MONTE-CARLO METHODS FOR ESTIMATING SYSTEM RELIABILITY

Michael G. Luby

Computer Science Department

ABSTRACT

An n**−component system** contains n components, where each component may be either **failing** or **working**. Each component i is failing with probability p_i independently of the other components in the system. A **system state** is a specification of the states of the n components. Let F, the set of **failure states**, be a specified subset of the set of all system states. In this paper we develop Monte Carlo algorithms to estimate $Pr[F]$, the probability that the system is in a failure state.

We now describe the two different formats for the representation of F considered in this paper. In the first format F is represented by m failure sets $F_1, F_2, \cdots, F_m$. Each failure set is specified by an n−tuple in the input. The set of failure states is then $F = \bigcup_{i=1}^{m} F_i$. We develop a formula for the probability of a union of events. A Monte Carlo algorithm which estimates the failure probability of an n−component system is developed based on this formula. One trial of the algorithm outputs an unbiased estimator of $Pr[F]$. Let Y be the average of the estimators pro-

duced by many trials of the algorithm. We show that, when the algorithm is run for an amount of time proportional to $n \cdot m$, Y is provably close to $Pr[F]$ with high probability.

The second format for the representation of F can be described as follows. A **network** is an undirected graph G, where the edges in the graph correspond to the components in the system. Let G be a planar network and let $x_1, \cdots, x_K$ be K specified nodes in G. For the **planar K-terminal problem**, the network is in a failing state if there is no path of working edges between some pair of specified nodes. We develop a Monte Carlo algorithm to estimate $Pr[F]$ for the planar K-terminal problem. The algorithm works especially well when the edge failure probabilities are small. In this case the algorithm produces an estimator Y which is provably close to $Pr[F]$ with high probability in time polynomial in the size of the graph. This compares very favorably with the execution times of other methods used for solving this problem.

Richard M. Karp / Manuel Blum

To my parents,

who carried me on their shoulders,

may I do as much for my children.

Acknowledgements

When I came to Berkeley I wasn't sure if I had the dedication, stamina or ability to complete a Ph.D. Without the help of the following people, I would never have finished.

Dick Karp has inspired me when I felt empty, filled me with ideas when I had none, picked me up when I was down, understood when I rebelled and has always been there when I needed a friend. I am eternally grateful for the influence he has had on my life.

My officemates, Shafi Goldwasser, Silvio Micali and Vijay Vazirani deserve special mention. They are the brightest and most original group of people I have ever known. I consider myself very lucky for having the pleasure of spending four years of my life working and playing with them. Shafi is the friend from whom I keep no secrets. I miss her dearly already. Silvio was the source of some of my happiest times in Berkeley (and Italy!). His friendship is one of my most valued treasures. I will greatly miss the playfulness and enthusiasm of Vijay, his wonderful Indian food and his music.

I couldn't think of better role models than Manuel Blum, Gene Lawler, Clark Thompson and Andy Yao. The knowledge they have shared with me through their refreshing courses and through close personal interaction has been a tremendous help. I would also like to thank Gene and Clark for their financial support when I needed it most.

I thank Sheldon Ross, Jim Pitman and J.W. Addison for their inspiring courses and for the hours they spent talking to me about research from a

perspective different from that of a computer scientist.

I thank Ravi Kannan, David Lichtenstein, Chip Martel, Barbara Simons and Michael Sipser for showing me that there is light at the end of the tunnel.

A special thanks goes to those with whom I have had the pleasure of working especially closely over the past four years: Eric Bach, Shafi Goldwasser, Howard Karloff, Narendra Karmarkar, Dick Karp, Gene Lawler, Alberto Marchetti (all those hours in the Roma!), Silvio Micali (the rush job in Toronto), Bruce Parker, Prabhakar Ragde, and Umesh and Vijay Vazirani (love that banana bread!).

I wish I had more space, because I really can't do justice to the amount I learned from the following friends: Shimon Even, Faith Fich, Zvi Galil, Andrey Goldberg, Donald Johnson, Debby Joseph, Wolfgang Maase, Joan Plumstead, Jeff Shallit, Michael Rabin, David Shmoys, Manfred Warmuth, Avi Wigderson and Alice Wong.

I am grateful to you all. You have given me incredible amounts of your time sharing your expertise and showing me how to abstract, create, synthesize and explain new ideas. On the personal side, I have never had such a good group of friends as I have had at Berkeley. I will never forget any of you.

Table of Contents

Chapter 1

Chapter 2

Chapter 1

1. Introduction

The focus of this dissertation is the problem of determining the reliability of a system. The standard approach is to try and calculate the exact reliability of the system. For systems with special structures and for very small systems this approach has been very successful. However, if the system is large and does not have a very special structure then the running time for all known algorithms to calculate the exact reliability of the system is exponential in the size of the system. For the classes of systems we consider, the computation of the exact reliability is either at least as hard as an NP-complete problem or is a #P-complete problem. Thus, it is unlikely that there will ever be a fast algorithm to calculate the reliability of these systems.

The difficulty in calculating the exact reliability of a system motivates the approach we have taken in this work. Instead of trying to calculate the exact reliability, we develop Monte Carlo algorithms to estimate the reliability. Our goal is to develop algorithms which output an estimate of the reliability which is provably close to the exact reliability of the system with high probability. Furthermore, the running time of the algorithms should be small. We have attained these goals for the n-component system described in subsection 2.1. We have attained these goals for the K-terminal problem on a class of graphs which occur often in practice. This problem is described in subsection 2.2.

The organization of the rest of this chapter follows. In section 2 we describe

the class of system reliability problems that will be considered. Then, we give a description of the the methods for calculating the exact reliability of the system in section 3. In section 4 we give evidence for why the calculation of the exact reliability is a hard problem. In section 5 we describe Monte Carlo algorithms and give our measure of what constitutes a good Monte Carlo algorithm. In section 6 we compare previous Monte Carlo algorithms for the systems described in section 2 with each other and with our algorithms. In section 7 we show how Monte Carlo algorithms can be effectively combined with methods for the exact computation of system reliability.

2. System Problem Description

In this section we describe the system problems which are the subject of this work. All of our systems are composed of n **components**. Each component is either **working** or **failing**. Each component i has a **failure probability** p_i associated with it, where i is failing with probability p_i and working with probability $1-p_i$ independently of the states of the other components in the system. A **system state** is an n-tuple $(b_1, \cdots, b_n)$ where $b_i = 0$ if component i is failing and $b_i = 1$ if component i is working. There are 2^n system states and the probability of any particular system state $(b_1, \cdots, b_n)$ is

$$\prod_{i=1}^{n} p_i^{1-b_i} \cdot (1-p_i)^{b_i} .$$

Let S be the set of all system states, and let $F \subseteq S$ be the set of **failing system states** and let $S-F$ be the set of all **working system states**. The **failure probability** of the system can be written as $Pr[F]$. The **reliability** of the system is $Pr[S-F] = 1-Pr[F]$. Notice that the exact computation of the reliability of the system is equivalent to the exact computation of the failure probability. However, a good estimate of the reliability of the system is not necessarily a good estimate of the failure probability. For instance, if the reliability of the system

is $1-10^{-10}$, then $1-10^{-6}$ is a good estimate of the reliability. On the other hand, 10^{-6} is not a good estimate of 10^{-10}, the failure probability of the same system. Usually, since systems are designed to be reliable, the failure probability is much smaller than the reliability. In this case, a good estimate of the failure probability is a good estimate of the reliability, but not necessarily vice-versa. Therefore, we discuss only the problem of trying to compute or estimate the failure probability of the system.

We have not described how the set of failure states F is presented in the problem input. We now describe two completely different formats for the representation of F.

2.1. n-Component System

This is the format we assume in chapter 2. The set of failing states F is the union of m failure sets F_i $(1 \le i \le m)$. The failure sets are specified in the problem input as n-tuples $(c_1, \cdots, c_n)$, where $c_i = 0, 1$ or $*$. Failure state $(b_1, \cdots, b_n)$ is in failure set $(c_1, \cdots, c_n)$ provided that $c_i = 0$ implies that $b_i = 0$, $c_i = 1$ implies that $b_i = 1$, and $c_i = *$ implies b_i may be either 0 or 1.

A wide class of systems can be put into this format. Many of these systems have the following **monotonic property** (In Barlow [3], this is called a binary coherent system). Define a partial order $\subseteq$ on system states as follows: $s \subseteq t$ if and only if the set of components failing in system state s is a subset of the set of components failing in t. An n-component system is monotone if for all failure states s and system states t, $s \subseteq t$ implies that t is also a failure state. Intuitively, a system is monotone if the system deteriorates in performance as more and more components fail. In any monotonic n-component system the set of failure states can be represented by failure sets which correspond naturally to minimal failure states (minimal with respect to the partial order $\subseteq$). The number of failure sets, m, is equal to the number of minimal failure states

of the system.

2.2. Networks

For the systems described in this subsection, the set of failure states, F, is represented implicitly by a **network**. We now describe a network. Let G be an undirected graph with n edges. Each edge e_i in G fails with probability p_i and works with probability $1-p_i$ independently of the other edges in the network. The edges are the components in the system. Nodes are assumed to always work.

In the **two-terminal problem** there are two specified nodes x and y. The set of failure states are those states where x and y are disconnected by a cut of failing edges. This is a monotonic system where the minimal failure states correspond to $x-y$ cuts. Therefore, this problem can be put into the same format as the n-component system described in the previous subsection by listing all the $x-y$ cuts. However, since the number of $x-y$ cuts can be exponential in the number of edges, this is not always an attractive alternative.

In the **K-terminal problem** there are K specified nodes. The problem is to compute the probability that there is a cut between any pair of the K specified nodes. The **all-terminal problem** is the special case of the K-terminal problem when all the nodes are specified. In chapter 3 we develop Monte Carlo algorithms for these three network problems when the underlying graph G is planar. These algorithms avoid listing the cuts in the graph.

3. Computation of the Exact Failure Probability

We first discuss previous work relevant to the computation of the exact failure probability for n-component systems presented in the format discussed in subsection 2.1. The **straightforward method** for calculating the failure probablity consists of listing all the failure states, computing their probabilities and

summing these probabilities to get $Pr[F]$. The running time for this algorithm is proportional to the number of failure states times n. However, the number of failure states can be exponential in the number of failure sets.

Another method is to use the principle of **inclusion-exclusion**, i.e.

$$Pr[F] = \sum_i Pr[F_i] - \sum_i \sum_j Pr[F_i \cap F_j] + \sum_i \sum_j \sum_k Pr[F_i \cap F_j \cap F_k] \ldots$$

However, the number of terms in this expression is 2^n. It is sometimes possible to use the first few terms in this expression to estimate $Pr[F]$. However, these partial sums may vary wildly.

A very powerful method is **factoring**. Let X be a system, let $X \cdot i$ be the same system as X except that component i is specified as working, and let $X - i$ be the same system as X except that component i is specified as failing. Thus, the failure states F_X of X are partitioned into $F_{X \cdot i}$, the set of all failure states in X such that component i is working, and F_{X-i}, the set of all failure states in X such that component i is failing. Then,

$$Pr[F_X] = p_i \cdot Pr[F_{X-i}] + (1-p_i) \cdot Pr[F_{X \cdot i}] \ .$$

Factoring can be used recursively to compute the failure probability of $X \cdot i$ and $X - i$. The factoring algorithm produces a binary tree structure, called the **decomposition tree**, where X corresponds to the root of the tree and $X \cdot i$ and $X - i$ are the children of X. A straightforward application of factoring results in a full binary tree with 2^n leaves, where each leaf corresponds to a one component system. However, the tree can be truncated at any node which corresponds to a system whose failure probability is easy to evaluate. Thus, the size of the tree can be substantially smaller than 2^n.

We now discuss previous work relevant to the computation of the exact failure probability for network problems. Let c be the number of cuts separating the two specified nodes in the two-terminal problem. Ball and Provan [2]

give a method for calculating the exact failure probability for which the running time is proportional to $n \cdot c^2$. Since c can be exponentially smaller than the number of failure states, this is sometimes a substantial savings over the straightforward method. Clearly, this method is better than inclusion-exclusion, for which the running time is proportional to $n \cdot 2^c$. Their method does not seem to extend to the K–terminal or all-terminal problems.

When the network has special structural properties it is possible to apply a **reduction**. Consider a network G. Suppose there is a pair of edges between nodes u and v in G with failure probabilities p_1 and p_2 respectively. A **parallel** reduction replaces these two edges with one edge with failure probability $p_1 \cdot p_2$. Suppose there is a node u in G of degree two, with one edge to v with failure probability p_1 and the other to w with failure probability p_2. A **series** reduction removes from the graph the node u and the two edges out of u, and adds a new edge between v and w with failure probability $1-(1-p_1)\cdot(1-p_2)$. G is called **series-parallel** if the graph can be reduced to a single edge via these two types of reductions.

Consider the K–terminal problem. A parallel reduction reduces the size of the network such that the resulting network and the original network both have the same failure probability. If there is a node u of degree two which is **not** one of the K specified nodes, then a series reduction can be applied which reduces the size of the network such that the resulting network and the original network both have the same failure probability. However, if u **is** one of the K specified nodes then the series reduction cannot be applied. Satyanarayana and Wood [13],[16] introduce a new set of reductions, called **polygon-to-chain** reductions. They show that if the underlying graph G is series-parallel then the exact failure probability for the K–terminal problem can be solved in time linear in n, the number of edges in G. In [10], Satyanarayana and Chang define a combinatorial

invariant of the graph called the **domination** of G. They show that the number of leaves in the optimal decomposition tree when factoring is combined with reductions is equal to the domination of G, and they show how to factor to achieve this optimal decompositon tree. Using the idea of the domination of the graph, Satyanarayana, Hagstrom and Prabhakar [11], [12], [3] show how many terms in the inclusion-exclusion formula can be combined to reduce the total number of terms.

In [13],[16], Satyanarayana and Wood show how a triconnected component in a K−terminal network can be replaced by a **chain**, where a chain is a path of degree two nodes. The failure probabilities of the edges in the chain are determined as follows. A triconnected component is connected to the rest of the graph through at most two nodes u and v. The triconnected component is first disconnected from the rest of the graph at u and v and considered as a network in its own right. Several K−terminal problems (at most 5) are solved using this network, with a different specification of the K nodes in each problem. The failure probabilities computed for these problems determine the failure probabilities of the edges in the chain. The chain is then added between u and v in the original network to replace the triconnected component.

4. Hardness Results

In this section we motivate the use of Monte-Carlo algorithms to estimate failure probabilities for the systems described in section 2. In [1], Cook proved that the **satisfiability** problem is as hard to solve as any problem in a class called NP. Thus, the satisfiability problem was the first problem in a new class of problems, called the NP-complete problems. Satisfiability is called a **decision problem**, because the answer to the problem is either "yes" or "no". Karp [7] showed that many other combinatorial decision problems were also NP-complete. Fast solutions to these NP-complete problems has been the aim of

many researchers for years, with no success. The worst case running time for the best algorithms for all these hard problems is exponential in the size of the problem input. Over the years, hundreds of problems have been added to the list of NP-complete problems [5]. Valiant [14] noted that many seemingly hard problems are not decision problems but counting problems. He went on to define a new class, called #P, and proved that there are problems in #P that are as hard to solve as any problem in #P; these are the #P-complete problems. #P-complete problems are at least as hard to solve as NP-complete problems, but they could be harder (nobody knows yet).

Provan and Ball [9] showed that the monotonic n-component problem with the input format as described in subsection 2.1 is #P-complete, even when there are at most two components per failure set and when all component failure probabilities are one-half. They also showed that the two-terminal problem is #P-complete, as well as the problem of approximating the failure probability for the two-terminal problem. The status of the planar two-terminal problem is unknown, but no polynomial time algorithm is known for the computation of the exact failure probability.

These results are strong evidence that it is hopeless to devise a polynomial time algorithm to compute the exact failure probability for the n-component problem described in subsection 2.1. Since there is no polynomial time algorithm known for computing the exact failure probability for the planar K-terminal problem, a Monte-Carlo approach seems attractive.

5. Monte Carlo Algorithms

A Monte Carlo algorithm is a randomized computational experiment for estimating some quantity. In this paper the quantity is the failure probability of the system and the experiment is cast in the following form. Let P be a probability measure on a finite set A and let f be a real-valued random variable on A

with respect to P. Let μ be the expected value of f, i.e.

$$\mu = \sum_{\alpha \in A} f(\alpha) \cdot P[\alpha] \ .$$

For our algorithms, we design the set A, the probability measure P and the random variable f with the following properties: (1) $\mu = Pr[F]$, (2) it is easy to randomly choose $\alpha \in A$ with probability $Pr[\alpha]$, and (3) given $\alpha \in A$, $f(\alpha)$ is easy to evaluate. One **trial** of a Monte Carlo algorithm can be described as follows:

1. Randomly choose an element in A such that $\alpha \in A$ is chosen with probability $P[\alpha]$.
2. Estimator $Y \leftarrow f(\alpha)$.

The expected value of Y, $E[Y]$, is $\mu = Pr[F]$. A Monte Carlo algorithm repeats many independent trial steps. Let N be the total number of trials and let Y_i be the estimator produced in the i^{th} trial. The algorithm outputs

$$\tilde{Y} = \frac{Y_1 + \ldots + Y_N}{N}$$

as the estimate of μ. By elementary probability theory, the greater the number of trials the more likely $\tilde{Y}$ is close to μ. We now quantify this statement. We say a Monte Carlo algorithm is an (ε,δ) algorithm if

$$Pr\left[\left|\frac{\tilde{Y}-\mu}{\mu}\right| < \varepsilon\right] > 1-\delta$$

where ε and δ are small positive real numbers. This is a **relative** measure of the closeness of $\tilde{Y}$ to μ. We discuss why this measure was chosen in chapter 3, subsection 2.3. Our goal is to design an algorithm so that the number of trials sufficient to achieve an (ε,δ) algorithm is provably small and so that the running time per trial is small. This goal may not be always possible. Therefore, we list our goals for Monte Carlo algorithm design here in increasing order of priority.

1. Design an (ε,δ) algorithm which is provably faster than an existing algorithm.
2. Design an (ε,δ) algorithm which is provably fast for an interesting subclass of the entire class of problems.
3. Design an (ε,δ) algorithm which is provably fast for the entire class of problem inputs.

6. Comparisons between Monte Carlo Algorithms

As before, let S be the set of all system states and let F be the set of all failure states. The easiest Monte Carlo algorithm to describe which outputs an estimate of $Pr[F]$ is **straight simulation**. One trial of the algorithm consists of randomly choosing a system state $s \in S$ such that s is chosen with probability $Pr[s]$. The estimator Y is zero if s is a working state and one if s is a failing state. Straight simulation is easy to implement. However, the number of trial N sufficient to achieve an (ε,δ) algorithm is proportional to $\frac{1}{Pr[F]}$. There are two serious problems here:

1. $Pr[F]$ can be arbitrarily small and thus $\frac{1}{Pr[F]}$ can be arbitrarily large.
2. The number of trials, N, is expressed in terms of the quantity we are trying to estimate. Thus, we can't put an **a priori** upper bound on N.

Let S_j, called the j^{th} **stratum**, be the set of all system states s such that there are exactly j failing components in s. The system states in an n-component system are partitioned into $n+1$ strata. The **sequential destruction** method (hereafter called SDM) of Easton and Wong [4] is a refinement of straight simulation. One trial of their algorithm can be described as follows:

1. Randomly choose a system state $s \in S$ such that s is chosen with probability $Pr[s]$. State s is chosen in exactly the same way as s is chosen using

straight simulation.

2. Suppose $s \in S_j$. Make a list of the n components such that the first j components on the list are a random permutation of the failing components in s, and the last $n-j$ components on the list are a random permutation of the working components in s. Let $\{c_1, \cdots, c_n\}$ be the ordered list of components.

3. Let s_j be the system state where components c_1 thru c_j are failing, and $c_{j+1}, \cdots, c_n$ are working. Thus, s_j is from the j^{th} stratum. The estimator Y is based on a weighted average of the estimator produced by straight simulation when presented with the system states $s_0, s_1, \cdots, s_n$.

Easton and Wong prove that the variance for SDM is less than the variance for straight simulation. Since the number of trials sufficient to achieve an (ε,δ) algorithm is directly proportional to the variance, SDM is better than straight simulation. They include some examples where SDM is substantially better than straight simulation. However, they give no a priori bounds on how much better SDM is then straight simulation. In contrast, our algorithm for the n-component system is better than straight simulation by a factor easily computable from the problem input. They also prove that the number of trials sufficient to achieve an (ε,δ) algorithm remains bounded by a constant as the component failure probabilities approach zero. However, their bound depends on the structure of the problem instance and cannot be easily computed. In contrast, we show that there is an easily computable upper bound on the number of trials sufficient to acheive an (ε,δ) algorithm for our n-component algorithm which approaches zero as the component failure probabilities approach zero.

Van Slyke and Frank [15] suggest **stratified sampling**. They consider only the case when all components have equal probability, but the technique can be

easily generalized to the case of unequal component failure probabilities. As before, let S_j be the j^{th} stratum set. Let F_j be the set of failure states in S_j. It is easy to compute $Pr[S_j]$ and to randomly choose $s \in S_j$ so that s is chosen with probability $\frac{Pr[s]}{Pr[S_j]}$. The failure probability of the system can be expressed as

$$Pr[F] = \sum_{j=0}^{n} Pr[F_j \mid S_j] \cdot Pr[S_j] \ .$$

The idea is to compute some of the quantities $Pr[F_j \mid S_j]$ by exact methods (for instance $Pr[F_j \mid S_j] = 0$ for $i < k$, where k is the minimum number of failing components in any failure state) and to estimate the rest of these quantities using straight simulation. The total number of trials they perform to estimate $Pr[F_j \mid S_j]$ by straight simulation is proportional to $Pr[S_j]$. Let S' be the union of the states in the stratum sets for which straight simulation is to be used, and let F' be the set of failure states in S'. Their algorithm can be thought of as using straight simulation to estimate $Pr[F' \mid S']$. They multiply this estimate by $Pr[S']$ and add this quantity to the failure probability they are able to compute using exact methods. This is their estimate of $Pr[F]$. They show by example that straitified sampling can be substantially better than straight simulation. In fact, it is not hard to prove that the variance for stratified sampling is smaller than for straight simulation.

Kumamato, et. al. [8] develop a technique which is similar in spirit to stratified sampling. Some of the failure probability they compute by exact methods, some they estimate by straight simulation. Their motivation is that in some network problems it is easy to find some important path sets and cut sets. Let PH be the set of system states which are in at least one of the important path sets, let CT be the set of system states which are in at least one of the important cut sets and let F' be the set of failure states in $S - PH - CT$. System

states in PH are known to be working states and system states in CT are known to be failing states. The idea is to use exact methods to compute $Pr[CT]$ and $Pr[PH]$ and use straight simulation to estimate $Pr[F' \mid S-PH-CT]$. Let $\hat{Y}$ be the estimate of $Pr[F' \mid S-PH-CT]$. The estimate of $Pr[F]$ is then $\hat{Y} \cdot Pr[S-PH-CT] + Pr[CT]$. The variance for this method is provably smaller than the variance for straight simulation. However, no a priori bound of the number of trials sufficient to achieve an (ε,δ) algorithm seem to be easily computable. Furthermore, not too many cuts or paths can be included in the set of important cuts and paths. The time to compute $Pr[CT]$ and $Pr[PH]$ exactly grows exponentially with the number of important cuts and paths.

Now we outline our main results. In chapter 2, we present an (ε,δ) algorithm for the n-component system described in subsection 2.1. The total running time of the algorithm is a priori bounded above by a value proportional to $\frac{1}{\delta}\frac{1}{\varepsilon^2} \cdot n \cdot m$, where n is the number of components and m is the number of failure sets. Thus, the algorithm is provably fast for the entire class of problem inputs.

In chapter 3 we present an (ε,δ) algorithm for the planar K-terminal problem. When the edge failure probabilities are small and the maximum number of edges bordering a face in the graph is small, the algorithm is provably fast. An upper bound on the running time of the algorithm can be computed a priori. The subclass of problems for which the algorithm runs provably fast are important in practice.

7. Combining Monte Carlo Methods with Deterministic Methods

In section 3 we described methods for computing the exact failure probability of a system. These techniques can be combined with the Monte Carlo methods we have developed. We now give several examples where these tech-

niques can be combined. Consider the decomposition tree for the factoring algorithm when applied to the K−terminal problem. We pointed out before that series-parallel and polygon-to-chain reductions should be applied whenever possible to the subnetworks in the decomposition tree to minimize the size of the tree. Our planar K−terminal algorithm can be used to estimate the failure probability for any network in this tree which is planar. Thus, the decomposition tree can be truncated at any node which corresponds to a planar subnetwork. It is an open (and probably hard) question as to what the optimal component selection rule is to minimize the size of the decomposition tree if the leaves of the tree are planar graphs.

Another place where our planar K−terminal algorithm could be useful is motivated by the work of Satyanarayana and Wood [13],[16]. As we described earlier, a triconnected component in the graph can be replaced by a chain without changing the failure probability of the entire network. The failure probabilities of the edges in the chain are computed by computing the failure probabilty for less than five K−terminal problems on the triconnected component. If the triconnected component is planar, then our planar K−terminal algorithm could be used to estimate the failure probability for these K−terminal problems.

In chapter 3, section 7, we point out that given a fast algorithm for computing the exact failure probability for the two-terminal problem for a class of networks there is a fast Monte Carlo (ε,δ) algorithm for the K−terminal problem for the same class of networks.

8. References

[1] Cook, S.A., *The Complexity of Theorem Proving Procedures*, Proc. 3rd Ann. ACM Symp. on Theory of Computing, Assoc. for Computing Machinery, 1971a, N.Y., pp. 151-158

[2] Ball, M.O. and Provan, J.S., unpublished paper

[3] Barlow, R.E. and Proschan, F., *Statistical Theory of Reliability and Life Testing*, Holt, Rinehart and Winston, New York, 1975

[4] Easton, M.C. and Wong, C.K., *Sequential Destruction Method for Monte Carlo Evaluation of System Reliability*, IEEE Transactions on Reliability, vol. R-29, no. 1, April 1980, pp. 27-32

[5] Garey, M.R. and Johnson, D.S., *Computers and Intractibility, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, 1979

[6] Hagstrom, J.N., *Combinatoric Tools for Computing Network Reliability*, Ph.D. dissertation, U.C. Berkeley, 1980

[7] Karp, R.M., *Reducibility Among Combinatorial Problems*, *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Thatcher, Plenum Press, N.Y., 1972, pp. 85-103

[8] Kumamoto, H., Tanaka, K. and Inoue, K., *Efficient Evaluation of System Reliability by Monte Carlo Method*, IEEE Transactions on Reliability, vol. R-26, no. 5, December 1977, pp.311-315

[9] Provan, J.S. and Ball, M.O., *The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected*, working paper MS/S 81-002, Management Science and Statistics, January 1981 (revised April 1981)

[10] Satyanarayana, A. and Chang, M.K., *Network Reliability and the Factoring Theorem*, ORC 81-012, U.C. Berkeley, 1981

[11] Satyanarayana, A. and Hagstrom, J.N., *New Algorithm for the Analysis of Multiterminal Network Reliability*, IEEE Transactions on Reliability, to appear

[12] Satyanarayana, A. and Prabhakar, A., *New Topological Formula and Rapid Algorithm for Reliability Analysis of Complex Networks*, IEEE Transactions on Reliability, 27, 1978, pp. 82-100

[13] Satyanarayana, A. and Wood, K., *Polygon-to-Chain Reductions and Network Reliability*, ORC 82-4, U.C. Berkeley, March 1982

[14] Valiant, L.G., *The Complexity of Enumeration and Reliability Problems*, SIAM J. Computing, 8, 1979, pp. 410-421

[15] Van Slyke, R. and Frank, H., *Network Analysis: Part 1*, NETWORKS, vol. 1, no. 3, 1972, pp. 279-290

[16] Wood, R.K., *Polygon-to-Chain Reductions and Extensions for Reliability Evaluation of Undirected Networks*, Ph.D. dissertation, ORC 82-12, U.C. Berkeley, 1982

Chapter 2

1. The Reliability Problem

We assume throughout this chapter that an instance of the n-component reliability problem is specified by the following data:

(a) for each component i, where $1 \leq i \leq n$, a failure probability p_i. Component i is **failing** with probability p_i and **working** with probability $1 - p_i$ independently of all other components in the system. The assumption made here that components are s-independent is convenient, but not essential, for the development that follows.

(b) a specification of the combinations of component states which cause the overall system to fail. As we describe below, this specification consists of a list of m **failure sets**.

Given these data, the problem is to estimate the failure probability of the system.

In order to discuss how failure sets are specified, and how failure sets together with the failure probabilities of components determine the failure probability of the system, we require further definitions. A **system state** is an n-tuple $(b_1, \ldots, b_n)$ where $b_i = 0$ if component i is failing, and $b_i = 1$ if component i is working. There are 2^n different system states and the probability of any particular system state $(b_1, \ldots, b_n)$ is $\prod_{i=1}^{n} p_i^{1-b_i} (1 - p_i)^{b_i}$. For example, in an eight-component system, the system state (0, 1, 1, 0, 1, 0, 1, 1) has proba-

bility $p_1(1-p_2)(1-p_3)p_4(1-p_5)p_6(1-p_7)(1-p_8)$.

Let $(c_1, \ldots, c_n)$ be an n-tuple, each component of which is either 0, 1, or *. Such an n-tuple represents a set F_k of system states, according to the following rule: $(b_1, \ldots, b_n)$ is an element of F_k provided that $c_i = 0$ implies $b_i = 0$, $c_i = 1$ implies $b_i = 1$, and $c_i = *$ implies b_i may be either 0 or 1. Thus, the 8-tuple (0, *, 1, 1, 1, *, 0, 1) represents the following set of four system states: {(0, 0, 1, 1, 1, 0, 0, 1), (0, 0, 1, 1, 1, 1, 0, 1), (0, 1, 1, 1, 1, 0, 0, 1) and (0, 1, 1, 1, 1, 0, 1, 1)}. A set F_k represented in this way by an n-tuple $(c_1, \ldots, c_n)$ is called a **failure set** provided that each system state in F_k is a failure state of the system.

Let F be the set of all failure states of the system. We assume that F is specified as the union of failure sets $F_1, F_2, \ldots, F_m$, each of which is described by an n-tuple of 0's, 1's and *'s. Thus $Pr[F]$, which is the failure probability of the system, can be written as $Pr[\bigcup_{k=1}^{m} F_k]$.

The probability of a failure set F_k is the sum of the probabilities of the system states contained in F_k:

$$Pr[F_k] = \sum_{s \in F_k} Pr[s] \tag{1.1}$$

Alternatively, if the n-tuple $(c_1, \ldots, c_n)$ specifies F_k, then

$$Pr[F_k] = \prod_{i=1}^{n} p_i^{c_i'} (1-p_i)^{c_i''} \text{ where}$$

$$c_i' = \begin{cases} 1 & \text{if } c_i = 0 \\ 0 & \text{if } c_i = 1 \text{ or } * \end{cases}$$

and

$$c_i'' = \begin{cases} 1 & \text{if } c_i = 1 \\ 0 & \text{if } c_i = 0 \text{ or } * \end{cases}$$

For example, $Pr\,[(0, *, 1, 1, 1, *, 0, 1)]$ is

$$p_1(1-p_3)(1-p_4)(1-p_5)p_7(1-p_8).$$

Let $Pr\,[s \mid F_k]$ denote the conditional probability of system state s given that s is drawn from the set of states F_k.

Then

$$Pr\,[s \mid F_k] = \begin{cases} \dfrac{Pr[s]}{Pr[F_k]} & s \in F_k \\ 0 & s \notin F_k \end{cases} \tag{1.2}$$

For example, $Pr\,[(0, 0, 1, 1, 1, 1, 0, 1) \mid (0, *, 1, 1, 1, *, 0, 1)]$ is

$$\frac{p_1p_2(1-p_3)(1-p_4)(1-p_5)(1-p_6)p_7(1-p_8)}{p_1(1-p_3)(1-p_4)(1-p_5)p_7(1-p_8)} = p_2(1-p_6)$$

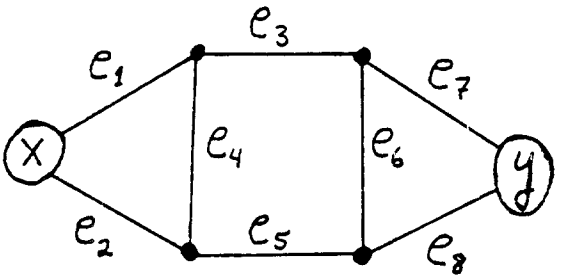
In network reliability and many other types of problems the n-component system has a **monotonic** property. Define a partial order $\subseteq$ on system states as follows:

$s \subseteq t$ **iff** the set of components failing in system state s is a subset of the set of components failing in state t.

An n-component system is monotone if for all failure states s and system states t, $s \subseteq t$ implies t is also a failure state. In any monotonic n-component system the set of failure states can be represented by failure sets which correspond naturally to minimal failure states (minimal with respect to the partial order $\subseteq$). If s is a minimal failure state, then the corresponding failure set F_k is the set of all states t such that $s \subseteq t$. Thus, F_k can be written as $(c_1, \ldots, c_n)$ where $c_i = 0$ if component i is failing in state s, and $c_i = *$ if component i is working in state s.

2. A Network Example

Figure 1 shows an undirected network with eight edges and two designated vertices, x and y. Each edge e_i is failing with probability p_i and working with probability $1 - p_i$. The network is said to fail if there is no path of working edges between x and y.



$$(p_1, p_2, \ldots, p_8) = (.1, .5, .4, .3, .2, .4, .1, .2)$$

Figure 1 - *Two - Terminal Reliability Problem*

An $x - y$ **cut set** is a minimal set of edges whose deletion leaves no path between x and y. Then the network fails if an only if all the edges in some $x - y$ cut set fail, and thus the set of failure states of the system can be expressed as the union of failure sets which correspond to the $x - y$ cuts. These failure sets are listed in Table 2, where the probability of each failure set is also given.

k	8-tuple representing failure set F_k								$Pr(F_k)$
	1	2	3	4	5	6	7	8	
1	*	*	*	*	*	*	0	0	.02
2	*	*	*	*	0	0	0	*	.008
3	0	*	*	0	*	0	*	0	.0024
4	*	*	0	*	*	0	*	0	.032
5	*	*	0	*	0	*	*	*	.08
6	0	*	*	0	0	*	*	*	.006
7	*	0	*	0	*	0	0	*	.008
8	*	0	0	0	*	*	*	*	.06
9	0	0	*	*	*	*	*	*	.05

Table 2 - *List of Failure Sets for Network of Figure 1*

3. Monte-Carlo Area Estimation

In this section we present a Monte-Carlo technique to estimate the area of a region in the Euclidean plane. Most of the Monte-Carlo algorithms presented in this paper for the estimation of the failure probability of a system are analogous to this area estimation technique, and the explanation of the algorithms will rely heavily upon this analogy.

Suppose a region E of known area $A(E)$ encloses the region U of unknown area in the plane. Furthermore, suppose region E is subdivided into b blocks such that the area of each block i is known to be a_i, thus $A(E) = \sum_{i=1}^{b} a_i$. Suppose the region U consists of some subset of these blocks. Let α_i indicate whether or not block i is in region U, i.e.,

$$\alpha_i = \begin{cases} 1 & \text{if block } i \in U \\ 0 & \text{if block } i \notin U \end{cases}$$

Then the area of region U, $A(U)$, can be written as $\sum_{i=1}^{b} a_i \cdot \alpha_i$.

A straightforward method for determining $A(U)$ is to compute the above sum, but if b is large, this is a costly calculation. In the applications we consider b is very large.

Suppose instead we have a method to randomly select block i out of the set of all blocks with probability $\frac{a_i}{A(E)}$. An unbiased estimator, Y, of the quantity $A(U)$ can be generated by randomly selecting block i with probability $\frac{a_i}{A(E)}$ and letting $Y = \alpha_i \cdot A(E)$. The expected value of Y, $E[Y]$, is $\sum_{i=1}^{b} \frac{a_i}{A(E)} \alpha_i \cdot A(E) = A(U)$.

The algorithm can be repeated many times yielding estimator Y_j in the j^{th} trial. $\bar{Y} = \frac{(Y_1 + \ldots + Y_N)}{N}$ is an unbiased estimator of $A(U)$.

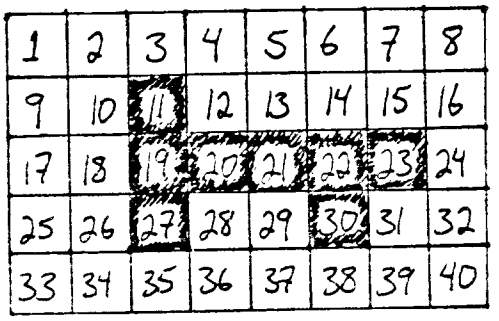


Figure 3 - *E is entire region*
U is the shaded region, E is subdivided into 40 blocks

$\alpha_5 = 0, \alpha_{11} = 1.$

We will show in the next section that the number of trials necessary to guarantee a specified degree of accuracy and confidence in the estimator is linearly proportional to $\frac{A(E)}{A(U)}$. It is therefore desirable that this ratio be small.

