Randomization in Graph Optimization Problems: A Survey

David R. Karger*

June 12, 1998

Abstract

Randomization has become a pervasive technique in combinatorial optimization. We survey our thesis and subsequent work, which uses four common randomization techniques to attack numerous optimization problems on undirected graphs.

1 Introduction

Randomization has become a pervasive technique in combinatorial optimization. Randomization has been used to develop algorithms that are faster, simpler, and/or better-performing than previous deterministic algorithms. This article surveys our thesis [Kar94], which presents randomized algorithms for numerous problems on undirected graphs. Our work uses four important randomization techniques:

Random Selection, which lets us easily choose a "typical" element of a set, avoiding rare "bad" elements;

Random Sampling, which provides a quick way to build a small, representative subproblem of a larger problem for quick analysis;

Randomized Rounding, which lets us transform fractional problem solutions into integral ones; and

Monte Carlo Simulation, which lets us estimate the probabilities of interesting events.

*MIT Laboratory for Computer Science, Cambridge, MA 02138.
email: karger@lcs.mit.edu
URL: http://theory.lcs.mit.edu/~karger.
Work supported in part by ARPA contract N00014-95-1-1246 and NSF contract CCR-9624239, and Fellowships from the Alfred P. Sloane and David and Lucille Packard Foundations.

We apply these techniques to numerous optimization problems on undirected graphs. The graph is one of the most common structures in computer science and optimization, modeling among other things roads, communication and transportation networks, electrical circuits, relationships between individuals, hypertext collections, resource allocations, project plans, database and program dependencies, and parallel architectures. Among the graph problems we address are finding a minimum spanning tree, finding a maximum flow, determining the connectivity (minimum cut) of a graph, network design, graph coloring, and estimating the reliability (disconnection probability) of a network with random edge failures. A great deal of work has been done on all of these problems. Due to space limitations, we are unable to discuss all of this related work. Such a discussion can be found in our thesis.

It is extremely important to note that we are not carrying out "expected case" analysis of algorithms running on random inputs. Rather, we consider *worst case* inputs and use random choices by the algorithm to solve them efficiently.

We begin this article with a survey of many of our applications of the four randomization techniques, sketching the methods, the problems, and the resulting algorithms. Afterwards, we will present a more detailed discussion of a few algorithms and proofs which will hopefully give some flavor of our work.

1.1 Notation

We address only undirected graphs; directed graphs have so far not succumbed to the techniques we apply here. Throughout our discussion, we will consider a graph G with m edges and n vertices. Each graph edge may have a *weight* (reflecting cost or capacity) associated with it. To simplify our presentation here, we often focus on unweighted graphs (that is, graphs with all edge weights equal to one), though many of our algorithms apply equally well to weighted graphs. Unless it has parallel edges (multiple edges with the same endpoints) a graph has $m \leq \binom{n}{2}$.

The notation $\tilde{O}(f)$ denotes $O(f \log^d n)$ for some constant d.

1.2 Overview of Results

We show how randomization can be used in several ways on problems of varying degrees of difficulty. For the "easy to solve" minimum spanning tree problem, where a long line of research has resulted in ever closer to linear-time algorithms, random sampling gives the final small increment to a truly linear time algorithm. For harder problems it improves running times by a more significant factor. For example, we improve the time needed to find minimum cuts from $\tilde{O}(mn)$ to $\tilde{O}(m)$, and give the first efficient parallel algorithm for the problem. Addressing some hard $\mathcal{NP}$-complete problems such as network design and graph coloring, where finding an exact solution in polynomial time is thought to be hopeless, we use randomized rounding to give better approximation algorithms than were previously known. Finally, for the problem of determining the reliability of a network (a $\sharp\mathcal{P}$-complete problem which is thought to be "even harder" than $\mathcal{NP}$-complete ones) we use Monte Carlo simulation to give the first efficient approximation algorithm.

1.3 Randomized Algorithms

Our work deals with randomized algorithms. Our typical model is that the algorithm has a source of "random bits"—variables that are mutually independent and take on values 0 or 1 with probability $1/2$ each. Extracting one random bit from the source is assumed to take constant time. If our algorithms use more complex operations, such as flipping biased coins or generating samples from more complex distributions, we take into account the time needed to simulate these operations in our unbiased-bit model. Event probabilities are taken over the sample space of random bit strings produced by the random bit generator. An event occurs *with high probability (w.h.p.)* if on problems of size n it occurs with probability greater than $(1 - \frac{1}{n^k})$ for some constant $k > 1$, and with *low probability* if its complement occurs with high probability.

The random choices that an algorithm makes can affect both its running time and its correctness. An algorithm that has a fixed (deterministic) running time but has a low probability of giving an incorrect answer is called *Monte Carlo (MC)*. If the running time of the algorithm is a random variable but the correct answer is given with certainty, then the algorithm is said to be *Las Vegas (LV)*. Depending on the circumstances, one type of algorithm may be better than the other. However, a Las Vegas algorithm is "stronger" in the following sense.

A Las Vegas algorithm can be made Monte Carlo by having it terminate with an arbitrary wrong answer if it exceeds the time bound $f(n)$. Since the Las Vegas algorithm is unlikely to exceed its time bound, the converted algorithm is unlikely to give the wrong answer. On the other hand, there is no universal method for making a Monte Carlo algorithm into a Las Vegas one, and indeed some of the algorithms we present are Monte Carlo with no Las Vegas version apparent. The fundamental problem is that sometimes it is impossible to check whether an algorithm has given a correct answer. However, the failure probability of a Monte Carlo optimization algorithm can be made arbitrarily small by repeating it several times and taking the best answer; we shall see several examples of this below. In particular, we can reduce the failure probability so far that other unavoidable events (such as a power failure) are more likely than an incorrect answer.

2 A Survey of Techniques and Results

In this section, we provide a high level overview of the four randomization techniques and the various algorithms we have been able to develop by using them.

2.1 Random Selection

The first and simplest randomization technique we discuss is random selection. The intuition behind this idea is that a single randomly selected individual is probably a "typical" representative of the entire population. Thus, random selection provides a good way to avoid choosing rare "bad" elements. This is the idea behind Quicksort [Hoa62], where the assumption is that the randomly selected pivot will be neither extremely large nor extremely small, and will therefore serve to separate the remaining elements into two roughly equal sized groups.

We apply this idea in a new algorithm for finding minimum cuts in undirected graphs. A *cut* is a partition of the graph vertices into two groups; the *value* of the cut is the number (or total weight) of edges with one endpoint in each group. The minimum cut problem is to identify a cut of minimum value. We distinguish the (global) minimum cut from an *s-t minimum cut* which is required to separate two specific vertices s and t. Finding minimum cuts is of great importance in analyzing network reliability and also plays a role in solving traveling salesman and network design problems.

We present the Recursive Contraction Algorithm (joint work with Clifford Stein [KS96]). The idea behind our algorithm is simple: a randomly selected edge is unlikely to cross a particular minimum cut, so its endpoints are probably on the same side. If we merge two vertices on the same side of this minimum cut, then we will not affect the minimum cut but will reduce the number of graph vertices by one. Therefore, we can find the minimum cut by repeatedly selecting a random edge and merging its endpoints until only two vertices remain and the minimum cut becomes obvious.

An efficient implementation of the above idea leads to a strongly polynomial $\tilde{O}(n^2)$-time algorithm for the minimum cut problem on weighted undirected graphs. In contrast, the best deterministic bound, due to Hao and Orlin [HO94], is $\tilde{O}(mn)$. Our algorithm actually finds *all* minimum cuts with high probability. It extends to enumerating approximately minimum cuts and minimum k-way cuts for constant k, as well as to constructing the *cactus* of a graph (a compact representation of all its minimum cuts). The algorithm is the first with a theoretically good parallelization. A *derandomization* of the algorithm (joint with Rajeev Motwani [KM97]) gave the first proof that there was a fast deterministic parallel algorithm for the minimum cut problem. An implementation experiment shows that the algorithm has reasonable time bounds in practice [CGK$^+$97].

