted is $2^{-\frac{c}{c}}$, or $\frac{1}{2}$.

Through the previous analysis, we know that a vertex has at most a $\frac{1}{2}$ chance of survival through one branch of the contraction algorithm. It follows that half of the vertices will have an incident edge contracted. Using this knowledge, we can find what percentage of the vertices remain after one level has been completed. In the worst case, the contracted vertices would form a matching, so that every two contracted vertices are connected by a contracted ed . In this case, each pair of vertices is contracted into one vertex. Thus, in the worst case we expect that at most $\frac{3}{4}$ the original number of vertices will remain after one level of the contraction algorithm. Using this information, we conclude that the depth of the tree is $\log_{\frac{4}{3}} n$, where n is the number of vertices in the original graph.

2.4.3 Finding Near Minimum Cuts

Since Karger's approximation scheme requires near minimum cuts to be enumerated, the RCA had to be modified to find cuts larger than the minimum cut. One modification we made was to change the contraction probability for an edge to assure that an α-minimum cut survives with probability at least $\frac{1}{2}$. Since changing the edge contraction probability affects the depth of the RCA tree, we also had to modify the number of iterations to ensure the same level of accuracy. In addition, the graph was only contracted until $\lceil 2\alpha \rceil$ vertices remained. The necessity of this modification is discussed in [KS93]. Finally, since our analysis only ensures that at least 1 α-minimum cut will be found with high probability, the number of iterations had to be adjusted again to account for the fact that we need to find all α-minimum cuts.

To modify the edge contraction probability, we assume that there exists at least 1 α-minimum cut with k edges, whose weights are $w_1 \ldots w_k$ respectively, and whose weight is less than or equal to αc. As discussed before in section 2.4.2, the probability that a particular edge survives contraction is $e^{-\beta w}$. The probability that the entire cut survives is at least $e^{-\beta \alpha c}$, again as before. Setting this probability to $\frac{1}{2}$ and solving for β, we get $\beta = \frac{\ln 2}{\alpha c}$. Substituting β in, we find that the probability that a particular edge should survive contraction is $e^{-(\frac{\ln 2}{\alpha c})w}$, which is simplified to $2^{-\frac{w}{\alpha c}}$.

The modification to edge contraction probability affects the depth in the following manner. In the α-minimum case, the vertex most likely to survive an iteration would still be one whose edges form a minimum cut. The probability that such a vertex would survive is $2^{-\frac{c}{\alpha c}}$, or $2^{-\frac{1}{\alpha}}$. Thus, we would expect that at least $1 - 2^{-\frac{1}{\alpha}}$ of the vertices would have an incident edge contracted and that $1 - \frac{1}{2}(1 - 2^{-\frac{1}{\alpha}})$ vertices would remain after each level. So, in the α-minimum case, the depth is $\log_{\frac{1}{1-\frac{1}{2}(1-2^{-\frac{1}{\alpha}})}} n$, where n is the number of vertices in the original graph. Currently, Karger is developing a more complete proof for determining depth that does not increase the calculation of depth by more than a constant factor.

Using the depth, we can find the likelihood that we will find a particular α-minimum cut. Since the recurrence in section 2.4.1 still applies, this probability is $O(\frac{1}{depth})$. We still, however, need to find how many trials of the RCA are needed to find all α-minimum cuts.

For this problem, assume there are i α-minimum cuts that we wish to find. Let c_i be the event that we find the ith α-minimum cut at any leaf of the RCA tree. We wish to find how many trials of the RCA it takes so that $\Pr[\cap c_i]$ is very high.

If the events were independent, $\Pr[\cap c_i]$ would be easy to calculate, as it would simply be the product of all c_i. Unfortunately, each event is not fully independent, since finding a cut at a particular leaf precludes the possibility of finding some cuts at neighboring leaves. This is true because contracting away some cuts at a higher level of the tree was necessary to find the cut.

So, instead of trying to calculate $\Pr[\cap c_i]$, we will instead find the probability of its complement $\Pr[\cup \overline{c_i}]$. Regardless of the events' independence, we know that $\Pr[\cup \overline{c_i}] \leq \sum \Pr[\overline{c_i}]$ using the Union Bound. By Theorem 2.2 from Karger's paper, we also know that there are at most $n^{2\alpha}$ α-minimum cuts [Kar95]. By forcing $\Pr[\overline{c_i}]$ to be less than $\frac{\beta}{n^{2\alpha}}$, where β is some constant to be determined, we know that $\sum \Pr[c_i]$ is at most β. Thus, $\Pr[\cup \overline{c_i}]$ is at most β, and $\Pr[\cap c_i]$ is at least $1 - \beta$.

Now we only need to know how many times to run the RCA to ensure that $\Pr[\overline{c_i}]$ is less than $\frac{\beta}{n^{2\alpha}}$. Let q be the probability that an iteration of the RCA does not find a particular α-minimum cut, where q is derived from the recurrence given in 2.3.1. If we repeat the algorithm k times, $\Pr[\overline{c_i}] = p^k$. Setting p^k to be less than or equal to $\frac{\beta}{n^{2\alpha}}$, we arrive at the following value for k.

$$k \geq \log_p \frac{\beta}{n^{2\alpha}}$$

Thus, as long as at least k trials of the modified RCA are run, we are guaranteed to find all α-minimum cuts of the input graph with probability 1 - β.

2.5 Self-Adjusting Coverage Algorithm

After the RCA is used to find the approximately minimum cuts, a formula in disjunctive normal form (DNF) is derived from the resultant cut list. Let m be the number of edges in the graph and l be the number of cuts in the cut list. The formula contains l clauses, $C_1 \ldots C_l$, where each clause is a conjunction of a subset of literals defined with respect to m boolean variables $X_1 \ldots X_m$. Each clause corresponds to a cut, where X_k is in the conjunction if and only if the kth edge of the input graph is in the cut. If an edge in the input graph fails, its corresponding variable in the formula is set to true. Thus, a cut causes the graph to become disconnected when all of its variables become true. By finding the probability that at least one clause is true, we can determine the failure probability due to the approximately minimum cuts. The Self Adjusting Coverage Algorithm (SACA) is used to find this probability.

The SACA is a Monte-Carlo algorithm used to solve the Union of Sets problem, which we now define [KLM89]. The input to the union of sets problem is s sets, $D_1 \ldots D_s$, and the goal is to compute the cardinality of $D = \bigcup_{i=1}^{s} D_s$. The DNF probability problem that we are trying to solve is a special case of this problem. Each clause can be thought of as a set whose elements are the variable configurations that cause the clause to be true. Solving the union of sets problem for the clauses counts the number of variable configurations that cause at least one of the clauses to be true. By adding the probabilities of each of these variable configurations, we can determine the overall probability that the DNF formula will be satisfied.

The SACA requires that a clause be chosen from the list of clauses, and that the variables be set so that the clause is satisfied. The following procedure was used to ensure that the variables were chosen with uniform randomness [KL83]:

1) Choose $i \in 1, 2, \ldots, l$ with probability $\frac{\Pr[D_i]}{\sum_{i=1}^{l} \Pr[D_i]}$.

2) Choose configuration $s \in D_i$ with probability $\frac{\Pr[s]}{\Pr[D_i]}$.

Step 1 was implemented by first calculating the truth probability of each clause.

This probability was the probability that all edges fail in the cut defined by a clause. A selection array was then created, and each clause was assigned a portion of the array whose size was proportional to the likelihood of the clause being true. A pointer was assigned randomly to the array, and the clause pointed to was assigned to i. Step 2 was implemented by setting all variables in clause i so that i is satisfied, and then randomly assigning values to the remaining variables according to the probability defined by the corresponding edge in the graph.

The SACA is a modification of the KLM's Coverage Algorithm [KLM89], which is also used to solve the Union of Sets problem. In the Coverage Algorithm, an ordering is defined for the clauses. For each trial, a clause is chosen at random and variables are set so that the clause is satisfied. The trial is considered a success if the clause chosen is the smallest clause as defined by the ordering that is made true by the configuration of variables. Thus, each configuration is only counted in at most one clause. An estimate of the size of the union of sets can be obtained by taking number of successes and dividing it by the number of trials. The algorithm improves on a naive Monte Carlo algorithm since it automatically assumes a variable configuration within the subset defined by the clauses. The most costly step in the Coverage Algorithm is determining if the clause chosen is the smallest one possible. The SACA improves the Coverage Algorithm by reducing the amount of time used to make this determination.

In the SACA, during each trial a clause and a variable configuration are chosen according to the method described before. Instead of running a set number of trials, however, the algorithm sets an upper limit on how many steps are run. A step in the SACA is choosing a random clause and seeing if the variable configuration satisfies the clause. A trial ends only when a step has succeeded in satisfying the randomly chosen clause. The success of the trial is determined by how many steps were needed before the trial completed. If more steps were needed, then the trial was considered more successful. The intuition behind performing steps is that if a large number of steps are used for a trial, the particular variable configuration must be in fewer of the clauses. Thus, it must be counted more heavily in determining the union of sets. If a trial takes fewer steps, it means that the variable configuration most likely satisfies many clauses. Thus, it will be chosen often in the selection of variable configurations and must be counted less heavily since the goal is to count each configuration only once.