4. Convergence of Monte-Carlo Algorithms

Suppose the Monte-Carlo algorithm is repeated N times. Let Y_t be the value of the estimator obtained from the t^{th} trial. Let $\bar{Y} = \frac{(Y_1 + Y_2 + \ldots + Y_N)}{N}$. A meaningful measure of the quality of the estimator $\bar{Y}$ is its **relative error**, given by

$$\left| \frac{\bar{Y} - u}{u} \right|$$

where u is the expected value of the estimator produced by the algorithm (u is the quantity we are trying to estimate). We next derive an upper bound on the number of trials N required to guarantee that the relative error will exceed a specified value ε with probability less than or equal to a specified value δ. For example, if we specify $\varepsilon = .05$ and $\delta = .1$, we are requiring N to be large enough that the relative error will be greater than 5% no more than 10% of the time. For the sake of brevity, a Monte-Carlo algorithm will be called an (ε, δ) **algorithm** if the algorithm achieves these guarantees.

Let σ^2 be the variance of Y, where Y is the value obtained in a single Monte-Carlo trial. Then the variance of $\tilde{Y}$ is $\frac{\sigma^2}{N}$. By Chebyshev's inequality

$$Pr\left[\left|\frac{\tilde{Y}-u}{u}\right|>\varepsilon\right] = Pr\left[\,|\,\tilde{Y}-u\,|>\varepsilon u\,\right] \le \frac{\sigma^2}{N\varepsilon^2u^2}.$$

Thus, in order that $Pr\left[\left|\frac{\tilde{Y}-u}{u}\right|>\varepsilon\right]$ be less than or equal to δ, it suffices that

$$N \ge \frac{\sigma^2}{u^2}\cdot\frac{1}{\delta\varepsilon^2}.$$

Notice that there are two factors in the right-hand side of the inequality: $\frac{\sigma^2}{u^2}$ which depends on the Monte-Carlo algorithm and problem instance; and $\frac{1}{\delta\cdot\varepsilon^2}$ which depends on the desired relative accuracy of $\tilde{Y}$ and the desired confidence level of obtaining this accuracy. The rest of this analysis will only be concerned with $\frac{\sigma^2}{u^2}$.

For the area estimation algorithm presented in the previous section the random variable Y, which is the estimator of the area of region U, $A(U)$, is a Bernoulli random variable multiplied by the area of region E, $A(E)$. Thus

$\sigma^2 = A(E)\cdot A(U) - A(U)^2$ and thus

$$\frac{\sigma^2}{u^2} = \frac{A(E)}{A(U)} - 1.$$

The number of trials, N, necessary to achieve an (ε, δ) algorithm is

$$\left[\frac{A(E)}{A(U)} - 1\right]\cdot\frac{1}{\delta\varepsilon^2}. \qquad (4.1)$$

We note that since Chebychev's inequality is true for any probability distribution this may be a very conservative upper bound on the number of trials.

If the area of region U is not much smaller than the area of region A, then the number of trials necessary is small. Our goal is to design an algorithm such that $\frac{A(E)}{A(U)}$ is small. We first present a standard algorithm to estimate the failure probability of an n-component system which can be viewed as an area estimation algorithm where for n-component systems typically encountered in practice, the ratio $\frac{A(E)}{A(U)}$ is very large.

5. Straight Simulation Monte-Carlo Method

A simple Monte-Carlo algorithm to estimate $Pr[F]$, the probability that the n-component system is in a failure state, follows.

Step 1 randomly select system state s with probability $Pr[s]$

Step 2 the estimator Y of $Pr[F]$ is $\begin{cases} 1 \text{ if } s \in F \\ 0 \text{ otherwise} \end{cases}$

The analogy to the area estimation technique goes as follows. The set of all system states corresponds to the enclosing region E. The system states correspond to the blocks into which the enclosing region is subdivided, where the area of each system state s is $Pr[s]$ and hence $A(E) = 1$. Region U comprises the set of all failure states F and hence the area of region U is $Pr[F]$. The expected value of Y is equal to $Pr[F]$, however if $Pr[F]$ is small compared to one (which is typical of n-component systems) the number of trials must be very large to estimate $Pr[F]$ accurately.

The motivation for this work is to design a Monte-Carlo algorithm which estimates $Pr[F]$ accurately with a small number of trials even when $Pr[F]$ is very small.

6. A Description of the Coverage Algorithm

Assume that F, the set of failure states of an n-component system, is specified as the union of failure sets $F_1, F_2, \ldots, F_m$. Then the failure probability of the system is given by

$$Pr[F] = Pr[\bigcup_{k=1}^{m} F_k].$$

We noted in Section 2 that it is easy to compute $Pr[F_k]$ where F_k is any one of the m failure sets. If the m failure sets were disjoint, then calculating $Pr[F]$ would be simply a matter of computing $\sum_{i=1}^{m} Pr[F_k]$. Unfortunately, the failure sets are not disjoint in general. Furthermore, the classical formulas for the probability of a union of sets do not lead to efficient algorithms for evaluating $Pr[F]$. For example, the inclusion-exclusion formula

$$Pr[F] = Pr[\bigcup_{k=1}^{m} F_k] = \sum_{k=1}^{m} Pr[F_k] - \sum_{k_1=1}^{m} \sum_{k_2=1}^{k_1-1} Pr[F_{k_1} \cap F_{k_2}]$$

$$+ \sum_{k_1=1}^{m} \sum_{k_2=1}^{k_1-1} \sum_{k_3=1}^{k_2-1} Pr[F_{k_1} \cap F_{k_2} \cap F_{k_3}] + \ldots + (-1)^{m+1} Pr[F_1 \cap F_2 \cap \ldots \cap F_m]$$

entails $2^m - 1$ terms. The terms can fluctuate wildly in value, making it impossible in general to obtain a good approximation by truncating the expansion after the first few terms.

Another well-known formula is

$$Pr[F] = Pr[\bigcup_{k=1}^{m} F_k] = Pr[F_1]$$

$$+ Pr[F_2 \cap \bar{F}_1] + Pr[F_3 \cap (\overline{F_1 \cup F_2})] + \ldots + Pr[F_m \cap (\overline{F_1 \cup \ldots \cup F_{m-1}})].$$

This expansion has only m terms, but the individual terms seem hard to compute, and the most obvious algorithms based on this formula require a number

of steps exponential in m. In fact, to compute $Pr[F]$ exactly is NP-hard [1].

We will not attempt to compute $Pr[F]$ exactly. Instead, the ability to easily calculate $\sum_{i=1}^{k} Pr[F_k]$ will be used to estimate $Fr[\bigcup_{k=1}^{m} F_k]$. Let **cov(s)**, the **coverage** of failure state s, be the number of failure sets containing s. For example, in the network reliability problem described in Figure 1 and Table 2, the coverage of failure state $s = (1, 0, 1, 0, 0, 0, 0, 0)$ is three because s is contained in the failure sets F_1, F_2, and F_7. As a second example, suppose the set of failure states is the union of three failure sets shown in Figure 4. Then

$$F = F_1 \cup F_2 \cup F_3, cov(s_1) = cov(s_2) = 1, cov(s_3) = 2, \text{ and } cov(s_4) = 3.$$

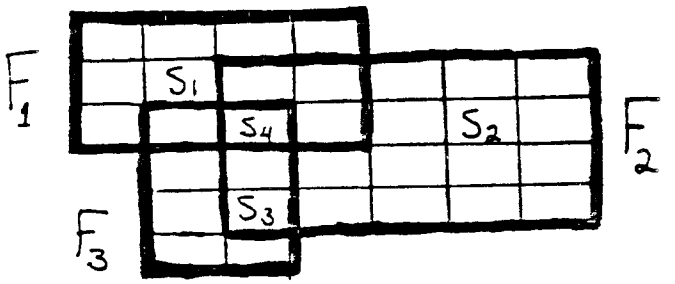


Figure 4 - *A representation of a set of failure states F as $F_1 \cup F_2 \cup F_3$*

Notice that each failure state s contributes a total of $cov(s) \cdot Pr[s]$ to $\sum_{k=1}^{m} Pr[F_k]$, because s contributes $Pr[s]$ to $Pr[F_k]$ for each F_k containing s. If instead we could arrange that each failure state s contributes a total of $Pr[s]$, then the total contribution of all failure states would be $Pr[F]$.

The new algorithm, called the **coverage algorithm**, is analogous to the area estimation algorithm. The area of the enclosing region E is $\sum_{k=1}^{m} Pr[F_k]$. The blocks that comprise region E are all ordered pairs (s, k), where s is a failure state contained in failure set F_k. Thus, failure state s will appear as the first component in exactly $cov(s)$ blocks. We let the area of each block (s, k),

denoted $\alpha(s, k)$, be equal to $Pr[s]$. Thus, the total area of all blocks in which s is the first component is $cov(s) \cdot Pr[s]$, and the total area of all blocks is indeed $\sum_{k=1}^{m} Pr[F_k]$.

Now we define region U, whose total area will be $Pr[F]$. For each failure state s we let exactly one the $cov(s)$ blocks in which s is the first component be in the region U. Thus, $\alpha(s, k) = 1$ for exactly one of the $cov(s)$ blocks in which s is the first component and $\alpha(s, k) = 0$ for the other $cov(s) - 1$ such blocks. Notice that it does not matter which of the $cov(s)$ blocks is in the region U. This makes it possible to select any one of the $cov(s)$ blocks in which s is the first component to be in region U.

Figure 5 illustrates the sample space for the set F of failure states shown in Figure 4. The region U is shaded in this figure. In this example block (s, k) is in region U if F_k is the smallest indexed failure set containing s.

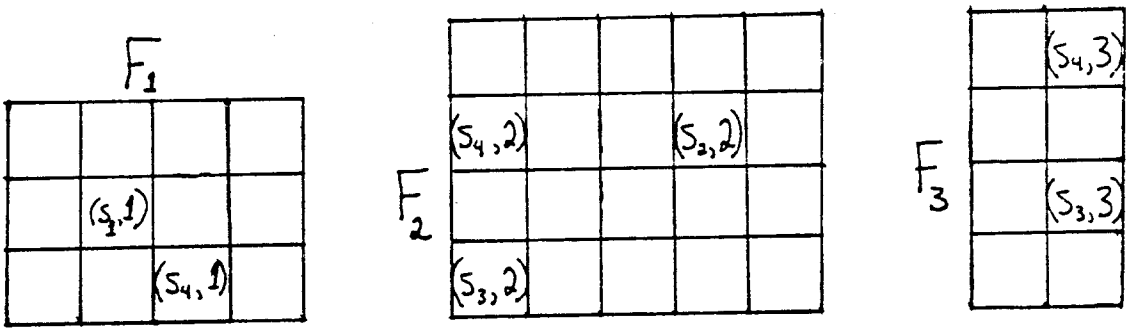


Figure 5 - *Sample space for F shown in Figure 4.*

How do we randomly select block (s, k) with probability $\frac{Pr[s]}{\sum_{k=1}^{m} Pr[F_k]}$?

This is a two-step process. First, we randomly select failure set F_k with probability $\frac{Pr[F_k]}{\sum_{k=1}^{m} Pr[F_k]}$. This is easy to do once the probability of each failure set has been computed. This selects the second component k of the block. Then we randomly select a failure state s from failure set F_k with probability $\frac{Pr[s]}{Pr[F_k]}$.

This is also easy to do, as we discuss in the next section. This selection picks the first component s of the block. Notice that this two-step process picks block (s, k) with probability $\frac{Pr[s]}{\sum_{k=1}^{m} Pr[F_k]}$.

The computation of $\alpha(s, k)$ is discussed in detail in the following sections. One trial of the coverage algorithm randomly selects block (s, k) by this two-step process, and returns $\alpha(s, k) \cdot \sum_{k=1}^{m} Pr[F_k]$ as the estimator of $Pr[F]$.

The advantage of this algorithm over the straight simulation algorithm is the number of trials, N, necessary to achieve an (ε, δ) algorithm. Recall from Formula (4.1) that for the area estimation algorithm, if N is greater than or equal to

$$\frac{\left(\frac{A(E)}{A(U)} - 1\right)}{\delta \varepsilon^2} \tag{6.1}$$

then the algorithm is an (ε, δ) algorithm. For the straight simulation algorithm this value of N is

$$\frac{\left(\frac{1}{Pr[F]} - 1\right)}{\delta \varepsilon^2} \tag{6.2}$$

But Formula (6.2) involves $Pr[F]$, the quantity the algorithm is attempting to estimate. There can be no upper bound on N derived from the formula since $Pr[F]$ can be arbitrarily small. Thus, without any prior information about $Pr[F]$ it is impossible, using only this formula, to put an upper bound on N a priori which will guarantee an (ε, δ) algorithm.

Let τ_i be the probability that a state s selected from among all 2^n states with probability $Pr[F]$ has $cov(s)$ equal to i, i.e.

$$\tau_i = \sum_{\substack{s \text{ a state s.t.} \\ cov(s)=i}} Pr[s] \tag{6.3}$$

Thus,

$$A(U) = Pr[F] = \sum_{i=1}^{m} \tau_i \tag{6.4}$$

and

$$A(E) = \sum_{i=1}^{m} Pr[F_i] = \sum_{i=1}^{m} i \cdot \tau_i \tag{6.5}$$

The new algorithm is an (ε, δ) algorithm if the number of trials is

$$\frac{\left\lfloor \frac{A(E)}{A(U)} - 1 \right\rfloor}{\delta \cdot \varepsilon^2} = \frac{\left\lfloor \frac{\sum_{i=1}^{m} i \cdot \tau_i}{\sum_{i=1}^{m} \tau_i} - 1 \right\rfloor}{\delta \cdot \varepsilon^2} \tag{6.6}$$

But,

$$\frac{\sum_{i=1}^{m} i \cdot \tau_i}{\sum_{i=1}^{m} \tau_i} \leq m \tag{6.7}$$

Thus, if we let $N = \frac{m}{\delta \varepsilon^2}$ for the new algorithm we will have an (ε, δ) algorithm. Since m is known before we run the algorithm it is possible to put an a priori upper bound on the number of trials necessary to guarantee an (ε, δ) algorithm. An even tighter upper bound is derived in Section 13.

Through the insight gained by this new algorithm, we are able to derive an easily computable upper bound on the number of trials to perform to guarantee on (ε, δ) algorithm for the straight simulation algorithm. The reasoning goes as follows: Equations (6.7), (6.5) and (6.4) imply that Formula (6.2) is less than or equal to

$$\frac{m}{\sum_{k=1}^{m} Pr[F_k]} \cdot \frac{1}{\delta \varepsilon^2} \tag{6.8}$$

Thus, the straight simulation algorithm is an (ε, δ) algorithm if the number of trials is given by to Formula (6.8).

We now compare the upper bounds on the number of trials derived for the straight simulation algorithm and the coverage algorithm. Let N be the upper bound on the number of trials for the straight simulation algorithm and $\tilde{N}$ be the upper bound on the number of trials for the coverage algorithm. We see that

$$\tilde{N} = N \cdot \sum_{k=1}^{m} Pr[F_k] \tag{6.9}$$

If $\sum_{k=1}^{m} Pr[F_k]$ is less than one, the upper bound of the number of trials to achieve an (ε, δ) algorithm for the new algorithm is less than the upper bound on the number of trials to achieve an (ε, δ) algorithm for the straight simulation algorithm. If $\sum_{k=1}^{m} Pr[F_k]$ is greater than one, the reverse inequality holds. Thus, a decision about which algorithm to use can be made based on the value of $\sum_{k=1}^{m} Pr[F_k]$. We expect $\sum_{k=1}^{m} Pr[F_k] \ll 1$ when the component failure probabilities are small, and therefore the upper bound on the number of trials for the new algorithm will be substantially less than the number of trials for the straight simulation algorithm.

7. An Implementation of the Coverage Algorithm

We next present an implementation of the new Monte-Carlo algorithm. The input to the algorithm is an n-component reliability problem in the format described previously. The output from one trial of the algorithm is a number which is an unbiased estimator of the system failure probability.

Preprocessing

For $k = 1, 2, \ldots, m$ compute $Pr[F_k]$.

If F_k is represented by the array $(c_1, \ldots, c_n)$ then, as described previously, $Pr[F_k] = \prod_{i=1}^{n} p_i^{c'_i}(1-p_i)^{c''_i}$

Allocate an array FS of size m.

For $k = 1, 2, \ldots, m$, $FS[k] \leftarrow \dfrac{\sum_{j=1}^{k} Pr[F_j]}{\sum_{j=1}^{m} Pr[F_j]}$

Array FS will be used to randomly select failure set F_k with probability $\dfrac{Pr[F_k]}{\sum_{j=1}^{m} Pr[F_j]}$.

Monte-Carlo Trial

Step 1 Randomly select a failure set F_k with probability $\dfrac{Pr[F_k]}{\sum_{j=1}^{m} Pr[F_j]}$. This can be done by picking a random number r from the uniform distribution over [0, 1] and determining

$k = \min \{ j \mid FS[j] \geq r \}$ using binary search.

Step 2 Randomly select $s \in F_k$ with probability $Pr[s \mid F_k] = \dfrac{Pr[s]}{Pr[F_k]}$.

If F_k is specified by the array $(c_1, c_2, \ldots, c_n)$ then $s = (b_1, b_2, \ldots, b_n)$ is chosen as follows:

if $c_i = 0$, then $b_i = 0$
if $c_i = 1$, then $b_i = 1$
if $c_i = *$, then choose $b_i = 0$ with probability p_i and $b_i = 1$ with probability $1 - p_i$

At this point block (s, k) has been selected.

Step 3 Compute $\alpha(s, k)$

$$\text{Define } \alpha(s, k) = \begin{cases} 1 \text{ if } F_k \text{ is the smallest indexed} \\ \qquad \text{failure set such that } s \in F_k \\ 0 \text{ otherwise} \end{cases}$$

Then $\alpha(s, k)$ can be computed by finding the smallest index i such that $s \in F_i$. If $k = i$ then $\alpha(s, k) = 1$, otherwise $\alpha(s, k) = 0$.

Step 4 The estimator, Y, of $Pr[F]$ is $\alpha(s, k) \cdot \sum_{k=1}^{m} Pr[F_k]$.

The time to perform the preprocessing step given the list of failure sets $F_1, \ldots, F_m$ is $O(m \cdot n)$. For the two-terminal problem the failure sets could be given implicitly by a data structure representing the graph. The preprocessing step consists of listing all the $x - y$ cuts in the graph and then proceeding as before. All of the cuts in the graph can be listed in time $O(m \cdot n)$ [2] (in this case m = # cuts in the graph, n = # edges in the graph), so the total preprocessing time is $O(m \cdot n)$ in this case also.

Step 1 of the trial takes time $O(\log m)$ using binary search to find k. Since there are 2^n system states, there can be at most 2^n failure sets, hence the time for Step 1 is $O(n)$. Step 2 also takes time $O(n)$, and Step 4 can be performed in constant time.

In this implementation the computation of $\alpha(s, k)$ in Step 3 takes time at most $O(m \cdot n)$. This can be seen as follows: $\alpha(s, k)$ is computed by sequentially searching through the failure sets until we find a failure set F_i such that $s \in F_i$, and then $\alpha(s, k) = 1$ if $i = k$, otherwise $\alpha(s, k) = 0$. Each test for membership of s in a failure set takes $O(n)$ time. In the worst case all m failure sets will be examined, thus the total running time is $O(m \cdot n)$.

In Section 6 we found that $\frac{m}{\delta \varepsilon^2}$ trials are sufficient to achieve an (ε, δ)

algorithm. Thus, the running time for all the trials is

$$O\left(\frac{m^2 n}{\delta\, \varepsilon^2}\right). \tag{7.1}$$

The running time per trial is dominated by the time to compute $\alpha(s, k)$ in Step 3 of the algorithm. In the following sections we will discuss methods to substantially reduce the running time of the algorithm based on alternative ways to compute α. First, we will generalize the definition of α in a way that helps us compute α quickly.

8. A Generalization of the Coverage Algorithm

The requirement that $\alpha(s, k) = 1$ for exactly one of the $cov(s)$ blocks in which s is the first component and $\alpha(s, k) = 0$ for the other $cov(s) - 1$ such blocks can be relaxed. A more general scheme is to allow $\alpha(s, k)$ to be a random variable such that

$$\sum_{\{k \mid s \in F_k\}} E[\alpha(s, k)] = 1 . \tag{8.1}$$

Any such scheme can be viewed as a probabilistic allocation of the probability of system state s to the set of blocks in which s is the first component. Any allocation scheme fulfilling these more general requirements will produce an unbiased estimator of $Pr[F]$. As an example, letting $\alpha(s, k) = \frac{1}{cov(s)}$ for all blocks in which s is the first component, fulfills these requirements. This particular allocation scheme has the smallest variance among all allocation schemes, which can be seen as follows. The variance σ^2 is equal to $E[Y^2] - E[Y]^2$, but, since $E[Y]^2 = Pr[F]^2$ for any choice of α, the allocation which minimizes $E[Y^2]$ will have the smallest variance. Now,

$$E[Y^2] = \left[\sum_{k=1}^{m} Pr[F_k]\right] \left[\sum_{s \in F_k} Pr[s] \cdot \left[\sum_{\{k \mid s \in F_k\}} E[\alpha^2(s, k)]\right]\right]. \tag{8.2}$$

The choice which minimizes $\sum_{\{k \mid s \in F_k\}} E[\alpha^2(s, k)]$ subject to $\sum_{\{k \mid s \in F_k\}} E[\alpha(s, k)] = 1$ will minimize the variance. A little algebraic manipulation shows that this is minimized when $\alpha(s, k) = \frac{1}{cov(s)}$ for all blocks (s, k) in which s is the first component.

9. A Hybrid Allocation Scheme - The Cutoff Method

Let c, the cutoff, be a positive integer. We will allocate the probability of failure state s among the blocks in which s is the first component as follows:

1.) If $cov(s) \leq c$ then allocate $Pr[s]$ equally among all $cov(s)$ blocks, i.e., $\alpha(s, k) = \frac{1}{cov(s)}$ for all $cov(s)$ such blocks.

2.) If $cov(s) > c$ then allocate $Pr[s]$ equally among c of the blocks, i.e., $\alpha(s, k) = \frac{1}{c}$ for c of the blocks and $\alpha(s, k) = 0$ for the other $cov(s) - c$ such blocks.

The reason that c is called the cutoff is because in the implementation of the hybrid allocation scheme the algorithm finds the first $\min\{c, cov(s)\}$ failure sets that contain s. The probability of state s is allocated equally among the blocks in which the second component is the index of one of these $\min\{c, cov(s)\}$ failure sets. Thus, $\alpha(s, k) = \frac{1}{\min\{c, cov(s)\}}$ if k is the index of one of these failure sets, otherwise $\alpha(s, k) = 0$. The value c is an upper bound, or cutoff, on the number of failure sets containing s that the algorithm must find in order to compute $\alpha(s, k)$.

When c is infinite then $\alpha(s, k) = \frac{1}{cov(s)}$ for all blocks (s, k); this is the minimum-variance case. When c is one then $\alpha(s, k) = 1$ for exactly one of the $cov(s)$ blocks in which s is the first component and $\alpha(s, k) = 0$ for the other $cov(s) - 1$ such blocks; this is the maximum-variance case for the hybrid

method. Recall from formulas (6.3), (6.4) and (6.5) the definition of r_i. Then $\frac{i r_i}{\sum_{i=1}^{m} i r_i}$ is the probability a system state with coverage i is randomly selected in one trial of the algorithm. Thus, for the cutoff method $\frac{\sigma^2}{u^2}$ can be expressed as

$$\left(\frac{\sum_{1 \le i < c} \frac{r_i}{i} + \frac{1}{c} \sum_{i \ge c} r_i}{\sum_{i \ge 1} r_i} \right) \left(\frac{\sum_{i \ge 1} i r_i}{\sum_{i \ge 1} r_i} \right) - 1 \qquad (9.1)$$

It is trivial to modify the coverage algorithm presented in Section 7 to incorporate the hybrid allocation scheme. The only change is in the computation of $\alpha(s, k)$, which can be described as follows:

Step 3 Sequentially search through the failure sets until either c failure sets F_i are found such that $s \in F_i$ or all the failure sets are searched. Let l be the number of failure sets found such that $s \in F_i$, then $l = \min(c, cov(s))$. If k is the index of one of the l failure sets found, then $\alpha(s, k) = \frac{1}{l}$, otherwise $\alpha(s, k) = 0$.

If we let $c = \infty$ (in which case $\alpha(s, k) = \frac{1}{cov(s)}$), then the time per trial is still $O(m \cdot n)$. The best upper bound we can prove on the number of trials necessary to achieve an (ε, δ) algorithm is still $\frac{m}{\delta \varepsilon^2}$, so the total running time is still $O\left[\frac{m^2 n}{\delta \varepsilon^2}\right]$. However, using the algorithm with $c = \infty$ will result in a stochastically better estimate of $Pr[F]$ then using the algorithm with a smaller value of c for the same number of trials.

10. A Substantially Faster Variation of the Coverage Algorithm

In this section we present an alternative implementation of the coverage algorithm presented in Section 7. We will prove that an upper bound on the running time of this new algorithm to guarantee an (ε,δ) algorithm is $O\left[\frac{m \cdot n}{\delta \varepsilon^2}\right]$.

Recall that for the previous implementation of the coverage algorithm we were able to prove an upper bound on the running time of $O\left[\frac{m^2 n}{\delta \varepsilon^2}\right]$. Since m is typically very large in comparison to n, this improvement in the running time is substantial. We call the new algorithm the **linear time coverage algorithm** to emphasis the fact that the running time is linear in the input size (which is $m \cdot n$) divided by $\delta \cdot \varepsilon^2$.

We assume the most general input format. The input consists of the failure probabilities of the n components and a list of the m failure sets in the format described in the first section of this paper.

We first present the algorithm. The preprocessing step is exactly the same as it is for the coverage algorithm. The first two steps, randomly selecting block (s, k) with probability $\frac{Pr[s]}{\sum_{k=1}^{m} Pr[F_k]}$, are also exactly the same as they are for the coverage algorithm. Once block (s, k) has been selected, the linear time coverage algorithm produces two unbiased estimators, $\alpha(s, k)$ and $\alpha'(s, k)$, of $\frac{1}{cov(s)}$ in Step 3 which are independent of one another. Note that both $\sum_{\{k \mid s \in F_k\}} E[\alpha(s, k)] = 1$ and $\sum_{\{k \mid s \in F_k\}} E[\alpha'(s, k)] = 1$. Thus, either $Y = \alpha(s, k) \sum_{k=1}^{m} Pr[F_k]$ or $Y' = \alpha'(s, k) \sum_{k=1}^{m} Pr[F_k]$ will be an unbiased estimator of $Pr[F]$.

A Description of the Linear Time Coverage Algorithm

$\bar{Y} \leftarrow 0$, $\bar{Y}' \leftarrow 0$

numtrials $\leftarrow 0$

time $\leftarrow 0$

Repeat steps 1-5 until *time* $= \dfrac{c \cdot m}{\delta \cdot \varepsilon^2}$

Step 1 randomly select a failure set F_k with probability $\dfrac{Pr[F_k]}{\sum_{k=1}^{m} Pr[F_k]}$ as before

Step 2 randomly select $s \in F_k$ with probability $\dfrac{Pr[s]}{Pr[F_k]}$ as before

Step 3 $l \leftarrow 0$

Do until a failure set F_i is selected such that $s \in F_i$

randomly select failure set F_i with probability $\frac{1}{m}$
$l \leftarrow l + 1$
Check to see if $s \in F_i$ (*)
time $\leftarrow$ *time* + 1

$$\alpha(s, k) = \frac{l}{m}$$

$$\alpha'(s, k) = \begin{cases} 1 \text{ if } i = k \\ 0 \text{ otherwise} \end{cases}$$

Step 4 The estimators, Y & Y', of $Pr[F]$ are

$$Y = \alpha(s, k) \cdot \sum_{k=1}^{m} Pr[F_k]$$

$$Y' = \alpha'(s, k) \cdot \sum_{k=1}^{m} Pr[F_k]$$

Step 5 *numtrials* $\leftarrow$ *numtrials* + 1 , $\bar{Y} \leftarrow \bar{Y} + Y$, $\bar{Y}' \leftarrow \bar{Y}' + Y'$

Go to step 1

Step 6 Estimator 1 = $\frac{\tilde{\gamma}}{numtrials}$

Estimator 2 = $\frac{\tilde{\gamma}'}{numtrials}$

One trial of the algorithm is the execution of steps 1-5. The value of the constant c is discussed in the following theorem, which establishes that the choice $c = 16$ gives an (ε, δ) algorithm. The running time for each trial is dominated by the time to perform the test marked with a (*) in Step 3. This test takes $O(n)$ time to perform. The total running time is therefore $\frac{c \cdot m \cdot n}{\delta \cdot \varepsilon^2}$. To simplify notation, we call the number of times (*) is performed per trial the length of the trial.

Now we will show that $E[\alpha(s, k)] = \frac{1}{cov(s)}$. Suppose (s, k) is picked in Steps 1 and 2 of one trial. We are interested in computing

$$E[\alpha(s, k)] = \frac{E[\text{length of trial}]}{m}.$$

Each time line (*) is executed there is a chance of $\frac{cov(s)}{m}$ that $s \in F_i$ since each failure set is picked with probability $\frac{1}{m}$ and s is an element of $cov(s)$ failure sets. Let $X(s, k)$ be the length of the trial given that (s, k) is picked in Steps 1 and 2 of the trial. Then $X(s, k)$ is a random variable geometrically distributed with rate $\frac{cov(s)}{m}$, and

$$E[\alpha(s, k)] = \frac{E[X(s, k)]}{m} = \frac{m}{cov(s)} \cdot \frac{1}{m} = \frac{1}{cov(s)}. \qquad (10.3)$$

Now we will show that $E[\alpha'(s, k)] = \frac{1}{cov(s)}$. Suppose (s, k) is picked in Steps 1 and 2 of one trial. Once the algorithm finds a failure set F_i such that $s \in F_i$, the probability that $i = k$ is exactly $\frac{1}{cov(s)}$ independently of the

length of the trial. Thus,

$$E[\alpha'(s,k)] = \frac{1}{cov(s)} \tag{10.4}$$

and $\alpha(s,k)$ and $\alpha'(s,k)$ are independent estimators of $\frac{1}{cov(s)}$.

We now compute $\frac{E[Y^2]}{E[Y]^2}$ and $\frac{E[Y'^2]}{E[Y']^2}$, which we call ϑ and ϑ' in the following discussion to simplify notation. These formulas are needed to analyze the running time of the algorithm. Using the notation introduced in formulas (6.3),(6.4) and (6.5) we see that

$$E[Y^2] = \left[\sum_{i=1}^{m} i\,r_i\right]\left[\sum_{i=1}^{m} \frac{r_i}{i}\left(2 - \frac{i}{m}\right)\right] \le 2\left[\sum_{i=1}^{m} i\,r_i\right]\left[\sum_{i=1}^{m} r_i\right] \tag{10.5}$$

and

$$E[Y'^2] = \left[\sum_{i=1}^{m} i\,r_i\right]\left[\sum_{i=1}^{m} r_i\right]. \tag{10.6}$$

Thus, if we let

$$\mu = \frac{Pr[F]}{\sum_{i=1}^{m} Pr[F_k]} = \frac{\sum_{i=1}^{m} r_i}{\sum_{i=1}^{m} i \cdot r_i} \tag{10.7}$$

we see that

$$\vartheta \le \frac{2}{\mu} \tag{10.8}$$

and

$$\vartheta' \le \frac{1}{\mu} \tag{10.9}$$

The standard technique to guarantee an (ε, δ) algorithm is to compute an a priori upper bound on the number of trials sufficient for an (ε, δ) algorithm. Typically Chebyshev's inequality is used to compute an upper bound on the

number of trials sufficient to guarantee an (ε, δ) algorithm. Instead, we put an upper bound on the number of times (*) in Step 3 must be executed to guarantee an (ε, δ) algorithm. We will prove that if (*) is executed $\frac{c \cdot m}{\delta \cdot \varepsilon^2}$ times during the course of the algorithm, we have an (ε, δ) algorithm (where c is a suitably chosen constant ≥ 1). The intuitive reason why this type of time bound will guarantee an (ε, δ) algorithm follows:

$$E[\text{length of a trial}] = m\,\mu .$$

If (*) is executed $t = \frac{c\,m}{\delta \cdot \varepsilon^2}$ times, then the expected number of trials completed by time t is approximately $\frac{t}{m\,\mu} = \frac{c}{\mu\,\delta \cdot \varepsilon^2}$. The upper bound on both ϑ and ϑ' can be expressed as $\frac{\bar{c}}{\mu}$ (where $\bar{c} = 2$ and 1, respectively). Thus, if the estimator Y (or Y') for each trial were independent of the length of the trial and if c were chosen to be suitably larger than $\bar{c}$, then an application of Chebyshev's inequality would give us the desired result. Since the estimator Y (or Y') from each trial depends on the length of the trial, we will use Kolmogorov's inequality (which is a stronger version of Chebyshev's inequality) to prove the result.

We first introduce some notation which will simplify the proof that executing (*) in Step 3 $O\left(\frac{m}{\delta \varepsilon^2}\right)$ times will guarantee an (ε,δ) algorithm. Let X_j be a random variable denoting the length of the j^{th} trial. Thus, $\{X_j\}$ is a sequence of i.i.d. random variables where

$$E[X_j] = \sum_{i=1}^{m} \frac{i \cdot \tau_i}{\sum_{i=1}^{m} i \cdot \tau_i} \left(\frac{m}{i}\right) = m\,\frac{Pr[F]}{\sum_{i=1}^{m} Pr[F_i]} = m\,\mu . \qquad (10.10)$$

The estimator Y_j (or Y_j') of $Pr[F]$ generated at the end of the j^{th} trial is also a random variable such that $E[Y_j]$ (or $E[Y_j']$) $= Pr[F]$. Y_j is not independent of X_j, but the sequence of ordered pairs $\{(X_j, Y_j)\}$ are independently and

identically distributed (readers familiar with renewal-reward theory will recognize that this is a renewal-reward process).

We let $S_n = \sum_{i=1}^{n} X_i$ be the time at which the n^{th} trial is completed, $R_n = \sum_{i=1}^{n} Y_i$ $(R_n' = \sum_{i=1}^{n} Y_i')$ be the sum of the estimates from the first n trials and $N(t)$ be the number of trials completed by time t.

The running time of the algorithm will depend upon the upper bound on ϑ (ϑ') if the algorithm uses Y (Y') as the estimator of $Pr[F]$. If we let $\bar{c} = 2 \cdot (\bar{c} = 1)$, then $\vartheta \le \frac{\bar{c}}{\mu}$ $\left(\vartheta' \le \frac{\bar{c}}{\mu}\right)$. In the following discussion we express the running time in terms of $\bar{c}$ and use the variables Y and R in place of Y and R or Y' and R' to avoid proving two theorems depending upon whether Y or Y' is used as the estimator of $Pr[F]$.

Theorem: The Linear Time Coverage Algorithm is an (ε, δ) algorithm when the estimator used is Estimator 1 and $c = 16$, or when the estimator used is Estimator 2 and $c = 8$.

Proof : We will prove that

$$Pr\left[\left|\frac{\frac{R_{N(t)}}{N(t)} - E[Y]}{E[Y]}\right| \ge \varepsilon\right] \le \delta$$

when

$$t = \frac{4m\bar{c}}{\delta\,\varepsilon^2}\left[1 + \frac{\varepsilon^{\frac{2}{3}}}{2\bar{c}^{\frac{1}{3}}}\right]\left[1 + \frac{\varepsilon^{\frac{2}{3}}}{\bar{c}^{\frac{1}{3}}}\right]$$

$$E[X] = m \cdot \mu, \quad \frac{E[X^2]}{E[X]^2} \le \frac{2}{\mu}, \quad \frac{E[Y^2]}{E[Y]^2} \le \frac{\bar{c}}{\mu}$$

Comment 1 : When $\bar{c} \geq 1$ and $\varepsilon \leq .25$, $\left[1 + \frac{\varepsilon^{\frac{2}{3}}}{2\bar{c}^{\frac{1}{3}}}\right]\left[1 + \frac{\varepsilon^{\frac{2}{3}}}{\bar{c}^{\frac{1}{3}}}\right] \leq 2$. Thus when Estimator 1 is used as the estimator ($\bar{c} = 2$), the algorithm is an (ε, δ) algorithm when $c = 16$, and when Estimator 2 is used as the estimator ($\bar{c} = 1$), the algorithm is an (ε, δ) algorithm when $c = 8$.

Comment 2 : $\frac{R_{N(t)}}{N(t)}$ is not an unbiased estimator of $Pr[F]$ because $N(t)$ is not a valid stopping time [3]. Nevertheless, the proof of the theorem is still valid. We could complete the trial in process at time t and use $\frac{R_{N(t)+1}}{N(t)+1}$ as the unbiased estimator of $Pr[F]$, but this would make the running time of the algorithm a random variable. We choose not to complete the trial in progress at time t and accept the small bias in the estimator. In the following discussion we use the term "stopping time" to mean the time the algorithm is stopped, we do not mean stopping time as it is defined technically in renewal theory.

Back to Proof : Fix $t' = \frac{d\,m}{\delta\,\varepsilon^2}$, $k = \frac{d}{\mu\,\delta\,\varepsilon^2}$ where d is a constant to be determined later. Let β be a constant (whose optimal value we will determine later) and let $t'' = t'(1 + \beta)$. First, we investigate what can be said using stopping time t''.

$$Pr\left[\left|\frac{\frac{R_{N(t'')}}{N(t'')} - E[Y]}{E[Y]}\right| \geq \varepsilon\right] =$$

$$Pr\left[\left|\frac{\frac{R_{N(t'')}}{N(t'')} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ and } N(t'') < k\right] + \qquad (10.11)$$

$$Pr\left[\left|\frac{\frac{R_{N(t'')}}{N(t'')} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ and } N(t'') \geq k\right] \tag{10.12}$$

We will compute upper bounds on formula (10.11) and (10.12) separately.