The Contraction Algorithm is Monte Carlo: it gives the correct answer with high probability, but does have a small chance of being wrong. As we are unaware of any algorithm for *verifying* that a cut is minimum, we have been unable to devise a Las Vegas version of the algorithm. This is a case where a willingness to occasionally be wrong seems to provide a significant speedup.

The Contraction Algorithm also gives an important new bound on the number of small cuts a graph may contain; this has important applications in network reliability analysis and graph sampling (see below).

In subsequent work [BK98], we used the Contraction Algorithm to efficiently solve the *graph augmentation problem:* adding the minimum possible capacity to a graph so as to increase its minimum cut to a given value (this work is joint with Andras Benczùr).

2.2 Random Sampling

A more general use of randomization than random selection is to generate small representative subproblems. The representative random sample is a central concept of statistics. It is often possible to gather a great deal of information about a large population by examining a small sample randomly drawn from it. This approach has obvious advantages in reducing the investigator's work, both in gathering and in analyzing the data.

Given an optimization problem, it may be possible to generate a small representative subproblem by random sampling (perhaps the most natural sample from a graph is a random subset of its edges). Intuitively, such a subproblem should form a microcosm of the larger problem. Our goal is to examine the subproblem and use it to glean information about the original problem. Since the subproblem is small, we can spend proportionally more time examining it than we would spend examining the original problem. In one approach that we use frequently, an optimal solution to the subproblem may be a nearly optimal solution to the problem as a whole. In some situations, such an approximation might be sufficient. In other situations, it may be easy to refine this good solution into a truly optimal solution.

Floyd and Rivest [FR75] use this approach in a fast and elegant algorithm for finding the median of an ordered set. They select a small random sample of elements from the set and show how inspecting this sample gives a very accurate estimate of the value of the median. It is then easy to find the actual median by examining only those elements close to the estimate. This algorithm, which is very simple to implement, uses fewer comparisons than any other known median-finding algorithm.

The Floyd-Rivest algorithm typifies three components needed in a random-sampling algorithm. The first is a definition of a *randomly sampled subproblem.* The second is an *approximation theorem* that proves that a solution to the subproblem is an approximate solution to the original problem. These two components by themselves will typically yield an obvious approximation algorithm with a speed-accuracy tradeoff. The third component is a *refinement algorithm* that takes the approximate solution and turns it into an actual solution. Combining these three components can yield an algorithm whose running time will be determined by that of the refinement algorithm; intuitively, refinement should be easier than computing a solution from scratch.

In an application of this approach, we present the first (randomized) linear-time algorithm for finding minimum spanning trees in the comparison-based model of computation. This result reflects joint work with Philip Klein and Robert E. Tarjan [KKT95]. A long stream of results reduced the best known deterministic time bound to almost linear [GGST86], but a linear time bound remained elusive. Our fundamental insight is that if we construct a subgraph of a graph by taking a random sample of the graph's edges, then the minimum spanning tree in the subgraph is a "nearly" minimum spanning tree of the entire graph. More precisely, very few graph edges can be used to improve the sample's minimum spanning tree. By examining these few edges, we can refine our approximation into the actual minimum spanning tree at little additional cost.

We also apply sampling to the minimum cut problem and several other related problems involving cuts in graphs, including maximum flows. The maximum flow problem is perhaps the most widely studied of all graph optimization problems, having hundreds of applications. Given vertices s and t and capacitated edges, the goal is to ship the maximum quantity of material from s to t without exceeding the capacities of the edges. The value of a graph's maximum flow is completely determined by the value of the minimum s-t cut in the graph.

We prove a cut sampling theorem that says that when we choose half a graph's edges at random we approximately halve the value of every cut. In particular, we halve the graph's connectivity and the value of all s-t minimum cuts and maximum flows. This theorem gives a random-sampling scheme for approximating minimum cuts and maximum flows: compute the minimum cut and maximum flow in a random sample of the graph edges. Since the sample has fewer edges, the computation is faster. At the same time, our sampling theorems show that this approach gives accurate estimates of the correct values. If we want to get exact solutions rather than approximations, we still can use our samples as starting points to which we can apply inexpensive refinement algorithms. This leads to a simple randomized divide-and-conquer algorithm for finding exact maximum flows.

We put our results on graph sampling into a larger framework by examining sampling from *matroids* [Kar98b]. We generalize our minimum spanning tree algorithm to the problem of finding a minimum cost matroid basis, and extend our cut-sampling and maximum flow results to the problem of matroid basis packing. Our techniques actually give a paradigm that can be applied to any *packing problem* where the goal, given a collection of *feasible* subsets of a universe, is to find a maximum collection of disjoint feasible subsets. For example, in the maximum flow problem, we are attempting to send units of flow from s to t. Each such unit of flow travels along a path from s to t, so the feasible edge-sets are the s-t paths. We apply the sampling paradigm to the problem of packing disjoint bases in a matroid, and get faster algorithms for approximating and exactly finding optimum basis packings.

We have continued to apply random sampling technique in work following our thesis. Our recent results include:

- An $O(m \log^3 n)$-time Monte Carlo algorithm for finding minimum cuts [Kar96],
- A *compression algorithm* that lets us transform any undirected graph into a graph with $\tilde{O}(n)$ edges but roughly the same cut values, speeding up any algorithm that depends only on cut values [BK96],

- As an application, an $O(n^2 \log n)$-time Monte Carlo algorithm for finding any constant factor approximation to an s-t minimum cut [BK96],
- An $\tilde{O}(m\sqrt{n})$-time Las Vegas algorithm for finding any constant factor approximation to an s-t max-flow [Kar98a],
- An $\tilde{O}(n^{2.22})$-time algorithm for finding a maximum flow in a simple (uncapacitated) graph [KL98] (joint work with Matt Levine).

All of these bounds are significantly better than the best general time bound for finding maximum flows in directed graphs ($\tilde{O}(mn)$ for a strongly polynomial bound [GT88], and recently $\tilde{O}(m^{3/2} \log U)$ for a scaling algorithm of Goldberg and Rao [GR97]). This suggests that perhaps a better bound for maximum flow can be achieved, at least on undirected graphs.

2.3 Randomized Rounding

Yet another powerful randomization technique is *randomized rounding*. This approach is used to find approximate solutions to $\mathcal{NP}$-hard *integer programs*. These problems typically ask for an assignment of 0/1 values to variables x_i such that linear constraints of the form $\sum a_i x_i = c$ are satisfied. If we *relax* the integer program, allowing each x_i to take any real value between 0 and 1, we get a linear program that can be solved in polynomial time, giving values p_i such that $\sum a_i p_i = c$. Raghavan and Thompson [RT87] observed that we could treat the resulting values p_i as probabilities. If we randomly set $x_i = 1$ with probability p_i and 0 otherwise, then the *expected value* of $\sum a_i x_i$ is $\sum a_i p_i = c$. Raghavan and Thompson presented techniques for ensuring that the randomly chosen values do in fact yield a sum near the expectation, thus giving approximately correct solutions to the integer program. We can view randomized rounding as a way of sampling randomly from a large space of *answers*, rather than subproblems as before. Linear programming relaxation is used to construct an answer-space in which most of the answers are good ones.

We use our graph sampling theorems to apply randomized rounding to *network design problems*. Such a problem is specified by an input graph G with each edge assigned a cost. The goal is to output a subgraph of G satisfying certain connectivity requirements at minimum cost (measured as the sum of the costs of edges used). These requirements are described by specifying a minimum number of edges that must cross each cut of G. This formulation easily captures many classic problems including perfect matching, minimum cost flow, Steiner tree, and minimum T-join. By applying randomized rounding, we improve the approximation bounds for a large class of network design problems, from $O(\log n)$ (due to Goemans et al [GGP$^+$94]) to $1 + o(1)$ in some cases. Our graph sampling theorems provide the necessary tools for showing that randomized rounding works well in this case.