Self-adjusting Coverage Algorithm Theorem II [KLM89] is used in the SACA to determine the number of steps, T, necessary for completion, where a step is as defined in the previous paragraph.

Theorem 2.5.1 (Self-adjusting Algorithm Theorem II) *When $\epsilon < 1$ and $T = \frac{8 \cdot (1+\epsilon) \cdot l \ln(\frac{3}{\delta})}{(1-\frac{\epsilon^2}{8})\epsilon^2}$, the self-adjusting coverage algorithm is an ϵ, δ approximation algorithm when estimator $Y = \frac{\sum_{i=1}^{T} Y_i}{N_T}$ is used.*

The SACA produces estimator Y, which is according to Self-Adjusting Coverage Algorithm II an (ϵ, α)-approximation for the probability that one of the cuts fails. Thus, by finding the approximately minimum cuts of a graph and then using the SACA to determine the failure probability of these cuts, we can approximate the failure probability of the graph. Since the RCA is accurate to within $1 \pm \epsilon$, and the SACA is also accurate to within $1 \pm \epsilon$ with high probability, our estimate is accurate to within $(1 \pm \epsilon)^2$ of the actual failure probability with high probability.

Chapter 3

Optimizing Performance

The theoretical algorithm attempts to balance the time used by the naive Monte Carlo Algorithm and the Recursive Contraction Algorithm/Self-Adjusting Coverage Algorithm by running the RCA/SACA only for $FAIL(p) < \frac{1}{n^4}$. This division of labor theoretically results in both paths of the algorithm having the same $O(n^4)$ running time. In practice, however, we noticed that the MCA approach took much more time than the RCA/SACA approach, even for values of $FAIL(p)$ larger than $\frac{1}{n^4}$. To avoid having to find the values of $FAIL(p)$ for which the RCA/SACA was faster for each particular graph, we decided to use an adaptive approach that would ensure that the faster algorithm would be run in all cases. The approach we decided to adopt was dovetailing, in which the MCA and RCA/SACA are essentially run in parallel.

While testing our implementation we also noticed that the theoretical algorithm lists more cuts than necessary during the RCA/SACA portion for most graphs. This situation occurs because the theoretical algorithm uses a pessimistic estimation of the graph failure probability, namely the probability that the minimum cut fails, during its calculation of how much error to allow. In many cases, however, the failure probability is much greater than this value. If the algorithm were somehow supplied a better estimation of graph failure, it could allow more leeway and thus examine fewer cuts. A natural way to come up with this estimate was to use Karger's approximation algorithm, except allowing a larger margin of error to decrease the running time for this application of the algorithm. We implemented the dovetail optimization as well as this optimization and analyzed their effects on the algorithm's speed.

3.1 Dovetailing

The first optimization we performed was to dovetail the naive Monte Carlo algorithm (MCA) portion of the implementation with the Recursive Contraction Algorithm/Self Adjusting Coverage Algorithm(RCA/SACA) portion of the implementation. Our rationale for this optimization was that during initial testing we discovered that the MCA portion potentially dominates running time. For a randomly generated graph with 10 nodes, ϵ set to 0.05, δ set to 0.05, and using the hypothesis that $FAIL(p) \geq \frac{1}{n^4}$, the Zero-One Estimator Theorem states that over 50,000,000 trials need to be run in order to assure that the Monte Carlo algorithm is an ϵ, δ approximation algorithm. By dovetailing, we are assured that the implementation terminates when the faster of the two portions completes.

The scheme we used to dovetail the MCA and RCA/SACA is as follows:

(1) Run the RCA to find the minimum cut of the graph. Using the minimum cut information, determine α to ensure that the probability that any cut of value greater than αc fails is less than ϵp^c, where c is the weight of a minimum cut and p^c is the probability that a particular minimum cut fails.

(2) Run the RCA/SACA approach using this α for some arbitrary amount of time. We chose to use one second. If the RCA/SACA completes in this time, use the output from the RCA/SACA as our answer.

(2) If the RCA/SACA does not complete, run the MC for the same amount of time. If the MCA completes in that amount of time, then use the output from the MCA as our answer.

(3) If neither algorithm gives us an answer, double the time used for computation and repeat steps 1 and 2.

Let y be the number of iterations it takes for the dovetail implementation to complete. In a worst case scenario, the algorithm optimally takes 2^y time for some y and terminates in the MCA before the RCA/SACA. In this situation, our dovetail implementation would run for $2^{y+1} + 2(2^y)$ time in the RCA/SACA, and then run for $2(2^y)$ in the MC. In this situation, the total time used is still within a factor of six of the optimal time.

3.1.1 Computing α for the Dovetail Implementation

Since we do not use the theory paper's assumption that $FAIL(p) < \frac{1}{n^4}$ to run the RCA/SACA for the dovetail implementation, we need to find the right value for α that makes the error less than ϵ with high probability for an arbitrary $FAIL(p)$. To find α, we use the following claim.

Claim 3.1.1 *Given α such that $n^{2\alpha}p^{\alpha c}(1 - \frac{\ln n^2}{\ln n^2 p^c})$ is less than ϵp^c, the probability that any cut of value greater than αc fails is less than ϵp^c.*

Proof: In Karger's proof of Theorem 3.1 from [Kar96] on this approximation scheme, he shows if we write p^c as $n^{-(2+\beta)}$, the probability that a cut larger than αc fails is $O(n^{-\beta\alpha})$. We need the exact probability, however, to calculate α accurately.

To calculate this probability, we examine the proof of Theorem 2.3 from the same paper. In this proof, Karger uses the following assumptions. Let r be the number of cuts in the graph, and let $c_1, ..., c_r$ be the values of the r cuts. Assume the values of c_i are in increasing order.

By Theorem 2.2 of the paper, we know there are at most $n^{2\alpha}$ cuts of value less than αc. Thus, cuts $c_1, ..., c_{n^{2\alpha}}$ can be cuts of any value greater than c and cuts $c_{n^{2\alpha}+1}, ..., c_r$ must have values larger than αc.

For cuts $c_1, ..., c_{n^{2\alpha}}$, the minimum contributing cut value to the failure probability is αc since we want the probability that a cut larger than αc fails. Thus, the contributions of the first $n^{2\alpha}$ cuts is bounded by $n^{2\alpha}p^{\alpha c}$. Karger shows that the contributions of the remaining cuts is bounded by the integral, $\int_{n^{2\alpha}}^{r} k^{-(1+\frac{\beta}{2})}dk$. This integral is less than $\frac{2n^{-\alpha\beta}}{\beta}$ if $\beta > 0$. Thus, the overall probability that a cut of value greater than αc fails is less than

$$n^{2\alpha}p^{\alpha c} + \frac{2n^{-\alpha\beta}}{\beta}$$

Solving for β in terms of p^c, we determine:

$$\beta = -(\frac{\ln p^c}{\ln n} + 2)$$

Substituting β into our bound, we find:

$$\Pr[\text{Cut} > \alpha c \text{ fails}] \leq n^{2\alpha} p^{\alpha c} - \frac{2n^{\alpha(\frac{\ln p^c}{\ln n}+2)}}{(\frac{\ln p^c}{\ln n}+2)} = n^{2\alpha} p^{\alpha c}(1 - \frac{\ln n^2}{\ln n^2 p^c})$$

We set this value to be less than ϵp^c and solve for α to determine the correct number of cuts to examine.

3.1.2 Monte-Carlo Algorithm Termination Criteria

By removing the $\frac{1}{n^4}$ barrier between the MCA and the RCA/SACA, the dovetail implementation raised another question. Since we could no longer assume that $FAIL(p)$ is larger than $\frac{1}{n^4}$ when running the MCA, we had to develop a termination criterion for the MCA that did not require an estimate of the failure probability. To do this, we defined the concept of self-verification. We say that the MCA self-verifies if it encounters enough disconnected graphs so it is very likely that its estimate of the graph failure probability is accurate to within ϵ of the actual failure probability. If the MCA self-verifies within its allotted amount of time according to the dovetail implementation, it has run a sufficient number of trials and its answer can be used. The following lemma answers how many disconnected graphs are needed for self-verification.

Lemma 3.1.2 *The naive Monte Carlo algorithm self-verifies if it encounters at least* $(1+\epsilon)\frac{4\ln(\frac{2}{\delta})}{\epsilon^2}$ *disconnected graphs.*

Proof: Before we prove this statement, recall the Zero-One Estimator Theorem and our analysis of the naive Monte Carlo algorithm from section 2.2. By the theorem, we know that we need to run at least $N = \frac{1}{FAIL(p)} \cdot \frac{4ln(\frac{2}{\delta})}{\epsilon^2}$ for the algorithm to be an ϵ, δ algorithm.