Upper bound on formula (10.11)

$$Pr\left[\left|\frac{\frac{R_{N(t'')}}{N(t'')} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ and } N(t'') < k\right] \leq Pr[N(t'') < k] =$$

$$Pr[S_k > t''] = Pr\left[\frac{S_k}{k} > \frac{t'(1+\beta)}{k}\right] \leq$$

$$Pr\left[\left|\frac{\frac{S_k}{k} - \frac{t'}{k}}{\frac{t'}{k}}\right| > \beta\right] = Pr\left[\left|\frac{\frac{S_k}{k} - m\mu}{m\mu}\right| > \beta\right]$$

Let $\varepsilon' = \beta$, $\delta' = \frac{2\delta\varepsilon^2}{d\beta^2}$. Since $k = \frac{d}{\mu\delta\varepsilon^2} = \left\lceil\frac{2}{\mu}\right\rceil \frac{1}{\delta'\varepsilon'^2}$, we use Chebyshev's inequality to conclude that an upper bound on formula (10.11) is

$$\delta' = \frac{2\,\delta\,\varepsilon^2}{d\,\beta^2} \tag{10.13}$$

Upper bound on formula (10.12)

$$Pr\left[\left|\frac{\frac{R_{N(t'')}}{N(t'')} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ and } N(t'') \geq k\right] \leq$$

$$Pr\left[\left|\frac{\frac{R_r}{r} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ for some } r \geq k \text{ and } N(t'') \geq k\right] \leq$$

$$Pr\left[\left|\frac{\frac{R_r}{r} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ for some } r \geq k\right] \tag{10.14}$$

We will use Kolmogorov's inequality to derive an upper bound on formula (10.14). We first state Kolmogorov's inequality [4] and then manipulate the inequality until it is in a form which is useful to derive an upper bound on formula (10.14).

Kolmogorov's Inequality

Let $Y_1, Y_2, \ldots, Y_n$ be independent random variables with the same distribution as Y such that $E[Y]$ and $\sigma^2[Y] = E[Y^2] - E[Y]^2$ are finite, and let $R_l = \sum_{i=1}^{l} Y_i$. For every $x > 0$,

$$Pr[\exists\, l \mid 1 \le l \le n \;\&\; |R_l - l \cdot E[Y]| > x\sqrt{n}\;\sigma[Y]] < \frac{1}{x^2} \qquad (10.15)$$

Substituting $\frac{\varepsilon n}{\sqrt{n}\;\sigma[Y]} E[Y]$ for x yields

$$Pr[\exists\, l \mid 1 \le l \le n \;\&\; |R_l - l \cdot E[Y]| > \varepsilon \cdot n \cdot E[Y]] < \frac{\sigma^2[Y]}{\varepsilon^2 \cdot n \cdot E[Y]^2}$$

$$\le \frac{\bar{c}}{\mu \varepsilon^2 n}. \quad \left[\text{Since } \frac{\sigma^2[Y]}{E[Y]^2} < \vartheta \;(\text{or } \vartheta') \le \frac{\bar{c}}{\mu}\right].$$

Once again, this can be rewritten as

$$Pr\left[\exists\, l \mid 1 \le l \le n \;\&\; \left|\frac{\frac{R_l}{l} - E[Y]}{E[Y]}\right| > \frac{\varepsilon \cdot n}{l}\right] < \frac{\bar{c}}{\mu \cdot \varepsilon^2 \cdot n}$$

Thus,

$$\frac{\bar{c}}{\mu \cdot \varepsilon^2 \cdot n \cdot 2^i} \ge Pr\left[\exists\, l \mid 1 \le l < n \cdot 2^i \;\&\; \left|\frac{\frac{R_l}{l} - E[Y]}{E[Y]}\right| > \frac{\varepsilon \cdot n \cdot 2^i}{l}\right] \ge$$

$$Pr\left[\exists\, l \mid n \cdot 2^{i-1} \le l < n \cdot 2^i \;\&\; \left|\frac{\frac{R_l}{l} - E[Y]}{E[Y]}\right| > \frac{\varepsilon \cdot n \cdot 2^i}{l}\right] \ge$$

$$Pr\left[\exists\, l \mid n \cdot 2^{i-1} \le l < n \cdot 2^i \;\&\; \left|\frac{\frac{R_l}{l} - E[Y]}{E[Y]}\right| > 2\varepsilon\right]$$

An upper bound on formula (10.14) can be derived as follows.

$$Pr\left[\left|\frac{\frac{R_\tau}{\tau} - E[Y]}{E[Y]}\right| \geq \varepsilon \text{ for some } \tau \geq k\right]$$

$$\leq \sum_{i=1}^{\infty} Pr\left[\exists\, l \mid k \cdot 2^{i-1} \leq l < k \cdot 2^{i} \;\&\; \left|\frac{\frac{R_l}{l} - E[Y]}{E[Y]}\right| \geq 2 \cdot \left[\frac{\varepsilon}{2}\right]\right]$$

$$\leq \frac{\bar{c}}{\mu \cdot \left[\frac{\varepsilon}{2}\right]^2 \cdot k} \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{4 \cdot \bar{c}}{\mu \cdot \varepsilon^2 \cdot k} = \frac{4 \cdot \bar{c} \cdot \delta}{d}$$

Thus, formulas (10.11) and (10.12) sum to less than

$$\frac{\delta}{d}\left[4 \cdot \bar{c} + \frac{2\varepsilon^2}{\beta^2}\right] \tag{10.16}$$

Using stopping time $t'' = \frac{d \cdot m}{\delta \cdot \varepsilon^2}(1 + \beta)$, we achieve an $\left[\varepsilon, \frac{\delta}{d}\left[4 \cdot \bar{c} + \frac{2\varepsilon^2}{\beta^2}\right]\right]$ algorithm. If we substitute $\delta' = \delta \cdot \left[\frac{d}{4 \cdot \bar{c} + 2\frac{\varepsilon^2}{\beta^2}}\right]$ for δ we achieve an (ε, δ) algorithm where now the stopping time is $\frac{d \cdot m}{\delta' \cdot \varepsilon^2}(1 + \beta) = \frac{m}{\delta \cdot \varepsilon^2}\left[4 \cdot \bar{c} + 2 \cdot \frac{\varepsilon^2}{\beta^2}\right](1 + \beta)$.

The value of β which minimizes this stopping time when ε is small is $\beta = \frac{\varepsilon^{\frac{2}{3}}}{\bar{c}}$.

Substituting this value for β yields

$$\frac{4\, m\, \bar{c}}{\delta\, \varepsilon^2}\left[1 + \frac{\varepsilon^{\frac{2}{3}}}{2\bar{c}^{\frac{1}{3}}}\right]\left[1 + \frac{\varepsilon^{\frac{2}{3}}}{\bar{c}^{\frac{1}{3}}}\right].$$

This completes the proof.

The total running time of the linear time coverage algorithm is then $O(m \cdot n)$ for preprocessing plus $O\left[m \cdot \frac{n}{\delta\, \varepsilon^2}\right]$ time to execute (*) in Step 3

$O\left[\frac{m}{\delta \varepsilon^2}\right]$ times, which guarantees an (ε, δ) algorithm for the linear time coverage algorithm.

11. Two-Terminal Network Reliability Coverage Algorithm

In this section we show how the two-terminal network reliability problem can be attacked using a variant of the algorithm of Section 7. We give a fast way to compute α when the input to the algorithm is a list of the edge failure probabilities together with an adjacency list for the graph. The preprocessing step in this case consists of listing all the $x-y$ cuts in the graph [2].

The first two steps of one trial of this algorithm, picking block (s, k), are exactly the same as they are for all the previously described coverage algorithms. We will use the cutoff method described in Section 9 to compute $\alpha(s, k)$. Let c be the value of the cutoff. The adjacency list representation is used to list cut sets occurring among failing edges in state s. The algorithm lists cut sets occurring among failing edges in state s until either c cuts are found $(cov(s) \geq c)$ or until all $cov(s)$ cut sets occurring among failing edges in state s are found. $\alpha(s, k) = \frac{1}{\min\{c, cov(s)\}}$ if k is the index of one of the failure sets, otherwise $\alpha(s, k) = 0$.

The time for listing each cut set is $O(n)$ [2], thus the time to compute $\alpha(s, k)$ is

$$\min\{c, cov(s)\} \cdot n . \tag{11.1}$$

The average time to compute α for one trial of the algorithm is

$$\frac{\left[\sum_{1 \leq i < c} i^2 r_i + c \sum_{i \geq c} i r_i\right]}{\sum_{i \geq 1} i r_i} \cdot n . \tag{11.2}$$

When $c = 1$, the time per trial is $O(n)$. An upper bound on the number of trials sufficient to guarantee an (ε, δ) algorithm is

$$\left(\frac{\sum_{k=1}^{m} Pr[F_k]}{Pr[F]} - 1 \right) \cdot \frac{1}{\delta \varepsilon^2} \leq \frac{m}{\delta \varepsilon^2}$$

as we saw in formula (6.1). Thus, an upper bound on the running time of $O\left[\frac{m \cdot n}{\delta \varepsilon^2}\right]$ is obtained if the number of trials is $\frac{m}{\delta \varepsilon^2}$ and $c = 1$.

In the next section we show that

$$\frac{\sum_{k=1}^{m} Pr[F_k]}{Pr[F]}$$

is less than or equal to $\prod_{i=1}^{n} (1 + p_i)$. Thus, if we execute

$$\frac{\left[\prod_{i=1}^{n} (1 + p_i) - 1 \right]}{\delta \varepsilon^2}$$

trials, we have an (ε, δ) algorithm. If we let $c = 1$, the total running time is

$$O\left[m \cdot n + \frac{\left[\prod_{i=1}^{n} (1 + p_i) - 1 \right] \cdot n}{\delta \varepsilon^2} \right].$$

12. An Upper Bound on the Number of Trails Necessary to Achieve an (ε, δ) Algorithm when the System is Monotonic

In this section we show for monotonic n-component systems

$$\frac{\sum_{k=1}^{m} Pr[F_k]}{Pr\left[\bigcup_{k=1}^{m} F_k\right]} \leq \prod_{i=1}^{n} (1 + p_i) . \tag{12.1}$$

Thus, the number of trials sufficient to guarantee an (ε, δ) algorithm for the coverage algorithms is $\dfrac{\left\lceil \prod_{i=1}^{n} (1 + p_i) - 1 \right\rceil}{\delta \cdot \varepsilon^2}$ for monotonic n-component systems. Note that

$$\prod_{i=1}^{n} (1 + p_i) \leq e^{\sum_{i=1}^{n} p_i} .$$

Thus, as $\sum_{i=1}^{n} p_i$ goes to zero, the number of trials necessary also goes to zero. In marked contrast, since $Pr[F]$ goes to zero as $\sum_{i=1}^{n} p_i$ goes to zero, the number of trials necessary for the straight simulation method becomes unbounded as $\sum_{i=1}^{n} p_i$ goes to zero.

Note that $\prod_{i=1}^{n} (1 + p_i)$ can be computed before any trials are performed. This calculation gives an a priori upper bound on the number of trials necessary. Thus upper bound suggests that the coverage algorithms work especially well when the failure probabilities are small. Reliability problems tend to have small failure probabilities associated with their components. These observations indicate that the coverage algorithms are well suited for solving problems that occur in practice.

The proof of equation (12.1) will be by induction. Let n-tuple $(b_1,\ldots,b_i,*,\ldots,*)$ be a specification of system states where each b_j is either zero or one. Let $D(b_1,\ldots,b_i,*,\ldots,*) = Pr[F \mid (b_1,\ldots,b_i,*,\ldots,*)]$, where F is the set of failure states.

Since $(*,\ldots,*)$ is the set of all system states, $D(*,\ldots,*) = Pr[F]$ which is equal to the denominator of the left-hand side of equation (12.1). Note that

$$D(b_1,\ldots,b_n) = \begin{cases} 1 \text{ if } (b_1,\ldots,b_n) \in F \\ 0 \text{ otherwise} \end{cases}.$$

D can be defined inductively as

$$D(b_1,\ldots,b_{i-1},*,\ldots,*) = p_i D(b_1,\ldots,b_{i-1},0,*,\ldots,*) + (1-p_i) D(b_1,\ldots,b_{i-1},1,*,\ldots,*).$$

If the n-component system has the monotonic property,

$$D(b_1,\ldots,b_{i-1},0,*,\ldots,*) \geq D(b_1,\ldots,b_{i-1},1,*,\ldots,*).$$

Let

$$N(b_1,\ldots,b_n) = \begin{cases} 1 \text{ if } (b_1,\ldots,b_n) \in F \\ 0 \text{ otherwise} \end{cases}.$$

Define N inductively as

$$N(b_1,\ldots,b_{i-1},*,\ldots,*) = p_i \cdot N(b_1,\ldots,b_{i-1},0,*,\ldots,*) + N(b_1,\ldots,b_{i-1},1,*,\ldots,*).$$

We claim $N(*,\ldots,*)$ is greater than $\sum_{k=1}^{m} Pr[F_k]$. This can be seen by observing the the contribution of failure state $(b_1,\ldots,b_n)$ to $N(*,\ldots,*)$ is equal to

$$\prod_{i=1}^{n} p_i^{c'_i} \text{ where } c'_i = \begin{cases} 1 \text{ if } b_i = 0 \\ 0 \text{ if } b_i = 1 \end{cases}.$$

Thus, the contribution of all failure states to $N(*,\ldots,*)$ is greater than the contribution of all minimal failure states to $N(*,\ldots,*)$ which is equal to the sum of the probabilities of all the failure sets.

We will show by an induction argument that

$$\frac{N(*,\ldots,*)}{D(*,\ldots,*)} \leq \prod_{i=1}^{n} (1 + p_i)$$

which will validate equation (12.1). The induction hypothesis is

$$N(b_1,\ldots,b_i,*,\ldots,*) \leq \prod_{j=i+1}^{n} (1 + p_j)\, D(b_1,\ldots,b_i,*,\ldots,*)$$

for all combinations of 0's and 1's substituted for $(b_1,\ldots,b_i)$. The basis of the induction argument is that for all system states $(b_1,\ldots,b_n)$, $N(b_1,\ldots,b_n) = D(b_1,\ldots,b_n)$ by definition of N and D. We will assume the induction hypothesis for i and show this implies it is true for $i - 1$.

$$N(b_1,\ldots,b_{i-1},*,\ldots,*) = p_i \cdot N(b_1,\ldots,b_{i-1},0,*,\ldots,*) + N(b_1,\ldots,b_{i-1},1,*,\ldots,*)$$

$$\leq \prod_{j=i+1}^{n} (1 + p_j)\,[p_i\, D(b_1,\ldots,b_{i-1},0,\ldots,\ldots,*) + D(b_1,\ldots,b_{i-1},1,\ldots,\ldots,*)] =$$

$$\prod_{j=i+1}^{n} (1+p_j) \left[\frac{p_i\, D(b_1,\ldots,b_{i-1},0,*,\ldots,*) + D(b_1,\ldots,b_{i-1},1,*,\ldots,*)}{p_i\, D(b_1,\ldots,b_{i-1},0,*,\ldots,*) + (1-p_i)\, D(b_1,\ldots,b_{i-1},1,*,\ldots,*)} \right] \quad (12.2)$$

$$\cdot\, D(b_1,\ldots,b_{i-1},*,\ldots,*)\,.$$

Formula (12.2) is maximized when $D(b_1,\ldots,b_{i-1},1,*,\ldots,*)$ is maximum, but since $D(b_1,\ldots,b_{i-1},1,*,\ldots,*) \leq D(b_1,\ldots,b_{i-1},0,*,\ldots,*)$, this implies (12.2) is maximized when $D(b_1,\ldots,b_{i-1},1,*,\ldots,*) = D(b_1,\ldots,b_{i-1},0,*,\ldots,*)$. Substituting this value into (12.2) yields the conclusion

$$N(b_1,\ldots,b_{i-1},*,\ldots,*) \leq \prod_{j=i}^{n} (1 + p_j) \cdot D(b_1,\ldots,b_{i-1},*,\ldots,*)\,.$$

Thus equation (12.1) is verified.

13. Deterministic Upper and Lower Bounds on $Pr[F]$, An Extension of Boole's Inequality

Boole's inequality states that

$$Pr\left[\bigcup_{k=1}^{m} F_k\right] \leq \sum_{k=1}^{m} Pr[F_k]. \tag{13.1}$$

We provide a lower bound on $Pr\left[\bigcup_{k=1}^{m} F_k\right]$, and show that this lower bound can be computed quickly. First, we present Boole's inequality in another form. We defined r_i in section 6 as

$$r_i = \sum_{s \text{ a state s.t. } cov(s)=i} Pr[s] = Pr[\, cov(s) = i \,]$$

Now we define a positive integer valued random variable Z such that $Pr[\, Z = i \,] = r_i$. Let I be the event $Z \geq 1$. Then

$$E[Z] = \sum_{i=0}^{m} i \cdot Pr[\, Z = i \,] = \sum_{i=0}^{m} i \cdot r_i = \sum_{k=1}^{m} Pr[F_k]$$

and

$$Pr[I] = \sum_{i=1}^{m} Pr[\, Z = i \,] = \sum_{i=1}^{m} r_i = Pr\left[\bigcup_{i=1}^{m} F_k\right]$$

Thus Boole's inequality can be rewritten as

$$Pr[I] \leq E[Z]. \tag{13.2}$$

which is a special case of Markov's inequality. Now we develop a lower bound on $Pr[I]$.

$$E[Z^2] = \sum_{i=1}^{m} i^2 \cdot Pr[\, Z = i \,] = \sum_{i=1}^{m} i^2 \cdot r_i = \sum_{k=1}^{m} \sum_{j=1}^{m} Pr\left[F_k \cap F_j\right]$$

We claim

Theorem :

$$\frac{(E[Z])^2}{E[Z^2]} \leq Pr[I] \tag{13.3}$$

or alternatively,

$$\frac{\left[\sum_{k=1}^{m} Pr[\,F_k\,]\right]^2}{\sum_{k=1}^{m}\sum_{j=1}^{m} Pr\left[F_k \cap F_j\right]} \leq Pr\left[\bigcup_{k=1}^{m} F_k\right] \qquad (13.4)$$

Proof :

This is equivalent to proving

$$\frac{\left[\sum_{i=1}^{m} i\cdot\tau_i\right]^2}{\sum_{i=1}^{m} i^2\cdot\tau_i} \leq \sum_{i=1}^{m}\tau_i$$

which is true if and only if

$$\left[\sum_{i=1}^{m} i\cdot\tau_i\right]\cdot\left[\sum_{j=1}^{m} j\cdot\tau_j\right] \leq \left[\sum_{i=1}^{m}\tau_i\right]\cdot\left[\sum_{j=1}^{m} j^2\cdot\tau_j\right]$$

Compare terms $i = l$, $j = k$ and $i = k$, $j = l$ from the left-hand side and right-hand side of the equation.

Left-hand side : $l\cdot\tau_l\cdot k\cdot\tau_k + k\cdot\tau_k\cdot l\cdot\tau_l = 2\cdot(k\cdot l)\cdot\tau_k\cdot\tau_l$.

Right-hand side : $\tau_l\cdot k^2\cdot\tau_k + \tau_k\cdot l^2\cdot\tau_l = (k^2 + l^2)\cdot\tau_k\cdot\tau_l$.

It is sufficient to show $2\cdot(k\cdot l) \leq k^2 + l^2$. But this follows since $(k - l)^2 \geq 0$. This completes the proof.

Notice that from the list of failure sets we can compute

$$\sum_{k=1}^{m}\sum_{j=1}^{m} Pr\left[F_k \cap F_j\right]$$

in time $O(m^2\cdot n)$. We now consider how good these bounds are by taking the ratio of the upper bound and the lower bound. This ratio is equal to

$$\frac{E[\,Z^2\,]}{E[\,Z\,]} = \frac{\sum_{i=1}^{m} i^2\cdot\tau_i}{\sum_{i=1}^{m} i\cdot\tau_i}$$

The best a priori upper bound on this ratio is m, but in practice we expect this ratio to be much smaller than m.

Inequality (13.3) can be rewritten as $\frac{E[Z^2]}{E[Z]} \geq \frac{E[Z]}{Pr[I]} = \frac{\sum_{i=1}^{m} Pr[F_i]}{Pr[F]}$.

Thus for the algorithms presented in Sections 7 and 11, $\frac{E[Z^2]}{E[Z]\cdot\delta\cdot\varepsilon^2}$ is an easily computable upper bound on the number of trials sufficient to guarantee an (ε, δ) algorithm.

14. A Computational Example

The new coverage algorithms was applied to the network reliability problem used as an example in Section 2. The failure probability of the system is .21254, there are nine failure sets,

$$\sum_{k=1}^{9} Pr[F_k] = .2644$$

and

$$\frac{\left[\sum_{k=1}^{9} Pr[F_k]\right]^2}{\sum_{k=1}^{9}\sum_{j=1}^{9} Pr\left[F_k \cap F_j\right]} = \frac{.2644^2}{.3953} = .1768$$

Four different versions of the coverage algorithm are used to estimate $Pr[F]$. Since randomly selecting block (s, k) in the first two steps of a trial is the same for all versions of the algorithm, all four versions use the same randomly selected block (s, k) on the same trial.

The four different versions of the coverage algorithm differ only in the computation of $\alpha(s, k)$. The method used to compute $\alpha(s, k)$ for each of the four versions is described in the following table.

Version	Computation of $\alpha(s, k)$
1	$\alpha(s, k) = \frac{1}{cov(s)}$
2	$i = \min_{j=1,\ldots,m} \{j \mid s \in F_j\}$ $\alpha(s, k) = \begin{cases} 1 \text{ if } i = k \\ 0 \text{ if } i \neq k \end{cases}$
3	randomly select F_i with probability $\frac{1}{m}$ until $s \in F_i$. Let l = number of failure sets picked until $s \in F_i$ $\alpha(s, k) = \frac{l}{m}$
4	let F_i be failure set s.t. $s \in F_i$ found using version 3 $\alpha(s, k) = \begin{cases} 1 \text{ if } i = k \\ 0 \text{ if } i \neq k \end{cases}$

Forty trials were conducted, with the following results. When Version 1 was used, the estimate of $Pr[F]$ determined by these forty trials is .2181. The estimates derived by treating each of the first, second, third and fourth sets of ten trials as though it were the entire sample are .2005, .2291, .2181 and .2247, respectively.

When Version 2 was used, the estimate of $Pr[F]$ determined by these forty trials is .2247. The estimates derived by treating each of the first, second, third and fourth sets of ten trials as though it were the entire sample are .2115, .2115, .2300 and .2380, respectively.

When Version 3 was used, the estimate of $Pr[F]$ determined by these forty trials is .2078. The estimates derived by treating each of the first, second, third and fourth sets of ten trials as though it were the entire sample are .1410, .2262, .1410 and .3232, respectively.

When Version 4 was used, the estimate of $Pr[F]$ determined by these forty trials is .2115. The estimates derived by treating each of the first, second, third and fourth sets of ten trials as though it were the entire sample are .2380, .2380,

.1851 and .1851, respectively.

A detailed table of these results follows.

k	system state $s \in F_k$	failure sets containing s	$\alpha(s,k)$ version 1	2	3	4	selected index of F_i s.t. $s \in F_i$
6	00100010	3,6,9	1/3	0	2/9	1	3
5	10000111	5,8	1/2	1	7/9	1	5
5	00000011	5,6,8,9	1/4	1	1/9	1	5
4	11011010	4	1	1	11/9	1	4
9	00001011	8,9	1/2	0	2/9	0	8
5	11010111	5	1	1	1/9	1	5
9	00111010	9	1	1	2/9	1	9
9	00011011	9	1	1	5/9	1	9
9	00111011	9	1	1	4/9	1	9
9	00111111	9	1	1	13/9	1	9
3	01101010	3	1	1	14/9	1	3
9	00100000	1,2,3,6,7,9	1/6	0	1/9	0	3
9	00110011	9	1	1	1/9	1	9
5	11010111	5	1	1	2/9	1	5
1	10011100	1	1	1	15/9	1	1
9	00110111	9	1	1	4/9	1	9
5	10010001	2,5	1/2	0	2/9	1	5
9	00111011	9	1	1	3/9	1	9
5	11000011	5	1	1	8/9	1	5
5	10010011	5	1	1	18/9	1	5
2	10000001	2,5,7,8	1/4	1	7/9	0	7
8	10000011	5,8	1/2	0	5/9	0	5
5	11000011	5	1	1	2/9	1	5
9	00011111	9	1	1	10/9	1	9
5	11010101	5	1	1	1/9	1	5
8	10001111	8	1	1	14/9	1	8
4	11000010	4,5	1/2	1	1/9	0	5
8	10001011	8	1	1	1/9	1	8
5	10010011	5	1	1	2/9	1	5
9	00111111	9	1	1	5/9	1	9
8	10001011	8	1	1	4/9	1	8
9	00101011	9	1	1	14/9	1	9
5	11000011	5	1	1	5/9	1	5
4	10011010	4	1	1	24/9	1	4
8	10000110	5,8	1/2	0	3/9	0	5
5	10000111	5,8	1/2	1	8/9	0	8
5	11010111	5	1	1	12/9	1	5
7	10001001	7,8	1/2	1	8/9	0	8
8	10001111	8	1	1	4/9	1	8
9	00011111	9	1	1	30/9	1	9

15. Conclusion

We have presented several highly effective Monte-Carlo methods for estimating the failure probability of an n-component system, given a list of failure sets. In many practical situations an n-component system is presented as a network or fault tree, and the failure sets are too numerous to be explicitly listed. In future work we will show that our coverage formula and the associated Monte Carlo method can sometimes be applied in such situations using implicit methods of sampling from among the failure sets of the system.

16. Acknowledgements

The authors would like to thank J. Pitman for pointing out the approach to proving that the Linear Time Coverage Algorithm is an (ε, δ) algorithm using Kolmogorov's Inequality.

17. Nomenclature

The nomenclature introduced in each section is listed here for easy reference.

Section 1 n - number of components in the system

m - number of failure sets

p_i - probability component i is failing

$(b_1,\ldots,b_n)$ - a specification of a system state

$b_i = 0$ if component i is failing

$b_i = 1$ if component i is working

$(c_1,\ldots,c_n)$ - a failure set specification

$c_i = 0$ if component i is failing

$c_i = 1$ if component i is working

$c_i = *$ if component i may be either failing or working

F_k - the k^{th} failure set

s - a system state

F - the set of all failure states

$Pr[F]$ - the probability the network is in a failing state

Section 2 x, y - the two designated nodes in the two terminal problem

Section 3 E - enclosing region

$A(E)$ - area of region E

U - unknown region

$A(U)$ - area of region U

b - number of blocks into which E is subdivided

a_i - area of block i in E

α_i - 1 if block $i \in U$, 0 if block $i \notin U$.

Y - random variable s.t. $E[Y] = A(U)$

Section 4 N - number of trials performed during the course of a Monte-Carlo algorithm

Y_i - estimator produced by the i^{th} trial

μ - $E[Y_i]$

σ^2 - variance of Y_i

ε - allowable relative error

δ - confidence level that the relative error is $\leq \varepsilon$

(ε, δ) - algorithm with confidence level δ the relative error of the estimator produced by the algorithm is less than or equal to ε.

Section 6 $cov(s)$ - the number of failure sets F_k s.t. $s \in F_k$

(s, k) - a block in the region E for the coverage algorithms $s \in F_k$

$\alpha(s, k)$ - indicates whether or not $(s, k) \in U$

$\alpha(s\ k) = 1$ if $(s, k) \in U$

$\alpha(s\ k) = 0$ if $(s, k) \notin U$

Section 7 FS - an array of size m used to pick failure sets

r_i - summation of the probability of all states with coverage i

Section 8 $\alpha(s, k)$ - generalized version of definition given in Section 8

$$\sum_{\{k \mid s \in F_k\}} E[\alpha(s, k)] = 1$$

Section 9 c - the cutoff

Section 10 Y - unbiased estimator of $Pr[F]$

Y' - a second unbiased estimator of $Pr[F]$

$\alpha(s, k)$ - used to produce estimator Y

$\alpha'(s, k)$ - used to produce estimator Y'

$X(s, k)$ - length of trial given that block (s, k) is selected

$$\vartheta - \frac{E[Y^2]}{E[Y]^2}$$

$$\vartheta' - \frac{E[Y'^2]}{E[Y']^2}$$

$$\mu - \frac{Pr[F]}{\sum_{k=1}^{m} Pr[F_k]}$$

t - number of times (*) in Step 3 of algorithm is executed during the course of the algorithm.

X_j - length of the j^{th} trial

S_n - $S_n = \sum_{i=1}^{n} X_i$

Y_j - value of Y on the j^{th} trial

Y_j' - value of Y' on the j^{th} trial

R_n - $\sum_{i=1}^{n} Y_i$

R_n' - $\sum_{i=1}^{n} Y_i'$

$N(t)$ - number of trials completed after (*) in Step 3 is executed t times

$\bar{c}$ - both ϑ and ϑ' are $\leq \frac{\bar{c}}{u}$

18. Linear Time Coverage Algorithm Pascal Program

In the following pages there is a listing of a Pascal program for the Linear Time Coverage Algorithm described in Section 10. After the problem data is input and the preprocessing is performed, the upper bound and the lower bound on $Pr[\ F\ ]$ described in Section 13 is computed and output. Following this are steps 1 thru 6 of the Linear Time Coverage Algorithm. Following this listing is a run of the program using as input data the example problem described in Section 2.

```
program montel (input,output) ;

{ this program computes the failure probability of an n-component system
  where the input is a list of failure sets -
  the running time of this algorithm is c * numcomponents * numfail
  divided by delta * epsilon * epsilon, where
  c is a small constant  - (see the proof that this algorithm is an
                            epsilon,delta algorithm)
  numcomponents is  the number of system components
  numfail is the number of specified failure sets
  delta is the confidence level
  epsilon is the allowable relative error }

  label 1 ;
  var i,j,k,l : integer ;
      numcomponents : integer ;
      numfail : integer ;
      Seed : integer ;
      prob : array [1..40] of real ;
      failset : array [1..100,1..40] of integer ;
      failprob : array [0..120] of real ;
      sumprob : real ;
      sstate : array [1..40] of integer ;
      numsteps : integer ;
      sumest1 : real ;
      sumest2 : real ;
      outest1 : real ;
      outest2 : real ;
      numtrials : integer ;
      c : real ;
      x : real ;
      z : real ;
      epsilon : real ;
      delta : real ;
      time : integer ;
      alpha1 : real ;
      alpha2 : real ;
      found : boolean ;

{select failure set }

function selectfail : integer ;
var low,high,pntl,pnth : integer ;
    x : real ;
    found : boolean ;
begin
   x := random (Seed) ;
   found := false ;
   low := 0 ;
   high := numfail ;
   while (not found) do
   begin
      pntl := (low + high) div 2 ;
      pnth := pntl + 1 ;
      if (failprob[pntl] >= x) then high := pntl ;
      if (failprob[pnth] < x) then low := pnth ;
      if ((failprob[pntl] <=x) and (failprob[pnth] >= x)) then found := true
   end ;
   selectfail := pnth
end ;

{ select system state }

procedure selectstate ;
   var i : integer ;
       x : real ;
   begin
        for i := 1 to numcomponents do
        begin
             sstate[i] := failset[k,i] ;
             if (sstate[i] = -1) then
             begin
                  x := random (Seed) ;
                  if (x <= prob[i]) then sstate[i] := 0
                  else sstate[i] := 1
             end
        end
   end ;

{ function to see if system state is in failure set }

function inset : boolean ;
var indic : boolean ;
    j : integer ;
begin
   indic := true ;
```

```
    for j := 1 to numcomponents do
    begin
        if ((failset[i,j] = 1) and (sstate[j] = 0)) then
            indic := false ;
        if ((failset[i,j] = 0) and (sstate[j] = 1)) then
            indic := false
    end ;
    inset := indic
end ;

{ input problem data }

  begin
     writeln ('Seed:') ;
     read (Seed) ;
     i := seed (Seed) ;
     writeln ('enter epsilon :') ;
     read (epsilon) ;
     writeln ('enter delta :') ;
     read (delta) ;
     writeln ('enter constant :') ;
     read (c) ;
     writeln ('number of components :') ;
     read (numcomponents) ;
     writeln ('input probability of components : ') ;
     for i := 1 to numcomponents do
     begin
         writeln ('input prob. of component ',i:3) ;
         read (prob[i])
     end ;
     writeln ('input number  of failure sets :') ;
     read (numfail) ;
     writeln ;
     writeln ('The failure set specification format is :') ;
     writeln ('1 - component must work for system state to be in failure set') ;
     writeln ('0 - component must fail for system state to be in failure set') ;
     writeln ('-1 - component may either work or fail (unspecified)') ;
     writeln ;
     for i := 1 to numfail do
     begin
        writeln ('input specifications for failure set ',i:3) ;
        for j := 1 to numcomponents do
           read (failset[i,j])
     end ;

{ preprocessing }

     failprob[0] := 0.0 ;
     sumprob := 0 ;
     for i := 1 to numfail do
     begin
         failprob[i] := 1.0 ;
         for j := 1 to numcomponents do
         begin
             if (failset[i,j] = 0) then
                 failprob[i] := failprob[i] * prob[j] ;
             if (failset[i,j] = 1) then
                 failprob[i] := failprob[i] * (1.0 - prob[j])
         end ;
         sumprob := sumprob + failprob[i]
     end ;
     writeln ;


{ Print upper bound on the failure probability }

     writeln ('Upper bound on the failure probability is ',sumprob:12:8) ;

{ Compute and print lower bound on the failure probability }

     z := 0.0 ;
     for i := 1 to numfail do
         for j := 1 to numfail do
         begin
             x := 1.0 ;
             for k := 1 to numcomponents  do
             begin
                 if ((failset[i,k]=1) or (failset[j,k]=1)) then
                     x := x * (1.0 - prob[k]) ;
                 if ((failset[i,k]=0) or (failset[j,k]=0)) then
                     x := x * prob[k] ;
                 if ((failset[i,k] <> failset[j,k]) and (failset[i,k] <> -1)
                     and (failset[j,k] <> -1)) then x := 0
             end ;
             z := z + x
         end ;
```

```
    writeln ; writeln (' E(Z*Z) = ',z:12:8) ;  writeln ;
    z := (sumprob * sumprob) / z ;
    writeln ;
    writeln ('Lower bound on the failure probability is ',z:12:8) ;
    writeln ;
    x := 0 ;
    for i := 1 to numfail do
    begin
        writeln ('The prob. of failure set ',i:3,' is ',failprob[i]:12:8) ;
        x := failprob[i] + x ;
        failprob[i] := x / sumprob
    end ;
    numsteps := trunc ((c * numfail) / (delta * epsilon * epsilon)) + 1 ;
    writeln ; writeln ('the number of steps will be ',numsteps:6) ;
    time := 0 ;
    numtrials := 0 ;
    sumest1 := 0.0 ;
    sumest2 := 0.0 ;

{ beginning of Monte-Carlo trial }

    while (time <= numsteps) do
    begin

{ step 1 - select failure set }

        k := selectfail ;

{ step 2 - select system state }

        selectstate ;

{ step 3 - compute estimators alpha1 and alpha2 }

        i := 0 ;
        found := false ;
        while (not found) do
        begin
            i := trunc (random(Seed) * numfail) + 1 ;
            if (i >= numfail) then i := numfail ;
            j := j + 1 ;
            time := time + 1 ;
            if (time > numsteps) then goto 1 ;
            found := inset
        end ;

{ step 4 - compute alpha1 and alpha2 }

        alpha1 := (1 / numfail) * sumprob ;
        if (k = i) then alpha2 := sumprob
        else alpha2 := 0 ;

{ step 5 - increment number of trials and total estimators }

        numtrials := numtrials + 1 ;
        sumest1 := sumest1 + alpha1 ;
        sumest2 := sumest2 + alpha2
    end  { while (time <= numsteps) } ;

{ step 6 - the simulation is completed, output the results }

1 : outest1 := sumest1 / numtrials ;
    outest2 := sumest2 / numtrials ;
    writeln ('number of trials completed = ',numtrials:5) ;
    writeln ('estimator 1 = ',outest1:12:8) ;
    writeln ('estimator 2 = ',outest2:12:8)
end  { of program } .
```

```
Seed:
39845
enter epsilon :
0.1
enter delta :
0.05
enter constant :
8.0
number of components :
8
input probability of components :
input prob. of component   1
0.1
input prob. of component   2
0.5
input prob. of component   3
0.4
input prob. of component   4
0.3
input prob. of component   5
0.2
input prob. of component   6
0.4
input prob. of component   7
0.1
input prob. of component   8
0.2
input number  of failure sets :
9

The failure set specification format is :
1 - component must work for system state to be in failure set
0 - component must fail for system state to be in failure set
-1 - component may either work or fail (unspecified)

input specifications for failure set   1
-1 -1 -1 -1 -1 -1  0  0
input specifications for failure set   2
-1 -1 -1 -1  0  0  0 -1
input specifications for failure set   3
 0 -1 -1  0 -1  0 -1  0
input specifications for failure set   4
-1 -1  0 -1 -1  0 -1  0
input specifications for failure set   5
-1 -1  0 -1  0 -1 -1 -1
input specifications for failure set   6
 0 -1 -1  0  0 -1 -1 -1
input specifications for failure set   7
-1  0 -1  0 -1  0  0 -1
input specifications for failure set   8
-1  0  0  0 -1 -1 -1 -1
input specifications for failure set   9
 0  0 -1 -1 -1 -1 -1 -1

Upper bound on the failure probability is   0.26440000

 E(Z*Z) =   0.39534400


Lower bound on the failure probability is   0.17682666

The prob. of failure set   1 is   0.02000000
The prob. of failure set   2 is   0.00800000
The prob. of failure set   3 is   0.00240000
The prob. of failure set   4 is   0.03200000
The prob. of failure set   5 is   0.00800000
The prob. of failure set   6 is   0.00640000
The prob. of failure set   7 is   0.00640000
The prob. of failure set   8 is   0.00800000
The prob. of failure set   9 is   0.05600000

the number of steps will be 144001
number of trials completed = 19864
estimator 1 =   0.21307693
estimator 2 =   0.21215656
```

19. References

[1] J. Scott Provan and Michael O. Ball, *The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected*, working paper MS/S 81-002, Management Science and Statistics, January 1981 (revised April 1981)

[2] S. Tsukiyama, I. Shirakawa, H. Ozaki, H. Ariyoshi, *An Algorithm to Enumerate All Cutsets of a Graph in Linear Time*, JACM, vol. 27, no. 4, October 1980, pp. 619-632

[3] S. Ross, *Applied Probability Models with Optimization Applications*, 1970, chapter 3, section 3.4, p. 37

[4] W. Feller, *An Introduction to Probability Theory and Its Applications*, third edition, vol. 1, 1968, chapter 9, section 7, pp. 234-235

Chapter 3

1. Introduction

In chapter 2 we presented a Monte-Carlo algorithm to estimate the failure probability of an n-component system. We extend this technique to estimate the failure probability for the reachability problem and the planar K–terminal problem. A direct application of the algorithm presented in chapter 2 to the planar K–terminal problem requires an explicit enumeration of the cuts in the graph. Since the number of cuts may be exponential in the size of the graph an explicit enumeration may be very costly. Similarly, direct application of this algorithm to the reachability problem requires an enumeration of the simple paths in the graph. The algorithms presented in this paper avoid such explicit enumerations.