We also apply randomized rounding to the classic *graph coloring problem*. No linear programs have yet been devised that provide a useful fractional solution, so we use more powerful *semidefinite programming* as our starting point. We show that any 3-colorable graph can be colored in polynomial time with $\tilde{O}(n^{1/4})$ colors, improving on the previous best bound of $\tilde{O}(n^{3/8})$ [Blu94]. We also give presently best results for k-colorable graphs. Along the way, we discover new properties of the *Lovász ϑ-function,* an object that has received a great deal of attention because of its connections to graph coloring, cliques, and independent sets. This work is joint with Rajeev Motwani and Madhu Sudan [KMS98]. We gave a slight improvement with Avrim Blum [BK97].

2.4 Monte Carlo Estimation

The last randomization technique we consider is Monte Carlo estimation. The technique is applied when we want to estimate the probability p of a given event E over some probability space. Monte Carlo estimation carries out repeated "trials" (samples from the probability space) and measures in what fraction of the trials the event E occurs. This gives a natural estimate of the event probability.

The Monte Carlo approach breaks down when the interesting probability p is very small. To estimate p, we need to carry out enough experiments to see at least a few occurrences of E. But we expect to see a first occurrence only after $1/p$ trials, which may be too many to carry out efficiently. A solution to this problem, explored by Karp, Luby and Madras [KLM89], is to carry out the Monte Carlo simulation in a different, "biased" way that makes the event E more likely to occur, so that we can get by with fewer trials. The trick is to choose the new simulation so that it gives us useful information about the *original* probability space.

We apply this technique to the *network reliability problem.* In this problem, we are interested in estimating the probability that a network is disconnected by random edge failures, so it is perhaps unsurprising that randomization is useful. We are given a graph G whose edges fail randomly and independently with certain specified probabilities. Our goal is to determine the probability that the graph becomes disconnected by edge failures.

Unfortunately, it is known to be $\sharp\mathcal{P}$-hard (even worse than $\mathcal{NP}$-hard) to exactly determine the reliability of a network. But we use Monte Carlo methods to give a *fully polynomial randomized approximation scheme (FPRAS)* for the network reliability problem [Kar98d]. Given a failure probability p for the edges, our algorithm, in time polynomial in n and $1/\epsilon$, returns a number P that estimates the probability $\mathrm{FAIL}(p)$ that the graph becomes disconnected. With high probability, P is in the range $(1 \pm \epsilon)\mathrm{FAIL}(p)$. The algorithm is Monte Carlo, meaning that the approximation is correct with high probability but that it is not possible to verify its correctness. It generalizes to the case where the edge failure probabilities are different, to computing the probability the graph fails to be k-connected (for any fixed k), and to the more general problem of approximating the *Tutte Polynomial* for a large family of graphs. Our algorithm is easy to implement and appears likely to have satisfactory time bounds in practice [Kar98d, CGK+97, KT97].

A natural way to estimate a network's failure probability is to carry out numerous simulations of the edge failures and check how often the graph is disconnected by them. But as mentioned above, this can take prohibitively many trials if the failure probability is extremely small. However, we use our cut counting and sampling theorems to prove that when P is small, only the small cuts in a graph are significantly likely to fail. We use our cut algorithms to enumerate these small cuts and then use the biased Monte Carlo technique developed by Karp, Luby and Madras [KLM89] to estimate the probability the one of the explicitly enumerated cuts fails.

3 The Contraction Algorithm

To give some flavor of our results, we begin by describing an algorithm for finding a minimum cut in an undirected graph [KS96]. For simplicity we discuss unweighted graphs, but the algorithm works equally well for graphs with edge weights.

The algorithm is based on the idea of *contracting* edges. Suppose we were somehow able to identify an edge that did not cross the minimum cut. This would tell us that both of its endpoints are on the same side of the cut. We can use this information to simplify the graph by contracting the two endpoints. To contract two vertices v_1 and v_2 we replace them by a vertex v, and let the set of edges incident on v be the union of the sets of edges incident on v_1 and v_2. We do not merge edges from v_1 and v_2 that have the same other endpoint; instead, we allow multiple instances of those edges. However, we remove self loops formed by edges originally connecting v_1 to v_2. Formally, we delete all edges (v_1, v_2), and replace each edge (v_1, w) or (v_2, w) with an edge (v, w). The rest of the graph remains unchanged. We will use $G/(v_1, v_2)$ to denote graph G with edge (v_1, v_2) contracted (by *contracting an edge*, we will mean contracting the two endpoints of the edge).

Note that a contraction reduces the number of graph vertices by one. We can imagine repeatedly selecting and contracting edges until every vertex has been merged into one of two remaining "metavertices." These metavertices define a cut of the original graph: each side corresponds to the vertices contained in one of the metavertices. It is easy to see that if we never contract an edge that crosses the minimum cut, then the two metavertices we end up with will correspond to the two sides of the minimum cut we are looking for.

So our subgoal is to devise a method for selecting an edge that does not cross the minimum cut. There are some sophisticated deterministic algorithms for doing this [NI92], but unfortunately they are slow. We instead rely on the following observation: almost none of the edges in a graph cross the minimum cut. Thus, if we choose a *random* edge to contract, we probably get a non-min-cut edge! This gives us a very fast edge selection algorithm; the trade-off is that we must be prepared for it occasionally to make mistakes. We describe our algorithm in Figure 1. Assume initially that we are given a multigraph $G(V, E)$ with n vertices and m edges. The *Contraction Algorithm*, which is described in Figure 1, repeatedly chooses an edge at random and contracts it.

Algorithm `Contract`(G)

repeat until G has 2 vertices

  choose an edge (v, w) uniformly at random from G

  let $G \leftarrow G/(v, w)$

return the unique cut defined by (the contracted) G

Figure 1: The Contraction Algorithm

It is relatively straightforward to implement this algorithm in $O(n^2)$ time.

Lemma 3.1. *A particular minimum cut in G is returned by the Contraction Algorithm with probability at least* $\binom{n}{2}^{-1}$.

Proof. Fix attention on some specific minimum cut C with c crossing edges. We will use the term *minimum cut edge* to refer only to edges crossing C. If we never select a minimum cut edge during the Contraction Algorithm, then the two vertices we end up with must define the minimum cut.

Observe that after each contraction, the minimum cut value in the new graph must still be at least c. This is because every cut in the contracted graph corresponds to a cut of the same value in the original graph, and thus has value at least c. Furthermore, if we contract an edge (v, w) that does not cross C, then the cut C corresponds to a cut of value c in $G/(v, w)$; this corresponding cut is a minimum cut (of value c) in the contracted graph.

Each time we contract an edge, we reduce the number of vertices in the graph by one. Consider the stage in which the graph has r vertices. Since the contracted graph has a minimum cut of at least c, it must have minimum degree c, and thus at least $rc/2$ edges. However, only c of these edges are in the minimum cut. Thus, a randomly chosen edge is in the minimum cut with probability at most $2/r$. To determine the probability that we *never* contract a minimum cut edge, we simply multiply all of the per-stage probabilities. This shows that the probability that we never contract a minimum cut edge through all $n-2$ contractions is at least

$$\begin{aligned}
\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{3}\right) &= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \binom{n}{2}^{-1} \\
&\geq 1/n^2.
\end{aligned}$$

□

Note that $\binom{n}{2}^{-1} \approx 1/n^2$, so the Contraction Algorithm described above has a relatively small chance of succeeding. But it is large enough to be useful.