If N trials are run, we expect that k, the number of failed trials, is approximately $FAIL(p)N$. The $FAIL(p)$ term in k cancels out with the $FAIL(p)$ term in N, resulting in $k = \frac{4\ln(\frac{2}{\delta})}{\epsilon^2}$. We now proceed to prove that running the MC until $(1+\epsilon)k$ failures are encountered and dividing the number of failures by the number of total trials is an ϵ, δ approximation of a graph's reliability.

For the purposes of this proof, let $l = (1+\epsilon)k$. To prove that searching for l failures is sufficient, we consider an infinite sequence of trials. The expected value of T_l, the trial at which the lth failure is encountered, is $\frac{l}{FAIL(p)}$, a number larger than N for $\epsilon > 0$. We will call this value $E[T_l]$. Consider two times $\frac{E[T_l]}{1+\epsilon}$ and $\frac{E[T_l]}{1-\epsilon}$. The Zero-One Estimator Theorem applies to both times since $\frac{E[T_l]}{1+\epsilon} = N$ and $\frac{E[T_l]}{1-\epsilon} > N$.

By the theorem, we know that at time $\frac{E[T_l]}{1-\epsilon}$ at least $(\frac{E[T_l]}{1-\epsilon})FAIL(p)(1-\epsilon)$ failures will be encountered with probability δ. This expression simplifies to $E[T_l]FAIL(p)$, or l, so we know that when l failures occur, $T_l < \frac{E[T_l]}{1-\epsilon}$ with probability δ.

Similarly, the theorem states that at time $\frac{E[T_l]}{1+\epsilon}$, at most $(\frac{E[T_l]}{1+\epsilon})FAIL(p)(1+\epsilon)$ failures are encountered with probability δ. This expression also simplifies to l. By the Union Bound Theorem, we know the following is true with a probability of at least $1-2\delta$.

$$\frac{E[T_l]}{1+\epsilon} < T_l < \frac{E[T_l]}{1-\epsilon}$$

To complete our proof, we substitute $\frac{l}{FAIL(p)}$ for $E[T_l]$, divide by l and take the reciprocal which results in the following:

$$(1-\epsilon)FAIL(p) < \frac{l}{T_l} < (1+\epsilon)FAIL(p)$$

Thus, by taking $(1+\epsilon)k$ over the number of trials it takes to find the $(1+\epsilon)k^{th}$ failure, we obtain an ϵ, δ approximation for $FAIL(p)$.

3.2 Optimizing for multiple minimum cuts

In graph geometries in which there are multiple minimum cuts or many cuts that are close in size to the minimum cut, the approximation scheme can be made more efficient. The wasted computation occurs because in this case, $FAIL(p)$ is much larger than p^c, the probability that the minimum cut fails. In the theoretical paper, α is set so that the error caused by not examining cuts larger than α-minimal cuts is less than ϵp^c. In actuality, however, α needs only to be set so that the error caused by cuts larger than α-minimal cuts is less than $\epsilon FAIL(p)$. Thus, in cases where p^c is much smaller than $FAIL(p)$, our calculation for α is unnecessarily conservative.

To optimize for this case, we decided to use Karger's approximation scheme to obtain a coarse estimate of the failure probability, and then use this estimate to calculate the α necessary to guarantee a close estimate. Thus, we initially ran Karger's algorithm using an ϵ of 0.5 to get a quick estimate, $EFAIL(p)$, that is at most $2FAIL(p)$ with probability δ. Solving for $FAIL(p)$, we know that $FAIL(p) \geq \frac{EFAIL(p)}{2}$. We compare this estimate of $FAIL(p)$ with the probability that a minimum cut fails and use the larger value to obtain α.

α was calculated from our estimate of $FAIL(p)$ in the following manner. Recall Claim 3.1.1 states that given $n^{2\alpha}p^{\alpha c}(1-\frac{\ln n^2}{\ln n^2 p^c})$ is less than ϵp^c, the probability that any cut of value greater than αc fails is less than ϵp^c. By solving α so that $n^{2\alpha}p^{\alpha c}(1-\frac{\ln n^2}{\ln n^2 p^c})$ is less than $\epsilon \frac{EFAIL(p)}{2}$, we account for our more accurate estimate of $FAIL(p)$.

Chapter 4

Results

To analyze the performance of the approximation scheme, we ran our implementation on several families of graphs. The families were chosen to highlight the worst case running time of the scheme, and also to give an indication of how well the scheme would perform on test inputs from the real world. Performance was measured by recording how much CPU time (in seconds) was needed to compute the reliability for graphs of various sizes and edge failure probabilities.

4.1 Network Topologies

We ran the implementation on cycles, Delaunay graphs, and nearest neighbor graphs. Cycles were used to test the worst case running time of the scheme, and the other two network topologies were used because they closely model actual telecommunications networks. Finally, we tested our implementation on four real-life networks.

4.1.1 Cycles

A cycle is a graph in which there are exactly as many edges as there are nodes and each node is connected to two other nodes. The only way for such a graph to be 2-connected is for the edges to form a ring. In a cycle, a minimum cut is any two edges, since the loss of any two edges will disconnect the graph. Thus, there is an abnormally large number of near minimum cuts in a cycle, which is actually the maximum number of minimum cuts for the number of vertices. Since the goal of the RCA is to find these cuts, solving the reliability for cycle was a essentially a worst case test for the RCA/SACA portion of the algorithm.

Another interesting aspect of the cycle is that its reliability can be calculated easily using simple analysis. We note that the only way a cycle is not disconnected is if either no edges fail, or if exactly one edge fails. The probability of no edges failing is $(1-p)^n$. The probability that exactly one edge fails is $np(1-p)^{n-1}$. Thus, the failure probability of a cycle can be expressed by $1-(1-p)^n - np(1-p)^{n-1}$. Using this exact expression for reliability, we can measure the accuracy of our implementation.

4.1.2 Delaunay Graphs

To create a Delaunay graph, we first randomly place nodes in a plane. We then perform a triangulation of that plane by connecting nodes with edges that subdivide the area into triangles. The Delaunay triangulation has the property that the circumcircle of every triangle does not contain any points of the triangulation [Kra95]. We call the result of a Delaunay triangulation on a set of nodes a Delaunay graph. Netpad, a graph manipulation program created at Bellcore, was used to create the Delaunay graphs for this research.

This family of graphs was particularly interesting because a Delaunay graph is a fairly good approximation of a telecommunications network. Each node in the graph could represent a city in the network, and each edge a physical link between two cities. The nature of the triangulation causes nodes to be connected only to other nearby nodes, which is a realistic analogy to what happens in actual telecommunications networks. Thus, this family of graphs allows us to get a sense of how the algorithm would perform on real networks.

4.1.3 Nearest Neighbor Graphs

Another model of how real networks are constructed is the nearest neighbor graph. The idea for this type of graph was suggested by Milena Mihail of Bellcore. In this family of graphs, we are given two parameters, a *range* and a *density*. The *range* specifies how many of a node's nearest neighbors should be considered for connection, and the *density* specifies how many of the considered nodes will actually be connected. A graph generator created by Professor Karger was used to create the nearest neighbor graphs. The generator works by first placing a random assortment of nodes in a plane. The generator then iterates through all nodes and connects each node to a random choice of *density* of its *range* nearest neighbors, where near is defined by least distance. The resultant graphs are also a fairly good approximation of a city's telecommunications network in which neighborhoods are connected to some nearby neighborhoods, but not all.

4.1.4 Real Networks

Finally, our implementation was tested on four telecommunications networks from real life. These networks were supplied by a telecommunications company that requested anonymity. By solving these networks, we were able to obtain a sense of how long the scheme would take to approximate the reliability of a network designed by an engineer for a specific task.

4.2 Performance

To test the performance of the algorithm, we ran it each of the network families with varying sizes of n and values of p. For the test runs, the C code was compiled using the GNU C Compiler using the -O2 flag for optimizations. The runs were done on a SPARC 20 with a 60 MHz TI, TMS390Z55 CPU using an ϵ value of 0.1.