Let G be an undirected connected graph with n nodes and m edges. For each edge e_i a probability p_i is specified. Edge e_i is failing (in a failing state) with probability p_i and working (in a working state) with probability $1 - p_i$ independently of the other edges in the graph. Nodes are always working. G, together with the specified edge probabilities, is called a **network**. We will use the terms network and graph interchangeably when the context is clear.

There are 2^m different combinations of edge states. These combinations are called the states of the graph. Let S be the set of all states of the graph. For all $s \in S$, the probability of s, denoted $Pr[s]$, is the product of the probabilities of the edge states specified in s. We will denote by F, where $F \subseteq S$, the set of **failure states** of the network. We are interested in computing

$Pr[F] = \sum_{s \in F} Pr[s]$, the **failure probability** of the network.

For the reachability problem there are two specified nodes x and y in the graph. A failure state is one in which there is a path of failing edges between x and y. For the planar K–terminal problem the graph G is planar. There are K specified nodes $x_1, \cdots, x_K$ in G. The problem is to compute the probability that there is no path of working edges between some pair of the specified nodes.

We develop Monte-Carlo algorithms for both the reachability problem and the planar K–terminal problem. Two important special cases of the planar K–terminal problem are the planar two-terminal problem (when $K = 2$) and the planar all-terminal problem (when $K = n$). For these special cases we present algorithms which have much simpler implementations than the more general planar K–terminal algorithm.

For the reachability problem we prove that when the edge failure probabilities are small and the node degrees are small the average value of the Monte-Carlo estimator over many trials of the algorithm converges quickly to the failure probability. For the planar K–terminal problem we prove that when the edge failure probabilities are small and the number of edges bordering any face in the graph is small the average value of the estimator over many trials of the algorithm converges quickly to the failure probability.

2. Model for Monte-Carlo Algorithms to Approximate Enumeration Problems

The Monte-Carlo algorithms we develop to estimate the failure probability for the reachability problem and the planar K–terminal problem can be explained abstractly as follows. The algorithm can be logically decomposed into two parts: the **preprocessing** step and the **trial** step. The preprocessing step is executed once at the beginning of the algorithm. The purpose of the preprocessing step is to perform calculations needed for every execution of the trial step.

Let μ denote the failure probability. Each execution of the trial step uses random choices to produce an unbiased estimator of μ. Each trial produces an estimator which is independent of the estimators produced from the other trials. We now give an abstract description of how μ is approximated in the trial step. Let U be a finite set, which we call the **universe**. Let wt (mnemonic for **weight**) be a measure on U (i.e. wt is a function from all subsets of U to the positive reals satisfying the requirement that for all $B \subseteq U$, $wt(B) = \sum_{b \in B} wt(b)$) and let α be a function from U to the reals such that

$$\mu = \sum_{a \in U} wt(a) \cdot \alpha(a) .$$

Let

$$wt(U) = \sum_{a \in U} wt(a)$$

be called the **weight of the universe**. One trial of the algorithm follows.

Trial to Estimate μ

1. Randomly choose $a \in U$ with probability $\frac{wt(a)}{wt(U)}$.
2. Compute $\alpha(a)$
3. Estimator $Y \leftarrow \alpha(a) \cdot wt(U)$

Lemma 2.a: $E[Y] = \mu$

Proof :

$$E[Y] = \sum_{a \in \mathcal{U}} \frac{wt(a)}{wt(U)} \alpha(a) \cdot wt(U) = \sum_{a \in \mathcal{U}} wt(a) \cdot \alpha(a) = \mu$$

■

2.1. Preliminaries for Estimating the Probability of a Union of Sets

In all of the problems described in the following sections we want to compute the probability of some subset F of system states. In each case F can be written as the union of subsets of states, $F = \bigcup F_i$. In this section we develop an algorithm to estimate the probability of a union of sets. This will be the basis for all the algorithms presented subsequently.

Let S be a finite set and let Pr be a probability measure defined on all subsets of S. Let $I = \{1,...,\psi\}$ and let $F = \bigcup_{i \in I} F_i$ where, for all $i \in I$, $F_i \subseteq S$. We are interested in computing $\mu = Pr[F]$. In our applications it is hard to compute μ exactly. Instead, we will be able to generate an unbiased estimator, Y, of μ using the general model discussed previously. We first describe the universe U and the functions wt and α. The **coverage** of $s \in F$ is the set

$$cov(s) = \{i \in I : s \in F_i\} .$$

Let the universe U be the set of all ordered pairs (s,i) such that $s \in F_i$. Then U can be written as

$$U = \{(s,i) : s \in F \text{ and } i \in cov(s)\} = \{(s,i) : i \in I \text{ and } s \in F_i\} .$$

Intuitively, U can be explained as follows. State (s,i) can be thought of as the copy of failure state s for set F_i. Each set F_i is represented in U as if though it did not intersect with any set F_j, where $j \neq i$, $j \in I$. For fixed $i \in I$ let

$$U_i = \{(s,i) : s \in F_i\}$$

U_i can be thought of as the copy of the set F_i in the universe U. Notice that the

universe is partitioned by the U_i sets. For all $(s,i) \in U$ let

$$wt(s,i) = Pr[s].$$

Then,

$$wt(U_i) = Pr[F_i]$$

and

$$wt(U) = \sum_{i \in I} Pr[F_i].$$

Let α be defined such that for all $s \in F$,

$$\sum_{i \in cov(s)} \alpha(s,i) = 1.$$

Intuitively, $\alpha(s,i)$ is the appropriate adjustment factor to the weight function to make the total contribution to μ of all copies of state s equal to $Pr[s]$. Thus,

$$\mu = \sum_{(s,i) \in U} wt(s,i) \cdot \alpha(s,i) = \sum_{s \in F} \sum_{i \in cov(s)} wt(s,i) \cdot \alpha(s,i)$$

$$= \sum_{s \in F} Pr[s] = Pr[F].$$

We now give an abstract description of one trial of the algorithm to estimate $Pr[F]$.

1. Randomly choose $i \in I$ with probability $\dfrac{Pr[F_i]}{\sum_{i \in I} Pr[F_i]}$.
2. Randomly choose $s \in F_i$ with probability $\dfrac{Pr[s]}{Pr[F_i]}$.

   (At this point (s,i) has been chosen with probability $Pr[s] = \dfrac{wt(s,i)}{wt(U)}$)
3. Compute $\alpha(s,i)$.
4. Estimator $Y \leftarrow \alpha(s,i) \cdot wt(U)$.

To develop some intuition for this algorithm, consider the special case when

$\alpha(s,i) = 1$ if i is the smallest index in $cov(s)$, and

$\alpha(s,i) = 0$ if i is not the smallest index in $cov(s)$.

An example of the set $F = \bigcup_{i \in I} F_i$ is shown as a Venn diagram in Figure 2.1.a, the universe U is shown in Figure 2.1.b.

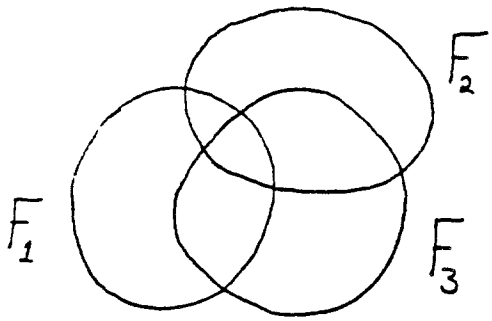


Figure 2.1.a $F = \bigcup F_i$

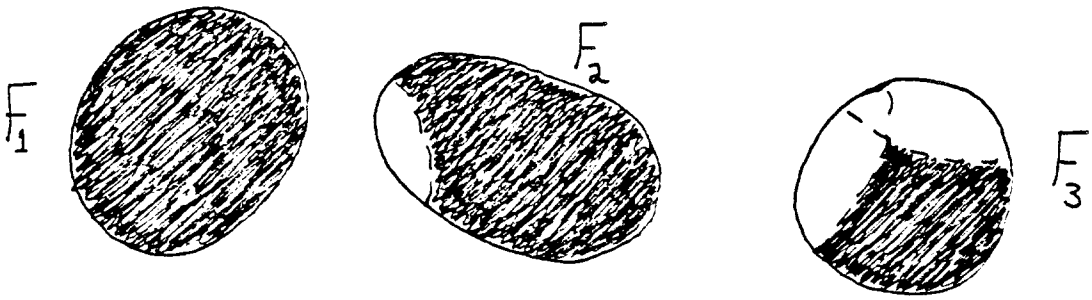


Figure 2.1.b *The universe U, the shaded portion is where $\alpha = 1$.*

The shaded portion in Figure 2.1.b corresponds to the area where $\alpha = 1$. Notice that the area of the shaded portion is $Pr[F]$, whereas the total area of U is $wt(U) = \sum_{i \in I} Pr[F_i]$. In this case α can be thought of as a Bernoulli random variable, where $Pr[\ \alpha=1\ ] = \frac{Pr[F]}{\sum_{i \in I} Pr[F_i]}$. Thus,

$$E[\alpha] = \frac{Pr[F]}{\sum_{i \in I} Pr[F_i]}$$

and

$$E[Y] = E[\alpha] \cdot \sum_{i \in I} Pr[F_i] = Pr[F] .$$

In the applications we consider in this paper it is hard to compute the weight of the universe as the universe is defined here. For example, in the reachability problem, calculating $\sum_{i \in I} Pr[F_i]$ is at least as hard as counting the number of $x-y$ simple paths. Counting the number of $x-y$ simple paths is a #P-complete problem. In the next subsection we give an abstract view of how this difficulty can be circumvented.

2.2. The Coverage Algorithm for Estimating the Probability of a Union of Sets

In all our applications it is hard to directly compute the weight of the universe as it was defined in the previous subsection. In this subsection we show how this problem can be circumvented. We call this algorithm the **coverage algorithm**.

In each of our applications we circumvent the problem of calculating the weight of the universe U by designing a set V, called the **sample set**, which is a superset of U. There is a wt measure defined on V which is an extension of the wt measure defined on U. The sample set V is designed such that $wt(V)$ is easy to calculate and such that it is easy to randomly choose elements in V with probability proportional to their weights.

We now describe the structure of the sample set V. Abstractly, V can be thought of as an extension of the universe U in two ways. Recall that for all $i \in I$, $U_i \subseteq U$ corresponds to F_i, i.e.

$$U_i = \{(s,i) : s \in F_i\} .$$

Let E_i be a superset of F_i and let

$$V_i = \{(t,i) : t \in E_i\} \ .$$

Each set V_i can be thought of as an expansion of the set U_i. The sample set V will contain $\bigcup_{i \in I} V_i$. Consequently, $U \subseteq V$.

Secondly, the sample set is an expansion of U in the following way. Let J be a superset of I. For each $j \in J-I$ we define a set V_j . The V_j sets are defined in such a way that they are mutually disjoint. These sets are also added to the sample set. Thus,

$$V = \bigcup_{j \in J} V_j \ .$$

We extend the wt measure defined on U to all elements in V. The weight of the sample space is

$$wt(V) = \sum_{j \in J} wt(V_j) \ .$$

We extend the α function so that for all $w \in V-U$, $\alpha(w) = 0$. We can now implement one trial of the coverage algorithm as follows.

Trial for Estimating $Pr[F]$

1. Randomly choose $j \in J$ with probability $\dfrac{wt(V_j)}{wt(V)}$
2. **If** $j \notin I$ **then** estimator $Y \leftarrow 0$ and **stop**
   Else $j = i \in I$

   (At this point i has been chosen with probability $\dfrac{wt(V_i)}{wt(V)}$)
3. Randomly choose $t \in E_i$ with probability $\dfrac{wt(t,i)}{wt(V_i)}$
4. **If** $t \notin F_i$ **then** estimator $Y \leftarrow 0$ and **stop**
   Else $t = s \in F_i$

   (At this point $(s,i) \in U$ has been chosen with probability $\dfrac{Pr[s]}{wt(V)}$)
5. Compute $\alpha(s,i)$
6. Estimator $Y \leftarrow \alpha(s,i) \cdot wt(V)$

Lemma 2.2.a : $E[Y] = Pr[F]$

Proof :

$$E[Y] = \sum_{j \in J-I} \frac{wt(V_j)}{wt(V)} \cdot 0 + \sum_{i \in I} \left[\sum_{t \in E_i - F_i} \frac{wt(t,i)}{wt(V)} \cdot 0 + \sum_{s \in F_i} \frac{Pr[s]}{wt(V)} \alpha(s,i) \cdot wt(V) \right]$$

$$= \sum_{s \in F} Pr[s] \cdot \left[\sum_{i \in cov(s)} \alpha(s,i) \right] = Pr[F]$$

■

Suppose α is a zero-one variable. The coverage algorithm can be used to randomly choose $s \in F$ as we now describe. Each $s \in F$ is **associated** with the unique $(s,i) \in U$ such that $\alpha(s,i) = 1$. We say $s \in F$ is **chosen** in a trial if and only if the element of U associated with s is chosen in the trial. If no $s \in F$ is chosen in a trial then we say the trial **fails**. The probability that a particular $s \in F$ is chosen in a trial is $\frac{Pr[s]}{wt(V)}$ and the probability that some $s \in F$ is chosen in a trial is $\frac{Pr[F]}{wt(V)}$. The ability to randomly choose a state $s \in F$ will be important when we discuss the planar K−terminal problem. The coverage algorithm for the planar K−terminal problem will use as a subroutine the coverage algorithm for the planar two-terminal problem. The subroutine will be used to randomly choose a failure state s for an appropriately defined planar two-terminal problem.

2.3. Convergence of the Algorithm to Estimate $Pr[F]$

There are three components to the running time of the algorithm: the preprocessing time, the number of trials and the time per trial. In this subsection we discuss the number of trials executed by the algorithm. Suppose a trial which produces an unbiased estimator of a real number μ is repeated N times. Let Y_i be the value of the estimator obtained from the i^{th} trial. Let

$\tilde{Y} = \frac{(Y_1 + Y_2 + \cdots + Y_N)}{N}$. In this subsection we address the question of how many trials are sufficient to "guarantee" that $\tilde{Y}$ is "close" to μ. First, we must define what we mean by guarantee and close. For this purpose, we introduce two parameters which will be a part of the input to the algorithm, ε and δ.

ε is a positive real number which is the measure of the closeness of $\tilde{Y}$ to μ. There are two natural measures of closeness:

1. $\tilde{Y}$ is close to μ in an **absolute** sense if

$$| \tilde{Y} - \mu | < \varepsilon$$

2. $\tilde{Y}$ is close to μ in a **relative** sense if

$$\left| \frac{\tilde{Y} - \mu}{\mu} \right| < \varepsilon$$

The relative measure is the stronger measure of closeness for our applications because $\mu = Pr[F] \leq 1$. In fact, to say that $\tilde{Y}$ is close to μ in an absolute sense when $\mu \ll 1$ is meaningless. For example, consider the case when we are trying to compare two networks to see which is more likely to fail. Suppose the failure probability of the first network is 10^{-5} and the failure probability of the second network is 10^{-7}. Suppose we choose ε to be 10^{-3}. Suppose the algorithm produces an estimator in both cases of 10^{-4}. For both networks we would say that $\tilde{Y}$ is close to μ in an absolute sense, even though the first network is 100 times more likely to fail then the second network! On the other hand, consider what happens if we use the relative measure of closeness. In this case, for the first network $\tilde{Y}$ must be between $.999 \cdot 10^{-5}$ and $1.001 \cdot 10^{-5}$ to be close to μ and for the second network $\tilde{Y}$ must be between $.999 \cdot 10^{-7}$ and $1.001 \cdot 10^{-7}$ to be close to μ. If the algorithm produces an estimator $\tilde{Y}$ which is close to μ for both networks then it is easy to tell which is the more reliable. In our analysis we will use the relative measure of closeness.

δ is a positive real number which is our measure of confidence that the algorithm produces an estimator $\tilde{Y}$ close to μ. We require that the algorithm output an estimator $\tilde{Y}$ which is close to μ with probability greater than $1-\delta$. Thus, for input parameters ε and δ, we say the algorithm is an (ε,δ) **algorithm** if

$$Pr\left[\left|\frac{\tilde{Y}-\mu}{\mu}\right| < \varepsilon\right] > 1-\delta$$

We now derive an upper bound on the number of trials, N, sufficient to produce an (ε,δ) algorithm based on the estimator presented in the previous subsection. We consider the case when α is a random variable such that $|\alpha| \leq 1$. For all of the algorithms presented in this paper, including those presented in the previous chapter, α fullfills this condition. This theorem is a stronger version of the upper bound on the number of trials derived in chapter 2, section 4.

Theorem 2.3.a : Let $\{Y_i\}$ be a sequence of independently and identically distributed random variables such that $|Y_i| \leq 1$. Let

$$\tilde{Y} = \frac{(Y_1 + Y_2 + \cdots + Y_N)}{N}$$

and let $p = E[Y_i] > 0$. For $\varepsilon \leq 2$, if $N = (\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}\frac{1}{p}$ then

$$Pr\left[\left|\frac{\tilde{Y}-p}{p}\right| < \varepsilon\right] > 1-\delta$$

Proof : This is equivalent to showing

$$Pr\left[\left|\frac{\tilde{Y}-p}{p}\right| \geq \varepsilon\right] \leq \delta$$

We will show that

$$Pr\left[\tilde{Y} \geq (1+\varepsilon)p\right] \leq \frac{\delta}{2} \qquad (2.3.b)$$

and that

$$Pr\left[\, \bar{Y} \le (1-\varepsilon)p \,\right] \le \frac{\delta}{2} \qquad (2.3.c)$$

when $N = (\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}\frac{1}{p}$. The proof uses Bernstein's improvement of Chebyshev's inequality (see [3], chapter VII, section 4). Let

$$M_{-}(\varepsilon) = E\left[\, e^{\varepsilon\cdot N\cdot(\bar{Y}-p)} \,\right]$$

and let

$$M^{-}(\varepsilon) = E\left[\, e^{\varepsilon\cdot N\cdot(p-\bar{Y})} \,\right] .$$

The following two inequalities follow from [3], chapter VII, section 1.

$$Pr\left[\, N\cdot\bar{Y} \ge N\cdot p + \frac{t + \ln M_{-}(\varepsilon)}{\varepsilon} \,\right] \le e^{-t} \qquad (2.3.d)$$

$$Pr\left[\, N\cdot\bar{Y} \le N\cdot p - \frac{t + \ln M_{-}(\varepsilon)}{\varepsilon} \,\right] \le e^{-t} \qquad (2.3.e)$$

Let $\sigma^2 = E[(Y_i - p)^2]$. In [3], chapter VII, section 4, $\ln M_{-}(\varepsilon)$ and $\ln M^{-}(\varepsilon)$ are both bounded above by

$$\frac{\varepsilon^2\cdot N\cdot\sigma^2}{2}\cdot\left[\, 1 + \frac{\varepsilon\cdot e^{\varepsilon}}{3} \,\right] . \qquad (2.3.f)$$

We consider only the case when $\varepsilon \le 1$. Then, $1 + \frac{\varepsilon\cdot e^{\varepsilon}}{3} \le 2$ and

$$\ln M_{-}(\varepsilon) \le \varepsilon^2\cdot N\cdot\sigma^2 .$$

Substituting into (2.3.d) yields

$$Pr\left[\, N\cdot\bar{Y} \ge N\cdot p + \frac{t + \varepsilon^2\cdot N\cdot\sigma^2}{\varepsilon} \,\right] \le e^{-t}$$

or

$$Pr\left[\, \bar{Y} \ge p\left[\, 1 + \frac{t + \varepsilon^2\cdot N\cdot\sigma^2}{N\cdot\varepsilon\cdot p} \,\right] \,\right] \le e^{-t} .$$

Let $t = \varepsilon^2 \cdot N \cdot p$. Then,

$$Pr\left[\tilde{Y} \geq p \cdot \left[1 + \varepsilon + \varepsilon \cdot \left[\frac{\sigma^2}{p}\right]\right]\right] \leq e^{\varepsilon^2 \cdot N \cdot p} .$$

Now, since $|Y_i| \leq 1$, we can bound $\frac{\sigma^2}{p}$ by 1, yielding

$$Pr\left[\tilde{Y} \geq p \cdot \left[1 + 2 \cdot \varepsilon\right]\right] \leq e^{\varepsilon^2 \cdot N \cdot p} .$$

Let $\varepsilon' = 2 \cdot \varepsilon$ (and consequently $\varepsilon' \leq 2$). Then,

$$Pr\left[\tilde{Y} \geq p \cdot (1 + \varepsilon')\right] \leq e^{\frac{\varepsilon'^2 \cdot N \cdot p}{4}} .$$

Thus, to satisfy (2.3.b), we want

$$e^{\frac{\varepsilon'^2 \cdot N \cdot p}{4}} \leq \frac{\delta}{2} .$$

Taking the natural log on both sides of this inequality, we can conclude that if $N = (\ln \frac{2}{\delta}) \cdot \frac{4}{\varepsilon^2} \frac{1}{p}$ then Inequality (2.3.b) holds. A similar calculation using (2.3.e) and (2.3.f) shows that Inequality (2.3.c) holds under the same conditions. Thus, the theorem follows. ∎

Corollary : When α is a random variable such that $|\alpha| \leq 1$ for the algorithm presented in the previous subsection, $N = (\ln \frac{2}{\delta}) \cdot \frac{4}{\varepsilon^2} \frac{1}{E[\alpha]}$ trials is sufficient to produce an (ε,δ) algorithm. ∎

Notice that the number of trials, N, depends upon two quantities : $(\ln \frac{2}{\delta}) \cdot \frac{4}{\varepsilon^2}$ which depends on the the desired relative accuracy of $\tilde{Y}$ and the desired confidence level of obtaining this accuracy; and $\frac{1}{E[\alpha]} = \frac{wt(U)}{Pr[F]}$, which depends on the algorithm design and the problem instance. The goal of the algorithm design is to make the ratio $\frac{wt(U)}{Pr[F]}$ provably small so that the number of trials

sufficient to produce an (ε,δ) algorithm is provably small.

3. Reachability Algorithm

The concepts developed for the reachability algorithm are the foundation for the algorithms to solve the more classical network reliability problems: the planar two-terminal, all-terminal and K-terminal problems. The solution to the reachability problem is primarily of interest for this reason. The reachability problem can be described as follows. Let G be a network and let x and y be two specified nodes in G. The network fails if there is a path of failing edges between x and y. Therefore, state s is a **failure state** if there is a path of failing edges between x and y in s. Let $F \subseteq S$ be the set of all failure states. The problem is to compute $Pr[F]$. Ball and Provan [2] show that computing the exact value of $Pr[F]$ is a #P-complete problem.

3.1. Basic Concepts and Definitions

An $a-b$ **simple path** is a path between nodes a and b which does not repeat edges or nodes. Therefore, state s is a failure state if and only if there is a $x-y$ simple path of failing edges in s. Let SP be the set of all $x-y$ simple paths in G. For all $sp \in SP$, let F_{sp} be the set of all states s such that all the edges in sp are failing in s. Then,

$$F = \bigcup_{sp \in SP} F_{sp} \tag{3.1.a}$$

Furthermore, $Pr[F_{sp}] = \prod_{e_i \in sp} p_i$. Let $WSP = \sum_{sp \in SP} Pr[F_{sp}]$.

3.2. Estimating $Pr[F]$, A Preliminary Version

F can be written as the union of sets as in equation (3.1.a). We first give a preliminary version of the algorithm to estimate $Pr[F]$ using the framework established in section 2.1. After presenting this algorithm in an abstract setting, we discuss a possible implementation. This will motivate the final version of the algorithm discussed in the following subsection.

The coverage of $s \in F$ is the set

$$cov(s) = \{sp \in SP : s \in F_{sp}\}$$

Let the universe be

$$U = \{(s,sp) : s \in F \text{ and } sp \in cov(s)\} = \{(s,sp) : sp \in SP \text{ and } s \in F_{sp}\}.$$

For all $(s,sp) \in U$,

$$wt(s,sp) = Pr[s].$$

For all $s \in F$, $\sum_{sp \in cov(s)} \alpha(s,sp) = 1$. Then, $wt(U) = WSP$. We now describe the trial step.

Preliminary Version of One Trial of the Reachability Algorithm

1. Randomly choose $sp \in SP$ with probability $\frac{Pr[F_{sp}]}{WSP}$
2. Randomly choose $s \in F_{sp}$ with probability $\frac{Pr[s]}{Pr[F_{sp}]}$.

   (At this point $(s,sp) \in U$ has been chosen with probability $\frac{Pr[s]}{WSP}$)
3. Compute $\alpha(s,sp)$
4. Estimator $Y \leftarrow \alpha(s,sp) \cdot WSP$

We now discuss a possible implementation of this algorithm. Step 2, the random choice of $s \in F_{sp}$, can be implemented as follows:

Randomly choose $s \in F_{sp}$

All edges in sp are failing in s.

For each edge $e_i \notin sp$, randomly choose e_i to
fail with probability p_i, or
work with probability $1-p_i$

The time for this step is $O(m)$.

Now we discuss one possible implementation of step 3, the computation of $\alpha(s,sp)$. For fixed s we need

$$\sum_{sp \in cov(s)} \alpha(s,sp) = 1$$

Let X be a fast algorithm for which the input is an undirected graph with m' edges and two specified nodes and which outputs a simple path between the two specified nodes. It is easy to design X such that the running time is $O(m')$. Let G_s be the graph consisting of the nodes in G together with the failing edges in s. The specified nodes in G_s are x and y.

Compute $\alpha(s,sp)$

Form G_s from G by deleting all working edges in s.
Let $sp' \leftarrow x-y$ simple path output by X when the input is G_s.

$$\alpha(s,sp) \leftarrow \begin{cases} 1 & \text{if } sp = sp' \\ 0 & \text{if } sp \neq sp' \end{cases}$$

For fixed s, X always outputs the same simple path $sp' \in cov(s)$. Therefore, $\sum_{sp \in cov(s)} \alpha(s,sp) = 1$. The time for this step is $O(m)$.

Now we discuss the implementation of steps 1 and 4. We could list all the $x-y$ simple paths in G. From this we could calculate $Pr[F_{sp}]$ for each $sp \in SP$ and then finally sum these values to obtain $WSP = \sum_{sp \in SP} Pr[F_{sp}]$. However, if the number of $x-y$ simple paths is large then this is a costly computation, both in terms of time and space. In general the number of $x-y$ simple paths is very large in comparison to the size of the graph. We would like to compute WSP without listing all the $x-y$ simple paths. However, if all the edge failure probabilities were equal to one, WSP is the number of $x-y$ simple paths in G. Counting the number of $x-y$ simple paths in an undirected graph, even if the graph is planar, is known to be NP-hard. This follows from an easy reduction from the planar Hamiltonian Circuit problem [4]. Therefore, we do not attempt to compute WSP directly. What we do instead in the next subsection is design a set W such that W is a superset of SP and the weight of W is easy to compute. We then use an implementation of the coverage algorithm described in subsection 2.2 to estimate $Pr[F]$, where the weight of W is the weight of the sample set.

3.3. Estimating $Pr[F]$, The Final Version

The algorithm we develop in this subsection corresponds abstractly to the coverage algorithm presented in section 2.2. For all $sp \in SP$, let

$$X_{sp} = \{(s, sp) : s \in F_{sp}\} .$$

Then $wt(X_{sp}) = Pr[F_{sp}]$. For notational simplicity we will use $wt(sp)$ to denote $wt(X_{sp})$. A walk is an ordered set of edges

$$\{(u_0,u_1),(u_1,u_2), \cdots ,(u_{k-1},u_k)\}$$

such that (u_{i-1},u_i) is an edge in the graph. Edges may be repeated in the walk. An $a-b$ walk is a walk which starts at node a and ends at node b. The **weight** of a walk wk, $wt(wk)$, is $\prod_{e_i \in wk} p_i$, where p_i is repeated in this product as many times as the edge e_i appears in the walk wk. Thus, the notion of weight extends naturally from simple paths to walks.

We now consider a class of walks which includes all $x-y$ simple paths but excludes a large portion (at least in terms of weight) of the $x-y$ walks. Let l be a positive integer, the value of which we will discuss in a later section. A **l-restricted walk** is a walk with at most $n-1$ edges such that every consecutive subsequence of l nodes along the walk is distinct. Let W be the set of all $x-y$ l-restricted walks, and let $WW = \sum_{wk \in W} wt(wk)$. Clearly, $SP \subseteq WW$ and

$WSP \leq WW$

In the following section we develop a dynamic programming subroutine to calculate WW. Furthermore, the calculation of WW allows us to randomly choose $wk \in W$ with probability $\frac{wt(wk)}{WW}$. This is the basis for the final algorithm which is abstractly described in section 2.2. The sample set is

$$V = U \cup \{wk : wk \in W-SP\}$$

The weight function is extended from U to V such that the weight of wk is $wt(wk)$. Then, $wt(V) = WW$. For all $wk \in W - SP$, $\alpha(wk) = 0$.

Coverage Algorithm to Estimate $Pr[F]$ for the Reachability Problem

Preprocessing

Compute WW using the subroutine discussed in the following section.

Trial

1. Randomly choose $wk \in W$ with probability $\frac{wt(wk)}{WW}$
2. **If** $wk \in W - SP$ (i.e. wk is not simple) **then** $Y \leftarrow 0$
   Else $wk = sp \in SP$, continue to step 3

   (At this point $sp \in SP$ has been chosen with probability $\frac{Pr[F_{sp}]}{WW}$)
3. Randomly choose $s \in F_{sp}$ with probability $\frac{Pr[s]}{Pr[F_{sp}]}$

   (At this point (s,sp) has been chosen with probability $\frac{Pr[s]}{WW}$)
4. Compute $\alpha(s,sp)$
5. Estimator $Y \leftarrow \alpha(s,sp) \cdot WW$

The preprocessing time is discussed in a subsequent section. Let d be the maximum node degree in G. As we show in the next section the time to perform step 1 is $O(n \cdot d)$. Step 2, determining whether or not wk is simple, can be implemented in time $O(m)$. The rest of the steps in this implementation can be performed in time $O(m)$, as discussed in the previous subsection. Thus, the running time per trial for this implementation is $O(n \cdot d)$.

4. Dynamic Programming Subroutine to Calculate WW

We present a dynamic programming subroutine to calculate the sum of the weights of $x-y$ l-restricted walks,

$$WW = \sum_{wk \in W} wt(wk) .$$

The dynamic programming subroutine can be used to implement steps 1 and 5 of the reachability algorithm presented in the previous section. Let $l \geq 2$ be the length of the shortest cycle allowed in a $x-y$ walk $wk \in W$. The value of l affects both the running time of the subroutine and the number of trials sufficient to obtain an (ε,δ) algorithm. We defer the choice of the value of l until a later section.

4.1. Concepts and Definitions

We first define some terminology which will simplify the following discussion. A **partial walk**, $(u_1,u_2,\cdots,u_{k-1})$, is a u_i-u_{k-1} simple path which fullfills the following conditions:

1.) The edges in the simple path are $e_i = (u_i,u_{i+1})$ for $i = 1,\ldots,k-2$.

2.) If node x appears in the path then it is the first node in the path u_1. Partial walks where x is the first node in the path are called **initial** partial walks.

3.) If node y appears in the path then it is the last node in the path u_{k-1}. Partial walks where y is the last node in the path are called **terminal** partial walks.

4.) k is equal to l unless x is the first node and y is the last node in the path. In this case k must be less than or equal to l, and the partial walk is a $x-y$ simple path of length less than or equal to $l-2$.

If $u_0 \neq u_{l-1}$ we say partial walk $(u_0,u_1,\cdots,u_{l-2})$ is a **predecessor** of partial walk $(u_1,\cdots,u_{l-2},u_{l-1})$ and that $(u_1,\cdots,u_{l-2},u_{l-1})$ is a **successor** of $(u_0,u_1,\cdots,u_{l-2})$. No initial partial walk can have a predecessor (by condition 2 above) nor can any terminal partial walk have a successor (by condition 3

above).

4.2. The Computation of WW

We now describe a function C, which will be computed using dynamic programming. The domain of C is the set of ordered pairs (lp,pw), where lp is an integer between 1 and $n-1$ and pw is a partial walk. We define $C(lp,(u_1,u_2,\cdots,u_{k-1}))$ to be the sum of the weights of all $x-u_{k-1}$ *l-restricted walks* of length lp for which the last portion of the walk is a partial walk $(u_1,u_2,\cdots,u_{k-1})$. From the definition of C we see that

$$WW = \sum_{lp=1}^{n-1} \sum_{\substack{\text{terminal} \\ \text{walk}}} \sum_{\substack{\text{partial} \\ (u_1,\cdots,u_{k-2},y)}} C\,(\,lp,(u_1,\cdots,u_{k-2},y))$$

The initial conditions for C are :

1.) $C\,(\,k-2,(x,u_2,\cdots,u_{k-2},y)) =$ weight of simple path $(x,u_2,\cdots,u_{k-2},y)$ for all simple paths of length $k-2 \le l-2$.

2.) $C\,(\,l-2,(x,u_2,\cdots,u_{l-1})) =$ weight of simple path $(x,u_2,\cdots,u_{l-1})$ for all other initial partial walks.

3.) All other values of C are initially zero.

The general formula for calculating C is

$$C\,(\,lp,(u_1,\cdots,u_{l-2},u_{l-1})) = p_{(u_{l-2},u_{l-1})} \cdot \sum_{\substack{\text{predecessors} \\ (u_0,u_1,\cdots,u_{l-2})}} C\,(lp-1,(u_0,u_1,\cdots,u_{l-2}))$$

for all $lp = l-1,\ \cdots,n-1$, for all partial walks which are not initial partial walks. The computation of C proceeds by calculating all the C values for a fixed lp in terms of the previously computed C values for $lp-1$. A more detailed description of the algorithm is given in the appendix.

Let d be the maximum node degree of any node in G. We now analyze the running time of this subroutine in terms of the parameters l and d. We first show an upper bound on the number of partial walks $(u_1,\cdots,u_{l-2},u_{l-1})$ in G.

There are at most n choices for u_{l-1}. Since u_l and u_{l+1} must be adjacent nodes in the graph, there are at most d^{l-2} choices for $u_1, \cdots, u_{l-2}$. Thus, there are at most $n \cdot d^{l-2}$ partial walks. The number of predecessors of any partial walk is at most d. Thus, the computation of C for all walks of some fixed length lp (which uses the values of C for walks of length $lp-1$) takes time at most $n \cdot d^{l-1}$. Since there are $n-1$ lengths for which the value of C is computed, the total time for the computation of WW is

$$O\left(n^2 d^{l-1}\right)$$

4.3. Choosing a Walk at Random

We now describe how to use the solution to the dynamic programming subroutine to randomly choose an *l−restricted walk* such that a particular walk wk is chosen with probability $\frac{wt(wk)}{WW}$.

Algorithm to Choose A Random *l*−*restricted walk wk*

Step 1 - Choose the length lp of wk and the terminal portion of wk, $(u_1, \cdots, u_{k-2}, y)$, such that lp and $(u_1, \cdots, u_{k-2}, y)$ are chosen with probability

$$\frac{C\,(\,lp, (u_1, \cdots, u_{k-2}, y))}{WW}$$

If $u_1 = x$ then $wk = (x, u_2, \cdots, u_{k-2}, y)$ is a simple path of length $lp = k-2$.
Otherwise $k = l$ and the length of wk is lp, where the terminal portion of wk is $(u_1, \cdots, u_{k-2}, y)$. In this case, continue to step 2 to choose the rest of wk.