To improve our chance of success, we may simply repeat the algorithm a large number of times. If we run the Contraction Algorithm $n^2 \ln n$ times, and take the best answer we see, then the probability that we fail to give the right answer is just the probability that *none* of the repetitions of the algorithm yield the right answer, which is at most

$$(1 - 1/n^2)^{n^2 \ln n} \approx 1/n,$$

which means the algorithm works with high probability. This "amplification through repetition" is standard for randomized algorithms: we can get an exponential decrease in the failure probability from a linear slowdown in the running time.

Since the Contraction Algorithm takes $O(n^2)$ time per iteration, we immediately get an algorithm that finds a minimum cut *with high probability* in $O(n^4 \log n)$ time. This is somewhat unsatisfactory, as algorithms based on flow [HO94] can be used to find the minimum cut in $\tilde{O}(mn)$ time.

3.1 The Recursive Contraction Algorithm

By adding another idea we can improve the running time of our minimum cut algorithm to $\tilde{O}(n^2)$. We aim to "share work" among the numerous iterations of the algorithm. Note that the failure probability of the Contraction Algorithm rises as its size decreases. In fact, if we contract G until it has k vertices rather than 2, then the probability that the algorithm does not destroy the minimum cut of G exceeds $(k/n)^2$ (this follows by truncating the product we used to analyze the original Contraction Algorithm). So the real problem with the Contraction Algorithm arises when the graph has gotten small. We might imagine switching over to a deterministic algorithm once the graph is small, and indeed this approach yields improved performance. But we can do even better with another application of the principle that "repetition improves your chances." When the graph gets small, in order to improve our odds of success, we (recursively) carry out *two* executions of the algorithm on what remains.

Let `Contract`(G, k) denote a subroutine that runs the Contraction Algorithm until G is reduced to k vertices. Consider the *Recursive Contraction Algorithm* in Figure 2. As can be seen, we perform two independent trials. In each, we first partially contract the graph, but not so much that the likelihood of the cut surviving is too small. By contracting the graph until it has $n/\sqrt{2}$ vertices, we ensure a 50% probability of not contracting a minimum cut edge, so we expect that on the average one of the two attempts will avoid contracting a minimum cut edge. We then recursively apply the algorithm to each of the two partially contracted graphs. As described, the algorithm returns only a cut value; it can easily be modified to return a cut of the given value. Alternatively, we might want to output every cut encountered, hoping to enumerate all the minimum cuts.

Next we analyze the running time of this algorithm.

Lemma 3.2. *Algorithm* `Recursive-Contract` *runs in* $O(n^2 \log n)$ *time.*

Algorithm `Recursive-Contract`(G, n)

input A graph G of size n.

if G has 2 vertices

then

  return the weight of (unique) cut in G

else repeat **twice**

  $G' \leftarrow$ `Contract`$(G, n/\sqrt{2})$

  `Recursive-Contract`$(G', n/\sqrt{2})$.

 return the smaller of the two resulting values.

Figure 2: The Recursive Contraction Algorithm

Proof. One level of recursion consists of two independent trials of contraction of G to $n/\sqrt{2}$ vertices followed by a recursive call. Performing a contraction to $n/\sqrt{2}$ vertices can be implemented by Algorithm `Contract` from the previous section in $O(n^2)$ time. We thus have the following recurrence for the running time:

$$T(n) = 2\left(n^2 + T\left(n/\sqrt{2}\right)\right). \tag{1}$$

This recurrence is solved by

$$T(n) = O(n^2 \log n).$$

□

We now analyze the probability that the algorithm finds the particular minimum cut we are looking for. We will say that the Recursive Contraction Algorithm *finds* a certain minimum cut if that minimum cut corresponds to one of the leaves in the algorithm's tree of recursive calls. Note that if the algorithm finds any minimum cut then it will output the minimum cut value.

Lemma 3.3. *The Recursive Contraction Algorithm finds a particular minimum cut with probability* $\Omega(1/\log n)$.

Proof. We give a recursive argument. The algorithm will find a particular minimum cut if, in one of its two iterations, the following two things happen: (i) the call to `Contract`$(G, n/\sqrt{2})$ preserves the minimum cut and (ii) the recursive call finds the particular minimum cut. The probability that an iteration succeeds is just the product of the probabilities of events (i) and (ii). The algorithm succeeds if *either* iteration succeeds, and thus fails only if *both* iterations fail. The probability this double failure happens is just the square of the probability that one iteration fails. Thus the success probability is one minus this squared quantity. This yields a recurrence $P(n)$ for a lower bound on the probability of success on a graph of size n:

$$\begin{aligned} P(2) &= 1 \\ P(n) &\geq 1 - \left(1 - \frac{1}{2} P\left(n/\sqrt{2}\right)\right)^2. \end{aligned}$$

We solve this recurrence through a change of variables. Write $z_k = 4/P(2^{k/2}) - 1$, so $P(2^{k/2}) = 4/(z_k + 1)$. Plugging this into the above recurrence and solving for z_k yields

$$\begin{aligned} z_1 &= 3 \\ z_{k+1} &= z_k + 1 + 1/z_k. \end{aligned}$$

Since clearly $z_k \geq 1$, it follows by induction that

$$k < z_k < 3 + 2k$$

Thus $z_k = \Theta(k)$ and thus that

$$P(n) = 4/(z_{2 \log n} + 1) = \Theta(1/\log n).$$

In other words, one trial of the Recursive Contraction Algorithm finds any particular minimum cut with probability $\Omega(1/\log n)$. □

Those familiar with branching processes might see that we are evaluating the probability that the extinction of contracted graphs containing the minimum cut does not occur before depth $2 \log n$.

Theorem 3.4 ([KS96]). *All minimum cuts in an arbitrarily weighted undirected graph with n vertices can be found with high probability in $O(n^2 \log^3 n)$ time.*

Proof. We will see below that there are at most $\binom{n}{2}$ minimum cuts in a graph. Repeating `Recursive-Contract` $O(\log^2 n)$ times gives an $O(1/n^4)$ chance of missing any particular minimum cut. Thus our chance of missing any one of the at most $\binom{n}{2}$ minimum cuts is upper bounded by $O(\binom{n}{2} \cdot n^{-4}) = O(1/n^2)$. □

3.2 Counting Cuts

Besides serving as an algorithm to find minimum cuts, the Contraction Algorithm tells us some interesting things about the number of minimum and, more generally, small cuts in a graph. These results are extremely useful when we consider our next topic, random sampling from graphs.

Definition 3.5. An α*-minimum cut* is a cut whose value is at most α times that of the (global) minimum cut.

Lemma 3.6. *There are at most $\binom{n}{2} < n^2$ minimum cuts.*

Proof. We showed that the Contraction Algorithm outputs a given minimum cut with probability at least $\binom{n}{2}^{-1}$. Suppose that there were more than k minimum cuts. Each is output with probability k. Since these output events are disjoint (the algorithm outputs only one cut), the probability that one of them is output is just the sum of their individual probabilities, namely $k/\binom{n}{2}$. This quantity, being a probability, is at most one. So $k \leq \binom{n}{2}$. □

Theorem 3.7 (Cut Counting [KS96]). *In a graph with minimum cut c, there are less than $n^{2\alpha}$ cuts of value at most αc.*

Proof. If we consider a cut of value αc, we can prove (by generalizing the argument we gave for minimum cuts in the obvious way) that the Contraction Algorithm outputs it with probability at least $1/n^{2\alpha}$. The argument then proceeds as in the previous lemma. □

4 Random Sampling

So far we have addressed random selection, which works by finding a "typical" element (eg a non-min-cut edge). We now turn to random sampling, where the goal is to build a small representative model of our input problem. We will describe algorithms for approximating and exactly finding maximum flows and minimum cuts in an undirected graph. For simplicity, we will restrict our discussion to graphs with unit-capacity edges (unweighted graphs) though many of the techniques that we discuss can be applied to weighted graphs as well. Due to space limitations, and because we are focusing on our thesis work rather than later improvements, we present algorithms that only work well when the minimum cut of the graph is large.