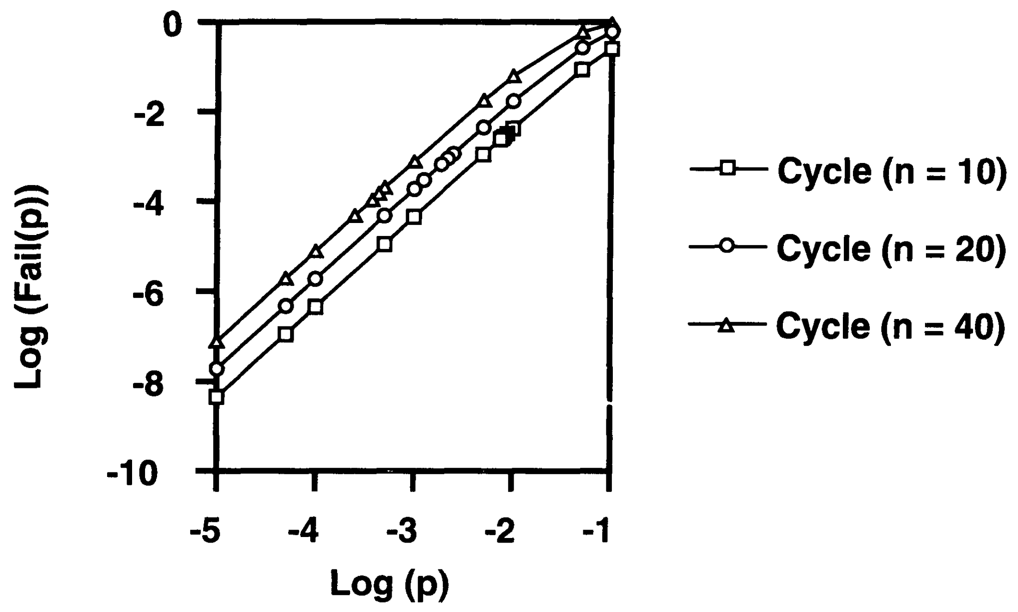

Figure 4-1: Graph Failure Probabilities for Cycles

4.2.1 Cycle Performance

With the cycle, we were able to use the exact analysis method to obtain correct values for $FAIL(p)$. These values were used to analyze the performance of the approximation algorithm in terms of accuracy. In addition, we measured the running time of the scheme for various cycle sizes and edge failure probabilities. Using these times, we were also able to analyze the speed of the algorithm for cycles.

Accuracy

The approximation scheme was run on cycles whose sizes ranged from 10 to 55 nodes using edge failure probabilities ranging from 10^{-1} to 10^{-5}. Figure 4-1 shows the results of our trials; it plots how the log of $FAIL(p)$ varies with the log of p for several sizes of cycles. As expected, the plot shows several straight lines that indicate graph failure probability increases polynomially with edge failure probability.

We used our exact analysis method to compute $FAIL(p)$ for the cycle sizes that

Nodes	Epsilon	MCA % Error	RCA % Error
10	0.10	2.124132%	0.460323%
15	0.10	1.149327%	0.585237%
20	0.10	0.849997%	0.721226%
25	0.10	2.141035%	0.635712%
30	0.10	2.141464%	0.493210%
40	0.10	1.356978%	0.680662%
45	0.10	1.905801%	0.677075%

Table 4.1: Average Error for Cycles

Nodes	Epsilon	MCA % Error	RCA % Error
30	0.10	2.141464%	0.493210%
30	0.15	2.732437%	0.825496%
30	0.20	4.822210%	1.244153%
30	0.25	4.013265%	1.106079%
30	0.30	8.459856%	1.923839%
30	0.40	4.952984%	2.485224%

Table 4.2: Average Error for Various ϵ for Cycles

we tested and compared these results with those obtained from the approximation scheme. Table 4.1 shows the average percentage error over all trials run for a particular cycle size and a particular ϵ that ended up using the MCA estimate, and the average error for trials that ended up using the RCA/SACA estimate. We note from the results that the accuracy of the scheme was on the average much better than we would guess from the value of ϵ used, especially with respect to the RCA/SACA estimates. Thus, we ran the scheme for several values of ϵ on a 30 node cycle with p varying from 10^{-1} to 10^{-5} to analyze the effect of ϵ on accuracy. Table 4.2 shows these results.

From Table 4.2, we see that even for values of ϵ as large as 0.4, we still obtain values for $FAIL(p)$ that are within 10 percent of the actual value on average. We also note that the difference between the MCA error and the RCA/SACA error increases as ϵ increases. This leads us to believe that using a different ϵ for the two parts of the algorithm could lead to a better balance of time used while still maintaining the same accuracy in practice.

Speed

To analyze the worst case speed of the algorithm for cycles, we first had to find the worst case values of p for each graph. This search was necessary because of the nature of our dovetail implementation. For large values of p, the MCA would complete quickly and terminate the algorithm. Similarly, for small values of p, the RCA/SACA would terminate first. We expect that the worst case time would occur when the MCA and RCA/SACA terminate simultaneously, so that no time is saved by dovetailing. Thus, we searched for values of p where this occurred using a binary search.

The next two figures show how the log of running time varied with $\log p$ and $\log FAIL(p)$ for several cycle sizes. We can see from the figures that the RCA/SACA is dominant at small values of p, and its running time increases slowly up to a point where the MCA begins to dominate. After this point, the running time falls off exponentially as p increases, as expected. It is interesting to note, however, that the worst case running time often did not occur where the binary search on p converged. This convergence is evident on Figure 4-2 as the p where there are several trials clustered. Instead of only one peak, there is a second peak in running time for a p that is slightly less than the p that the binary search found.

Finding this result unusual, we examined the trials for these values of p more carefully. We found that the odd peak in running time was caused by the coarse estimation process. Looking at the division of time between the coarse estimation and the more exact estimation, we found that the coarse estimation time dominated at the peak that was at lower values of p.

Looking at the trials, it appeared as if at high values of ϵ, the MCA is sped up greatly but the RCA/SACA is not affected as much. When we studied the effect of varying ϵ on the accuracy of the algorithm, we also noted the effect on running time. We were able to use this data to clarify our observation. We ran the approximation

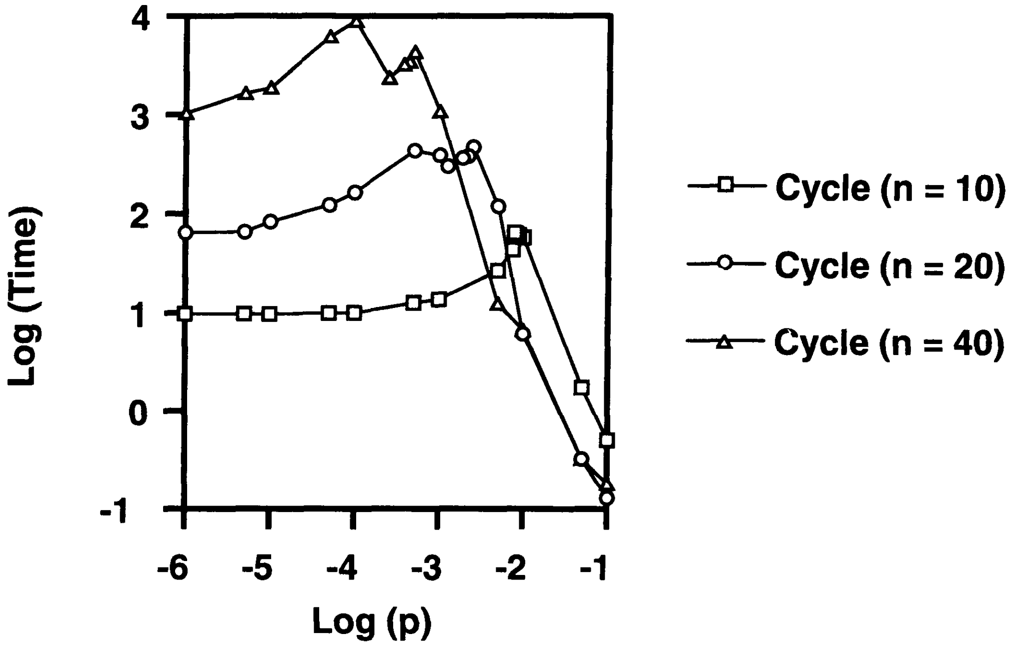

Figure 4-2: Running Time versus Edge Failure Probability for Cycles

algorithm on a 30 node cycle with varying ϵ for two values of p, one in which the MCA dominated ($p = 0.001$) and one in which the RCA/SACA dominated ($p = 0.0001$). Table 4.3 shows how running time was reduced as ϵ increased.