Step 2 - DO UNTIL $u_1 = x$

Choose a predecessor partial walk of $(u_1, \cdots, u_{l-2}, u_{l-1})$ such that partial walk $(u_0, u_1, \cdots, u_{l-2})$ is chosen with probability

$$\frac{p_{(u_{l-2}, u_{l-1})} \cdot C\,(\,lp-1, (u_0, u_1, \cdots, u_{l-2}))}{C\,(\,lp, (u_1, \cdots, u_{l-2}, u_{l-1}))}$$

$wk \leftarrow (u_0, u_1) + wk$
$lp \leftarrow lp - 1$
$(u_1, \cdots, u_{l-2}, u_{l-1}) \leftarrow (u_0, u_1, \cdots, u_{l-2})$

Lemma 4.3.a : *l*−*restricted walk* wk is chosen by the algorithm with probability $\frac{wt(wk)}{WW}$

Proof :

Simple path $(x, u_2, \cdots, u_{k-2}, y)$ is chosen in step 1 of the algorithm with probability

$$\frac{wt(x, u_2, \cdots, u_{k-2}, y)}{WW}$$

since $C\,(\,k-2, (x, u_2, \cdots, u_{k-2}, y)) = wt(x, u_2, \cdots, u_{k-2}, y)$. Similarly, terminal partial walk $(u_1, \cdots, u_{l-2}, y)$ and length lp are chosen together in step 1 with probability

$$\frac{C\,(\,lp, (u_1, \cdots, u_{l-2}, y))}{WW}$$

The predecessor partial walk $(u_0,u_1,\cdots,u_{l-2})$ of $(u_1,\cdots,u_{l-2},y)$ is chosen with probability

$$\frac{p_{(u_{l-2},y)}\cdot C\,(\,lp-1,(u_0,u_1,\cdots,u_{l-2}))}{C\,(\,lp,(u_1,\cdots,u_{l-2},y))}$$

so that the last portion of a walk of length lp such that $(\,u_0,u_1,u_2,\cdots,u_{l-2},y\,)$ is the final portion of the walk is

$$\frac{p_{(u_{l-2},y)}\cdot C\,(\,lp-1,(u_0,u_1,\cdots,u_{l-2}))}{WW}$$

because $C\,(\,lp,(u_1,\cdots,u_{l-2},y))$ appears both in the numerator and the denominator. Similarly, by induction, the probability that walk $wk = (\,x=u_1,u_2,\cdots,u_{l-1},u_l,\cdots,u_{lp},u_{lp+1}=y\,)$ of length lp is chosen is

$$\frac{\prod_{k=l-1}^{lp} p_{(u_k,u_{k+1})}\cdot C\,(\,l-2,(x,u_2,\cdots,u_{l-1}))}{WW}$$

which is equal to

$$\frac{\prod_{k=1}^{lp} p_{(u_k,u_{k+1})}}{WW} = \frac{wt\,(wk\,)}{WW}$$

■

We now sketch the extra preprocessing necessary for a fast implementation of step 1 of the algorithm to choose a random $l-$ restricted walk. Let nt be the number of terminal partial walks. We associate each terminal partial walk uniquely with a number between 1 and nt. We allocate an array T of length $nt\cdot(n-1)$. Each index into this array can be written in the form

$$lp\ +\ (n-1)\cdot(j-1)$$

where $1\le lp\le n-1$ and $1\le j\le nt$. $T(lp+(n-1)\cdot(j-1))$ is equal to $C(lp,(u_1,\cdots,u_{k-2},y))$, where $(u_1,\cdots,u_{k-2},y))$ is the terminal partial walk associated with index j. Notice that

$$WW = \sum_{a=1}^{nt\cdot(n-1)} T(a) .$$

We use T to form another array T' as follows, where the indices of T' range between 0 and $nt\cdot(n-1)$.

$$T'(0) = 0$$

$$T'(a) = \frac{\sum_{b=1}^{a} T(b)}{WW}, \text{ for } 1 \le a \le nt\cdot(n-1) .$$

Notice that

$$T'(a) - T'(a-1) = \frac{T(a)}{WW} ,$$

the entries in T' are in increasing order and $T'(nt\cdot(n-1)) = 1$.

Step 1 can be implemented as follows. First, choose a random number r from the uniform [0,1] distribution. Then, use binary search to find the index a such that

$$T'(a-1) < r \le T'(a) .$$

The probability that a particular index a is chosen is equal to $\frac{T(a)}{WW}$. Then, find the unique indices lp and j such that

$$a = lp + (n-1)\cdot(j-1) .$$

The length of the chosen walk is lp and the last portion of the walk is the terminal partial walk corresponding to j. The time for the binary search is bounded above by

$$O\left(lg\left(nt\cdot(n-1)\right)\right) .$$

This quantity is $O(n)$. Therefore, step 1 takes at most $O(n)$ time. We discuss a slightly different implementation in greater detail in the appendix.

In step 2 we choose a predecessor partial walk $lp - l$ times and there are at most d predecessors to any partial walk. Thus, the entire walk can be chosen in step 2 in $O(n \cdot d)$ time.

In this section and the previous section we have given an algorithm to estimate the failure probability for the reachability problem. The basis of this algorithm is the coverage algorithm described in section 2. The primary difference between the algorithm presented here and a straightforward implementation of the algorithm presented in chapter 2 is the preprocessing step. The preprocessing step for the algorithm presented in chapter 2 requires listing all the $x-y$ simple paths in the graph, whereas the preprocessing step here is the dynamic programming subroutine presented in this section. Since the number of $x-y$ simple paths can be exponential in the size of the graph, the savings can be substantial. In a later section we formally analyze this algorithm and prove an upper bound on the running time for an (ε,δ) algorithm which is independent of the number of $x-y$ simple paths.

5. $x-y$ Planar Two-Terminal Problem

The two-terminal problem can be described as follows. Let G be a network and let x and y be two specified nodes in G. State s is a **failure state** if there is no path of working edges between x and y in s. Let $F \subseteq S$ be the set of all failure states. The problem is to compute $Pr[F]$. In this section we discuss a special case of the two-terminal problem. We first introduce some graph concepts which will simplify the discussion.

5.1. Graph Concepts

An $a-b$ **cut** is a minimal set of edges whose removal from the graph disconnects the graph into two components, where node a is in one component and node b is in the other. State s is a failure state if and only if there is a $x-y$ cut of failing edges in s. Let CT be the set of all $x-y$ cuts in G. For $ct \in CT$, let F_{ct} be the set of all states s such that all edges in ct are failing in s. Then,

$$F = \bigcup_{ct \in CT} F_{ct}$$

Furthermore, $Pr[F_{ct}] = \prod_{e_i \in ct} p_i$. Let $WCT = \sum_{ct \in CT} Pr[F_{ct}]$.

We denote the graph obtained by **deleting** edge e from G as $G-e$. We denote the graph obtained by **contracting** edge e in G as $G \cdot e$. We denote the graph obtained by the **addition** of edge e to G as $G+e$. An $a-b$ **dummy edge** is an edge added between nodes a and b in G, where the failure probability of the dummy edge is one. Note that adding dummy edges to G does not change $Pr[F]$.

Let G be a planar network. The **dual network** G' of G is formed as follows (see Figure 5.1.a): Each face in G corresponds to a unique node in G' and each face in G' corresponds to a unique node in G. For each edge e_i in G there is a unique edge e'_i in G' such that e_i and e'_i cross each other in the plane. The

failure probability of e'_i is equal to the failure probability of e_i. There is a one to one correspondence between states in G and G'. State s in G corresponds to the unique state s' in G' where e_i is failing in s if and only if e'_i is failing in s'. Clearly, $Pr[s] = Pr[s']$.

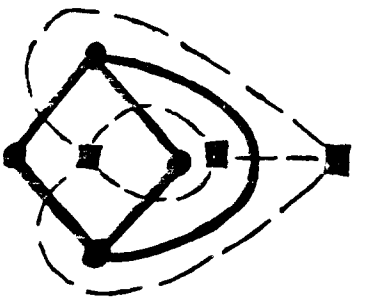


Figure 5.1.a - *G and the dual network G'*

We say G is $a-b$ **planar** if $G+e$ is planar, where e is a new edge added between nodes a and b in G.

5.2. Estimating $Pr[F]$

Let G be $x-y$ planar. We discuss the two-terminal problem for this special case. Consider $G+e$, where e is a $x-y$ dummy edge added to G (see Figure 5.2.a). Let $(G+e)'$ be the dual network of $G+e$. Label the two nodes in $(G+e)'$ corresponding to the two faces bordering edge e in $G+e$ as x' and y' respectively. Let $G' = (G+e)'-e'$, where e' is the edge in $(G+e)'$ corresponding to dummy edge e.

Consider the reachability problem for G', where x' and y' are the specified nodes. Let SP be the set of $x'-y'$ simple paths in G', and let F'_{sp} be the set of states s' such that all edges in sp are failing in s'. Notice that every $x-y$ cut $ct \in CT$ in G corresponds to a unique $x'-y'$ simple path $sp \in SP$ in G' where $Pr[F_{ct}] = Pr[F'_{sp}]$. Similarly, every failure state s of G for the two-terminal

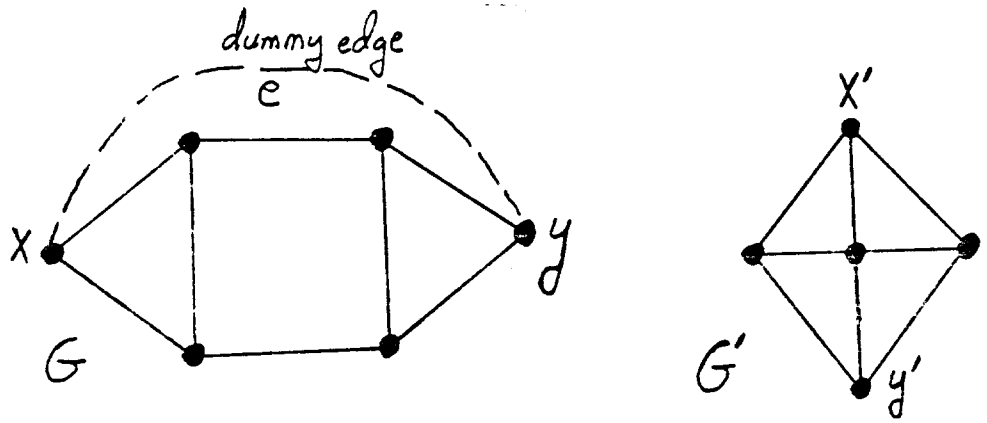


Figure 5.2.a - *The formation of G' for the x –y planar two-terminal problem*

problem corresponds to a unique failure state s' of G' for the reachability problem, where $Pr[s] = Pr[s']$. Therefore, we can estimate the failure probability for the two-terminal problem on G by forming G' and estimating the failure probability for the reachability problem on G'.

6. Planar Two-Terminal Problem

We discuss the two-terminal problem when G is planar in this section. In the previous section we discussed a special case of this problem, when G is $x-y$ planar. Notice that G can be planar without being $x-y$ planar (see Figure 6.1.b). We first introduce some concepts which will simplify the discussion.

6.1. Preliminary Concepts

Let F be the set of all failure states in G. State s is a **failure state** if and only if there is a $x-y$ cut of failing edges in s. Let CT be the set of all $x-y$ cuts in G. For $ct \in CT$, let F_{ct} be the set of all states s such that all edges in ct are failing in s. Then,

$$F = \bigcup_{ct \in CT} F_{ct}$$

Since the set of failure states F can be written as a union of sets, we could try to implement the abstract algorithm described in subsection 2.1 directly, where the universe is

$$U = \{(s,ct) : ct \in CT \text{ and } s \in F_{ct}\},$$

$wt(s,ct) = Pr[s]$ and $wt(ct) = Pr[F_{ct}]$. The main difficulty with this approach, as was mentioned at the end of subsection 2.1, is that it is hard to compute $wt(U) = \sum_{ct \in CT} Pr[F_{ct}]$. To overcome this difficulty we do two things. First, we work with the dual problem in the dual network G'. Secondly, we define a sample set V such that the universe $U \subseteq V$ as described in subsection 2.2 so that $wt(V)$ is easy to compute. We use a variant of the dynamic programming subroutine described in section 4 to compute $wt(V)$. These same basic techniques will also be used for the planar K-terminal and all-terminal algorithms.

We introduce some terminology to describe the dual problem. A **cycle** is a walk that starts and ends at the same node. A **simple cycle** is a cycle with no

repeated nodes or edges, except that the start and end node are the same. Let G' be the dual network of G. We say simple cycle cy **separates** face x from face y if cy corresponds to an $x-y$ cut in G. Let CY be the set of simple cycles that separate face x from face y. State s is a failure state in G' if and only if there is a simple cycle $cy \in CY$ among the failing edges in s. For $cy \in CY$, let F'_{cy} be the set of all states s in G' such that all edges in cy are failing in s. Let F' be the set of all failure states in G'. Then, the set of failure states in the dual network is

$$F' = \bigcup_{cy \in CY} F'_{cy} \tag{6.1.a}$$

The dual problem is to compute $Pr[F'] = Pr[F]$. The universe for the dual problem is

$$U = \{(s,cy) : cy \in CY \text{ and } s \in F'_{cy}\} ,$$

where $wt(s,cy)) = Pr[s]$ and $wt(cy) = Pr[F'_{cy}]$. Thus,

$$wt(U) = \sum_{cy \in CY} Pr[F'_{cy}] .$$

We now introduce the concepts needed to explain how the sample set V is defined. The sample set will be defined in such a way that $wt(V)$ can be computed using a dynamic programming subroutine. Consider $x-y$ simple path

$$sp = \{e_{i_1}, e_{i_2}, \cdots, e_{i_{ls}}\}$$

of length ls in G. Let $\{e'_{i_1}, \cdots, e'_{i_{ls}}\}$ be the edges in G' corresponding to $\{e_{i_1}, e_{i_2}, \cdots, e_{i_{ls}}\}$ in G. Every simple cycle $cy \in CY$ which separates face x from face y must contain an odd number of edges from $\{e'_{i_1}, \cdots, e'_{i_{ls}}\}$ (see Figure 6.1.b). Another way of viewing this is that every simple cycle $cy \in CY$ includes a first edge $e'_{i_s} \in \{e'_{i_1}, \cdots, e'_{i_{ls}}\}$ and an even number of edges from $\{e'_{i_{s+1}}, \ldots, e'_{i_{ls}}\}$. Thus, we can partition CY into $CY_1, CY_2, \cdots, CY_{ls}$, where CY_s

is the set of simple cycles $cy \in CY$ such that the first edge from $\{e'_{i_1}, \cdots, e'_{i_z}\}$ is e'_{i_z} and for which there are an even number of edges from $\{e'_{i_{z+1}}, \cdots, e'_{i_u}\}$ in cy.

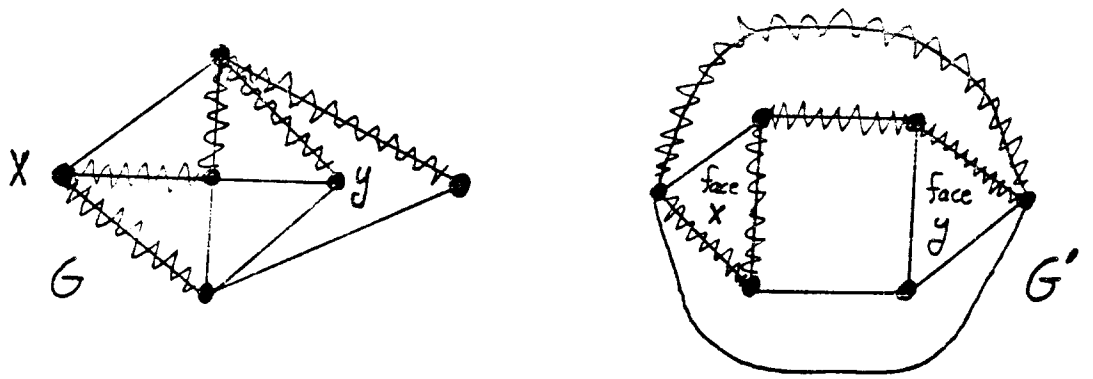


Figure 6.1.b - *An $x-y$ cut in G corresponds to a simple cycle in G'*

Let $G'_z = G' - \{e'_{i_1}, \cdots, e'_{i_z}\}$, where the two endpoints of e'_{i_z} in G' are labelled x'_z and y'_z respectively in G'_z. A **z-simple path** is a $x'_z - y'_z$ simple path in G'_z including an even number of edges from $\{e'_{i_{z+1}}, \cdots, e'_{i_u}\}$. Let SP_z be the set of all $z-$simple paths. For $sp \in SP_z$, let F'_{sp} be the set of all states s in G' such that all edges in sp are failing in s. Every simple cycle $cy \in CY_z$ is a $z-$simple path $sp \in SP_z$ together with edge e'_{i_z}, such that

$$Pr[F'_{cy}] = Pr[F'_{sp}] \cdot p_{i_z}$$

Let

$$WSP_z = \sum_{sp \in SP_z} Pr[F'_{sp}], \quad WCY_z = \sum_{cy \in CY_z} Pr[F'_{cy}]$$

and

$$WCY = \sum_{cy \in CY} Pr[F'_{cy}]$$

Then, $WCY_z = WSP_z \cdot p_{i_z}$ and

$$wt(U) = WCY = \sum_{s=1}^{ls} WCY_s \ .$$

6.2. Estimating $Pr[F']$

F' can be written as the union of sets as in (6.1.a). We would like to be able to implement the algorithm abstractly described in subsection 2.1. This requires the computation of WCY and a procedure to randomly choose $cy \in CY$. However, it is infeasible to compute WCY exactly, because if all the failure probabilities were equal to one then this is equivalent to counting the number of simple cycles, which is NP-hard. This follows from an easy reduction from the planar Hamiltonian Circuit problem [4]. We will use a modification of the dynamic programming subroutine presented in section 4 to circumvent these problems.

Let W_s be a suitably restricted set of walks in G' such that $SP_s \subseteq W_s$ and let $WW_s = \sum_{wk \in W_s} wt(wk)$. Let

$$W = \bigcup_{s=1}^{ls} \{wk + e'_{i_s} : wk \in W_s\}$$

Notice that $CY \subseteq W$. The sample set will be

$$V = U \cup \{wk : wk \in W - CY\}$$

where, for $wk \in W - CY$, $wt(wk)$ is the weight of walk wk. Let WW be the weight of sample set V. Then

$$wt(V) = WW = \sum_{s=1}^{ls} WW_s \cdot p_{i_s} \ .$$

In the next subsection we develop an algorithm to compute WW_s which allows us to randomly choose $wk \in W_s$ with probability $\frac{wt(wk)}{WW_s}$. This allows us to randomly choose $cy \in W$ with probability $\frac{wt(cy)}{WW}$. We now present the algorithm

to estimate $Pr[F']$ and then discuss the implementation details.

Algorithm to Estimate $Pr[F']$ for the Planar Two-Terminal Problem

Preprocessing

For $z = 1,\ldots,ls$ compute WW_z. Then, $wt(V) = WW = \sum_{z=1}^{ls} WW_z \cdot p_{i_z}$.

Trial

1. Randomly choose $z \in \{1, \cdots, ls\}$ with probability $\frac{WW_z \cdot p_{i_z}}{WW}$.
2. Randomly choose $wk \in W_z$ with probability $\frac{wt(wk)}{WW_z}$

   (At this point $wk + e'_{i_z} \in W$ has been chosen with probability $\frac{wt(wk + e'_{i_z})}{WW}$).
3. **If** $wk + e'_{i_z}$ is not simple **then** $Y \leftarrow 0$
   Else let $cy = wk + e'_{i_z}$ $(cy \in CY_z)$, and continue to step 4

   (At this point $cy \in CY$ has been chosen with probability $\frac{Pr[F'_{cy}]}{WW}$)
4. Randomly choose $s \in F'_{cy}$ with probability $\frac{Pr[s]}{Pr[F'_{cy}]}$

   (At this point (s,cy) has been chosen with probability $\frac{Pr[s]}{WW}$)
5. Compute $\alpha(s,cy)$
6. Estimator $Y \leftarrow \alpha(s,cy) \cdot WW$

The preprocessing time is discussed in a subsequent section. The time to perform step 1 is $O(n)$ since $ls \leq n$. Let d be the maximum node degree in G' (which is equal to the maximum number of edges bordering any face in G). The time to perform step 2 is $O(n \cdot d)$, which we show in the next subsection. Step 3, determining whether or not wk is simple, can be performed in time $O(m)$. Step 4 is analogous to step 2 for the reachability algorithm in section 3.3, the running time is $O(m)$. The running time for step 6 is $O(1)$.

Step 5, the computation of $\alpha(s,cy)$, can be implemented in an analogous manner to the computation of α for the reachability problem. The primary difference is that in the reachability problem s contains a $x-y$ simple path of

failing edges in G while here s contains a $x-y$ cut of failing edges in G (or equivalently, a cycle separating face x from face y in G'). For $s \in F'$,

$$cov(s) = \{cy \in CY : s \in F'_{cy}\}.$$

For fixed s we need

$$\sum_{cy \in cov(s)} \alpha(s,cy) = 1$$

Let X be the algorithm presented in [5] for which the input is an undirected graph with m' edges and two specified nodes a and b; and which outputs an $a-b$ cut ct'. The running time for X is $O(m)$. Let G_s be the graph formed from G by contracting all working edges in s. The edges in G_s correspond to the failing edges in s, and the nodes in G_s correspond to the components of G which are connected by working edges in s. The two specified nodes x and y in G_s correspond to the two components in G containing x and y. Nodes x and y in G are in different components because s contains a $x-y$ cut of failing edges.

Compute $\alpha(s,cy)$

Form G_s from G by contracting all working edges in s.
Let $ct' \leftarrow x-y$ cut output by X when the input is G_s.
Let cy' be the simple cycle in G' corresponding to ct'

$$\alpha(s,cy) \leftarrow \begin{cases} 1 & \text{if } cy = cy' \\ 0 & \text{if } cy \neq cy' \end{cases}$$

For fixed s, X always outputs the same cut ct', which corresponds to $cy' \in cov(s)$. Therefore, $\sum_{cy \in cov(s)} \alpha(s,cy) = 1$. The time for this step is $O(m)$.

The total time for all steps of the trial is $O(n \cdot d)$.

6.3. Modified Dynamic Programming Subroutine

We would like to be able to compute WSP_s. WSP_s is equal to $\sum_{sp \in SP_s} Pr[F'_{sp}]$, where SP_s is the set of all $z-$simple paths in G'_s. Except for the restriction that the number of occurrences of edges $\{e'_{i_{s+1}}, \ldots, e'_{i_{s'}}\}$ must

be even in every simple path, this is exactly the same problem the dynamic programming algorithm presented in section 4 was designed to circumvent. We now present a modified version of that algorithm for the problem at hand.

A zl**-restricted walk** is a x'_z-y'_z l-restricted walk in G'_z with an even number of occurrences of edges from the set $\{e'_{i_{z+1}}, \ldots, e'_{i_{z_2}}\}$. Let W_z be the set of all zl-restricted walks and let $WW_z = \sum_{wk \in W_z} wt(wk)$. Then, $SP_z \subseteq W_z$ and $WSP_z \leq WW_z$. We now present a dynamic programming subroutine to calculate WW_z.

We first describe a function C, which will be computed using dynamic programming . The domain of C is the set of ordered triples (par, lp, pw), where par is either *even* or *odd*, lp is an integer between 1 and $n-1$ and pw is a partial walk. We define C (*even*, lp, $(u_1, u_2, \cdots, u_{k-1})$) to be the sum of the weights of all l*-restricted walks* of length lp where the last portion of the walk is partial walk $(u_1, u_2, \cdots, u_{k-1})$, and where the number of occurrences of edges $\{e'_{i_{z+1}}, \ldots, e'_{i_{z_2}}\}$ in the walk is even. Similarly, C (*odd*, lp, $(u_1, u_2, \cdots, u_{k-1})$) has the same definition except the number of occurrences of edges $\{e'_{i_{z+1}}, \ldots, e'_{i_{z_2}}\}$ in the walk is odd. Then,

$$WW_z = \sum_{lp=1}^{n-1} \sum_{(u_1, \cdots, u_{k-2}, y'_z)} C\,(\,even, lp, (u_1, \cdots, u_{k-2}, y'_z))$$

The initial conditions for C are

1.) C (*even*, $k-2$, $(x'_z, u_2, \cdots, u_{k-2}, y'_z)$) = weight of z-simple path $(x'_z, u_2, \cdots, u_{k-2}, y'_z)$ of length $k-2 \leq l-2$.

2.) C (*even*, $l-2$, $(x'_z, u_2, \cdots, u_{l-1})$) = weight of simple path $(x'_z, u_2, \cdots, u_{l-1})$ for all other initial paths for which the number occurrences of edges $\{e'_{i_{z+1}}, \ldots, e'_{i_{z_2}}\}$ is even.

3.) C (*odd*, $l-2$, $(x'_z, u_2, \cdots, u_{l-1})$) = weight of simple path $(x'_z, u_2, \cdots, u_{l-1})$ for all other initial paths for which the number of occurrences of edges $\{e'_{i_{z+1}}, \ldots, e'_{i_{z_2}}\}$ is odd.

4.) All other values of C are initially zero.

Let

$$flip(par) = \begin{cases} odd & \text{if } par = even \\ even & \text{if } par = odd \end{cases}$$

The general formula for calculating C is

$$C(par', lp, (u_1, \cdots, u_{l-2}, u_{l-1})) = p_{(u_{l-2}, u_{l-1})} \cdot \sum_{\substack{\text{predecessors} \\ (u_0, u_1, \cdots, u_{l-2})}} C(par, lp-1, (u_0, u_1, \cdots, u_{l-2}))$$

where

$$par = \begin{cases} par' & \text{if } (u_{l-2}, u_{l-1}) \notin \{e'_{i_{s+1}}, \ldots, e'_{i_{s'}}\} \\ flip(par') & \text{if } (u_{l-2}, u_{l-1}) \in \{e'_{i_{s+1}}, \ldots, e'_{i_{s'}}\} \end{cases}$$

Thus, using this modified dynamic program on graph G'_s we can calculate WW_s.

We now analyze the running time for this subroutine. Let d be the maximum node degree of any node in G' (which is equal to the maximum number of edges bordering any face in G). The only difference between the running time for this subroutine and the time for the subroutine presented in section 4 is that there are at most twice as many C values that need to be computed. Thus, the running time for the subroutine here is at most twice the running time for the subroutine presented in section 4. The total time for the computation of W_s is

$$O(n \cdot d^{l-1})$$

In an analogous manner to that used in section 4 we can use this dynamic programming solution to randomly choose a zl-restricted walk wk with probability $\frac{wt(wk)}{WW_s}$. The entire walk can be chosen in time

$$O(n \cdot d)$$

6.4. Adding Dummy Edges to Reduce the Preprocessing Time

The number of dynamic programming problems that have to be solved to implement the algorithm is proportional to l_s, the length of the $x-y$ simple path in G. Thus, we want this path to be as short as possible. We pointed out earlier that adding dummy edges to G does not change the failure probability $Pr[F]$. Thus, we can sometimes reduce the preprocessing time by adding dummy edges to G to create a shorter path. The only restriction is that the graph must remain planar after the dummy edges are added (see Figure 6.4.a). This is the generalization of the technique we used to solve the $x-y$ planar two-terminal problem, where we added exactly one dummy edge between x and y (and consequently had to solve only one dynamic programming problem).

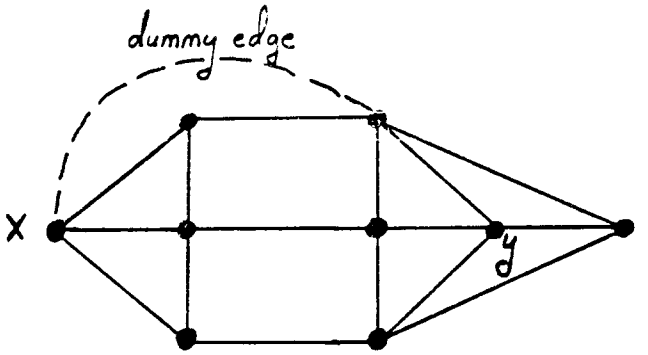


Figure 6.4.a - *Adding dummy edges to G to reduce the preprocessing time*

7. Planar K–Terminal Problem

The K–terminal problem can be described as follows. Let G be a network and let $x_1, \cdots, x_K$ be K specified nodes in G. State s is a **failure state** if there is a pair of specified nodes x_i and x_j which cannot be joined by a path of working edge in s. Let $F \subseteq S$ be the set of all failure states. The problem is to compute $Pr[F]$. In this section we present an algorithm to estimate $Pr[F]$ when G is planar.

7.1. Preliminary Concepts

For the K–terminal problem, state s is a failure state if and only if there is a $j \in \{2, \cdots, K\}$ such that there is a x_1–x_j cut of failing edges in s. Let F_j be the set of failure states for the x_1–x_j two-terminal problem. Then,

$$F = \bigcup_{j \in \{2, \cdots, K\}} F_j$$

and

$$Pr[F] = Pr\left[\bigcup_{j \in \{2, \cdots, K\}} F_j\right] .$$

Any algorithm for the exact solution of the two-terminal problem can be used as the basis for a Monte-Carlo algorithm for the K–terminal problem. We now describe a Monte-Carlo implementation of the K–terminal algorithm using any two-terminal algorithm which computes the exact failure probability. The abstract description of the algorithm is given in subsection 2.1. The universe is

$$U = \{(s,j) : s \in F_j \text{ and } j \in \{2, \cdots, K\}\} ,$$

where $wt(s,j) = Pr[s]$ and $wt(U) = \sum_{j \in \{2, \cdots, K\}} Pr[F_j]$. The preprocessing step is to compute $Pr[F_j]$ for $j \in \{2, \cdots, K\}$. From this, $wt(U)$ can be computed. The first step of a trial is to randomly choose $j \in \{2, \cdots, K\}$ with probability $\frac{Pr[F_j]}{wt(U)}$. The implementation of this step is straightforward. The second step

of a trial is to randomly choose $s \in F_j$ with probability $\frac{Pr[s]}{Pr[F_j]}$. This can be implemented as follows:

Initially, let H be the network G.

Do Until the state of all edges in s are determined

For edge e in H, compute the failure probability
p_1 of network $H-e$, and
p_2 of network $H \cdot e$.

With probability
$\frac{p_1}{p_1+p_2}$ choose edge e to **fail** in s and let $H \leftarrow H-e$.

$\frac{p_2}{p_1+p_2}$ choose edge e to **work** in s and let $H \leftarrow H \cdot e$.

The computation of α can be implemented as follows. Let j' be the smallest index such that there is no path of working edges between x_1 and $x_{j'}$ in s. Then,

$$\alpha(s,j) = \begin{cases} 1 & \text{if } j = j' \\ 0 & \text{if } j \neq j' \end{cases}.$$

An upper bound on the running time per trial is $O(m\ t)$, where m is the number of edges in G and t is the time to compute the failure probability for a two-terminal problem with at most m edges. The number of trials N necessary to achieve an (ε,δ) algorithm is $K \cdot \ln\frac{2}{\delta}\ \frac{1}{\varepsilon^2}$.

We now describe the planar K-terminal algorithm. The algorithm just described is the abstract basis for the planar K-terminal algorithm. However, since it is not feasible to calculate the exact failure probability for the planar two-terminal problem, we will not be able to implement this algorithm directly. We will use the Monte-Carlo planar two-terminal algorithm described in section 3 in place of an algorithm to compute the exact failure probability for the planar two-terminal problem. We will use a technique different than that described above to randomly choose a failure state.

We first describe the dual problem in the dual network. Let G' be the dual network of G. Every cut in G corresponds to a unique simple cycle in G'. We say cycle cy **separates** face x_1 from face x_j if cy corresponds to a x_1–x_j cut ct in G. Let CY_j be the set of all cycles separating face x_1 from face x_j in G'. Let F'_{cy} be the set of all states s in G' such that all edges in cycle cy are failing in s. Let F'_j be the set of all states s in G' such that there is a cycle of failing edges in s separating face x_1 from face x_j. Let F' be the set of all states s in G' such that there is a cycle of failing edges in s separating face x_1 from face x_j for some $j \in \{2, \cdots, K\}$. Then,

$$F'_j = \bigcup_{cy \in CY_j} F'_{cy} \ .$$

and

$$F' = \bigcup_{j \in \{2, \cdots, K\}} F'_j \ .$$

The dual problem is to compute $Pr[F'] = Pr[F]$. The universe for the planar K–terminal algorithm is

$$U = \{(s,j) : s \in F'_j \text{ and } j \in \{2, \cdots, K\}\}$$

where $wt(s,j) = Pr[s]$ and $wt(j) = Pr[F'_j]$.

7.2. Estimating $Pr[F']$

Now we describe the implementation of the coverage algorithm for the planar K–terminal problem. The coverage algorithm uses the planar two-terminal algorithm to randomly choose $s \in F'_j$. Because we have two coverage algorithms we need to distinguish between the α_{two} function associated with the planar two-terminal algorithm and the α_K function associated with the planar K–terminal algorithm. The α_{two} function for each of the planar two-terminal problems is defined and computed the same way as described in subsection 6.2. The α_K function will be described after we present the algorithm to estimate

$Pr[F']$.

We now describe abstractly how the planar two-terminal algorithm is used to randomly choose $s \in F'_j$. As described in subsection 6.1, the universe for the x_1–x_j planar two-terminal algorithm is

$$U_j = \{(s,cy) : cy \in CY_j \text{ and } s \in F'_{cy}\} .$$

Let W_j be the set of l–restricted walks for the x_1–x_j planar two-terminal problem. As described in subsection 6.2, the sample set for the x_1–x_j planar two-terminal algorithm is

$$E_j = U_j \cup \{wk : wk \in W_j - CY_j\} .$$

As described in subsection 2.2, each $s \in F'_j$ can be **associated** with the unique $(s,cy) \in U_j$ such that $\alpha_{two}(s,cy) = 1$. Note that $wt(s,cy) = Pr[s]$. We say $s \in F'_j$ is **chosen** in a trial of the planar two-terminal algorithm if $(s,cy) \in U_j \subseteq E_j$ is chosen in the trial where s is associated with (s,cy). For simplicity of notation, we write $s \in E_j$ instead of $(s,cy) \in E_j$ if s is associated with (s,cy). We say the trial **fails** if no $s \in F'_j$ is chosen in the trial. The probability a particular $s \in F'_j$ is chosen in a trial is $\frac{Pr[s]}{wt(E_j)}$.

We now describe the sample set V for the planar K–terminal algorithm. Let

$$V_j = \{(w,j) : w \in E_j\}$$

where $wt(w,j) = wt(w)$ and $wt(V_j) = wt(E_j)$. Then,

$$V = \bigcup_{j \in \{2,\cdots,K\}} V_j .$$

The wt function as defined on each V_j carries over naturally to V. In subsection 6.3 we showed how to calculate $wt(E_j) = wt(V_j)$. Thus, $wt(V) = \sum_{j \in \{2,\cdots,K\}} wt(V_j)$ is also easy to compute. We now present the algorithm to estimate $Pr[F']$ and then discuss the implementation details.

Algorithm to Estimate $Pr[F']$ for the Planar K-Terminal Problem

Preprocessing

For $j \in \{2, \cdots, K\}$ perform the preprocessing step for the x_1-x_j planar two-terminal algorithm to compute $wt(V_j)$. Let
$WW = wt(V) = \sum_{j \in \{2, \cdots, K\}} wt(V_j)$.

Trial

1. Randomly choose $j \in \{2, \cdots, K\}$ with probability $\frac{wt(V_j)}{WW}$.
2. Randomly choose $s \in F'_j$ with probability $\frac{Pr[s]}{wt(V_j)}$ using the planar two-terminal algorithm as a subroutine.

   If the trial **fails** then $Y \leftarrow 0$
   Else $s \in F'_j$ is chosen, continue to step 3.

   (At this point $(s,j) \in U$ has been chosen with probability $\frac{Pr[s]}{WW}$).
3. Compute $\alpha_K(s,j)$
4. Estimator $Y \leftarrow \alpha_K(s,j) \cdot WW$

The preprocessing time is at most $K-1$ times the preprocessing time for the planar two-terminal algorithm. All the steps in the trial are straightforward except for the computation of α_K. We now discuss the computation of α_K. For $s \in F'$,

$$cov(s) = \{j : s \in F'_j\}$$

For fixed s we need

$$\sum_{j \in cov(s)} \alpha_K(s,j) = 1 \ .$$

Let G_s be the graph formed from G by contracting all working edges in s. Let $j' \in \{2, \cdots, K\}$ be the smallest index such that x_1 and $x_{j'}$ are not contracted into the same node in G_s. There must be at least one such node or else s is not a failure state. Then,

$$\alpha_K(s,j) = \begin{cases} 1 & \text{if } j = j' \\ 0 & \text{if } j \neq j' \end{cases}$$

For fixed s, j' is uniquely determined. Therefore, $\sum_{j \in cov(s)} \alpha_K(s,j) = 1$. The time for this step is $O(m)$. The total time for all steps of the trial is $O(n \cdot d)$.

8. Planar All-Terminal Problem

The all-terminal problem is the special case of the K−terminal problem when all the nodes in G are specified. We could use the algorithm presented in section 7 for the all-terminal problem. However, we present a much simpler algorithm in this section for this very important special case.

8.1. Graph Concepts

State s is a **failure state** if there is a cut of failing edges in s whose removal would disconnect G. Let F be the set of all failure states in G. Let CT be the set of all cuts in G. For $ct \in CT$, let F_{ct} be the set of all states s such that all edges in ct are failing in s. Then,

$$F = \bigcup_{ct \in CT} F_{ct}$$

Let G' be the dual network of G. Let CY be the set of simple cycles in G'. For $cy \in CY$, let F'_{cy} be the set of states s in G' such that all edges in cy are failing in s. Let

$$F' = \bigcup_{cy \in CY} F'_{cy} \qquad (8.1.a)$$

Each cut $ct \in CT$ corresponds to a unique simple cycle $cy \in CY$, such that $Pr[F_{ct}] = Pr[F'_{cy}]$. Clearly, $Pr[F] = Pr[F']$. We now focus on estimating $Pr[F']$.