In unweighted graphs, the *s-t maximum flow problem* is to find a maximum set, or *packing,* of edge-disjoint s-t paths. It is known [FF62] that the value of this flow is equal to the value of the minimum s-t cut. In fact, the only known algorithms for finding an s-t minimum cut simply identify a cut that is saturated by an s-t maximum flow.

In unweighted graphs, a classic algorithm for finding such a maximum flow is the *augmenting path* algorithm (cf. [Tar83, AMO93]). Given a graph and an s-t flow of value f, a linear-time depth first search of the so-called *residual graph* will either show how to augment the flow to one of value $f+1$ or prove that f is the value of the maximum flow. This algorithm can be used to find a maximum flow of value v in $O(mv)$ time by finding v augmenting paths. Of course, since the algorithm's running time depends on the edge count and flow value, we can make it faster by reducing one or both quantities. We show how random sampling can be used to do this.

4.1 A Sampling Theorem

Our algorithms are all based upon the following model of random sampling in graphs. We are given an unweighted graph $G = (V, E)$ with a *sampling* *probability* p for each edge e, and we construct a random subgraph, or *skeleton,* on the same vertices V by placing each edge e in the skeleton independently with probability p. We denote the skeleton by $G(p)$. Note that if a given cut has k edges crossing it in G, then the expected number of edges crossing that cut in $G(p)$ is pk. In particular, if the s-t minimum cut in G has value v, then we might expect that the s-t minimum cut in $G(p)$ has value pv.

Unfortunately, samples invariably *deviate* from their expectations. In order to effectively make use of a skeleton, we need to show that these deviations are small. If they are, then the skeleton will tell us things about the original graph that are approximately correct. Let c be the minimum cut of graph G. Our main theorem says that so long as pc (the minimum expected cut value in the skeleton) is sufficiently large, every cut in the skeleton takes on roughly its expected value.

Theorem 4.1 ([Kar98c]). *Let* $\epsilon = \sqrt{3(d+2)(\ln n)/pc}$ *(so* $p = \Theta((\ln n)/\epsilon^2 c)$*). If* $\epsilon \leq 1$*, then with probability* $1 - O(1/n^d)$*, every cut in* $G(p)$ *has value between* $1-\epsilon$ *and* $1+\epsilon$ *times its expected value.*

This result is somewhat surprising. A graph has exponentially many (2^{n-1}) cuts. Naively, even if each cut is unlikely to deviate far from its expected value, with so many cuts one probably will. We are saved by the cut counting theorem discussed in the previous section. The central limit theorem (as quantified by the Chernoff bound [Che52, MR95b]) says that as the expected value of a sample gets larger, its sample value becomes more and more tightly concentrated about its expectation. In particular, as a cut value grows, its probability of deviating by a given ratio ϵ from its expectation decays exponentially with the cut value. The cut counting theorem says that the number of cuts of a given value increases "only" exponentially with the cut value. The parameters of Theorem 4.1 are chosen so that the exponential decrease in deviation probability dominates the exponential increase in the number of cuts.

4.2 Applications

We now show how the skeleton approach can be applied to minimum cuts and maximum flows. We use the following definitions:

Definition 4.2. An *α-minimum s-t cut* is an s-t cut whose value is at most α times the value of the s-t minimum cut. An *α-maximum s-t flow* is an s-t flow whose value is at least α times the optimum.

We have the following immediate extension of Theorem 4.1:

Theorem 4.3. *Let* G *be any graph with minimum cut* c *and let* $p = \Theta((\ln n)/\epsilon^2 c)$ *as in Theorem 4.1. Suppose the* s-t *minimum cut of* G *has value* v*. Then with high probability, the* s-t *minimum cut in* $G(p)$ *has value between* $(1-\epsilon)pv$ *and* $(1+\epsilon)pv$*, and the minimum cut has value between* $(1-\epsilon)pc$ *and* $(1+\epsilon)pc$*.*

Corollary 4.4. *Assuming* $\epsilon < 1/2$*, the* s-t *min-cut in* $G(p)$ *corresponds to a* $(1+4\epsilon)$*-minimum* s-t *cut in* G*.*

Proof. Assuming that Theorem 4.3 holds, the minimum cut in G is sampled to a cut of value at most $(1+\epsilon)c$ in $G(p)$. So $G(p)$ has minimum cut no larger. And (again by the previous theorem) this minimum cut corresponds to a cut of value at most $(1+\epsilon)c/(1-\epsilon) < (1+4\epsilon)c$ when $\epsilon < 1/2$. $\square$

This means that if we use augmenting paths to find maximum flows in a skeleton, we find them faster than in the original graph for two reasons: the sampled graph has fewer edges, and the value of the maximum flow is smaller. The maximum flow in the skeleton reveals an s-t minimum cut in the skeleton, which corresponds to a near-minimum s-t cut of the original graph. An extension of this idea lets us find near-maximum flows: we randomly partition the graph's edges into many groups (each a skeleton), find maximum flows in each group, and then merge the skeleton flows into a flow in the original graph. Furthermore, once we have an approximately maximum flow, we can turn it into a maximum flow with a small number of augmenting path computations. This leads to an algorithm called `DAUG` that finds a maximum flow in $O(mv\sqrt{(\log n)/c})$ time, improving on the basic augmenting paths algorithm when c is large.

In the following subsections, we detail the algorithms we just sketched. We lead into `DAUG` with some more straightforward algorithms.

4.2.1 Approximate s-t Minimum Cuts

The most obvious application of Theorem 4.3 is to approximate s-t minimum cuts. We can find an approximate s-t minimum cut by finding an s-t minimum cut in a skeleton.

Lemma 4.5. *In a graph with minimum cut c, a $(1+\epsilon)$-approximation to the s-t minimum cut of value v can be computed in $\tilde{O}(mv/\epsilon^3c^2)$ time (with a low probability of error).*

Proof. Given ϵ, determine the corresponding $p = \Theta((\log n)/\epsilon^2 c)$ from Theorem 4.3. Suppose we compute an s-t maximum flow in $G(p)$. By Theorem 4.3, $1/p$ times the value of the computed maximum flow gives a $(1+\epsilon)$-approximation to the s-t min-cut value (with high probability). Furthermore, any flow-saturated (and thus s-t minimum) cut in $G(p)$ will be a $(1+\epsilon)$-minimum s-t cut in G.

By the Chernoff bound [Che52, MR95b], the skeleton has $O(pm)$ edges (that is, about its expectation) with high probability. Also, by Theorem 4.3, the s-t minimum cut in the skeleton has value $O(pv)$. Therefore, the standard augmenting path algorithm can find a skeletal s-t maximum flow in $O((pm)(pv)) = O(mv\log^2 n/\epsilon^4c^2)$ time. Our improved augmenting paths algorithm `DAUG` in Section 4.2.4 lets us shave a factor of $\Theta(\sqrt{pc/\log n}) = \Theta(1/\epsilon)$ from this running time, yielding the claimed bound. $\square$

4.2.2 Approximate Maximum Flows

A slight variation on the previous algorithm will compute approximate maximum flows.

Lemma 4.6. *In a graph with minimum cut c and s-t maximum flow v, a $(1-\epsilon)$-maximum s-t flow can be found in $\tilde{O}(mv/\epsilon c)$ time (with a low probability of error).*

Proof. Given p as determined by ϵ, randomly partition the edges into $1/p$ groups, creating $1/p$ graphs. Each graph looks like (has the distribution of) a p-skeleton, and thus with high probability has an s-t minimum cut of value at least $pv(1-\epsilon)$. It has an s-t maximum flow of the same value that can be computed in $O((pm)(pv))$ time as in the previous section (the skeletons are not independent, but even the sum of the probabilities that any one of them violates the sampling theorem is negligible). Adding the $1/p$ flows that result gives a flow of value $v(1-\epsilon)$. The running time is $O((1/p)(pm)(pv)) = O(mv(\log n)/\epsilon^2 c)$. If we use our improved augmenting path algorithm `DAUG` in Section 4.2.4, we improve the running time by an additional factor of $\Theta(1/\epsilon)$, yielding the claimed bound. □

4.2.3 A Las Vegas Algorithm

Our max-flow and min-cut approximation algorithms are both Monte Carlo, since they are not *guaranteed* to give the correct output (though the error probability can be made arbitrarily small). However, by combining the two approximation algorithms, we can certify the correctness of our results and obtain a *Las Vegas* algorithm for both problems—one that is guaranteed to find the right answer, but has a small probability of taking a long time to do so. This is a standard example of turning a Monte Carlo (error-prone) algorithm into a Las Vegas (correct but occasionally slow) one by checking the correctness of the output and trying again if it is wrong.