From Table 4.3, we see that increasing ϵ greatly reduces running times on trials for which the MCA is dominant, but has a much smaller effect on trials for which the RCA/SACA is dominant. As the running times for the MCA are greatly reduced as ϵ increases, we can deduce that at high values for ϵ, the MCA is effective at lower levels of p than it is at small values for ϵ. Since the RCA/SACA is not affected to the same degree as ϵ increases, we can assume that the value of p for which the MCA and the RCA/SACA take the same amount of time decreases as ϵ increases. Thus, the balance point for the coarse estimation must be different than the balance point for the more exact estimation. Since the coarse estimation process dominates time for cycles, it is that balance point that is creating the higher peak. Figure 4-4 overlays time used for coarse estimation on the time used for the exact estimation to clarify

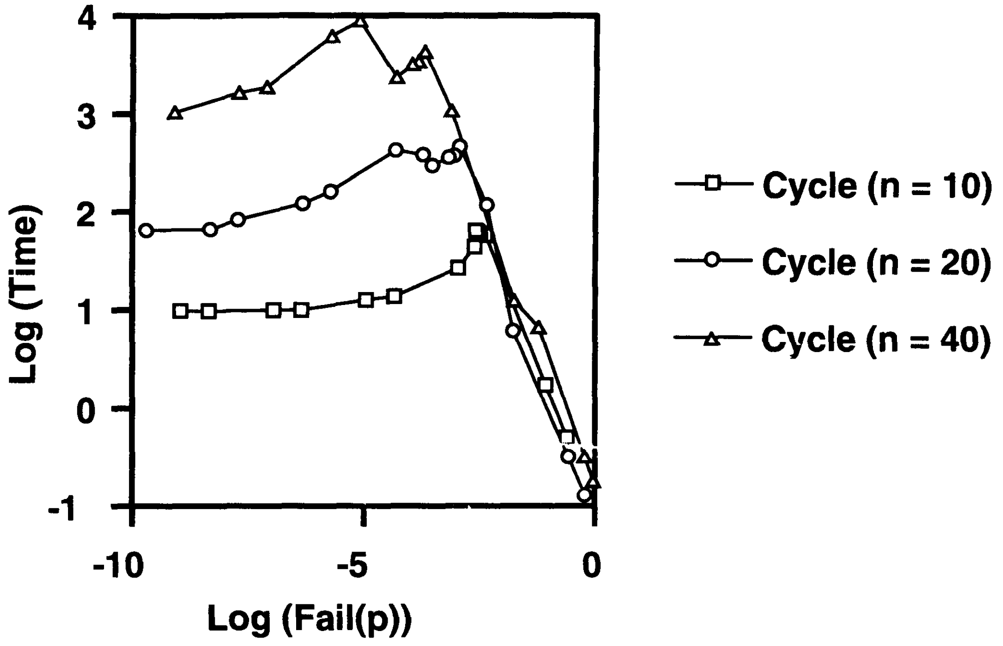

Figure 4-3: Running Time versus Graph Failure Probability for Cycles

our explanation for the double peak.

Figure 4-5 shows the increase in running time as the size of the cycle increases by plotting the log of running time versus the log of the number of nodes in the cycle. The worst case times for each cycle were used for this plot. Looking at the plot, we see that it is approximately a straight line. By taking the slope of the line, we can find the exponent on the running time. This slope was approximately 3.34, so we know that the algorithm runs in $O(n^{3.34})$ time for cycles, which is significantly better than the theoretical running time of $O(n^4)$.

4.2.2 Delaunay Graph Performance

We were satisfied with the algorithm's accuracy through its performance on cycles, and the reliability for the other families of graphs could not be determined analytically, so most of the rest of the tests were focused on analyzing the algorithm's speed for various graph topologies. We did, however, perform tests on the Delaunay graphs to

Nodes	Epsilon	MCA (p = 0.001)	RCA/SACA (p = 0.0001)
30	0.1	1971.12	1714.00
30	0.15	959.30	1472.72
30	0.2	525.42	1484.03
30	0.25	302.55	922.87
30	0.3	292.67	1422.33
30	0.4	176.90	1377.00

Table 4.3: Times for Various ϵ for Cycles

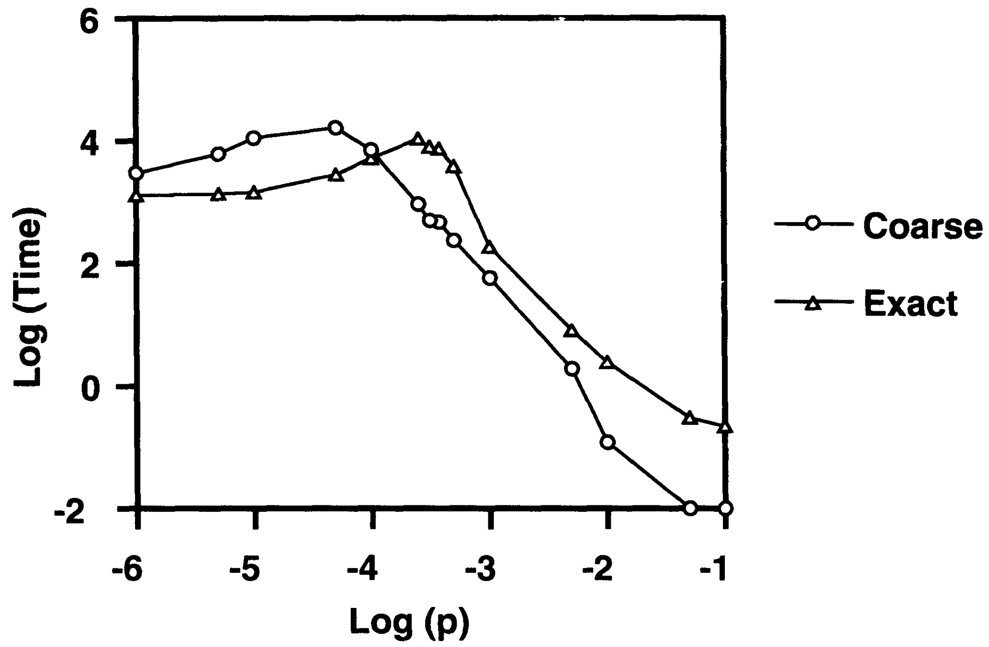

Figure 4-4: Coarse and More Exact Estimation for Cycles

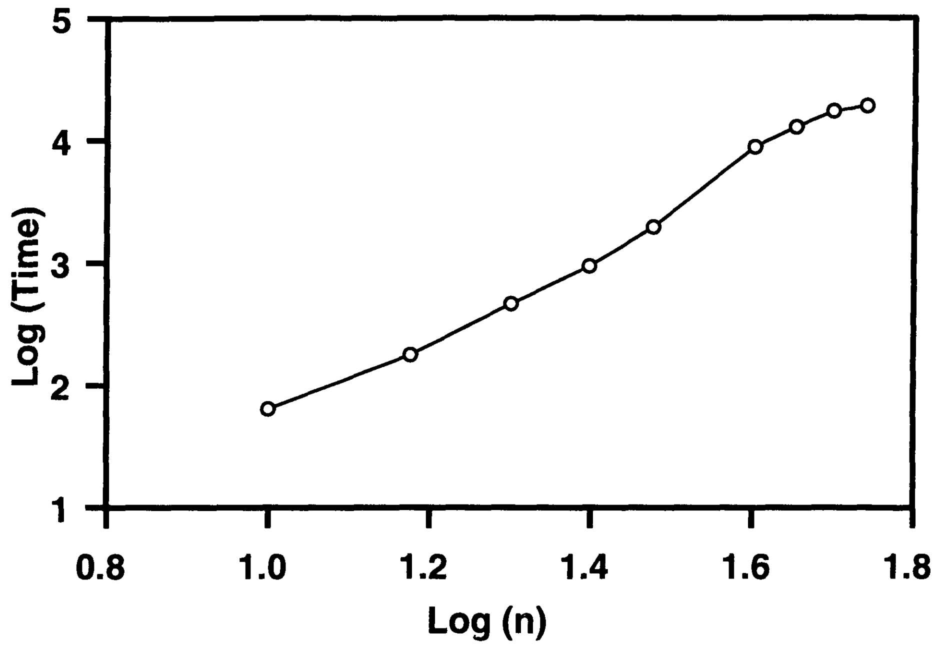

Figure 4-5: Worst Case Running Time for Cycles

further analyze the effects of changing ϵ on the algorithm's accuracy and speed. In addition, we measured the running time of the algorithm for various sizes and edge failure probabilities.

Effects of ϵ on Accuracy and Speed

To test the effects of varying ϵ, we ran the algorithm using values of ϵ from 0.1 to 0.4. We compared the reliability estimates obtained from higher ϵ values with the reliability estimate we got for $\epsilon = 0.1$ to ascertain the effects on accuracy. Runs were made on a 40 node Delaunay graph using several different edge failure probabilities to find the average change in accuracy for both the MCA estimate and the RCA/SACA estimate as ϵ increased.

To test the effect on speed, we varied ϵ from 0.1 to 0.4 and measured the time required to compute the reliability for two values of edge failure probability. One value ($p = 0.05$) was chosen to show the effect on the MCA, and the other ($p = 0.01$)

Nodes	Epsilon	MCA % Change	RCA % Change
40	0.15	1.484369%	2.841093%
40	0.20	2.964967%	2.044593%
40	0.25	6.041112%	0.639655%
40	0.30	6.641002%	2.960856%
40	0.40	10.577722%	0.581327%

Table 4.4: Average Change from $\epsilon = 0.1$ for Delaunay Graphs

Nodes	Epsilon	MCA (p = 0.05)	RCA/SACA (p = 0.01)
40	0.10	2448.92	112.05
40	0.15	988.50	105.57
40	0.20	631.23	66.12
40	0.25	464.90	65.17
40	0.30	367.70	86.75
40	0.40	277.62	63.50

Table 4.5: Times for Various ϵ for Delaunay Graphs

was chosen to show the effect on the RCA/SACA portion of the algorithm. Table 4.4 and Table 4.5 show the results of our tests.