Let $\{e'_1, \cdots, e'_{n-1}\}$ be an ordered list of some subset of edges in G' for which the corresponding edges $\{e_1, \cdots, e_{n-1}\}$ in G form a spanning tree. Each simple cycle in G' includes a first edge e'_z from this list. Thus, we can partition CY into $CY_1, \cdots, CY_{n-1}$, where CY_z is the set of simple cycles cy such that the smallest indexed edge in cy from $\{e'_1, \cdots, e'_{n-1}\}$ is e'_z.

Let $G'_z = G' - \{e'_1, \cdots e'_z\}$, where the two endpoints of e'_z are labelled x'_z and y'_z respectively in G'_z. Let SP_z be the set of all $x'_z - y'_z$ simple paths in G'_z.

For $sp \in SP_z$, let F'_{sp} be the set of all states s in G' such that all edges in sp are failing in s. Every simple cycle $cy \in CY_z$ is a $x'_z - y'_z$ simple path $sp \in SP_z$ together with edge e'_z, such that

$$Pr[F'_{cy}] = Pr[F'_{sp}] \cdot p_z$$

Let

$$WSP_z = \sum_{sp \in SP_z} Pr[F'_{sp}], \quad WCY_z = \sum_{cy \in CY_z} Pr[F'_{cy}]$$

and

$$WCY = \sum_{cy \in CY} Pr[F'_{cy}].$$

Then, $WCY_z = WSP_z \cdot p_z$ and $WCY = \sum_{z=1}^{n-1} WCY_z$.

8.2. Estimating $Pr[F']$

F' can be written as the union of sets as in equation (8.1.a). The universe is

$$U = \{(s,cy) : cy \in CY \text{ and } s \in F'_{cy}\}$$

where $wt(s,cy) = Pr[s]$ and $wt(cy) = Pr[F'_{cy}]$. Thus, $wt(U) = WCY$. We would like to be able to implement the algorithm abstractly described in subsection 2.1. This requires the computation of WCY and a procedure to randomly choose $cy \in CY$. However, it is infeasible to compute WCY exactly. Instead, we will use the techniques developed in section 3.

Let W_z be the set of all $x'_z - y'_z$ l-restricted walks in G'_z, and let $WW_z = \sum_{wk \in W_z} wt(wk)$. Let

$$W = \bigcup_{z=1}^{n-1} \{wk + e'_z : wk \in W_z\}.$$

Notice that $CY \subseteq W$. The sample set will be

$$V = U \cup \{wk : wk \in W - CY\}$$

where, for $wk \in W-CY$, $wt(wk)$ is the weight of walk wk. Let WW be the weight of the sample set V. Then

$$wt(V) = WW = \sum_{z=1}^{n-1} WW_z \cdot p_z$$

We now present the algorithm to estimate $Pr[F']$ and then discuss the implementation details.

Algorithm to Estimate $Pr[F']$ for the Planar All-Terminal Problem

Preprocessing

For $z = 1,\ldots,n-1$ compute WW_z. (Each WW_z computation corresponds to a computation of WW in section 4)
$WW = \sum_{z=1}^{n-1} WW_z \cdot p_z$.

Trial

1. Randomly choose $z \in \{1, \cdots, n-1\}$ with probability $\frac{WW_z \cdot p_z}{WW}$.
2. Randomly choose $wk \in W_z$ with probability $\frac{wt(wk)}{WW_z}$
3. **If** wk is not simple **then** $Y \leftarrow 0$
   Else let $cy = wk + e'_z$ ($cy \in CY_z$), and continue to step 4

   (At this point $cy \in CY$ has been chosen with probability $\frac{Pr[F'_{cy}]}{WW}$)
4. Randomly choose $s \in F'_{cy}$ with probability $\frac{Pr[s]}{Pr[F'_{cy}]}$

   (At this point (s,cy) has been chosen with probability $\frac{Pr[s]}{WW}$)
5. Compute $\alpha(s,cy)$
6. Estimator $Y \leftarrow \alpha(s,cy) \cdot WW$

The preprocessing time is at most $n-1$ time the preprocessing time for the reachability problem, since there are $n-1$ WW_z computations. All the steps in the trial are straightforward except the computation of α, which we now discuss. For $s \in F'$,

$$cov(s) = \{cy \in CY : s \in F'_{cy}\}.$$

For fixed s we need

$$\sum_{cy \in cov(s)} \alpha(s,cy) = 1$$

We can use the same algorithm X as described in section 6.2 to compute α. Let G_s be the graph formed from G by contracting all working edges in s. Let the nodes in G be indexed as $x_1,x_2,\ldots,x_n$. One of the specified nodes in G_s is the node corresponding to the component containing x_1 in G. The other specified node in G_s is the node corresponding to the component in G which contains the smallest indexed node x_j not in the same component as x_1.

Compute $\alpha(s,cy)$

Form G_s from G by contracting all working edges in s.
Let $ct' \leftarrow x_1-x_j$ cut output by X when the input is G_s.
Let cy' be the simple cycle in G' corresponding to ct'

$$\alpha(s,cy) \leftarrow \begin{cases} 1 \text{ if } cy = cy' \\ 0 \text{ if } cy \neq cy' \end{cases}$$

For fixed s, X always outputs the same cut ct', which corresponds to $cy' \in cov(s)$. Therefore, $\sum_{cy \in cov(s)} \alpha(s,cy) = 1$. The time for this step is $O(m)$.

The total time for all steps of the trial is $O(n \cdot d)$.

9. Running Time Analysis for the Reachability Problem

In this section we analyze the running time for the reachability algorithm presented in sections 3 and 4. There are three components to the running time: the preprocessing time, the time to perform a trial and the number of trials, N, sufficient to guarantee an (ε,δ) algorithm.

Let d be the maximum node degree in G. We showed in section 4 that the time per trial is $O(n \cdot d)$. The preprocessing step, presented in section 4, computes WW, the sum of the weights of $x-y$ l-restricted walks. The weights of walks which have cycles of length less than l are not considered in this sum. In section 4 we showed that the preprocessing time is $O(n^2 \cdot d^{l-1})$. In this section we develop an upper bound on N. This upper bound will depend upon the value of l. We show here that for a judicious choice of l both the preprocessing time and N are small.

In all the algorithms presented, α is a zero-one variable. In section 2.3 we show that if α is a zero-one variable and

$$N = \frac{1}{E[\alpha]}(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}$$

then we have an (ε,δ) algorithm. For the reachability problem,

$$E[\alpha] = \frac{Pr[F]}{WW} = \frac{Pr[F]}{\sum_{sp\ \in SP} Pr[F_{sp}]}\cdot\frac{WSP}{WW}.$$

Thus, N is equal to

$$\frac{\sum_{sp\ \in SP} Pr[F_{sp}]}{Pr[F]} \tag{9.a}$$

times

$$\frac{WW}{WSP} \tag{9.b}$$

times $(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}$.

9.1. An Upper Bound on (9.a)

We now derive an upper bound on (9.a). Let G be a function from F to the positive real numbers such that, for $s \in F$,

$$G[s] = \prod_{e_i \text{ failing in } s} p_i$$

and let

$$G[F] = \sum_{s \in F} G[s] .$$

In chapter 2, section 12, we prove that

$$\frac{G[F]}{Pr[F]} \leq \prod_{i=1}^{m} (1 + p_i) \leq e^{\sum_{i=1}^{m} p_i} .$$

For the problems discussed in this paper the set of failure states, F, has the following **monotonicity property**. Let $s \in F$. If t is a state such that the set of edges failing in s is a subset of the edges in t (in which case we say $s \subseteq t$) then $t \in F$ also. State $s \in F$ is a **minimal failure state** if there is no state t such that $t \subseteq s$, $t \neq s$ and $t \in F$. For the reachability problem s is a minimal failure state if the failing edges in s are exactly the edges in some $x-y$ simple path sp. Thus, $G[s] = Pr[F_{sp}]$. From this discussion we conclude that

$$G[F] \geq \sum_{sp \in SP} Pr[F_{sp}]$$

and

$$\frac{\sum_{sp \in SP} Pr[F_{sp}]}{Pr[F]} \leq \prod_{i=1}^{m} (1 + p_i) \leq e^{\sum_{i=1}^{m} p_i} .$$

Thus, if the edge failure probabilities are small (which is common in practice) then so is (9.a).

9.2. An Upper Bound on (9.b)

We now derive an upper bound on $\frac{WW}{WSP}$, which will depend on the value of l. Our strategy for deriving an upper bound will be the following.

1. Associate each $x-y$ l-restricted walk uniquely with a $x-y$ simple path. This is done by defining a function

$$H : W \rightarrow SP .$$

2. We derive an upper bound on the maximum, over all $sp \in SP$, of the ratio of the sum of the weights of the walks associated with sp divided by the weight of sp. To do this we derive an upper bound on

$$\max_{sp \in SP} \frac{\sum_{wk \in H^{-1}(sp)} wt(wk)}{wt(sp)}.$$

The upper bound derived in 2 is an upper bound on $\frac{WW}{WSP}$, because

$$\frac{WW}{WSP} = \frac{\sum_{wk \in W} wt(wk)}{\sum_{sp \in SP} wt(sp)} \leq \max_{sp \in SP} \frac{\sum_{wk \in H^{-1}(sp)} wt(wk)}{wt(sp)}. \qquad (9.2.a)$$

9.2.1. The Definition of H

We first show how a $x-y$ walk can be uniquely associated with a $x-y$ simple path. Let $wk = (x=u_1,u_2,\cdots,u_{lp-1},u_{lp}=y)$ be a $x-y$ walk in G (see Figure 9.2.1.a). The $x-y$ simple path sp associated with wk will consist of disjoint connected segments of wk that, when spliced together, form a $x-y$ simple path. Simple path $sp = H(wk)$ will be created by proceeding along wk from x to y until a node u_k is repeated along the walk. The portion of the walk between the first occurrence of u_k and the second is a cycle cy. We call u_k the **origin** of cy. Cycle cy is discarded from the simple path. Let $DCY(wk)$ be the set of discarded cycles in wk when forming $H(wk)$. We add cy to $DCY(wk)$ at this point.

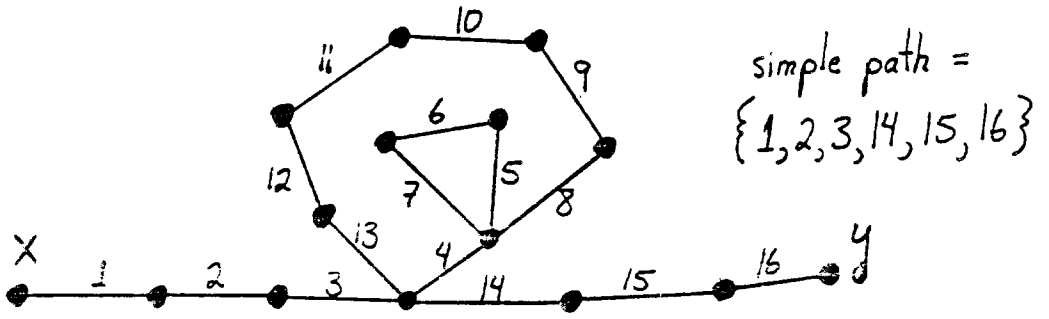


Figure 9.2.1.a - *A $x-y$ walk and the associated $x-y$ simple path*

If cy contains another cycle cy' such that $cy' \in DCY(wk)$, then cy' is deleted from $DCY(wk)$ when cy is added (see Figure 9.2.1.b).

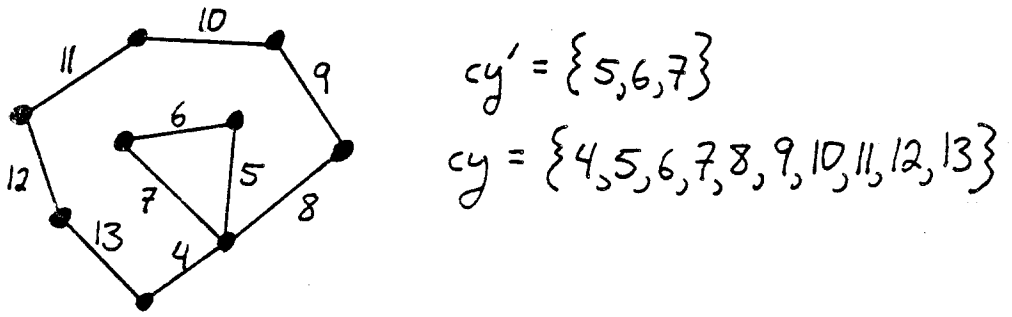


Figure 9.2.1.b - *When cy is added to $DCY(wk)$, cy' is deleted*

The extension of the simple path proceeds from u_k along walk wk. Eventually the simple path will extend from x *to* y.

Notice that the $x-y$ walk wk can be reconstructed from the $x-y$ simple path sp by splicing the cycles in $DCY(wk)$ back into sp.

9.2.2. Upper Bound Derivation

Let $adj(v)$ be the set of edges with node v as one endpoint. For this analysis, we asume there is a **constant** c strictly less than one such that

$$\sum_{e_i \in adj(v)} p_i \leq c$$

for all nodes v in G.

Theorem 9.2.2.a :

$$\frac{WW}{WSP} \leq \sum_{i=0}^{\infty} \left\lceil \frac{n \cdot c^i}{1-c} \right\rceil^i$$

Comment : We will choose l so that $\frac{n \cdot c^l}{1-c} \leq \frac{1}{2}$. In this case, $\frac{WW}{WSP} \leq 2$.

Before proving this theorem, we state and prove a lemma which is at the core of the theorem. For $k \geq 1$, for all nodes v in G, let $W(k,v)$ be the set of all walks with k edges that start at v, and let $WW(k,v)$ be the sum of the weights of all the walks in $W(k,v)$.

Lemma 9.2.2.b : For all $k \geq 1$, for all nodes v in G,

$$WW(k,v) \leq c^k$$

Proof : The proof will be by induction on k. For all nodes v in G,

$$WW(1,v) = \sum_{(v,w) \in adj(v)} p_{(v,w)} \leq c$$

by the definition of c. Suppose $WW(k-1,v) \leq c^{k-1}$ for $k-1 \geq 1$, for all nodes v in G. Then

$$WW(k,v) = \sum_{(v,w) \in adj(v)} p_{(v,w)} \cdot WW(k-1,w) \leq c \cdot c^{k-1} \leq c^k \ .$$

■

Corollary : For all v in G, the sum of the weights of all walks with at least k edges that start at v is less than or equal to $\frac{c^k}{(1-c)}$. ■

Now we return to the proof of Theorem 9.2.2.a.

Proof of Theorem 9.2.2.a :

$$\frac{WW}{WSP} \leq \max_{sp \in SP} \frac{\sum_{wk \in H^{-1}(sp)} wt(wk)}{wt(sp)}$$

by (9.2.a). Consider a generic simple path sp of length lp. We will show

$$\frac{\sum_{wk \in H^{-1}(sp)} wt(wk)}{wt(sp)} \leq \sum_{i=0}^{\infty} \left[\frac{n \cdot c^l}{1-c} \right]^i .$$

thus proving the theorem. All walks $wk \in H^{-1}(sp)$ consist of sp together with cycles of length at least l with origins along the simple path. Note that these cycles may not be simple cycles. Let NC_i be the set of all walks $wk \in H^{-1}(sp)$ such that wk has exactly i cycles, and let WNC_i be the sum of the weights of all walks in NC_i. We will show that

$$\frac{WNC_i}{wt(sp)} \leq \left[\frac{n \cdot c^l}{1-c} \right]^i ,$$

thus proving the theorem. Note that $NC_0 = \{sp\}$ and thus

$$\frac{WNC_0}{wt(sp)} = 1 .$$

For $i \geq 1$, NC_i can be further partitioned into $lp^i \leq n^i$ sets, depending upon which set of i nodes along sp are specified as the origins of the i cycles in the walks. Let $v_1, \cdots, v_i$ be a generic set of i nodes along sp. Consider the set of all walks in NC_i where $v_1, \cdots, v_i$ are the origins of the i cycles in the walks. By the Corollary to Lemma 9.2.2.b, the total weight of the cycles of length at least l with origin v_k $(1 \leq k \leq i)$ in these walks is at most $\frac{c^l}{1-c}$. It follows that the total weight of all the walks in this set is at most

$$wt(sp) \cdot \left[\frac{c^l}{1-c} \right]^i .$$

Since there are at most n^i such sets,

$$\frac{WNC_i}{wt(sp)} \leq \left\lceil \frac{n \cdot c^i}{1-c} \right\rceil^k .$$

■

9.3. The Choice of l

By Theorem 9.2.2.b,

$$\frac{WW}{WSP} \leq \sum_{i=0}^{l} \left\lceil \frac{n \cdot c^i}{1-c} \right\rceil^k .$$

We choose the value of l such that $\frac{n \cdot c^l}{1-c} \leq \frac{1}{2}$, in which case $\frac{WW}{WSP} \leq 2$. This restriction is satisfied if

$$l \geq \frac{lgn + 1 + lg\frac{1}{1-c}}{lg\frac{1}{c}} .$$

We point out that this value of l is easily computable from the problem input.

Now we analyze the preprocessing time for this choice of l. For simplicity, assume that $c \leq \frac{1}{3}$. In this case, the value of l can be chosen so that

$$l \leq \frac{lgn}{lg\frac{1}{c}} + 1 . \qquad (9.3.a)$$

In section 4 we show that the preprocessing time is $O\left(n^2 \cdot d^{l-1}\right)$. If we substitute the right-hand side of inequality (9.3.a) for l we see that the preprocessing time is

$$O\left(n^{2 + \frac{lgd}{lg\frac{1}{c}}}\right) .$$

If $d \cdot c \leq 1$ the preprocessing time is $O(n^3)$, but in any case it is $O(n^{lgn})$ since $d \leq n$.

9.4. Summary of the Running Time

If there is a $c \leq \frac{1}{3}$ (bounds can be derived for all $c < 1$) such that

$$\sum_{e_i \in adj(v)} p_i \leq c$$

for all nodes v in G, then the preprocessing time is

$$O\left[n^{2+\frac{\lg d}{\lg \frac{1}{c}}}\right],$$

and the number of trials, N, sufficient to guarantee an (ε,δ) algorithm is

$$2\cdot\prod_{e_i}(1+p_i)\cdot(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}.$$

The time per trial is $O(n\cdot d)$.

Note that all these bounds are easily computable from the problem input. The bounds on the preprocessing time and on the number of trials both contain a lot of slack. The actual performance of the algorithm should be significantly better than projected by these bounds. However, this analysis proves that the algorithm works especially well when the edge failure probabilities are small.

10. Running Time Analysis for the Planar Two-Terminal, All-Terminal and K-Terminal Algorithms

The running time analysis for the planar two-terminal, all-terminal and K-terminal algorithms is very similar to the analysis for the reachability problem. One of the differences is that for the planar problems the algorithms are used on the dual network G' instead of on the original network G. The analysis is in terms of the dual graph G'. Thus, we let d be the maximum node degree in G' (which is the maximum number of edges bordering any face in G) and we let c be a constant such that for all nodes v in G',

$$\sum_{e'_i \in adj(v)} p_i \leq c$$

(which is the same as, for all faces v in G, $\sum_{e_i \text{ bordering } v} p_i \leq c$). There are three components to the running time: the preprocessing time, the time to perform a trial and the number of trials, N, sufficient to guarantee an (ε,δ) algorithm. For all the algorithms the running time per trial is $O(n \cdot d)$.

10.1. Preprocessing Time

The preprocessing time for the reachability problem is the basis of the preprocessing time analysis for the other algorithms. Let PT denote the upper bound on the preprocessing time for the reachability problem. Let ls be the length of the shortest $x-y$ path in G for the two-terminal problem (where the shortest path may be artificially shortened by the addition of dummy edges, as explained in subsection 6.4). For each edge in this path the preprocessing step performs the dynamic programming algorithm presented in subsection 6.3 on a graph formed by deleting edges from G'. The running time for each dynamic programming algorithm is at most the same order as PT. Therefore, the preprocessing time for the two-terminal algorithm is

$$O\,(ls \cdot PT)\,.$$

The analysis for the preprocessing time for the all-terminal algorithm is very similar. The preprocessing step first finds a spanning tree consisting of $n-1$ edges in G. For each edge in this tree the dynamic programming algorithm presented in section 4 is executed on a graph formed by deleting edges from G'. The total preprocessing time is

$$O\,(n \cdot PT)\,.$$

Let ls_i be the length of the shortest x_1-x_i path in G for the K-terminal problem. For each specified node x_i (where $i = 2,\ldots,K$) the preprocessing step executes ls_i dynamic programming algorithms of the type presented in subsec-

tion 6.3. The running time for each x_i is

$$O\,(ls_i \cdot PT)$$

and thus the total preprocessing time is

$$O\left[\sum_{i=2}^{K} ls_i \cdot PT\right]$$

10.2. Number of Trials

The upper bounds on the preprocessing time and the number of trials, N, sufficient to guarantee an (ε,δ) algorithm are intimately linked thru the variable l. In subsection 2.3 we show that if α is a zero-one variable and $N = \frac{1}{E[\alpha]}(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}$ then we have an (ε,δ) algorithm.

For the planar two-terminal algorithm,

$$E[\alpha] = \frac{Pr[F']}{WW} = \frac{Pr[F']}{\sum_{cy\,\in\,CY} Pr[F'_{cy}]}\,\frac{WCY}{WW}$$

Thus, N is less than or equal to

$$\frac{\sum_{cy\,\in\,CY} Pr[F'_{cy}]}{Pr[F']} \qquad (10.2.a)$$

times

$$\frac{WW}{WCY} \qquad (10.2.b)$$

times $(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}$. Using the same reasoning as in subsection 9.1, we get an upper bound on (10.2.a) of

$$\prod_{i=1}^{m}(1+p_i) \le e^{\sum_{i=1}^{m} p_i}$$

To derive an upper bound on (10.2.b) we use the same technique as was

used in section 9.2. The difference in the analysis is how a walk is uniquely associated with a simple cycle. Each l-restricted walk in W is a walk that starts and ends at the same node in G' which includes an odd number of edges from $\{e'_{i_1}, \cdots, e'_{i_s}\}$. We associate each walk in W uniquely with a simple cycle which includes an odd number of edges from $\{e'_{i_1}, \cdots, e'_{i_s}\}$ in a manner similar to that used in subsection 9.2. If walk wk is not a simple cycle then wk can be partitioned into two walks wk_1 and wk_2 such that both wk_1 and wk_2 start and end at the same common node. Since wk contains an odd number of edges from $\{e'_{i_1}, \cdots, e'_{i_s}\}$ then either wk_1 or wk_2 must also contain an odd number of edges from $\{e'_{i_1}, \cdots, e'_{i_s}\}$. Suppose that wk_1 contains an odd number of edges from $\{e'_{i_1}, \cdots, e'_{i_s}\}$. We add wk_2 to the set of discarded cycles and recursively try to associate wk with a simple cycle occurring within wk_1. The rest of the analysis follows directly from subsection 9.2.

The result is that if $c \leq \frac{1}{3}$ and l is chosen to be $\frac{lgn}{lg\frac{1}{c}} + 1$, then

$$PT = O\left[n^{2+\frac{lgd}{lg\frac{1}{c}}}\right] \tag{10.2.c}$$

and

$$N = 2 \cdot \prod_{i=1}^{m} (1+p_i) \cdot (\ln \frac{2}{\delta}) \cdot \frac{4}{\varepsilon^2} \tag{10.2.d}$$

Thus, the total preprocessing time for the planar two-terminal algorithm is

$$O\left[ls \cdot n^{2+\frac{lgd}{lg\frac{1}{c}}}\right]$$

and the total time for all trials is

$$O\left[2 \cdot \prod_{i=1}^{m} (1+p_i) \cdot (\ln \frac{2}{\delta}) \cdot \frac{4}{\varepsilon^2} nd\right] .$$

The analysis for the planar all-terminal algorithm follows directly from subsections 9.1 and 9.2. The result is that if $c \leq \frac{1}{3}$ and l is chosen to be $\frac{lgn}{lg\frac{1}{c}} + 1$, then both (10.2.c) and (10.2.d) hoid. Thus, the total preprocessing time for the planar all-terminal algorithm is

$$O\left[n \cdot n^{2+\frac{lgd}{lg\frac{1}{c}}}\right]$$

and the total time for all trials is

$$O\left[2 \cdot \prod_{i=1}^{m} (1+p_i) \cdot (\ln\frac{2}{\delta}) \cdot \frac{4}{\varepsilon^2} nd\right] .$$

For the planar K-terminal algorithm, let

$$WCY_j = \sum_{cy \in CY_j} wt(cy) = \sum_{cy \in CY_j} Pr[F'_{cy}]$$

Then,

$$E[\alpha] = \frac{Pr[F']}{WW} = \frac{Pr[F']}{\sum_{j=2}^{K} \sum_{cy \in CY_j} Pr[F'_{cy}]}$$

times

$$\frac{\sum_{j=2}^{K} WCY_j}{\sum_{j=2}^{K} wt(V_j)} .$$

Thus, N is less than or equal to

$$\frac{\sum_{j=2}^{K} \sum_{cy \in CY_j} Pr[F'_{cy}]}{Pr[F']} \qquad (10.2.e)$$

times

$$\frac{\sum_{j=2}^{K} wt(V_j)}{\sum_{j=2}^{K} WCY_j} \qquad (10.2.f)$$

times $(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}$. Using the same reasoning as in subsection 9.1 we see that

$$\frac{\sum_{cy \in CY_j} Pr[F'_{cy}]}{Pr[F'_j]} \leq \prod_{i=1}^{m}(1+p_i) .$$

Then, since $Pr[F'_j] \leq Pr[F']$, (10.2.e) is bounded above by

$$(K-1)\cdot\prod_{i=1}^{m}(1+p_i) .$$

The upper bound on (10.2.f) is derived by placing an upper bound on

$$\frac{wt(V_j)}{WCY_j} \text{ , for } j = 2,\ldots,K .$$

The upper bound on $\frac{wt(V_j)}{WCY_j}$ is the same as the upper bound on (10.2.b). The result is that if $c \leq \frac{1}{3}$ and l is chosen to be $\frac{lgn}{lg\frac{1}{c}} + 1$, then (10.2.c) holds and

$$N = 2\cdot(K-1)\cdot\prod_{i=1}^{m}(1+p_i)\cdot(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2} .$$

Thus, the total preprocessing time for the planar K-terminal algorithm is

$$O\left[\sum_{i=2}^{K} ls_i\cdot n^{2+\frac{lgd}{lg\frac{1}{c}}}\right]$$

and the total time for all the trials is

$$O\left[2\cdot(K-1)\cdot\prod_{i=1}^{m}(1+p_i)\cdot(\ln\frac{2}{\delta})\cdot\frac{4}{\varepsilon^2}nd\right] .$$

11. Improvements and Fine Tuning of the Reachability Algorithm

There are several input parameters to the reachability algorithm which affect its performance. The proper choice of these parameters can substantially increase the efficiency of the algorithm in terms of the ratio of the running time to the accuracy of the estimator. In this section we highlight these parameters

and indicate some heuristic guidelines for fine tuning the algorithm.

Throughout this paper we have assumed the estimator produced by one trial of the algorithm is the zero-one variable α multipiied by an appropriate constant. As we pointed out in this paper, an unbiased estimate of the failure probability is produced in one trial if

$$\sum_{sp \in cov(s)} \alpha(s,sp) = 1 \ .$$

In chapter 2, sections 8 and 9, we discuss a hybrid method for computing α. We now briefly present the hybrid method for computing $\alpha(s,sp)$. Let **cutoff** be a positive integer. Let

$$nc = min(cov(s),cutoff) \ .$$

We first list nc simple paths occurring among the failing edges in state s. An algorithm to list these simple paths is straightforward. Let

$$\{sp_1, \cdots, sp_{nc}\}$$

be the simple paths listed by the algorithm. Then,

$$\alpha(s,sp) = \begin{cases} \frac{1}{nc} \text{ if } sp \in \{sp_1, \cdots, sp_{nc}\} \\ 0 \quad otherwise \end{cases} \ .$$

This definition of α produces an unbiased estimator of $Pr[F]$ with lower variance at the cost of more computation per trial. A judicious choice of the value of $cutoff$ could substantially improve the performance of the algorithm. This same technique can also be applied to the planar two-terminal, K-terminal and all-terminal algorithms.

One choice which greatly affects the behavior of the algorithm is the choice of the value of l. In this paper we prove theoretical results about the choice of l, but in practice the theoretical value of l seems to be too large. A large value

of l can make the preprocessing prohibitively expensive with little reduction in the sum of the weights of the walks. Our limited experience with the algorithm to date suggests a small value of l is always appropriate for the first run of the algorithm. One heuristic suggestion is to first try $l = 1$ and compute the sum of the weights of the walks, then try $l = 2$ and compare the time to compute the preprocessing for this value of l with the reduction in the sum of the weights of the walks (which directly shows how much the variance per trial decreases). This will indicate whether or not it is appropriate to proceed to larger values of l.

12. References

[2] J. Scott Provan and Michael O. Ball, *The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected*, working paper MS/S 81-002, Management Science and Statistics, January 1981 (revised April 1981)

[3] Renyi, A., *Probability Theory*, North-Holland Publishing Company, Amsterdam, 1970

[4] Garey, M.R., Johnson, D.S. and Tarjan, R.E., *The Planar Hamiltonian Circuit Problem is NP-Complete*, SIAM J. of Computing, 5, 1976, pp. 702-714

[5] S. Tsukiyama, I. Shirakawa, H. Ozaki, H. Ariyoshi, *An Algorithm to Enumerate All Cutsets of a Graph in Linear Time*, JACM, vol. 27, no. 4, October 1980, pp. 619-632

Appendix 1 - PASCAL Implementation of the Reachability Algorithm

We now present a PASCAL implementation of the coverage algorithm for the reachability problem discussed in sections 3 and 4.

Program Input

1. The *Seed* for the random number generator. The program uses the system supplied function *random* to generate a sequence of numbers uniformly distributed between zero and one. The function *seed* with the value *Seed* is used to initialize the random number generator.
2. The number of trials to run.
3. The value of l. *WW* will be the sum of all l*–restricted walks*.
4. The value of *cutoff*. *Cutoff* is described in section 11.
5. The output indicator. The value of zero produces standard output, the value of one produces more detailed output that is used primarily for debugging the program.
6. The number of nodes, *numnodes*, in the graph G. The convention is that nodes 1 and *numnodes* are automatically considered to be the specified nodes x and y.
7. The edges in the graph. Each edge is input as a triple i , j , p. The edge has as its endpoints nodes i and j, and the failure probability of the edge is p. The triple 0 , 0 , 0 indicates the end of the edge input.

Program Output

1. Data concerning the construction of the dynamic programming structure. Each line of data is of the form

$$l = \quad \#dynrecord = \quad \#predrecords = \quad time =$$

The dynamic programming structure for *l−restricted walks* is constructed from the structure for *l*−1*−restricted walks*. The time is the number of seconds used to construct the structure for *l* from the structure for *l*−1 (in the case *l*=1, this is the time to initialize the structure for *l*=1). *#dyn-records* and *#predrecords* are values representing the total amount of space dynamically allocated so far for the construction of the structure.

2. Data concerning the computation of WW using the dynamic programming structure. The *time* is the number of seconds used to compute WW. *#car-ray* represents the size of the array necessary to hold the intermediate values of C in the computation (the size is in increments of 1000 real words).
3. The time in seconds to run all the trials.
4. The fraction of yes answers is the sum of the α values for all the trials divided by the number of trials.
5. The sum of the walk weights is the value of WW.
6. The average value of the estimator is the value of Y estimating PF produced by the program.
7. The total running time of the program in seconds.

Main Program logic

1. The input parameters are read.
2. The graph input is read and the edge list graph structure is constructed (*readgraph*).
3. The dynamic programmng structure for i=1 is constructed from the edge lists (*convgrdyn*).

4. The dynamic programming structure for $i+1$ is constructed from the dynamic programming structure for i (*dynlenti*1). This step is repeated until the dynamic programming structure for l has been constructed.

5. The dynamic programming structure is used to compute the value of WW (*computeC*).

At this point the preprocessing is complete, and the running of the trials begins. Each trial consists of

6. Choose a walk j from node *numnodes* to 1 using the dynamic programming solution (*choosewalk*).

7. Choose a state s of the graph in which the edges in the walk chosen in step 6 are failing (*choosestate*).

8. Compute the value of $\alpha(s,j)$ (*alpha*).

9. Total the result of $\alpha(s,j)$ from this trial with the results from the previous trials (*yesans* = *yesans* + x). After all the trials have been completed, the estimator Y is computed and output.

Global Data Structures and Variables

noderec records - These are the records for the nodes in the edge list representation of the graph. *Nodearr* is an array of these records, where *nodearr*[i] is the nodes record for node i. *Numnodes* is the largest index for any node record, and is also the index for node y (node x is node 1). *Topedge* is the pointer to the beginning of the edge list for the node. Both *listptr* and *lastnode* are used by the function *alpha* when performing a depth first search of the graph. *Listptr* keeps track of which edge in the edge list shold be examined next. *Lastnode* points to the predecessor node in the depth first search tree.

edgerec records - These are the records for the edges in the edge list representation of the graph. These records are dynamically allocated as they are needed. Edge (i, j) appears on the edge lists for both nodes i and j. *Nodep* is an array of size two used to store the indices of the nodes i and j. *Edgep* is an array of size two used to point to the next edge on the edge lists of nodes i and j respectively. *Prob* is the edge failure probability. Each node has a pointer (*topedge*) to the first edge on its edge list, which does not correspond to a real edge but is used exclusively as the top of the edge list record for the edge; *topedge* .*edgep*[1] points to the first real edge on the edge list and *topedge* .*edgep*[2] points to the last real edge on the edge list. *Use* is used to indicate the status of the edge as follows:

use	meaning
-1	edge is the *topedge* for the edge list of some node
0	no choice for the edge has been made
1	edge has been chosen to fail in state s
2	edge has been chosen to fail in state s
3	edge is in the chosen walk j

Dyneptr is an array of size two used in the construction of the dynamic programming structure for $l=1$. *Dyneptr* points to the (up to two) *dynrec* records created from the edge. An example of the data structures associated with storing a graph is shown in Figure A.1.1 and Tables A.1.2 and A.1.3.

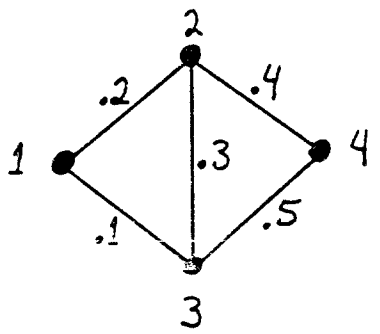


Figure A.1.1 - *Example graph G*

e1, ... ,e9 pointers to edgerec records						
edgerec pointer	prob	use	nodep		edgep	
			1	2	1	2
e1	0	-1	1	0	e5	e6
e2	0	-1	2	0	e6	e8
e3	0	-1	3	0	e5	e9
e4	0	-1	4	0	e8	e9
e5	0.1	0	1	3	e6	e7
e6	0.2	0	2	1	e7	e1
e7	0.3	0	3	2	e9	e8
e8	0.4	0	2	4	e2	e9
e9	0.5	0	3	4	e3	e4

Table A.1.2 -*Example of data structure for storage of graph G*

nodearr	
index	topedge
1	e1
2	e2
3	e3
4	e4

Table A.1.3 -*Example of data structure for storage of graph G*

dynrec records - These are the records for the dynamic programming structure, where each record corresponds to a partial walk. These records are dynamically allocated as they are needed. *Numdyn* is the number of records in use so far. *Dyngarb* is a pointer to the top of a list used for garbage collection. If a *dynrec* is no longer needed it is linked into this list for reuse. *Stnode* and *endnode* are the first and last nodes in the partial walk, and *lastedge* is the pointer to the edge record for the last edge.

Auxptr and *auxptr*1 are used to build and link together the dynamic programming structure. After the structure has been build, L is the pointer to the top of the list of all partial walks, where *auxptr* is the pointer used to link together this list. *Topred* is the pointer to a linked list of *predrec* records for the partial walk. The *predrec* records point to the predecessor

Typdyn indicates the type of partial walk.	
typdyn	meaning
start	node 1 is the first node in the partial walk, i.e. the walk is an initial partial walk
terminal	Node *numnodes* is the last node in the partial walk, i.e. the walk is a terminal partial walk
both	Node 1 is the first node and node *numnodes* is the last node in the partial walk,i.e. the walk is a simple path from node 1 to node *numnodes*.
neither	A partial walk which is neither an initial nor a terminal partial walk

partial walks if the partial walk is of type *terminal* or *neither*. The *predrec* records point to the edges in the partial walk if the partial walk is of type *start* or *both*. *Cval* is equal to the product of the probabilities of the edges in the partial walk. *Cptr* and *cindex* are used to point to an array of real variables used to store the values of *C* for the dynamic programming computation. *Cptr* points to an array of 1000 real variables, and *cindex* is an indice in this array which is the first indice used by the partial walk for the storage of its *C* values.

predrec records - These are the records used to point to predecessor partial walks (if the partial walk is of type *terminal* or *neither*) or to the edges in the partial walk (if the partial walk is of type *start* or *both*). These records are dynamically allocated as they are needed. *Numpred* is the number of records in use so far. *Predgarb* is a pointer to the top of a list used for garbage collection. If a *predrec* record is no longer needed it is linked into this list for reuse. The *predrec* records are linked together via the *npredrec* pointer. If a partial walk is of type *terminal* or *neither*, then the *topred* pointer for the partial walk points to a linked list of *predrec* records, one record for each predecessor partial walk. The *todyn* pointers of the *predrec* records are used to point to the predecessor partial walk. If a partial walk is of type *start* or *both* then the *topred* pointer for the partial walk points to a linked list of *predrec* records, one record for each edge in the

partial walk. The *toedge* pointer of the *predrec* are used to point to the edges. In this case the *predrec* lists are not necessarily distinct for each partial walk because the first few edges in a partial walk may agree with the edges in some other partial walk. This is why the *predrec* records pointed to by a partial walk *dynrec* record which is of type *start* or *both* cannot be returned to the *predgarb* list when the *dynrec* record is returned to the *dyngarb* list (see procedure *retdyn*).

crec records - These are the arrays of real variables used to compute the dynamic programming values of C. These arrays are allocated dynamically as they are needed. *Sizec* is the size of each array (currently set at 1000). *Numc* is the number of arrays allocated so far. Each partial walk which is of type *terminal* or *neither* is allocated *maxlength* consecutive real variables in a *crec* record, where *maxlength* is an upper bound of the length of the longest simple path from node 1 to node *numnodes* in G (*maxlength* is set to *numnodes* for simplicity in the program). In the *dynrec* record for the partial walk, *cptr* points to the *crec* record and *cindex* is the beginning indice of the real variables allocated to the partial walk in the *crec* record array.

termrec records - T is an array (currently of size 5000) of *termrec* records used to choose the terminal portion of a walk as the first step in choosing a walk at random. Each entry in T corresponds to a partial walk of type *terminal* or *both*, where t is the number of such partial walks. *Termdyn* points to the *dynrec* record corresponding to the partial walk and *termprob* is the sum of the weights of all walks from node 1 to node *numnodes* where the last portion of the walk is the partial walk.

fwalkarr array - This is an array used to store the pointers to the edges in the walk chosen. *Flenwalk* is the length of the walk chosen.