Corollary 4.7. *In a graph with minimum cut c and s-t maximum flow v, a $(1-\epsilon)$-maximum s-t flow and a $(1+\epsilon)$-minimum s-t cut can be found in $\tilde{O}(mv/\epsilon c)$ time by a Las Vegas algorithm.*

Proof. Run both the approximate min-cut and approximate max-flow algorithms, obtaining (with high probability) a $(1-\epsilon/2)$-maximum flow of value v_0 and a $(1+\epsilon/2)$-minimum cut of value v_1. We know that $v_0 \le v \le v_1$, so to verify the correctness of the results all we need do is check that $(1+\epsilon/2)v_0 \ge (1-\epsilon/2)v_1$, which happens with high probability. To make the algorithm Las Vegas, we repeat both algorithms until each demonstrates the other's correctness (or switch to a deterministic algorithm if the first randomized attempt fails). We are right on the first try with high probability, so the algorithm runs fast with high probability. □

4.2.4 Exact Maximum Flows

We now use the above sampling ideas to speed up the familiar augmenting paths algorithm for maximum flows. This section is devoted to proving the following theorem:

Theorem 4.8 ([Kar98c]). *In a graph with minimum cut value c, a maximum flow of value v can be found in $\tilde{O}(mv/\sqrt{c})$ time by a Las Vegas algorithm.*

We assume for now that $v \geq \log n$. Our approach is a randomized divide-and-conquer algorithm that we analyze by treating each subproblem as a (non-independent) random sample. This technique gives a general approach to solving packing problems with an augmentation algorithm (including packing bases in a matroid [Kar98b]). The flow that we are attempting to find can be seen as a packing of disjoint s-t paths. We use the algorithm in Figure 3, which we call `DAUG` (Divide-and-conquer AUGmentation).

1. Randomly split the edges of G into two groups (each edge goes to one or the other group with probability $1/2$), yielding graphs G_1 and G_2.
2. Recursively compute s-t maximum flows in G_1 and G_2.
3. Add the two flows, yielding an s-t flow f in G.
4. Use augmenting paths to increase f to a maximum flow.

Figure 3: Algorithm `DAUG`

Note that we cannot apply sampling in `DAUG`'s cleanup phase (Step 4) because the residual graph we manipulate there is directed, while our sampling theorems apply only to undirected graphs. We have left out a condition for terminating the recursion; when the graph is sufficiently small (say with one edge) we use the basic augmenting path algorithm.

The outcome of Steps 1–3 is a flow. Regardless of its value, Step 4 will transform this flow into a maximum flow. Thus, our algorithm is clearly correct; the only question is how fast it runs. Suppose the s-t maximum flow is v. Consider G_1. Since each edge of G is in G_1 with probability $1/2$, we can apply Theorem 4.3 to deduce that with high probability the s-t maximum flow in G_1 is at least $(v/2)(1-\tilde{O}(\sqrt{1/c}))$ and the global minimum cut is $\Theta(c/2)$. The same holds for G_2 (the two graphs are not independent, but this is irrelevant). It follows that the flow f has value $v(1-\tilde{O}(1/\sqrt{c})) = v - \tilde{O}(v/\sqrt{c})$. Therefore the number of augmentations that must be performed in G to make f a maximum flow is $\tilde{O}(v/\sqrt{c})$. Each augmentation takes $O(m)$ time on an m-edge graph. Intuitively, this suggests the following recurrence for the running time of the algorithm in terms of m, v, and c:

$$T(m, v, c) = 2T(m/2, v/2, c/2) + \tilde{O}(mv/\sqrt{c}).$$

(where we use the fact that each of the two subproblems expects to contain $m/2$ edges). If we solve this recurrence, it evaluates to $T(m, v, c) = \tilde{O}(mv/\sqrt{c})$.

Unfortunately, this argument does not constitute a proof because the actual running time recurrence is in fact a *probabilistic recurrence*: the number of edges and sizes of cuts in the subproblems are random variables not guaranteed to equal their expectations. In particular, the recursion arguments is likely to be false when $c = o(\log n)$. Actually proving the result requires some additional work [Kar98c].

5 Randomized Rounding

Next we turn to Randomized Rounding. Randomized Rounding is a powerful method for approximately solving integer programming problems. The basic idea is to take the values of some *relaxation* of the problem (eg a linear program) and use them to generate integer values that define a solution to the integer program. There are two elements of a randomized rounding approach: a good relaxation that preserves much of the structure of the original intractable problem but can be solved efficiently, and a rounding strategy that transforms the relaxed solution into an integer one (along with a proof that it works well).

We apply randomized rounding to two $\mathcal{NP}$-complete problems: network design and graph coloring. Both rounding approaches are slightly unusual. In the network design problem, we simultaneously round against exponentially many constraints. For graph coloring, we use *semidefinite programming* instead of the more traditional linear programming to determine a structure-preserving relaxation.

5.1 Network Design

The network design problem is a mirror to the minimum cut problem. The input is a set of vertices and a collection of candidate edges, each of which can be purchased for some specified cost. The goal is to design a network whose cuts are "sufficiently large." For example, one might wish to build (at minimum cost) a network that is k-connected. Alternatively one might want a network with sufficient capacity to route a certain amount of flow v between two vertices s and t (thus, the network must have s-t minimum cut v). Network design also covers many other classic problems, often $\mathcal{NP}$-complete, including perfect matching, minimum cost flow, Steiner tree, and minimum T-join. A minimum cost 1-connected graph is just a minimum spanning tree, but for larger values of k the minimum-cost k-connected graph problem is $\mathcal{NP}$-complete even when all edge costs are 1 or infinity [ET76].

Whenever a network design problem can be formulated in terms of (lower bound) constraints on the capacity or number of edges crossing each cut, one can write it as an integer linear program with a 0/1 variable for each edge that may be purchased and a constraint for each cut. To make the problem more tractable, we can relax the requirement that variables take 0/1 values and allow them to take fractional values in the interval $[0, 1]$. This gives rise to a linear programming relaxation that can often be solved in polynomial time. Sometimes the linear programs can be represented compactly and solved with standard methods. At other times, even though the relaxation has exponentially many constraints, it has a good separation oracle (e.g. a minimum cut computation for the k-connected subgraph problem) and can thus be solved with the ellipsoid algorithm.

Solving the relaxation yields a fractional solution. Randomized rounding is used to convert the fractional solution back into an integral one. Given fractional variable values $x_1, \ldots, x_m$, we convert them to integer values $y_1, \ldots, y_m$ by setting $y_i = 1$ with probability x_i and 0 otherwise. Note that $E[y_i] = x_i$. It follows that if $ax = b$ for some constraint vector a and scalar b, then $E[ay] = b$. In other words, y is "expected" to satisfy then same constraint that x did.

The problem, of course, is that random experiment deviate somewhat from their expectation. Raghavan and Thompson [RT87] showed that these deviations are often (provably) small enough that the resulting rounded solution is an approximately optimal solution to the integer program. Unfortunately, their analysis is focused on problems with a small number of constraints, which lets them argue that massive deviations from expectation are unlikely to happen. The network design problem has exponentially many constraints, so even unlikely large deviations are likely to occur in some of them. Fortunately, an analogue to our cut sampling theorem bounds these deviations, with the conclusion that randomized rounding can be applied to "fractional graphs" with much the same approximation guarantees as the original Raghavan-Thompson analysis. Among the results this yields is a $1 + O((\log n)/k)$ approximation algorithm for the minimum k-connected subgraph problem [Kar98c].