The results for Delaunay graphs are very similar to our results for cycles. The error increases more quickly for the MCA as ϵ increases, but the error is still much below ϵ for both the MCA and RCA/SACA. The running time decreased greatly for the MCA as ϵ increased, but the increase has little effect on the running time of the RCA/SACA. Thus, although a higher value for ϵ probably can be used for the RCA/SACA without sacrificing accuracy for the overall algorithm, it is unknown how much we would need to increase ϵ before a significant decrease in the RCA/SACA running time would occur.

Speed

To analyze the algorithm's performance on Delaunay graphs, we performed the same binary search to find the worst case p. The Delaunay graphs were more reliable than

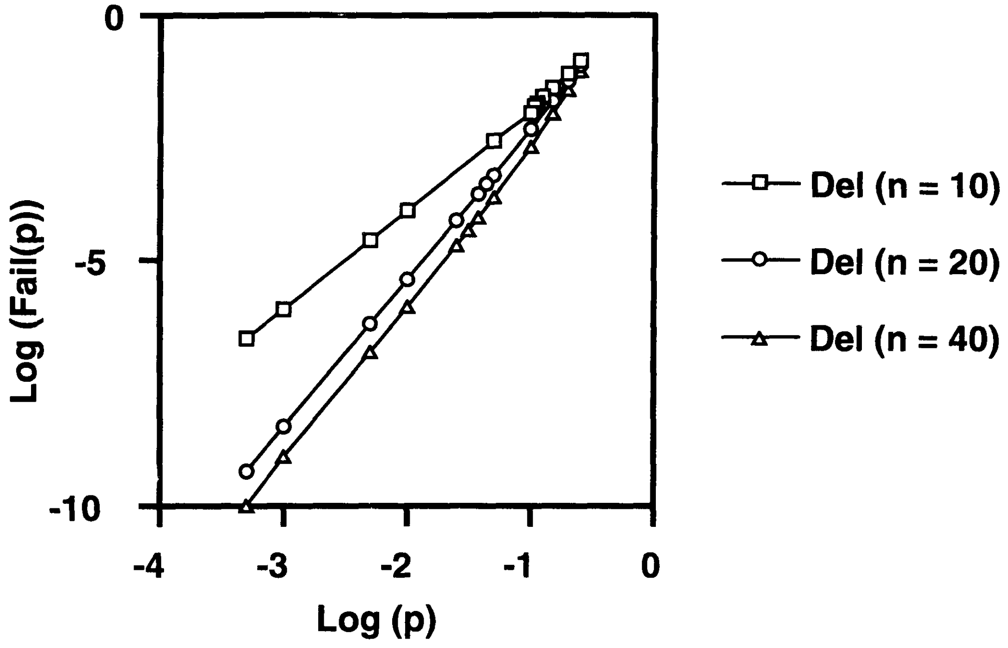

Figure 4-6: Graph Failure Probabilities for Delaunay Graphs

cycles, as their minimum cuts ranged from three to five edges, so we ranged our values of p from 0.75 to 10^{-4}. The failure probability of the graph became insignificant and easy to compute using the RCA/SACA approach at values of p smaller than 10^{-4}. The search was performed on five instances of each Delaunay graph size.

Figure 4-6 shows the increase in $FAIL(p)$ as p increases by plotting $\log FAIL(p)$ versus $\log p$ for one instance of each size of Delaunay graph. It is interesting to note from this figure that the graph failure increased polynomially with respect to the the value of p, as evidenced by the plots being virtually straight lines. We guessed that the cause was that almost all of the failure probability was due to the minimum cuts of the graph.

We plotted several instances of a 40 node Delaunay graph in Figure 4-7 to check this theory. Of the five instances that we plotted, four of them had identical slopes. We checked the minimum cuts of the instances and found that the four identical instances all had minimum cuts with three edges. The single instance with a different

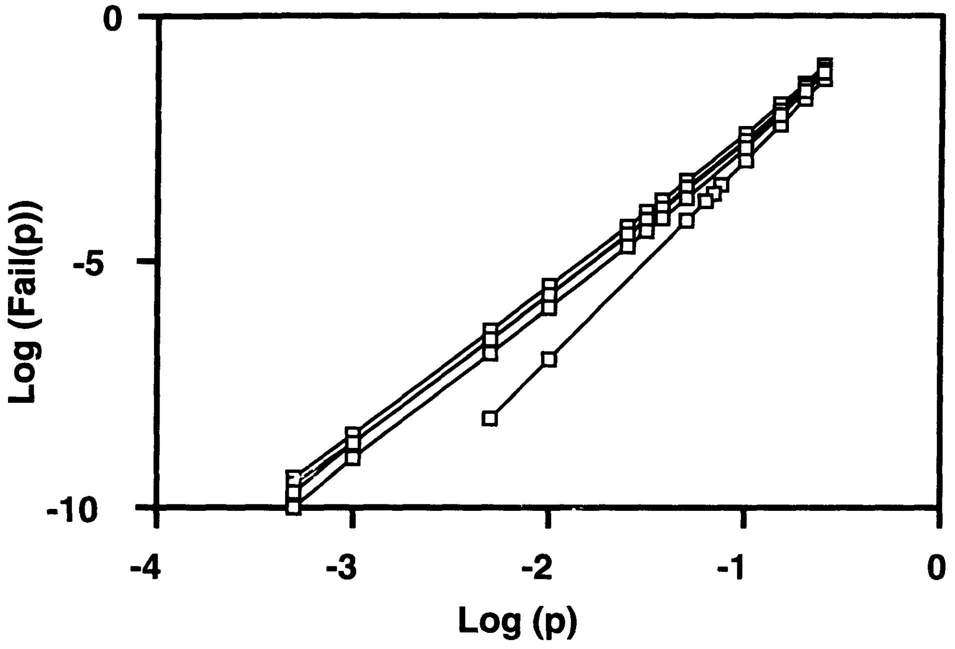

Figure 4-7: Graph Failure Probabilities for 40 Node Delaunay Graphs

slope had a minimum cut of four edges. The larger minimum cut meant that as the edge failure probability was reduced, graph failure probability was reduced more quickly than for a smaller minimum cut, resulting in the steeper slope. In addition, we discovered that the number of minimum cuts determined the height of the line. For example, of the four lines of the same slope, the highest line has 3 minimum cuts of size 3, and the lowest line has 1 minimum cut of size 3. The two lines in the middle that virtually overlap have 2 minimum cuts of size 3. This plot also reinforces our conclusion that almost all of the failure probability is caused by minimum cut failure for Delaunay graphs.

Figure 4-8 and Figure 4-9 show how the log of running time varied with $\log p$ and $\log FAIL(p)$ for different sizes of Delaunay graphs. These figures are very similar to Figures 2 and 3, except that the second peak is missing. We explain the absence of the second peak during Delaunay graph trials with the fact that the coarse estimation optimization had much less effect on Delaunay trials. Since the number of minimum

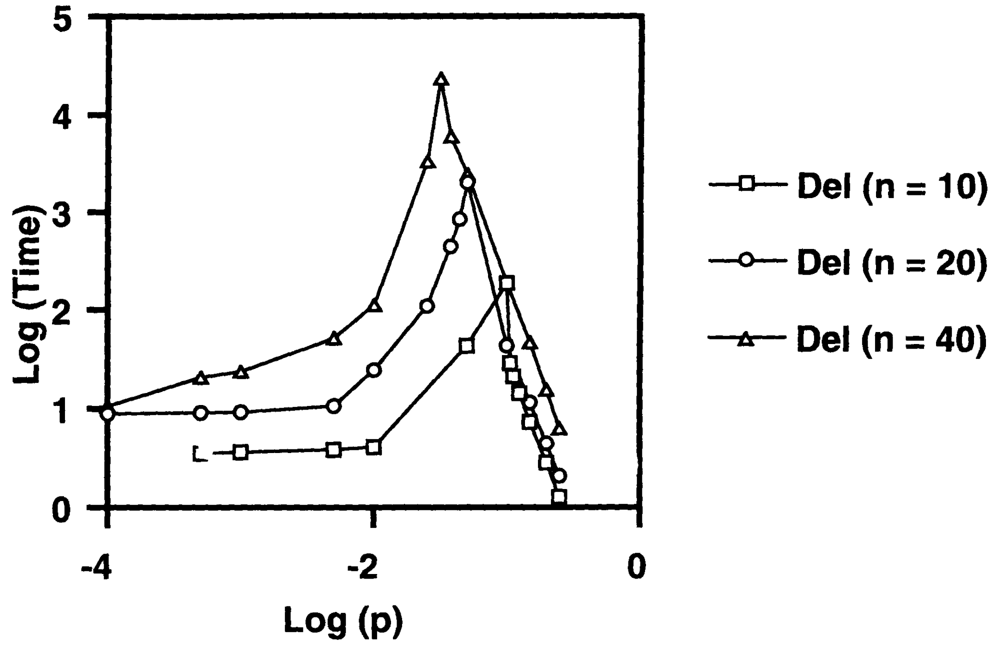

Figure 4-8: Running Time versus Edge Failure Probability for Delaunay Graphs

cuts in Delaunay graphs is much smaller than for cycles, the optimization was not able to reduce the more exact estimate's time by as much. Thus, the coarse estimation was never able to dominate time for any values of p.