Other Global Variables

name	use
looplength	The dynamic programming structure is constructed to sum the weights of all *looplength–restricted walks*.
i	i is incremented from 1 to *looplength*.
cutoff	The maximum number of simple walks to look for in the chosen state s
Seed	The seed for the random number generator
printind	Indicates standard output (0) or debugging output (1).
numtrials	The number of trials to be performed
trial	*trial* is the current trial number in progress
totaltime	The total running time of the program in seconds
intertime	The running time for various portions of the program
x	The returned value of *alpha* is stored here
yesans	The sum of the α values produced by the trials
WW	The sum of the weights os all *looplength–restricted walks*
Y	The estimator of $Pr[S]$

Detailed Description of the Most Important Procedures and Functions

readgraph procedure - This procedure read in the number of nodes in the graph, *numnodes*, and the edges in the graph. The edge list representation of the graph is constructed as the edges are read in (see Figure A.1.1, and Tables A.1.2 and A.1.3)

convgrdyn procedure - This procedure creates the dynamic programming structure for $i=1$ from the edge list representation of the graph. This is a two stage process. In the first stage the edge list for every node in the graph is traversed. Suppose the edge list for node *fnode* is being traversed, and edge (*fnode* , *lnode*) is being examined. A partial walk (*dynrec* record) with *stnode* equal to *fnode*, *endnode* equal to *lnode*, and *lastedge* pointing to (*fnode* , *lnode*) is created if *fnode* is not node *numnodes* and *lnode* is not node 1. If *fnode* = 1 the partial walk is of type *start* or *both* and a *predrec* record is created to point to edge (*fnode* = *lnode*). The *dyneptr* pointer for the edge points to the newly created partial walk record, this pointer is used in stage two.

Each edge gives rise to either one or two partial walks; two if neither *fnode* nor *lnode* are node 1 or node *numnodes*, and one otherwise. Each *dynrec* record is added to the list L via the *auxptr*.

In stage two the list L created in stage one is traversed to create the pointers to the predecessor partial walks. Each partial walk which is of type *neither* or *terminal* will have predecessor partial walks. The links to these predecessors are created by traversing the edge list for node *stnode* and using the *dyneptr* pointer for the edges set in stage one as the pointer to the predecessors.

*dynlenii*1 procedure - This procedure creates the dynamic programmng structure for $i+1$ from the structure for i. This is a three stage process. The first stage is to create the *dynrec* records for $i+1$ from the *dynrec* records for i. We will use the following terminology in this description. An *i-partial walk* is a partial walk which is of length i. A partial walk which is of type *terminal* or *neither* will be called a *noninitial* partial walk, while a partial walk which is of type *start* or *both* will be called an *initial* partial walk. Each *noninitial i-partial walk x* will mate with each of its predecessor *i-partial walks y* to create one *child $i+1$-partial walk yx*, where x will be called the *primary* parent and y will be called the *secondary* parent of yx. Consider the partial graph shown in Figure A.2.1. The dynamic programming structure for $i=3$ relevant to this example is shown in Tables A.2.2 and A.2.3. We will show how the structure for $i=4$ is created, concentrating in particular on the *dynrec* record corresponding to the *3-partial walk* (7,8,9,10). Predecessor partial walks (6,7,8,9) and (3,7,8,9) will mate with (7,8,9,10) to create children (6,7,8,9,10) and (3,7,8,9,10) respectively. Suppose (3,7,8,9) was an *initial* partial walk,

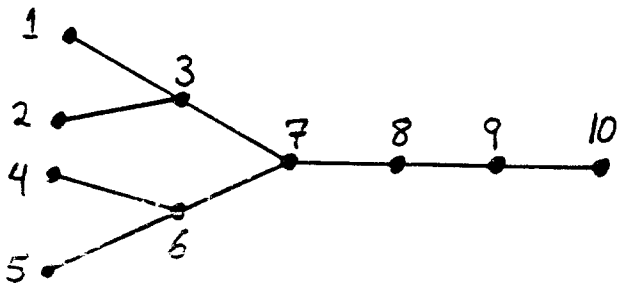


Figure A.2.1 - *Part of a graph to demonstrate dynamic programming structure*

dynrec records L = d1					
dynrec pointer	stnode	endnode	lastedge	auxptr	topred
d1	7	10	(9,10)	d2	p1
d2	6	9	(8,9)	d3	p2
d3	3	9	(8,9)	d4	p3
d4	4	8	(7,8)	d5	-
d5	5	8	(7,8)	d6	-
d6	1	8	(7,8)	d7	-
d7	2	8	(7,8)	-	-

Table A.2.2 - *Dynamic programming structure for* $i=3$

predrec records		
predrec pointer	npredrec	todyn
p1	p4	d2
p2	p5	d4
p3	p6	d6
p4	nil	d3
p5	nil	d5
p6	nil	d7

Table A.2.3 - *Dynamic programming structure for* $i=3$

then the child partial walk (3,7,8,9,10) will also be an *initial* partial walk. In this case a *predrec* record is created to point to edge (9,10) and this record is linked into the list of *predrec* records points to edges (3,7),(7,8), and (8,9) from (3,7,8,9). On the other hand, if (as is

true in the example shown) (3,7,8,9) is a *noninitial* partial walk then (3,7,6,9,10) is of the same type as (7,8,9,10). The primary parent is linked to its children via the *auxptr* 1 pointers. Each child points to its secondary parent using the *auxptr* pointer. See Table A.2.4 for the relevant dynamic programming structures after the completion of stage one.

dynrec records L = d1						
dynrec pointer	stnode	endnode	lastedge	auxptr	topred	auxptr1
d1	7	10	(9,10)	d2	p1	d8
d2	6	9	(8,9)	d3	p2	d10
d3	3	9	(8,9)	d4	p3	d12
d4	4	8	(7,8)	d5	-	-
d5	5	8	(7,8)	d6	-	-
d6	1	8	(7,8)	d7	-	-
d7	2	8	(7,8)	-	-	-
d8	3	10	(9,10)	d3	-	d9
d9	6	10	(9,10)	d2	-	nil
d10	5	9	(8,9)	d5	-	d11
d11	4	9	(8,9)	d4	-	nil
d12	2	9	(8,9)	d7	-	d13
d13	1	9	(8,9)	d6	-	nil

Table A.2.4 - *Relevant structure after stage one*

The second stage of *dynlenii* 1 is to create the links to the predecessor partial walks for the newly created children. The list of *i-partial walks* is traversed. For each *noninitial i-partial walk*, which is the primary parent of the children on the list pointed to by *auxptr* 1, the child list is traversed to create the predecessor links for the children. A child *yx* will have predecessors only if its secondary parent *y* is a *noninitial* partial walk. If secondary parent *y* is a *noninitial* partial walk, then the prececessors of *yx* are the children *ey* for which *y* is the primary parent, as long as the *stnode* of *ey* is not the same as the *endnode* of *yx*. See Tables A.2.5 and A.2.6 for the relevant data structure after the completion of state two.

dynrec records L = d1						
dynrec pointer	stnode	endnode	lastedge	auxptr	topred	auxptr1
d1	7	10	(9,10)	d2	p1	d8
d2	6	9	(8,9)	d3	p2	d10
d3	3	9	(8,9)	d4	p3	d12
d4	4	8	(7,8)	d5	-	-
d5	5	8	(7,8)	d6	-	-
d6	1	8	(7,8)	d7	-	-
d7	2	8	(7,8)	-	-	-
d8	3	10	(9,10)	d3	p7	d9
d9	6	10	(9,10)	d2	p9	nil
d10	5	9	(8,9)	d5	-	d11
d11	4	9	(8,9)	d4	-	nil
d12	2	9	(8,9)	d7	-	d13
d13	1	9	(8,9)	d6	-	nil

Table A.2.5 - *Relevant structure after stage 2*

predrec records		
predrec pointer	npredrec	todyn
p1	p4	d2
p2	p5	d4
p3	p6	d6
p4	nil	d3
p5	nil	d5
p6	nil	d7
p7	p8	d13
p8	nil	d12
p9	p10	d11
p10	nil	d10

Table A.2.6 - *Relevant structure after stage 2*

Stage three of *dynlenii* 1 discards the old *i–partial walks* (unless they are of type *both*, in which case they are always saved) and links the children of *i*+1*–partial walks* together via the *auxptr* pointers into the list *L*. In the example, *dynrec* records $d1,d2,d3,d4,d5,d6$ and $d7$ and *predrec* records $p1,p2,p3,p4,p5$ and $p6$ are discarded and *dynrec* records $d8,d9,d10,d11,d12$ and $d13$ are linked together via *auxptr* on list *L*.

computeC procedure - This procedure performs the dynamic programming calculation of C, which in turn yields the value of *WW*. The first step is to allocate space for each *noninitial* partial walk to calculate and save the values of C. The number of real variables allocated for each partial walk is *maxlength*, where *maxlength* is set equal to *numnodes*. *Crec* records are allocated to provide this space.

The second step is to compute C for walks of length $l2 = l1 + 1$ from the C values for walks of length $l1$ as described in section 5. The value of $l1$ is not the actual walk length, the true walk length is $l1$ + *looplength*. However, since no walk of length less than or equal to *looplength* can have as the final portion of the walk a *noninitial looplength–partial walk*, the C values for walks of length 1,...,*looplength* are all zero and need not be computed. When $l1 = 0$ only the *initial* partial walks will contribute their weight to the C values for ancestor partial walks. When $l1 > 0$ only *noninitial* partial walks will contribute their C values to the C values of their ancestors.

The third step is to accumulate the C values, so that C ($l1$, *partial walk*) is the sum of the weights of all walks of length less than or equal to $l1$ + *looplength* for which the final portion of the walk is *partial walk*.

The fourth step is to calculate *WW* from the sum of the weights of the walks for which the last portion of the walk is a *noninitial* partial walk. In addition, the array T is initialized, where $T[i]$.*termdyn* points to the *dynrec* record for the *noninitial* partial walk i, and $T[i]$.*probab* is the sum of the weights of the walks for which the last portion of the walk is partial walk i.

The final step is to revise the *probab* entries in the T array so that procedure *choosewalk* can use the T array to choose the terminal portion of a walk with the appropriate probability as described in section 5.

choosewalk procedure - This procedure is used to choose a walk such that walk j is chosen with probability $\frac{wt(j)}{\bar{w}\bar{\pi}}$. This algorithm is described in section 5. The first step is to initialize the *use* of all edges in the edge list structure to zero. The edges in the chosen walk will have their *use* set to three to indicate they are in the walk. The second step is to choose the terminal portion of the path using the T array initialized by *computeC* and using binary search as described in section 5. Once the terminal portion of the walk has been selected the rest of the walk is selected as described in section 5. *Fwalkarr* is used to store the pointers to the edges in the walk chosen. *Flenwalk* is the length of the chosen walk.

choosestate procedure - This procedure chooses a state s of the graph once the walk j has been chosen, as described in section 4.

alpha real function - This function computes the value of $\alpha(s,j)$, as described in section 12. A depth first search of the graph is executed using only edges chosen to fail in *choosestate* (*use* = 1) or edges in the chosen walk (*use* = 3). Every time a simple path i is found from node 1 to node *numnodes*, the edges in i are compared with the edges in walk j to see if they are the same. The pointers to the edges in j are stored in reverse order in the *fwalkarr* array. The edges in i are listed in reverse order using the *lastnode* value to point to the previous node in the path and *listptr* to the edge in the path. If a match is found then *matchpath* is set to true. The graph is explored until *numfnd*, the number of simple paths from node 1 to node *numnodes*

found so far among the failing edges in the graph, is equal to *cutoff*; or until all the simple paths in the graph are found. The value of *alpha* is then $\frac{1}{numfnd}$ if walk *j* was among the simple paths listed (*matchpath* = *true*) or zero otherwise.

Brief Description of Other Procedures and Functions

function or procedure	use
setclok	a real function (written in the C language) to initialize the running time to zero.
readclok	a real function (written in the C language) which returns the running time in seconds since the last time setclok was called.
seed (Seed)	an integer function (system supplied) which initializes the seed value to *Seed* for the random number generator.
random (Seed)	a real function (system supplied) which returns a random real number uniformly distributed between zero and one
newdyn (x)	allocates *x*, a new *dynrec* record
newpred (x)	allocates *x*, a new *predrec* record
retdyn (x)	returns for reuse *x*, a *dynrec* record. If *x* is a *noninitial* partial walk then the *predrec* records are also returned for reuse.
printgraph	prints the edge list structure, used only for debugging the program
direc (edgeptr, nodenum)	an integer function which returns the value of the indice in the array *nodep* for edge *edgeptr* which points to node *nodenum*.
nextedge (mask, lastedge,nodenum)	a function which returns an edge pointer to the next edge after *lastedge* on node *nodenum's* edge list. The *mask* is a boolean variable. The function is called with *mask* = *true* from the function *alpha* to ignore edges which are not failing in the state *s*.
othernode (edge, nodenum)	an integer function which returns the node number which is at the other endpoint of edge *edge* from *nodenum*, where *nodenum* is one of the endpoints of edge *edge*.

```
program monte2sim (input,output) ;

{ this programs estimatates the probability there is a path between
  two specified nodes in an undirected graph where edges have failure
  probabilities associated with them- the reachability problem   }

type eptr = ^ edgerec ;
  dynptr = ^ dynrec ;
  predptr = ^ predrec ;
  ptrc = ^ crec ;
  crec = record
    c : array [0..999] of real
  end ;
  dynrec = record
    stnode : integer ;
    endnode : integer ;
    typdyn : (start,terminal,both,neither) ;
    auxptr : dynptr ;
    auxptr1 : dynptr ;
    topred : predptr ;
    lastedge : eptr ;
    cval : real ;
    cptr : ptrc ;
    cindex : integer
  end ;
  predrec = record
    npredrec : predptr ;
    todyn : dynptr ;
    toedge : eptr
  end ;
  termrec = record
    termdyn : dynptr ;
    termprob : real
  end ;
  edgerec = record
    prob : real ;
    use : integer ;
    nodep : array [1..2] of integer ;
    edgep : array [1..2] of eptr ;
    dyneptr : array [1..2] of dynptr
end ;
  noderec = record
    topedge : eptr ;
    listptr : eptr ;
    lastnode : integer
end ;
var numc,sizec,numpred,numdyn,i,flenwalk,cutoff,t,numnodes : integer ;
    T : array [0..5000] of termrec ;
    nodearr : array [1..100] of noderec ;
    fwalkarr : array [1..100] of eptr ;
    Seed,printind,looplength,maxlength,numtrials,trial : integer ;
    totaltime,intertime,x,yesans,WW,Y : real ;
    L,dyngarb : dynptr ;
    predgarb : predptr ;

#include "decls.h"
function direc (edgeptr : eptr ; nodenum : integer) : integer ;
var i : integer ;
begin
    if (edgeptr^.nodep[1] = nodenum) then i := 1 ;
    if (edgeptr^.nodep[2] = nodenum) then i := 2 ;
    direc := i
end  { direc } ;

function nextedge (mask:boolean;lastedge:eptr;nodenum:integer):eptr ;

{mask - true - find next edge on edge list with use=3 or use=1 or use=-1
        false - find next edge on edge list regardless of use
 lastedge - current edge on edge list of nodenum
 nodenum - node whose edge list is being traversed }

var j:integer ; edge:eptr ; correct:boolean ;
begin
   edge := lastedge ;
   repeat
      j := direc (edge,nodenum) ;
      edge := edge^.edgep[j] ;
      correct := true ;
      if mask then correct := ((edge^.use = 1) or (edge^.use = 3) or
                               (edge^.use = -1))
   until (correct) ;
   nextedge := edge
end { nextedge } ;

function othernode (edge:eptr; nodenum:integer):integer ;
var i,j:integer ;
```

```
begin
   i := direc(edge,nodenum) ;
   if (i=1) then j:=2
   else j :=1 ;
   othernode := edge^.nodep[j]
end { othernode } ;

procedure newdyn (var x:dynptr) ;

{ used to get a new dynrec record }

begin
   numdyn := numdyn + 1 ;
   if (dyngarb = nil) then new(x)
   else begin
      x := dyngarb ;
      dyngarb := x^.auxptr ;
      x^.auxptr := nil
   end {else}

end {newdyn} ;

procedure newpred (var x:predptr) ;

{ used to get a new predrec record }

begin
   numpred := numpred + 1 ;
   if (predgarb = nil) then new(x)
   else begin
      x := predgarb ;
      predgarb := x^.npredrec ;
      x^.npredrec := nil
   end {else}
end {newpred} ;

procedure retdyn (var x:dynptr) ;

{ used to return a dynrec record, and also some predrec records attached
  to it as long as they are not pointed to by any other dynrec record,
   i.e. x^.typdyn <> start or both }

var z,w : predptr ;
begin
   x^.auxptr := dyngarb ;
   dyngarb := x ;
   numdyn := numdyn - 1 ;
   if ((x^.typdyn <> start) and (x^.typdyn <> both)) then begin
      w := x^.topred ;
      while (w<>nil) do begin
         z := w^.npredrec ;
         w^.npredrec := predgarb ;
         predgarb := w ;
         numpred := numpred - 1 ;
         w := z
      end {while (w<>nil) }
   end { if ((x^.typdyn <> start) and (x^.typdyn <> both)) }
end {retdyn} ;

procedure convgrdyn ;

{ converts the graph to the first stage of the dynamic programming
  procedure format }

var usetype : boolean ;
    k,onode,fnode,lnode : integer ;
    edge,edge1 : eptr ;
    x,y : dynptr ;
    z : predptr ;
begin
   usetype := false ;
   L := nil ;
   for fnode := 1 to numnodes do begin
      edge := nodearr[fnode].topedge ;
      edge1 := nextedge(usetype,edge,fnode) ;
      while (edge1 <> edge) do begin
         lnode := othernode (edge1,fnode) ;
         if ((fnode <> numnodes) and (lnode <> 1)) then begin
            newdyn (x) ;
            k := direc (edge1,lnode) ;
            edge1^.dyneptr[k] := x ;
            x^.auxptr := L ;
            L := x ;
            x^.auxptr1 := nil ;
            x^.stnode := fnode ;
            x^.endnode := lnode ;
```

```
                x^.lastedge := edge1 ;
                x^.cval := edge1^.prob ;
                x^.topred := nil ;
                if (fnode=1) then begin {its a start state}
                   newpred (z) ;
                   x^.topred := z ;
                   z^.npredrec := nil ;
                   z^.todyn := nil ;
                   z^.toedge := edge1 ;
                   if (lnode=numnodes) then x^.typdyn := both
                   else x^.typdyn := start ;
                end {if start state}
                else if (lnode=numnodes) then x^.typdyn := terminal
                     else x^.typdyn := neither
             end {if ((fnode <> numnodes) and (lnode <> 1)) } ;
             edge1 := nextedge (usetype,edge1,fnode)
          end { while (edge1 <> edge) }
       end {for fnode := 1 to numnodes } ;

{ L = list of all states - now initialize predecessors }

   x := L ;
   while (x <> nil) do begin
      if ((x^.typdyn = neither) or (x^.typdyn = terminal)) then begin
         fnode := x^.stnode ;
         lnode := x^.endnode ;
         edge := nodearr[fnode].topedge ;
         edge1 := nextedge (usetype,edge,fnode) ;
         while (edge1 <> edge) do begin
            onode := othernode (edge1,fnode) ;
            if ((onode <> lnode) and (onode <> numnodes)) then begin
               k := direc (edge1,fnode) ;
               y := edge1^.dyneptr[k] ;
               newpred (z) ;
               z^.toedge := nil ;
               z^.npredrec := x^.topred ;
               z^.todyn := y ;
               x^.topred := z
            end {if ((onode <> lnode) and (onode <> numnodes)) } ;
            edge1 := nextedge (usetype,edge1,fnode)
         end {while (edge1 <> edge) }
      end { if ((x^.typdyn = neither) or (x^.typdyn = terminal)) } ;
      x := x^.auxptr
   end { while (x <> nil) }
end { convgrdyn } ;

procedure dynlen111 ;

{ builds dynamic programming structure excluding loops of length i+1
  from the structure excluding loops of length i - where i is implicitly
  increased by one each time this routine is called   }

var z,w : predptr ;
    x,y,yx,ey : dynptr ;

begin

{ create next level of states }

x := L ;
while (x <> nil) do begin
   x^.auxptr1 :=nil ;
   if ((x^.typdyn = terminal) or (x^.typdyn = neither)) then begin
      z := x^.topred ;
      while (z <> nil) do begin
         y := z^.todyn ;
         newdyn (yx) ;
         yx^.auxptr := y ;
         yx^.auxptr1 := x^.auxptr1 ;
         x^.auxptr1 := yx ;
         yx^.stnode := y^.stnode ;
         yx^.endnode := x^.endnode ;
         yx^.lastedge := x^.lastedge ;
         yx^.cval := y^.cval * x^.lastedge^.prob ;
         yx^.topred := nil ;
         if (y^.typdyn = neither) then yx^.typdyn := x^.typdyn
         else begin  {y^.typdyn = start}
            if (x^.typdyn = neither) then yx^.typdyn := start
            else yx^.typdyn := both ;
            newpred (w) ;
            w^.toedge := x^.lastedge ;
            w^.todyn := nil ;
            w^.npredrec := y^.topred ;
            yx^.topred := w
         end {y^.typdyn = start} ;
         z := z^.npredrec
```

```
            end { while (z<>nil) }
         end { if ((x^.typdyn = terminal) or (x^.typdyn = neither)) } ;
         x := x^.auxptr
      end { while (x <> nil) } ;

      { create predecessor links }

      x := L ;
      while (x <> nil) do begin
         if ((x^.typdyn = neither) or (x^.typdyn = terminal)) then begin
            yx := x^.auxptr1 ;
            while (yx <> nil) do begin
               y := yx^.auxptr ;
               if (y^.typdyn = neither) then begin
                  ey := y^.auxptr1 ;
                  while (ey <> nil) do begin
                     if (ey^.stnode <> yx^.endnode) then begin
                        newpred (w) ;
                        w^.toedge := nil ;
                        w^.todyn := ey ;
                        w^.npredrec := yx^.topred ;
                        yx^.topred := w
                     end { if (ey^.stnode <> yx^.endnode) } ;
                     ey := ey^.auxptr1
                  end { while ( ey <> nil) }
               end { if (y^.typdyn = neither) } ;
               yx := yx^.auxptr1
            end { while (yx <> nil) }
         end { if ((x^.typdyn = neither) or (x^.typdyn = terminal)) } ;
         x := x^.auxptr
      end { while (x <> nil) } ;

      { link up next level of states onto L }

      x := L ;
      L := nil ;
      while (x <> nil) do begin
         y := x^.auxptr ;
         if (x^.typdyn = both) then begin
            x^.auxptr := L ;
            L := x ;
            x^.auxptr1 := nil
         end { if (x^.typdyn = both) }
         else begin  {x^.typdyn <> both}
            yx := x^.auxptr1 ;
            retdyn (x) ;
            while (yx <> nil) do begin
               ey := yx^.auxptr1 ;
               yx^.auxptr := L ;
               L := yx ;
               yx^.auxptr1 := nil ;
               yx := ey
            end {while (yx <> nil) }
         end { x^.typdyn <> both } ;
         x := y
      end { while (x <> nil) }
      end { dynlen111 } ;

      procedure computeC ;

      { performs the dynamic programming procedure once the structure has been
        built and initialized }

      var x,y : dynptr ;
          w : predptr ;
          cpoint : ptrc ;
          indc,i,i1,i2 : integer ;
          sum2,sum,probab : real ;
      begin
          indc := sizec ;

      { initialize C values }

      x := L ;
      while (x <> nil) do begin
         if ((x^.typdyn = neither) or (x^.typdyn = terminal)) then begin
            if (indc+maxlength >= sizec) then begin
               indc := 5 ;
               numc := numc + 1 ;
               new (cpoint)
            end {if (indc+maxlength >= sizec) } ;
            x^.cptr := cpoint ;
            x^.cindex := indc ;
            indc := indc + maxlength ;
            x^.cptr^.c[x^.cindex] := 8.8
         end { if ((x^.typdyn = neither) or (x^.typdyn = terminal)) } ;
```

```
        x := x^.auxptr
      end { while ( x <> nil ) } ;

      { compute C values for all lengths }

      for l1 := 0 to maxlength - 2 do begin
        l2 := l1 + 1 ;
        x := L ;
        while (x <> nil) do begin
          if ((x^.typdyn = neither) or (x^.typdyn = terminal)) then begin
            sum := 0.0 ;
            probab := x^.lastedge^.prob ;
            w := x^.topred ;
            while (w <> nil) do begin
              y := w^.todyn ;
              if ((y^.typdyn = start) and (l1 = 0)) then sum := sum + y^.cval ;
              if (y^.typdyn = neither) then sum := sum + y^.cptr^.c[y^.cindex + l1] ;
              w := w^.npredrec
            end { while (w <> nil) } ;
            x^.cptr^.c[x^.cindex + l2] := sum * probab
          end { x^.typdyn = neither or x^.typdyn = terminal } ;
          x := x^.auxptr
        end { while (x <> nil) }
      end {for l1 := 1 to maxlength - 1 } ;

      { accumulate C values so that C[i] is sum of values from 1 to i  }

      x := L ;
      while (x <> nil) do begin
        if ((x^.typdyn = neither) or (x^.typdyn = terminal)) then
          for l1 := 1 to maxlength -1 do x^.cptr^.c[x^.cindex + l1] :=
              x^.cptr^.c[x^.cindex + l1] + x^.cptr^.c[x^.cindex + l1 - 1] ;
        x := x^.auxptr
      end { while (x <> nil) } ;

      { build up list of terminal states }

      sum := 0.0 ;
      x := L ;
      while (x <> nil) do begin
        if ((x^.typdyn = both) or (x^.typdyn = terminal)) then begin
          t := t + 1 ;
          T[t].termdyn := x ;
          if (x^.typdyn = both) then probab := x^.cval
          else probab := x^.cptr^.c[x^.cindex + maxlength - 1] ;
          T[t].termprob := probab ;
          sum := sum + probab
        end { if ((x^.typdyn = both) or (x^.typdyn = terminal)) } ;
        x := x^.auxptr
      end { while (x <> nil) } ;
      WW := sum ;

      { normalize terminal states so that they add up to one }

      sum2 := 0.0 ;
      for i := 1 to t do begin
        sum2 := sum2 + T[i].termprob / sum ;
        T[i].termprob := sum2
      end { for i := 1 to t }
      end { computeC } ;

      procedure choosewalk ;

      { chooses a walk at random from the dynamic programming solution }

      var found,usetype,choose : boolean ;
          low,high,pnt1,pnth,i,j,l2 : integer ;
          w : predptr ;
          x,y : dynptr ;
          save,sum2,sum : real ;
          edge,edge1 : eptr ;

      begin

      { choose a path }

      { first initialize use to zero for all edges }

        usetype := false ;
        flenwalk := 0 ;
        for i := 1 to numnodes do begin
          edge := nodearr[i].topedge ;
          edge1 := nextedge (usetype,edge,i) ;
          while (edge1 <> edge) do begin
            edge1^.use := 0 ;
            edge1 := nextedge(usetype,edge1,i)
```

```
            end { while (edge1 <> edge) }
         end { for i := 1 to numnodes } ;

      { pick terminal state from T }

         save := random(Seed) ;
         found := false ;
         low := 0 ;
         high := t ;
         while ( not found) do begin
            pnt1 := (low + high) div 2 ;
            pnth := pnt1 + 1 ;
            if (T[pnt1].termprob >= save) then high := pnt1 ;
            if (T[pnth].termprob < save) then low := pnth ;
            if ((T[pnt1].termprob <= save) and (T[pnth].termprob >= save))
               then found := true
         end { while ( not found) } ;
         j := pnth ;

      { now pick the rest of the walk }

         l2 := maxlength ;
         y := T[j].termdyn ;
         while ((y^.typdyn = neither) or (y^.typdyn = terminal)) do begin
            flenwalk := flenwalk + 1 ;
            edge := y^.lastedge ;
            fwalkarr[flenwalk] := edge ;
            edge^.use := 3 ;
            sum2 := 0.0 ;
            sum := y^.cptr^.c[y^.cindex + l2 - 1] / edge^.prob ;
            l2 := l2 - 1 ;
            w := y^.topred ;
            save := random(Seed) ;
            choose := false ;
            while ( not choose) do begin
               x := w^.todyn ;
               if ((x^.typdyn = neither) or (x^.typdyn = terminal)) then
                  sum2 := sum2 + x^.cptr^.c[x^.cindex + l2 - 1]
               else sum2 := sum2 + x^.cval ;
               if (save <= (sum2 / sum)) then begin
                  y := x ;
                  choose := true
               end { if (save <= (sum2 / sum)) }
               else w := w^.npredrec
            end { while (not choose) }
         end { while ((y^.typdyn = neither) or (y^.typdyn = terminal)) } ;

      { at this stage y is a start state }

         w := y^.topred ;
         while (w <> nil) do begin
            edge := w^.toedge ;
            edge^.use := 3 ;
            flenwalk := flenwalk + 1 ;
            fwalkarr[flenwalk] := edge ;
            w := w^.npredrec
         end { while (w <> nil) }
      end { choosewalk } ;

      procedure readgraph ;

      var edge,edge1,edge2,edge1b,edge2b : eptr ;
          i,node1,node2: integer ;
          probab : real ;
      begin
          writeln ('input the reachability problem graph') ;
          writeln ('first input the number of nodes in the graph') ;
          readln (numnodes) ;
          writeln ('the number of nodes is : ',numnodes:4) ;
          for i := 1 to numnodes do begin
              new (edge) ;
              edge^.use := -1 ;
              edge^.edgep[1] := edge ;
              edge^.edgep[2] := edge ;
              edge^.nodep[1] := i ;
              edge^.nodep[2] := 0 ;
              nodearr[i].topedge := edge ;
              nodearr[i].listptr := edge ;
              nodearr[i].lastnode := 0
          end ;
          writeln ('now input the edges one at a time as i , j, probability ') ;
          writeln ('(the last edge is 0 0 0)') ;
          i := 0 ;
          repeat
              i := i + 1 ;
              readln (node1,node2,probab) ;
```

```
        writeln ('edge ',i:4,' is : ( ',node1:3,' , ',node2:3,' , ',
                 probab:6:4,' )') ;
        if (node1 > 0) then begin
            new (edge) ;
            edge^.nodep[1] := node1 ;
            edge^.nodep[2] := node2 ;
            edge^.prob := probab ;
            edge1 := nodearr[node1].topedge ;
            edge2 := nodearr[node2].topedge ;
            edge1b := edge1^.edgep[2] ;
            edge2b := edge2^.edgep[2] ;
            edge1b^.edgep[direc(edge1b,node1)] := edge ;
            edge2b^.edgep[direc(edge2b,node2)] := edge ;
            edge^.edgep[1] := edge1 ;
            edge^.edgep[2] := edge2 ;
            edge1^.edgep[2] := edge ;
            edge2^.edgep[2] := edge ;
            edge^.prob := probab ;
            edge^.use := 0
        end
    until (node1 = 0)
end { readgraph } ;

procedure printgraph ;

var i,m :integer ; edge, edge2 : eptr ; usetype : boolean ;
{ now print out the graph }
begin
    usetype := false ;
    for i := 1 to numnodes do
    begin
        writeln ;
        writeln (' node ',i:4,' lastptr ',nodearr[i].lastnode:4) ;
        edge := nodearr[i].topedge ;
        edge2 := nextedge (usetype,edge,i) ;
        while (edge2 <> edge) do begin
            m := othernode (edge2,i) ;
            write ('(',m:4,',',edge2^.prob:6:2,',', edge2^.use:2,') ') ;
            edge2 := nextedge(usetype,edge2,i)
        end
   end
end { printgraph } ;

procedure choosestate ;

{  use = -1 - top of edge list
          0 - unchosen edge
          1 - chosen edge and is failing
          2 - chosen edge and is working
          3 - edge is in the walk }

var i:integer ; probab:real ; usetype:boolean ; edge,edge1:eptr ;
begin
   usetype := false ;

{ choose state }

   for i := 1 to numnodes do
   begin
      nodearr[i].lastnode := 0 ;
      edge := nodearr[i].topedge ;
      edge1 := nextedge (usetype,edge,i) ;
      while (edge1 <> edge ) do
      begin
         if (edge1^.use = 0) then
         begin
            probab := random (Seed) ;
            if (edge1^.prob >= probab) then edge1^.use := 1
            else edge1^.use := 2
         end ;
         edge1 := nextedge (usetype,edge1,i)
      end
   end
end { choosestate } ;

function alpha : real ;

{ this function lists numfnd = min ( cov(s), cutoff) paths between
  nodes 1 and numnodes,
  the value of alpha is  1 / numfnd if chosen walk is among these paths,
  otherwise the value of alpha is 0. }

var edge:eptr ; stopsearch,matchpath,usetype,success :boolean ;
    savenode,numfnd,curnode,j : integer ;
    x : real ;
begin
```

```
    numfnd := 0 ;
    matchpath := false ;
    success := true ;
    usetype := true ;
    curnode := 1 ;
    edge := nodearr[1].topedge ;
    nodearr[1].lastnode := 1 ;
    repeat { until ((not success) or (numfnd >= cutoff)) }
    repeat { until ((not success) or (curnode = numnodes)) }
       edge := nextedge (usetype,edge,curnode) ;
       nodearr[curnode].listptr := edge ;
       if ( edge = nodearr[curnode].topedge) then

{ exhausted edge list - back up }

          if (curnode = 1) then success := false
          else begin
             savenode := curnode ;
             curnode := nodearr[curnode].lastnode ;
             nodearr[savenode].lastnode := 0 ;
             edge := nodearr[curnode].listptr
          end
       else begin
          j := othernode (edge,curnode) ;
          if (nodearr[j].lastnode = 0) then

{ new unexplored node found - explore from it }

          begin
             nodearr[j].lastnode := curnode ;
             curnode := j ;
             edge := nodearr[curnode].topedge
          end
       end
    until ((not success) or (curnode = numnodes)) ;
    if (curnode = numnodes) then numfnd := numfnd + 1 ;
    if ((curnode = numnodes) and (not matchpath)) then begin

{ see if walk is equal to this path from node I to numnodes }

       stopsearch := false ;
       j := 0 ;
       savenode := numnodes ;
       while (not stopsearch) do begin
          j := j + 1 ;
          savenode := nodearr[savenode].lastnode ;
          if ((j=flenwalk) or (savenode=1)) then stopsearch := true ;
          if (fwalkarr[j] <> nodearr[savenode].listptr) then stopsearch := true
       end {while (not stopsearch) } ;
       if (fwalkarr[j] = nodearr[savenode].listptr) then matchpath := true
    end {if ((numnodes = curnodes) and (not matchpath)) } ;

{ now back up from numnodes and continue search for more paths }

    savenode := curnode ;
    curnode := nodearr[curnode].lastnode ;
    nodearr[savenode].lastnode := 0 ;
    edge := nodearr[curnode].listptr
    until ((not success) or (numfnd >= cutoff)) ;
    if (matchpath) then x := 1.0 / numfnd
    else x := 0.0 ;
    alpha := x
end { alpha } ;

{ this is the beginning of the main program }

begin
    L := 0 ;
    T[0].termprob := 0.0 ;
    T[0].termdyn := nil ;
    L := nil ;
    writeln (' this is monte2sim.p in action, enter the seed') ;
    readln (Seed) ;
    writeln (Seed:8) ;
    trial := seed(Seed) ;
    writeln (' enter the number of trials for simple simulations') ;
    readln (numtrials) ;
    writeln (numtrials:6) ;
    writeln ('enter the value of shortest loop allowable in dynamic programming');
    readln (looplength) ;
    writeln (looplength:6) ;
    writeln ('enter the value of the cutoff') ;
    readln (cutoff) ;
    writeln (cutoff:4) ;
    writeln ('input the output indicator: 0 - short, 1 - long') ;
    readln (printind) ;
```

```
writeln (printind:4) ;

sizec := 1000 ;
numc := 0 ;
numpred := 0 ;
numdyn := 0 ;
dyngarb := nil ;
predgarb := nil ;

readgraph ;       {***}
if (printind=1) then printgraph ;
maxlength := numnodes ;

totaltime := 0.0 ;
intertime := setclok ;

convgrdyn ;       {***}
if (printind=1) then printgraph ;

i := 1 ;

intertime := readclok ;
totaltime := totaltime + intertime ;
writeln ;
writeln ('i = ',i:2,' #dynrecords = ',numdyn:6,' #predrecord = ',
     numpred:6,' time = ',intertime:12:4) ;
intertime := setclok ;

while (i < looplength) do begin
   dynlen111 ;   {***}

   i := i + 1 ;
   intertime := readclok ;
   totaltime := totaltime + intertime ;
   writeln ;
   writeln ('i = ',i:2,' #dynrecords = ',numdyn:6,' #predrecord = ',
        numpred:6,' time = ',intertime:12:4) ;
   intertime := setclok ;

   if (printind=1) then printgraph

end { while ( i < looplength ) } ;

computeC ;        {***}

intertime := readclok ;
totaltime := totaltime + intertime ;
writeln ;
writeln ('#carray = ',numc:4,' time to compute WW = ',intertime:12:4) ;
intertime := setclok ;

if (printind=1) then printgraph ;

yesans := 0.0 ;

for trial := 1 to numtrials do begin

   choosewalk ;       {***}
   if (printind=1) then printgraph ;

   choosestate ;      {***}
   if (printind=1) then printgraph ;

   x := alpha ;       {***}
   if (printind=1) then writeln ('the value of alpha is: ',x:12:8) ;

   yesans := yesans + x ;   {***}

end  {for trial := 1 to numtrials} ;

Y := yesans / numtrials ;

intertime := readclok ;
totaltime := totaltime + intertime ;
writeln ;
writeln ('time to run all the trials is = ',intertime:12:4) ;

writeln ;
writeln ('the fraction of yes answers is: ',Y) ;
writeln ('the sum of the walk weights is: ',WW) ;
Y :=Y * WW ;
writeln ('the average value of the estimator is: ',Y) ;

writeln ;
writeln ('the total running time is = ',totaltime:12:4)
```

```
end {monte2sim}.
```

```
#include <sys/types.h>
#include <sys/times.h>

static struct tms *p, timeblok;
static int start;

float setclok()          /* returns 0 */
{
p = &timeblok;
times(p);
start = p->tms_utime + p->tms_stime;
return( 0.0 );
}

float readclok()         /* system time + user time in seconds */
{
float now;
p = &timeblok;
times(p);
now = p->tms_utime + p->tms_stime - start;
return( now / 60.0 );
}
```

```
function setclok: real; external;
function readclok: real; external;
```

Appendix 2 - Sample Runs of the PASCAL Program

In this appendix we describe six example networks and the results of the reachability algorithm presented in appendix 1 when presented with these networks. In examples 1,2 and 3 the underlying network is series-parallel. We did not apply any reductions to the networks before running the algorithm. These examples were chosen because it is easy to calculate the exact failure probability of the network to check the results of the Monte Carlo algorithm.