5.2 Graph Coloring

We also apply randomized rounding to the problem of graph coloring. This problem is $\mathcal{NP}$-complete and has recently been proven extremely hard even to approximate well on graphs with large chromatic number [LY93]. However, there still remains some hope that it might be possible to do reasonably well coloring a graph with small chromatic number. In our thesis, we focus on 3-colorable graphs, and show how to color them with $\tilde{O}(n^{1/4})$ colors. The technique extends to give new performance ratios for graphs with larger chromatic number. This work is joint with Rajeev Motwani and Madhu Sudan [KMS98] and built upon the exciting work of Goemans and Williamson [GW95] on the maximum cut problem. We later improved it in joint work with Avrim Blum [BK97].

To attack graph coloring, we turned to the recently developed technique of *semidefinite programming.* Instead of rounding fractional-valued scalars to integers, we round *vectors.* To illustrate, we describe the relaxation of our graph coloring problem. We aim to assign a unit-length *vector* v_i to each vertex i of our graph such that for any two adjacent vertices i and j, the dot product $v_i \cdot v_j \leq -1/2$. To see that this can be done to any three-colorable graph, consider a "star" of three vectors on the unit circle with 120° angles between them, for example $(1,0)$, $(-1/2, \sqrt{3}/2)$, and $(-1/2, -\sqrt{3}/2)$. Each has unit length and has dot product $-1/2$ with the other two vectors. Given a 3-colored graph, we can solve the vector problem by assigning the first vector to all red vertices, the second to all green, and the third to all blue vertices. This proves that any 3-colorable graph has a feasible solution to our vector problem, which means that it is a valid relaxation.

Solving the relaxation can be formulated as finding a feasible (vector) solution to the following *semidefinite program* (where E denotes the set of edges in the graph G).

$$\begin{aligned} v_i \cdot v_j &\leq -1/2 \text{ if } (i,j) \in E \\ v_i \cdot v_i &= 1. \end{aligned}$$

The fact that such a system of constraints (on any linear combination of dot products) can be solved (to within a negligibly small error) in polynomial time is a difficult result [GLS88] which we can fortunately use as a black box.

Unfortunately, there are many feasible assignments to this semidefinite program—most in a dimension much higher than 2. We cannot constrain the solution to be two dimensional (and still solve the problem in polynomial time) so we must decide how to take a high dimensional relaxed solution and transform it into a coloring. Our method for doing so is quite straightforward: we choose a number of *random* unit vectors as *centers*, and color vertex i with the center closest to v_i. We show that if the number of centers is sufficiently large, no two adjacent vertices are likely to be assigned to the same center—that is, we get a legal coloring. The intuition behind our argument is simple. The vectors for adjacent vertices i and j point "away" from each other thanks to the semidefinite constraints. Thus, if i is "near" a random center, j will be "far" from that center and is thus likely to end up attached to some other center. Some technical arguments involving Gaussian distributions suffice to prove that $\tilde{O}(n^{1/4})$ centers suffice to make the probabilities work out.

6 Monte Carlo Estimation

The last randomization technique we consider is Monte Carlo estimation. The technique is applied when we want to estimate the probability p of a given event over some probability space. Monte Carlo estimation carries out repeated "trials" (samples from the probability space) and measures how often the given event occurs. This gives a natural estimate of the event probability.

We use Monte-Carlo estimation to attack the *all-terminal network reliability problem*: given a network on n vertices, each of whose m links is assumed to fail (disappear) independently with some probability, determine the probability that the surviving network is connected. The practical applications of this question to communication networks are obvious, and the problem has therefore been the subject of a great deal of study. A comprehensive survey can be found in [Col87]. As mentioned in Section 2, this problem is $\sharp\mathcal{P}$-hard to solve exactly, so we give a *fully polynomial randomized approximation scheme (FPRAS)* that gives an answer accurate to within a relative error of ϵ in time polynomial in n and $1/\epsilon$. Although our algorithm is quite general [Kar98d], we restrict discussion here to the case where every edge fails independently with the same probability p. We let $\mathrm{FAIL}(p)$ denote the failure probability of G when edges fail with probability p.

The basic approach of our FPRAS is to consider two cases. When $\mathrm{FAIL}(p)$ is large, we estimate it in polynomial time by direct Monte Carlo simulation of edge failures. That is, we randomly fail edges and check whether the graph remains connected. Since $\mathrm{FAIL}(p)$ is large, a small number of trials gives enough data to estimate it well. When $\mathrm{FAIL}(p)$ is small, we show that we can focus on the small cuts in a graph. We enumerate them with our cut algorithms and then use a biased Monte Carlo estimation technique to determine their failure probability.

Observe that a graph becomes disconnected precisely when all of the edges in some cut of the graph fail. If each edge fails with probability p, then the probability that a k-edge cut fails is p^k. Thus, the smaller a cut, the more likely it is to fail. It is therefore natural to focus attention on the small graph cuts. In particular, the probability that the graph becomes disconnected is at least p^c (since this is the probability that a minimum cut fails). At the same time, the probability that any one α-minimum cut fails is $p^{\alpha c}$.

We can now describe our two cases. When $\mathrm{FAIL}(p) \geq p^c \geq n^{-3}$, we use direct Monte Carlo simulation to estimate the failure probability. A single experiment consists of flipping coins to see which edges fail and then checking whether the graph is connected. If we carry out roughly $(\log n)/\epsilon^2 \mathrm{FAIL}(p) = \tilde{O}(n^3/\epsilon^2)$ experiments (a polynomial number), we will see about $(\log n)/\epsilon^2$ failures. This provides enough "evidence" to give a good estimate of the failure probability [Che52, KLM89].

Unfortunately, when $\mathrm{FAIL}(p)$ is small, we need too many simulations to develop a good baseline (note that we do not expect to see a single failure until we perform $1/\mathrm{FAIL}(p)$ experiments; this number can be super-polynomial). We instead turn to an enumeration of the small cuts. When $p^c \leq n^{-3}$, we know that a given α-minimum cut fails with probability $p^{\alpha c} \leq n^{-3\alpha}$. But we argued in our Cut Counting theorem that the number of α-minimum cuts is only $n^{2\alpha}$. It follows that the probability that *any* α-minimum cut fails is less than $n^{-\alpha}$—that is, exponentially decreasing with α. Thus, for a relatively small α, the probability that a greater than α-minimum cut fails is negligible. We can therefore approximate $\mathrm{FAIL}(p)$ by approximating the probability that some less than α-minimum cut fails. We do so by enumerating the α-minimum cuts (using a modification of the Contraction Algorithm [KS96]) and then applying a *DNF counting* algorithm developed by Karp, Luby, and Madras [KLM89]. The algorithm of [KLM89] is also based on Monte-Carlo methods, but uses biased sampling to ensure that we see failures often so that a good estimate of their likelihood can be constructed quickly. The contribution of our work is to show that it is possible to build a small formula that can be fed to the DNF counting algorithm to produce a meaningful answer.

7 Conclusion

Randomization has become an essential tool in the design of optimization algorithms. Randomization leads to algorithms that are faster, simpler, and/or better-performing than their deterministic counterparts. The basic techniques of random selection, random sampling, randomized rounding and Monte Carlo estimation let us draw on our intuitions about common cases and representative samples: whenever we expect that something should "usually" happen or be "typical," randomization may give us a way to turn our suspicion into an algorithm. We have demonstrated this approach on numerous basic optimization problems. But a great deal of work remains to be done.