Figure 4-10 plots the log of running time versus the log of the number of nodes to get a sense of how the running time increases as the size of the graph increases. The average of the five worst case times for each size of Delaunay graph were used for this plot. Looking at the plot, we see that it is approximately a straight line, except that running time seems to flatten out as the graph size increases. Taking a pessimistic view, we try to fit a line to the curve and ignore the apparent flattening. By taking the slope of this line, we find that the algorithm runs in $O(n^{4.02})$ time for Delaunay graphs. If we take an optimistic view, however, and take the least slope, we find that the algorithm potentially runs in $O(n^{2.65})$ time.

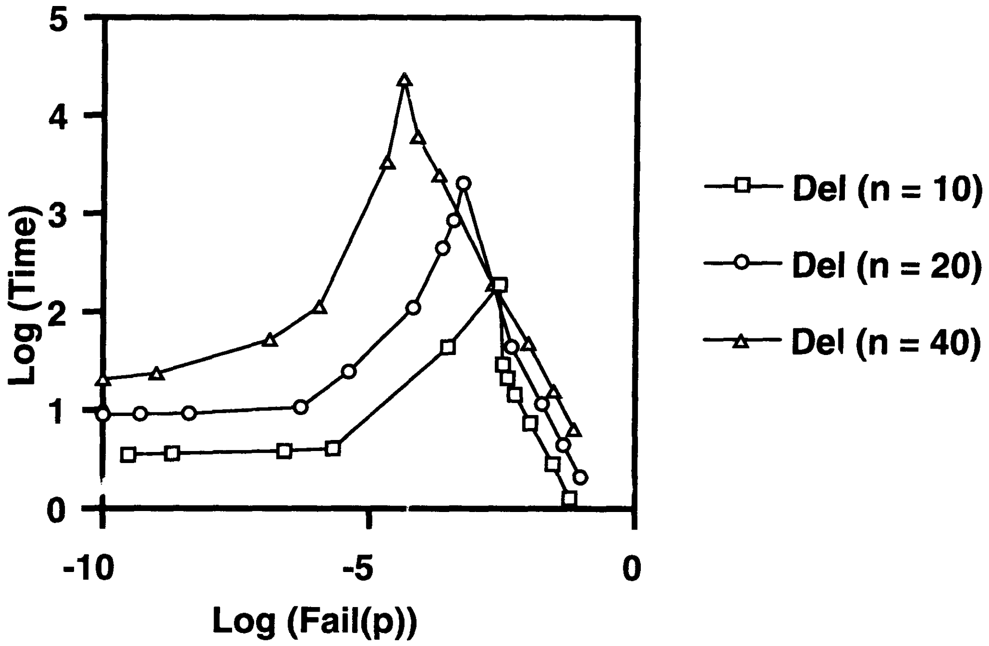

Figure 4-9: Running Time versus Graph Failure Probability for Delaunay Graphs

4.2.3 Nearest Neighbor Performance

When we ran the algorithm on the nearest neighbor family of graphs with *range* set to 8 and *density* set to 3, we found that for all reasonable values of p (less than 0.1), the graph was so reliable that the RCA/SACA dominated for all p used. Since we expect the algorithm to be used primarily on telecommunications networks, where link failure probabilities of even 1 percent would be unacceptable, this bound on p is acceptable. We used the nearest neighbor graphs to see how the time of the RCA/SACA increased as graph size increased for a fixed p.

We created five instances of each size for graphs ranging from 10 nodes to 320 nodes. The algorithm was then run on the graphs using $p = 0.1$ and $p = 0.15$. The values of $FAIL(p)$ produced varied from instance to instance, but in general it increased as the number of nodes increased. Even at 320 nodes, however, the failure probability averaged less than 10^{-9} for $p = 0.1$.

Figure 4-11 shows the more interesting data generated from this family of graphs,

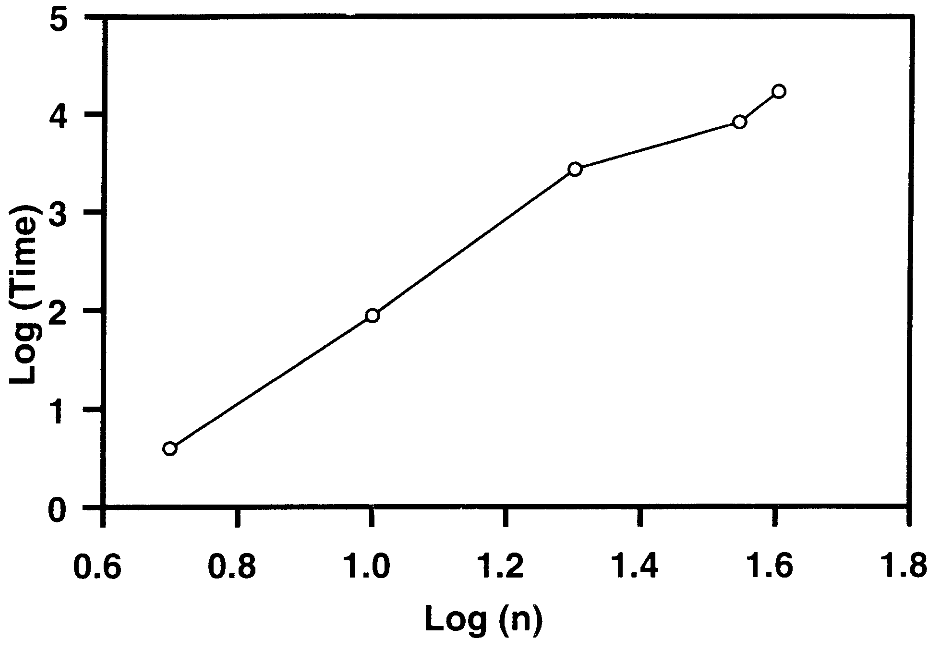

Figure 4-10: Worst Case Running Time for Delaunay Graphs

the effect of size on the running time of the RCA/SACA. In the figure, we plotted the log of the graph size versus the log of the total running time used. We note from this figure that the plot is not a straight line, so the running time grows more than polynomially as the graph size increases. This result, however, does not indicate that the overall running time of the algorithm is more than polynomial, as for the worst case of p, the MCA takes over before the RCA/SACA time begins growing at a super-polynomial rate.

We also ran the approximation scheme on nearest neighbor graphs for several values of p from 0.75 to 0.99, to study the effect of p on running time of the RCA/SACA. Figure 4-12 shows the results of these tests by plotting the log of p against the log of the running time for one instance of each size of graph. Of interest is the sudden jump in running time as p for a 10 node graph goes from 0.05 to 0.10, and the sudden jump in running time as p for a 40 node graph goes from 0.15 to 0.20. In both cases, the jumps were accompanied by a sharp increase in the number of cuts found. This

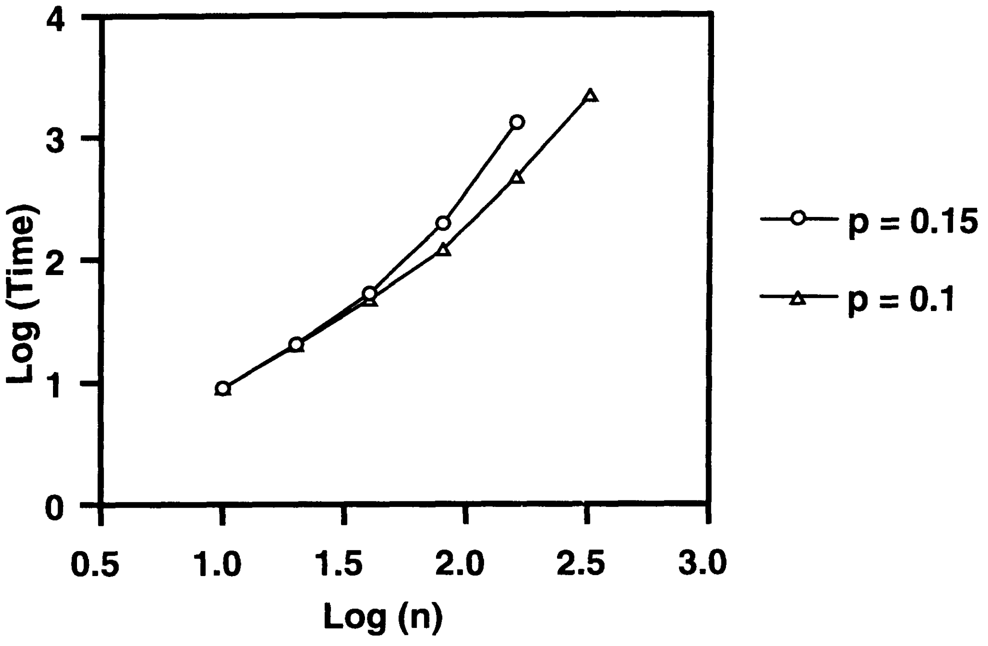

Figure 4-11: Running Time versus Size for Nearest Neighbor Graphs

result confirms Karger's statement in [Kar95] that the running time of the RCA varies with the number of cuts found.