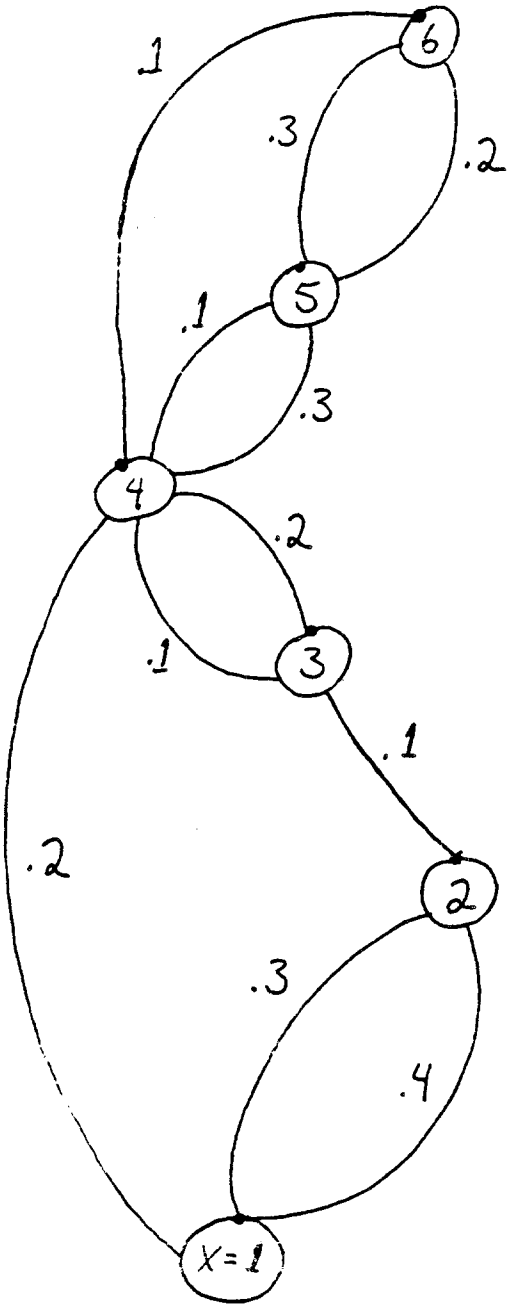
As we described in section 5, when the network is $x-y$ planar we can create a network G' from the original network G such that the failure probability for the reachability problem on G' is the same as the failure probability for the two-terminal problem on G. For examples 4 and 5 we are interested in estimating the failure probability for the two-terminal problem. Since both networks are $x-y$ planar, we use the reachability algorithm on the network G' created from the original network G. Notice that example 4 is the sample problem used in chapter 2.

To get an idea about how well the Monte Carlo algorithm performed on these examples, we first give the exact failure probabilities. The exact failure probabilities are $5.251 \cdot 10^{-2}$, $1.634 \cdot 10^{-4}$, $8.948 \cdot 10^{-3}$, $2.1254 \cdot 10^{-1}$ and $2.91 \cdot 10^{-5}$ for the first five examples respectively. We did not compute the exact failure probability for example 6. However, based on the output and the following discussion, the estimate is probably very close to the actual failure probability of the network.

We can get some idea about how well the Monte Carlo algorithm performs by concentrating on two parameters in the output. The first parameter is "the sum of the walk weights", which we call WW. If the $cutoff$ is one, WW measures how well the reachability algorithm performs compared to straight simulation. The reachability algorithm needs to perform only WW times as many trials as need

by straight simulation to get the same level of confidence and degree of accuracy in the estimator. Example 3 was chosen to be an especially bad example. Since WW is exactly one, the reachability algorithm and straight simulation are expected to perform exactly the same. If WW is greater than one then the reachability algorithm actually performs worse than straight simulation.

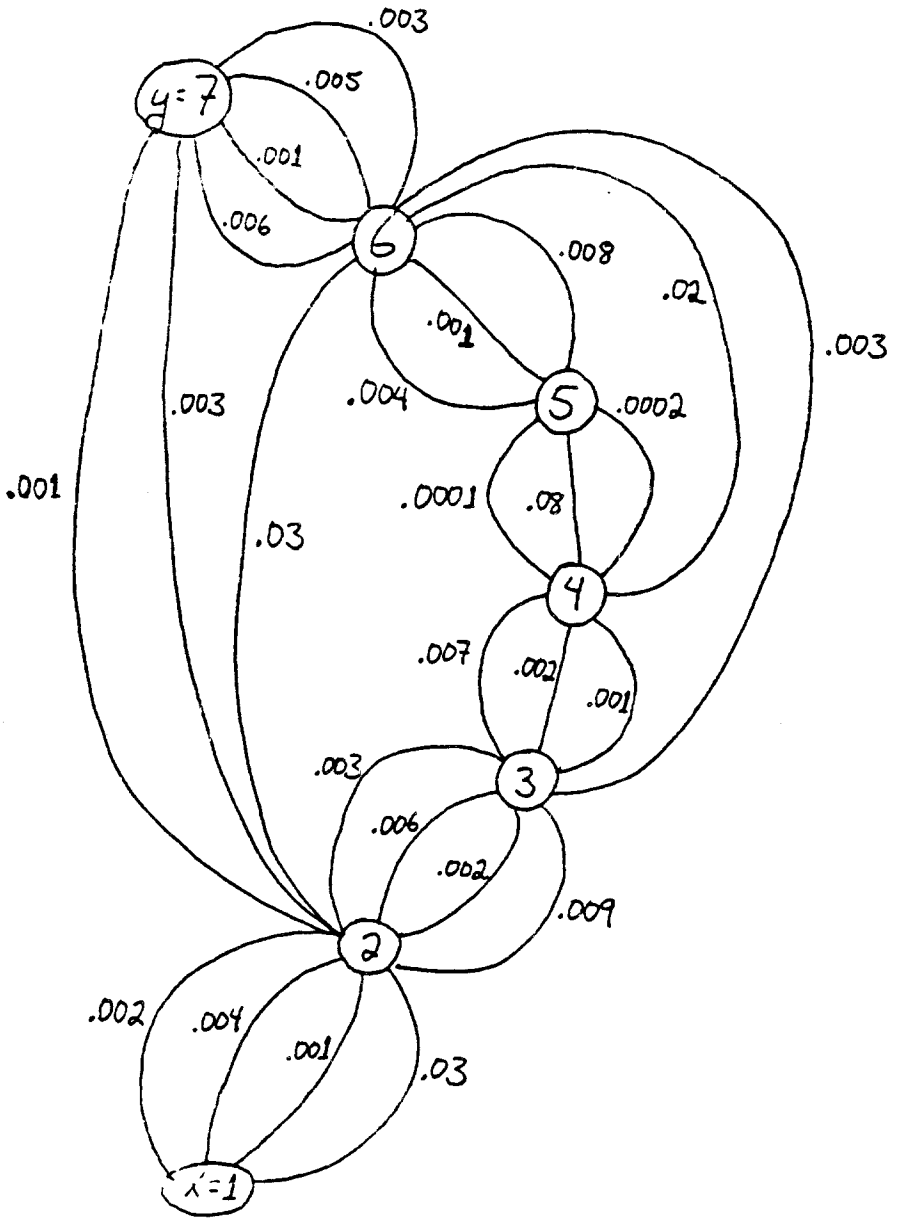
The second parameter of interest is "the fraction of yes answers", which we call X. The estimator of the failure probability is X multiplied by WW. Thus, the expected value of X is $\frac{Pr[F]}{WW}$. The closer $\frac{Pr[F]}{WW}$ is to one the better the Monte Carlo algorithm should perform. Thus, if X is close to one this indicates that $\frac{Pr[F]}{WW}$ is close to one and thus the algorithm should perform well. We note that the closeness of X to one only gives an indication about how the algorithm is performing. The value of X cannot be used directly to determine the number of trials sufficient to produce an (ε,δ) algorithm.



Example 1

```
this is monte2sim.p in action, enter the seed
  58483
enter the number of trials for simple simulations
  300
enter the value of shortest loop allowable in dynamic programming
   1
enter the value of the cutoff
  1
input the output indicator: 0 - short, 1 - long
  0
input the reachibility problem graph
first input the number of nodes in the graph
the number of nodes is :     6
now input the edges one at a time as i , j, probability
(the last edge is 0 0 0)
edge     1 is : (   1 ,   2 , 0.4000 )
edge     2 is : (   1 ,   2 , 0.3000 )
edge     3 is : (   2 ,   3 , 0.1000 )
edge     4 is : (   3 ,   4 , 0.2000 )
edge     5 is : (   3 ,   4 , 0.1000 )
edge     6 is : (   1 ,   4 , 0.2000 )
edge     7 is : (   4 ,   5 , 0.1000 )
edge     8 is : (   4 ,   5 , 0.3000 )
edge     9 is : (   5 ,   6 , 0.3000 )
edge    10 is : (   5 ,   6 , 0.2000 )
edge    11 is : (   4 ,   6 , 0.1000 )
edge    12 is : (   0 ,   0 , 0.0000 )

1 =  1 #dynrecords =      16 #predrecord =      30 time =        0.0333

#carray =     1 time to compute WW =        0.0167

time to run all the trials is =        2.3167

the fraction of yes answers is:  7.56666666666667e-01
the sum of the walk weights is:  6.63000000000000e-02
the average value of the estimator is:  5.01670000000000e-02

the total running time is =        2.3667
```

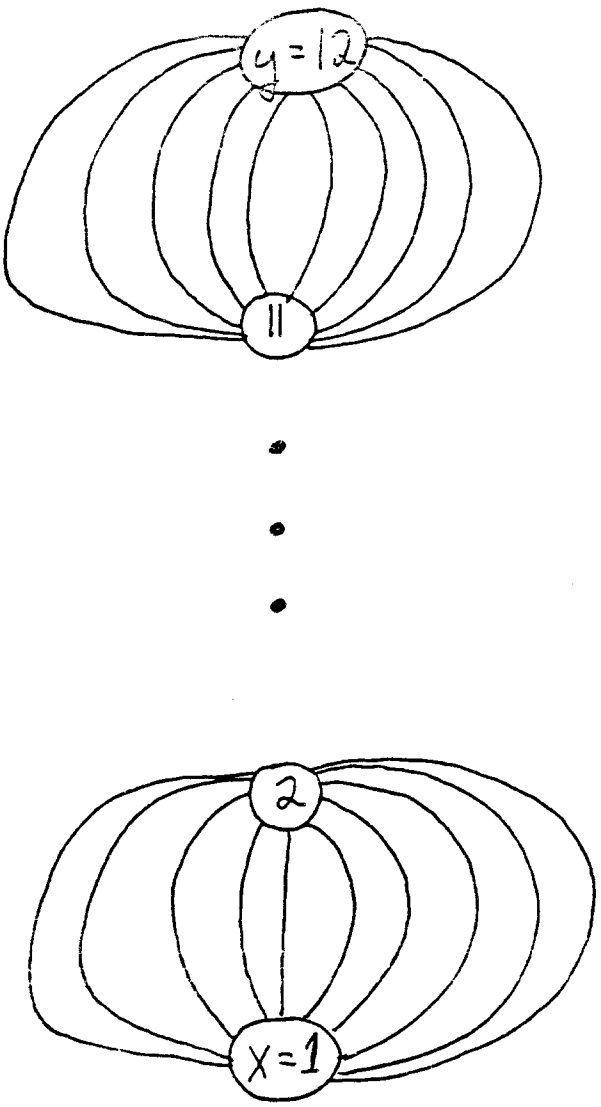
Example 1


y=7
6
5
4
3
2
x=1
.003
.005
.001
.006
.008
.02
.003
.001
.004
.0002
.003
.001
.0001
.08
.03
.007
.002
.001
.003
.006
.002
.009
.002
.004
.001
.03

Example 2

```
 this is monte2sim.p in action, enter the seed
1625268129
 enter the number of trials for simple simulations
   300
enter the value of shortest loop allowable in dynamic programming
     3
enter the value of the cutoff
    1
input the output indicator: 0 - short, 1 - long
   0
input the reachibility problem graph
first input the number of nodes in the graph
the number of nodes is :      7
now input the edges one at a time as i , j, probability
(the last edge is 0 0 0)
edge     1 is : (    1 ,    2 , 0.0020 )
edge     2 is : (    1 ,    2 , 0.0040 )
edge     3 is : (    1 ,    2 , 0.0010 )
edge     4 is : (    1 ,    2 , 0.0300 )
edge     5 is : (    2 ,    3 , 0.0030 )
edge     6 is : (    2 ,    3 , 0.0060 )
edge     7 is : (    2 ,    3 , 0.0020 )
edge     8 is : (    2 ,    3 , 0.0090 )
edge     9 is : (    3 ,    4 , 0.0070 )
edge    10 is : (    3 ,    4 , 0.0020 )
edge    11 is : (    3 ,    4 , 0.0010 )
edge    12 is : (    4 ,    5 , 0.0001 )
edge    13 is : (    4 ,    5 , 0.0800 )
edge    14 is : (    4 ,    5 , 0.0002 )
edge    15 is : (    5 ,    6 , 0.0040 )
edge    16 is : (    5 ,    6 , 0.0010 )
edge    17 is : (    5 ,    6 , 0.0080 )
edge    18 is : (    4 ,    6 , 0.0200 )
edge    19 is : (    3 ,    6 , 0.0030 )
edge    20 is : (    2 ,    6 , 0.0300 )
edge    21 is : (    6 ,    7 , 0.0060 )
edge    22 is : (    6 ,    7 , 0.0010 )
edge    23 is : (    6 ,    7 , 0.0050 )
edge    24 is : (    6 ,    7 , 0.0030 )
edge    25 is : (    2 ,    7 , 0.0030 )
edge    26 is : (    2 ,    7 , 0.0010 )
edge    27 is : (    0 ,    0 , 0.0000 )

l =   1 #dynrecords =      42 #predrecord =     184 time =        0.1333

l =   2 #dynrecords =     180 #predrecord =     630 time =        0.4000

l =   3 #dynrecords =     606 #predrecord =    1624 time =        1.9333

#carray =     4 time to compute WW =        0.9833

time to run all the trials is =        3.8333

the fraction of yes answers is:  9.93333333333333e-01
the sum of the walk weights is:  1.64685637831152e-04
the average value of the estimator is:  1.63587733578944e-04

the total running time is =        7.2833
```

Example 2



Example 3

```
 this is monte2sim.p in action, enter the seed
   42389
 enter the number of trials for simple simulations
   300
enter the value of shortest loop allowable in dynamic programming
    1
enter the value of the cutoff
   1
input the output indicator: 0 - short, 1 - long
   0
input the reachibility problem graph
first input the number of nodes in the graph
the number of nodes is :   12
now input the edges one at a time as i , j, probability
(the last edge is 0 0 0)
edge    1 is : (   1 ,   2 , 0.1000 )
edge    2 is : (   1 ,   2 , 0.1000 )
edge    3 is : (   1 ,   2 , 0.1000 )
edge    4 is : (   1 ,   2 , 0.1000 )
edge    5 is : (   1 ,   2 , 0.1000 )
edge    6 is : (   1 ,   2 , 0.1000 )
edge    7 is : (   1 ,   2 , 0.1000 )
edge    8 is : (   1 ,   2 , 0.1000 )
edge    9 is : (   1 ,   2 , 0.1000 )
edge   10 is : (   1 ,   2 , 0.1000 )
edge   11 is : (   2 ,   3 , 0.1000 )
edge   12 is : (   2 ,   3 , 0.1000 )
edge   13 is : (   2 ,   3 , 0.1000 )
edge   14 is : (   2 ,   3 , 0.1000 )
edge   15 is : (   2 ,   3 , 0.1000 )
edge   16 is : (   2 ,   3 , 0.1000 )
edge   17 is : (   2 ,   3 , 0.1000 )
edge   18 is : (   2 ,   3 , 0.1000 )
edge   19 is : (   2 ,   3 , 0.1000 )
edge   20 is : (   2 ,   3 , 0.1000 )
edge   21 is : (   3 ,   4 , 0.1000 )
edge   22 is : (   3 ,   4 , 0.1000 )
edge   23 is : (   3 ,   4 , 0.1000 )
edge   24 is : (   3 ,   4 , 0.1000 )
edge   25 is : (   3 ,   4 , 0.1000 )
edge   26 is : (   3 ,   4 , 0.1000 )
edge   27 is : (   3 ,   4 , 0.1000 )
edge   28 is : (   3 ,   4 , 0.1000 )
edge   29 is : (   3 ,   4 , 0.1000 )
edge   30 is : (   3 ,   4 , 0.1000 )
edge   31 is : (   4 ,   5 , 0.1000 )
edge   32 is : (   4 ,   5 , 0.1000 )
edge   33 is : (   4 ,   5 , 0.1000 )
edge   34 is : (   4 ,   5 , 0.1000 )
edge   35 is : (   4 ,   5 , 0.1000 )
edge   36 is : (   4 ,   5 , 0.1000 )
edge   37 is : (   4 ,   5 , 0.1000 )
edge   38 is : (   4 ,   5 , 0.1000 )
edge   39 is : (   4 ,   5 , 0.1000 )
edge   40 is : (   4 ,   5 , 0.1000 )
edge   41 is : (   5 ,   6 , 0.1000 )
edge   42 is : (   5 ,   6 , 0.1000 )
edge   43 is : (   5 ,   6 , 0.1000 )
```

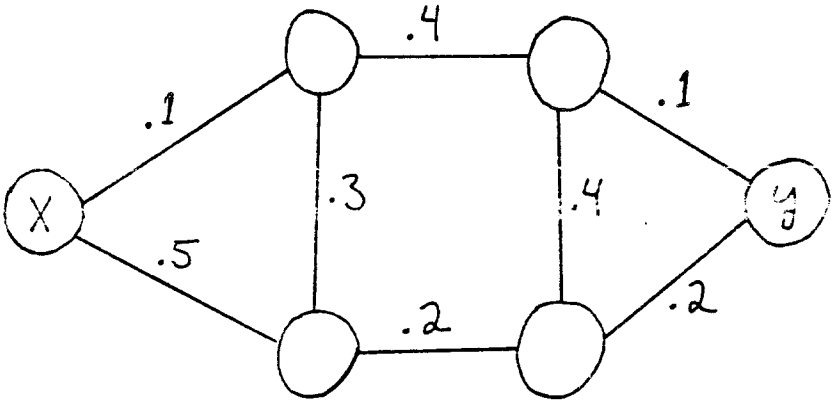
Example 3

```
edge   44 is : (   5 ,   6 , 0.1000 )
edge   45 is : (   5 ,   6 , 0.1000 )
edge   46 is : (   5 ,   6 , 0.1000 )
edge   47 is : (   5 ,   6 , 0.1000 )
edge   48 is : (   5 ,   6 , 0.1000 )
edge   49 is : (   5 ,   6 , 0.1000 )
edge   50 is : (   5 ,   6 , 0.1000 )
edge   51 is : (   6 ,   7 , 0.1000 )
edge   52 is : (   6 ,   7 , 0.1000 )
edge   53 is : (   6 ,   7 , 0.1000 )
edge   54 is : (   6 ,   7 , 0.1000 )
edge   55 is : (   6 ,   7 , 0.1000 )
edge   56 is : (   6 ,   7 , 0.1000 )
edge   57 is : (   6 ,   7 , 0.1000 )
edge   58 is : (   6 ,   7 , 0.1000 )
edge   59 is : (   6 ,   7 , 0.1000 )
edge   60 is : (   6 ,   7 , 0.1000 )
edge   61 is : (   7 ,   8 , 0.1000 )
edge   62 is : (   7 ,   8 , 0.1000 )
edge   63 is : (   7 ,   8 , 0.1000 )
edge   64 is : (   7 ,   8 , 0.1000 )
edge   65 is : (   7 ,   8 , 0.1000 )
edge   66 is : (   7 ,   8 , 0.1000 )
edge   67 is : (   7 ,   8 , 0.1000 )
edge   68 is : (   7 ,   8 , 0.1000 )
edge   69 is : (   7 ,   8 , 0.1000 )
edge   70 is : (   7 ,   8 , 0.1000 )
edge   71 is : (   8 ,   9 , 0.1000 )
edge   72 is : (   8 ,   9 , 0.1000 )
edge   73 is : (   8 ,   9 , 0.1000 )
edge   74 is : (   8 ,   9 , 0.1000 )
edge   75 is : (   8 ,   9 , 0.1000 )
edge   76 is : (   8 ,   9 , 0.1000 )
edge   77 is : (   8 ,   9 , 0.1000 )
edge   78 is : (   8 ,   9 , 0.1000 )
edge   79 is : (   8 ,   9 , 0.1000 )
edge   80 is : (   8 ,   9 , 0.1000 )
edge   81 is : (   9 ,  10 , 0.1000 )
edge   82 is : (   9 ,  10 , 0.1000 )
edge   83 is : (   9 ,  10 , 0.1000 )
edge   84 is : (   9 ,  10 , 0.1000 )
edge   85 is : (   9 ,  10 , 0.1000 )
edge   86 is : (   9 ,  10 , 0.1000 )
edge   87 is : (   9 ,  10 , 0.1000 )
edge   88 is : (   9 ,  10 , 0.1000 )
edge   89 is : (   9 ,  10 , 0.1000 )
edge   90 is : (   9 ,  10 , 0.1000 )
edge   91 is : (  10 ,  11 , 0.1000 )
edge   92 is : (  10 ,  11 , 0.1000 )
edge   93 is : (  10 ,  11 , 0.1000 )
edge   94 is : (  10 ,  11 , 0.1000 )
edge   95 is : (  10 ,  11 , 0.1000 )
edge   96 is : (  10 ,  11 , 0.1000 )
edge   97 is : (  10 ,  11 , 0.1000 )
edge   98 is : (  10 ,  11 , 0.1000 )
edge   99 is : (  10 ,  11 , 0.1000 )
edge  100 is : (  10 ,  11 , 0.1000 )
edge  101 is : (  11 ,  12 , 0.1000 )
```

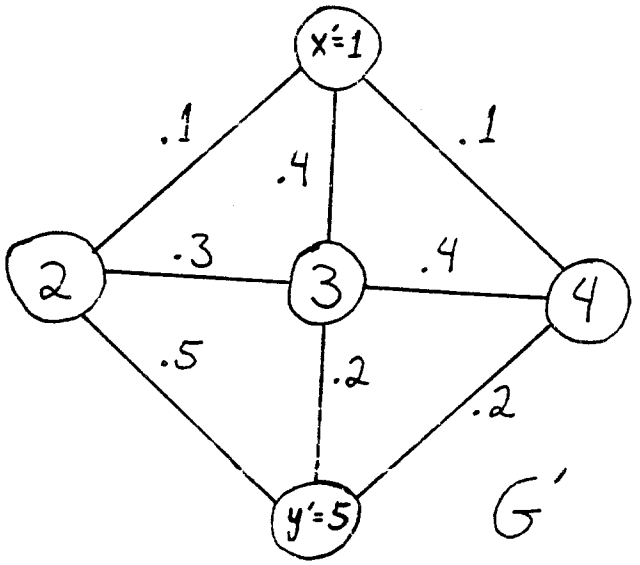
Example 3

```
edge  102 is : (  11 ,  12 , 0.1000 )
edge  103 is : (  11 ,  12 , 0.1000 )
edge  104 is : (  11 ,  12 , 0.1000 )
edge  105 is : (  11 ,  12 , 0.1000 )
edge  106 is : (  11 ,  12 , 0.1000 )
edge  107 is : (  11 ,  12 , 0.1000 )
edge  108 is : (  11 ,  12 , 0.1000 )
edge  109 is : (  11 ,  12 , 0.1000 )
edge  110 is : (  11 ,  12 , 0.1000 )
edge  111 is : (   0 ,   0 , 0.0000 )

l =  1 #dynrecords =    200 #predrecord =   1810 time =        1.7667

#carray =    3 time to compute WW =       1.2500

time to run all the trials is =      18.1500

the fraction of yes answers is:  1.00000000000000e-02
the sum of the walk weights is:  1.00000000000000e+00
the average value of the estimator is:  1.00000000000000e-02

the total running time is =      21.1667
```

Example 3



G

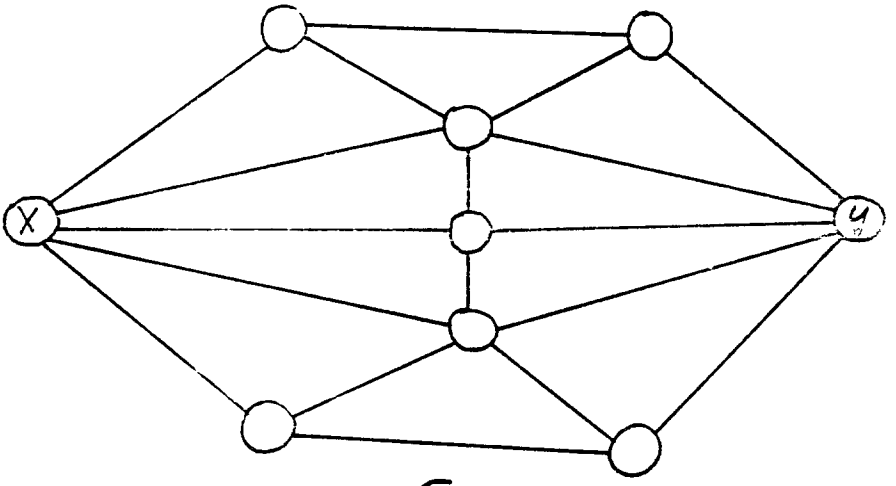


G'

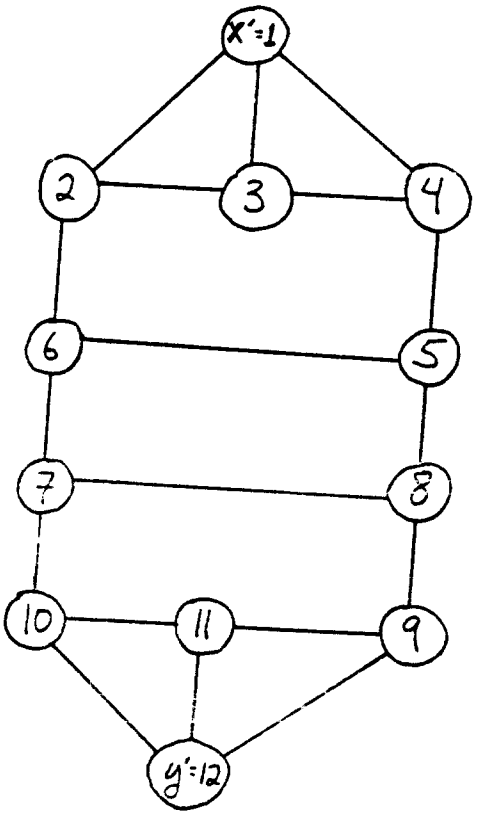
Example 4

```
 this is monte2sim.p in action, enter the seed
  289374
 enter the number of trials for simple simulations
  1000
enter the value of shortest loop allowable in dynamic programming
     1
enter the value of the cutoff
   4
input the output indicator: 0 - short, 1 - long
   0
input the reachibility problem graph
first input the number of nodes in the graph
the number of nodes is :     5
now input the edges one at a time as i , j, probability
(the last edge is 0 0 0)
edge     1 is : (   1 ,   2 , 0.1000 )
edge     2 is : (   1 ,   3 , 0.4000 )
edge     3 is : (   1 ,   4 , 0.1000 )
edge     4 is : (   3 ,   2 , 0.3000 )
edge     5 is : (   4 ,   3 , 0.4000 )
edge     6 is : (   2 ,   5 , 0.5000 )
edge     7 is : (   3 ,   5 , 0.2000 )
edge     8 is : (   4 ,   5 , 0.2000 )
edge     9 is : (   0 ,   0 , 0.0000 )

1 =   1 #dynrecords =      10 #predrecord =      16 time =        0.0167

#carray =     1 time to compute WW =        0.0000

time to run all the trials is =        7.2167

the fraction of yes answers is:  7.99000000000000e-01
the sum of the walk weights is:  2.64400000000000e-01
the average value of the estimator is:  2.11255600000000e-01

the total running time is =        7.2333
```

Example 4



G

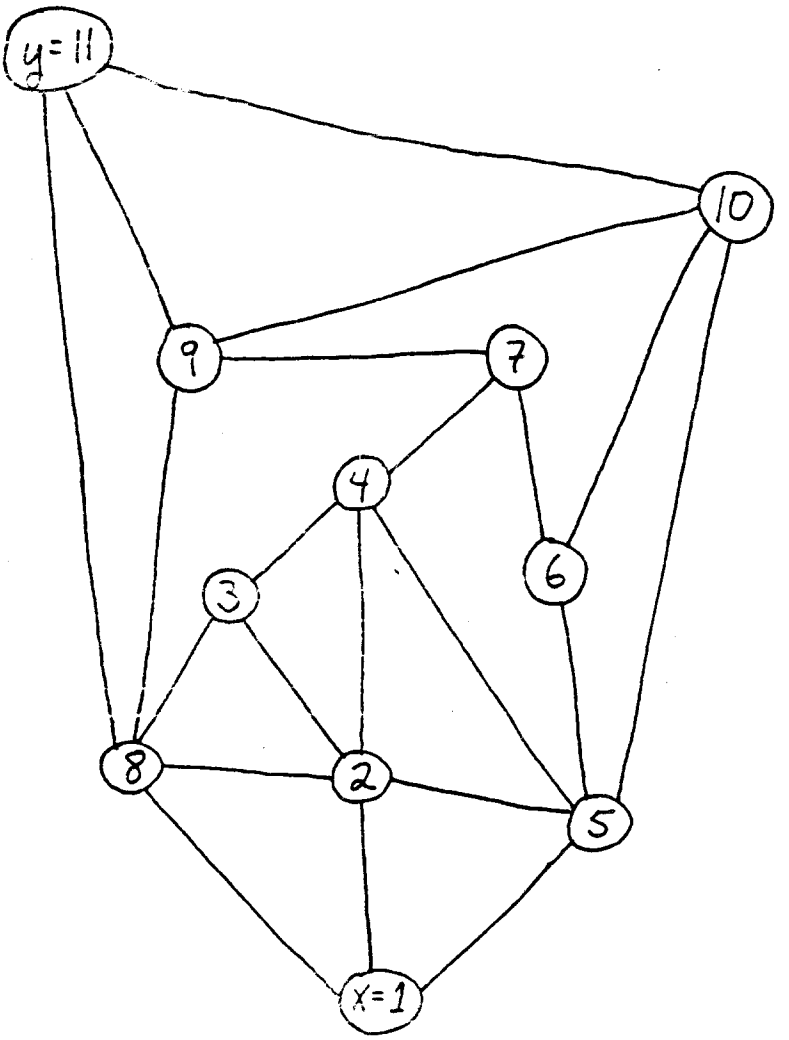


G'

Example 5

```
this is monte2sim.p in action, enter the seed
 789453
enter the number of trials for simple simulations
  300
enter the value of shortest loop allowable in dynamic programming
   1
enter the value of the cutoff
  3
input the output indicator: 0 - short, 1 - long
  0
input the reachibility problem graph
first input the number of nodes in the graph
the number of nodes is :   12
now input the edges one at a time as i , j, probability
(the last edge is 0 0 0)
edge    1 is : (   1 ,   2 , 0.1000 )
edge    2 is : (   1 ,   3 , 0.1000 )
edge    3 is : (   1 ,   4 , 0.1000 )
edge    4 is : (   2 ,   3 , 0.1000 )
edge    5 is : (   3 ,   4 , 0.1000 )
edge    6 is : (   2 ,   6 , 0.1000 )
edge    7 is : (   4 ,   5 , 0.1000 )
edge    8 is : (   6 ,   7 , 0.1000 )
edge    9 is : (   5 ,   8 , 0.1000 )
edge   10 is : (   5 ,   6 , 0.1000 )
edge   11 is : (   7 ,   8 , 0.1000 )
edge   12 is : (   7 ,  10 , 0.1000 )
edge   13 is : (   8 ,   9 , 0.1000 )
edge   14 is : (  10 ,  11 , 0.1000 )
edge   15 is : (   9 ,  11 , 0.1000 )
edge   16 is : (  10 ,  12 , 0.1000 )
edge   17 is : (  11 ,  12 , 0.1000 )
edge   18 is : (   9 ,  12 , 0.1000 )
edge   19 is : (   0 ,   0 , 0.0000 )

l =  1 #dynrecords =     30 #predrecord =     51 time =       0.0500

#carray =    1 time to compute WW =       0.0667

time to run all the trials is =       4.2000

the fraction of yes answers is:  9.80000000000000e-01
the sum of the walk weights is:  2.98203960000000e-05
the average value of the estimator is:  2.92239880800000e-05

the total running time is =       4.3167
```

Example 5



Example 6

```
 this is monte2sim.p in action, enter the seed
 8092785
 enter the number of trials for simple simulations
  1000
enter the value of shortest loop allowable in dynamic programming
    2
enter the value of the cutoff
   1
input the output indicator: 0 - short, 1 - long
   0
input the reachibility problem graph
first input the number of nodes in the graph
the number of nodes is :   11
now input the edges one at a time as i , j, probability
(the last edge is 0 0 0)
edge     1 is : (   1 ,   2 , 0.1000 )
edge     2 is : (   1 ,   5 , 0.1000 )
edge     3 is : (   2 ,   5 , 0.1000 )
edge     4 is : (   1 ,   8 , 0.1000 )
edge     5 is : (   2 ,   8 , 0.1000 )
edge     6 is : (   3 ,   2 , 0.1000 )
edge     7 is : (   4 ,   2 , 0.1000 )
edge     8 is : (   4 ,   5 , 0.1000 )
edge     9 is : (   5 ,   6 , 0.1000 )
edge    10 is : (   5 ,  10 , 0.1000 )
edge    11 is : (   6 ,   7 , 0.1000 )
edge    12 is : (   7 ,   4 , 0.1000 )
edge    13 is : (   4 ,   3 , 0.1000 )
edge    14 is : (   3 ,   8 , 0.1000 )
edge    15 is : (   8 ,   9 , 0.1000 )
edge    16 is : (   9 ,   7 , 0.1000 )
edge    17 is : (  10 ,   9 , 0.1000 )
edge    18 is : (   6 ,  10 , 0.1000 )
edge    19 is : (  10 ,  11 , 0.1000 )
edge    20 is : (   9 ,  11 , 0.1000 )
edge    21 is : (   8 ,  11 , 0.1000 )
edge    22 is : (   0 ,   0 , 0.0000 )

1 =  1 #dynrecords =     36 #predrecord =     96 time =        0.0667

1 =  2 #dynrecords =     93 #predrecord =    215 time =        0.1167

#carray =    1 time to compute WW =       0.1667

time to run all the trials is =      12.0167

the fraction of yes answers is:  9.74000000000000e-01
the sum of the walk weights is:  1.39084889960000e-02
the average value of the estimator is:  1.35468682821040e-02

the total running time is =      12.3667
```

Example 6