4.2.4 Real Networks

The real networks that we received ranged from 9 to 57 nodes in size. We ran tests by starting p at 0.1 and lowering p until $FAIL(p)$ dropped below 10^{-6}. We estimate that a typical telecommunications network will accept a failure rate between 10^{-5} and 10^{-7}, or about between 0.1 and 10 seconds of failure a year. These values motivated our settings for p, as a user would most likely set p such that $FAIL(p)$ is between the two bounds above.

Figure 4-13 plots $\log FAIL(p)$ versus $\log p$. Looking at this figure, we find again that $FAIL(p)$ increases polynomially as p increases. We also notice that three of the problems have the same slope, while one has a smaller slope. From our experience with Delaunay graphs, we would guess that the network with a smaller slope has a

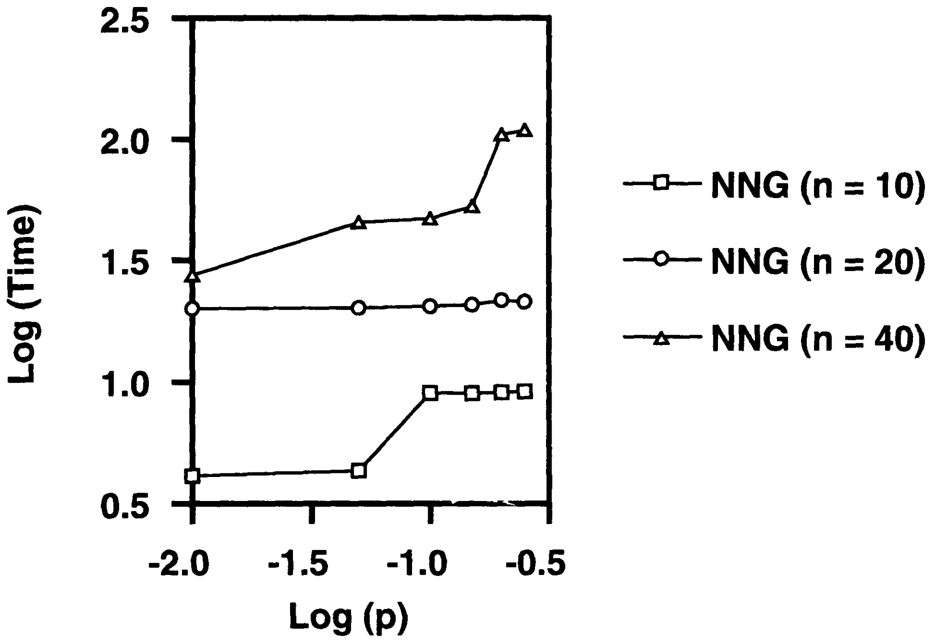

Figure 4-12: Running Time versus Edge Failure Probability for Nearest Neighbor Graphs

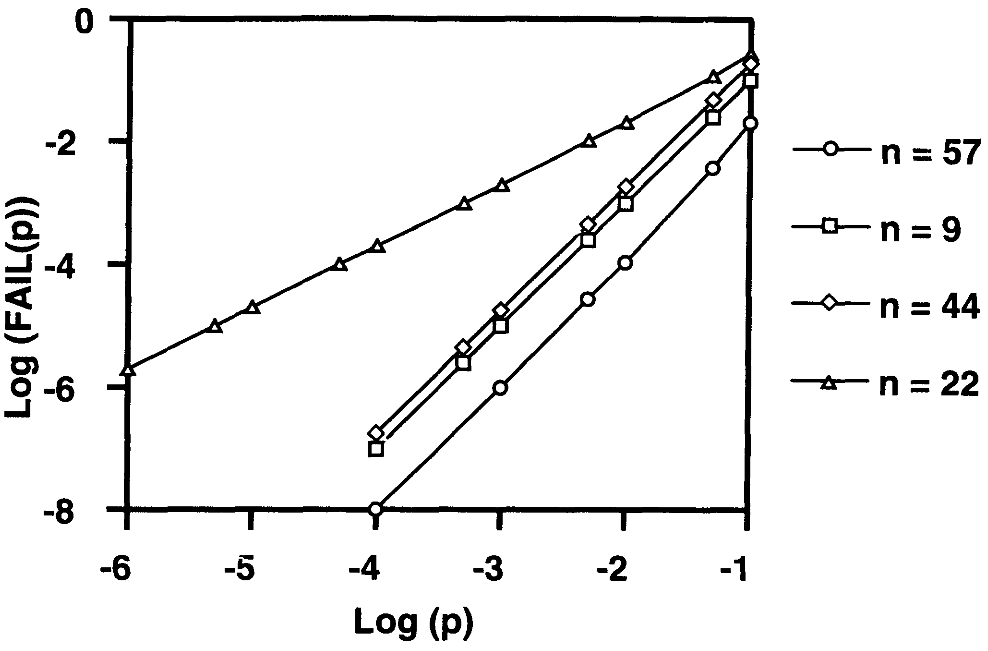

Figure 4-13: Graph Failure Probabilities for Real Networks

smaller minimum cut than the other three networks. By enumerating the minimum cuts, we find that our hypothesis is true. The network with a different slope has a minimum cut of 1 while the other networks have minimum cuts with 2 edges, causing its failure probability to increase more slowly than the other graphs.

Chapter 5

Conclusions and Future Work

By implementing Karger's approximation scheme for solving the All Terminal Reliability Problem, we have shown that the $O(n^4)$ time solution proved in theory also works in practice. For cycles and for Delaunay graphs, the implementation performed in approximately $O(n^4)$ time, and there were indications that the exponent was beginning to drop as the graph size increased.

Through our tests on several families of graphs, we were also able to gain some insight into the main causes of failure in graphs. From our plots of $FAIL(p)$ versus p for several families of graphs, we saw that graph failure increases polynomially as p increases, and with the same slope that the minimum cut would indicate. Thus, we conclude that the minimum cuts contribute overwhelmingly to failure probability of the sizes and types of graphs that we studied. Using this information, we propose that an extremely quick and reasonably accurate way of obtaining reliability is find the minimum cuts and multiply the probability of a minimum cut failing by the number of minimum cuts found.

Several extensions can be made to this thesis. Karger proposed in [Kar95] a method of extending the algorithm to include graphs where the edge failure probability is not identical for all edges. If this extension were implemented, it could give insight into the effects of different edge failure probabilities on the overall graph failure probability.

Another extension would be to study the effectiveness of the algorithm for larger graphs and to see apparent drop in the exponent on running time was valid. In addition, the algorithm could be run on a wider variety of graphs to see if our observations on reliability apply in general.

In conclusion, we have shown that Karger's time bounds for his approximation scheme apply in practice. For many values of p, the time used by the approximation scheme is much less than the worst case time. For these values of p, the algorithm can be used to analyze relatively large graphs in a reasonable amount of time. Thus, we hope we have created the foundation for a useful tool for analyzing real life networks.

REFERENCES

[Col87] Charles J. Colbourn. *The Combinatorics of Network Reliability*, volume 4 of *The International Series of Monographs on Computer Science*. Oxford University Press, 1987.

[Kar95] David R. Karger. A Randomized Fully Polynomial Time Approximation Scheme for the All Terminal Network Reliability Problem. Symposium on the Theory of Computing 1995.

[KL83] Richard M. Karp and Michael G. Luby, Monte-Carlo Algorithms for enumeration and reliability problems. *Proceedings of the 24th IEEE Foundations of Computer Science Symposium*, pp. 56-64.

[KLM89] Richard M. Karp and Michael G. Luby. Monte carlo algorithms for enumeration problems. *Journal of Algorithms* . 10(3):429-448, September 1989.

[Kra95] Jorg Kramer. Delaunay Triangulation in two and three dimensions. Master thesis, December 1995.

[KS93] David R. Karger and Clifford Stein. An $O(n^2)$ algorithm for minimum cuts. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, ACM Press, May 1993, pages 757-765.

[PB83] J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a network remains connected. *SIAM Journal on Computing*, 12(4):777-788, 1983.

[Val79] Leslie Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8:410-421, 1979.