The most direct open question is how far our particular results can be pushed. The minimum spanning tree and minimum cut problems are essentially "done," one with a linear time algorithm and the other with a linear-times-polylog time algorithm; but our results on s-t minimum cuts and maximum flows seem very incomplete: no lower bounds are evident, and our upper bounds are "odd" (e.g. $\tilde{O}(n^{20/9})$ for flows in simple graphs [KL98]) in ways that suggest that it must be possible to improve them (e.g. to $O(n^2)$). Our approximation algorithms apply well to both capacitated and uncapacitated problems, but our exact algorithms so far apply best to uncapacitated problems. We suspect that more can be done here.

More questionable is whether any of our technology can be applied to *directed* graphs. Absolutely none of the results discussed in this article extend to directed graphs: the Contraction Algorithm fails on them, and as a result we have been unable to prove a sampling theorem, a cut counting theorem (in fact a directed graph can have exponentially many minimum cuts), a sampling theorem, a rounding theorem, or anything about directed reliability. One possible explanation for this is that undirected graphs form natural matroids while directed graphs do not [Kar98b].

Thinking more broadly, a fundamental question about randomization is whether it is truly "necessary." Often, after a randomized algorithm gives insight into a problem, one can devise a deterministic algorithm with some of the same properties. Within theoretical computer science, there is an entire subfield devoted to *derandomization*—the development of techniques that will mechanically convert a randomized algorithm into a deterministic one. For example, the randomized rounding procedure for (polynomial size) linear programs can be made deterministic [Rag88], as can our randomized rounding algorithm for graph coloring [MR95a]. We have also derandomized our Contraction Algorithm [KM97].

Even when it is possible to derandomize an algorithm, it may not be worth doing so. The derandomization can add complexity, either computational (e.g. in the case of the Contraction Algorithm, where the derandomization drastically slows the algorithm) or conceptual (e.g. for randomized rounding, where the intuitive expectation argument is replaced by a more complex numeric calculation).

However, there are still motivations for exploring the derandomization question. Perhaps the strongest is the wish for an algorithm with predictable behavior. In a situation with lives at stake, it would be unsatisfactory to be right *most* of the time, or *usually* fast enough. This problem is particularly acute with our Monte Carlo algorithms, where one cannot even tell whether the answer is correct! An obvious place to begin is the minimum cut problem, where a Monte Carlo algorithm can solve the problem (with high probability) in $\tilde{O}(m)$

time but the best known deterministic running time is $\tilde{O}(mn)$. Another specific question is whether there is a *deterministic* linear-time minimum spanning tree algorithm (which would finally put the problem to rest for good). A more abstract question is the following: we have proven that any graph has a sparse "skeleton" that accurately approximates its cuts; this seems to have assorted uses. Can such a skeleton be constructed deterministically in polynomial time?

Comments and questions on this survey are most welcome.

References

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[BK96] A. A. Benczúr and D. R. Karger. Approximate s–t min-cuts in $\tilde{O}(n^2)$ time. In G. Miller, editor, *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing*, pages 47–55. ACM, ACM Press, May 1996.

[BK97] A. Blum and D. R. Karger. Improved approximation for graph coloring. *Information Processing Letters*, 61(1):49–53, January 1997.

[BK98] A. A. Benczúr and D. R. Karger. Augmenting undirected edge connectivity in $\tilde{O}(n^2)$ time. In H. Karloff, editor, *Proceedings of the* 9^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 500–509. ACM-SIAM, January 1998.

[Blu94] A. Blum. New approximation algorithms for graph coloring. *Journal of the ACM*, 41(3):470–516, May 1994.

[CGK+97] C. C. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In M. Saks, editor, *Proceedings of the* 8^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333. ACM-SIAM, January 1997.

[Che52] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.

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

[ET76] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5:653–665, 1976.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, New Jersey, 1962.

[FR75] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[GGP+94] M. X. Goemans, A. Goldberg, S. Plotkin, D. Shmoys, É. Tardos, and D. Williamson. Improved approximation algorithms for network design problems. In D. D. Sleator, editor, *Proceedings of the* 5^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232. ACM-SIAM, January 1994.

[GGST86] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning tree in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[GLS88] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer-Verlag, 1988.

[GR97] A. Goldberg and S. Rao. Beyond the flow decomposition barrier. In *Proceedings of the* 30^{th} *Annual Symposium on the Foundations of Computer Science*, pages 2–11. IEEE, IEEE Computer Society Press, October 1997.

[GT88] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 1995.

[HO94] Hao and Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994. A preliminary version appeared in Proceedings of the 3^{rd} Annual ACM-SIAM Symposium on Discrete Algorithms.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[Kar94] D. R. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Stanford University, Stanford, CA 94305, 1994. Contact at `karger@lcs.mit.edu`. Available from `http://theory.lcs.mit.edu/~karger`.

[Kar96] D. R. Karger. Minimum cuts in near-linear time. In G. Miller, editor, *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing*, pages 56–63. ACM, ACM Press, May 1996.

[Kar98a] D. R. Karger. Better random sampling algorithms for flows in undirected graphs. In H. Karloff, editor, *Proceedings of the* 9^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 490–499. ACM-SIAM, January 1998.

[Kar98b] D. R. Karger. Random sampling and greedy sparsification in matroid optimization problems. *Mathematical Programmming B*, 82(1–2):41–81, June 1998. A preliminary version appeared in Proceedings of the 34^{th} Annual Symposium on the Foundations of Computer Science.

[Kar98c] D. R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 1998. To appear. A preliminary version appeared in Proceedings of the 26^{th} ACM Symposium on Theory of Computing.

[Kar98d] D. R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. *SIAM Journal on Computing*, 1998. To appear. A preliminary version appeared in Proceedings of the 27^{th} ACM Symposium on Theory of Computing.

[Kar98e] H. Karloff, editor. *Proceedings of the 9^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, January 1998.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[KL98] D. R. Karger and M. Levine. Finding maximum flows in simple undirected graphs seems faster than bipartite matching. In *Proceedings of the 29^{th} ACM Symposium on Theory of Computing*. ACM, ACM Press, May 1998.

[KLM89] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, September 1989.

[KM97] D. R. Karger and R. Motwani. Derandomization through approximation: An $\mathcal{NC}$ algorithm for minimum cuts. *SIAM Journal on Computing*, 26(1):255–272, 1997. A preliminary version appeared in Proceedings of the 25^{th} ACM Symposium on Theory of Computing, p. 497.

[KMS98] D. R. Karger, R. Motwani, and M. Sudan. Approximate graph coloring by semidefinite programming. *Journal of the ACM*, 45(2):246–265, March 1998.

[KS96] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, July 1996. Preliminary portions appeared in SODA 1992 and STOC 1993.

[KT97] D. R. Karger and R. P. Tai. Implementing a fully polynomial time approximation scheme for all terminal network reliability. In M. Saks, editor, *Proceedings of the 8^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 334–343. ACM-SIAM, January 1997.

[LY93] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. In A. Aggarwal, editor, *Proceedings of the* 25^{th} *ACM Symposium on Theory of Computing*, pages 286–293. ACM, ACM Press, May 1993.

[Mil96] G. Miller, editor. *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing*. ACM, ACM Press, May 1996.

[MR95a] S. Mahajan and H. Ramesh. Derandomizing semidefinite programming based approximation algorithms. In *Proceedings of the* 36^{th} *Annual Symposium on the Foundations of Computer Science*, pages 162–169. IEEE, IEEE Computer Society Press, October 1995.

[MR95b] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[NI92] H. Nagamochi and T. Ibaraki. Linear time algorithms for finding k-edge connected and k-node connected spanning subgraphs. *Algorithmica*, 7:583–596, 1992.

[Rag88] P. Raghavan. Probabilistic construction of deterministic algorithms: Approximate packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–43, October 1988.

[RT87] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[Sak97] M. Saks, editor. *Proceedings of the* 8^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms.* ACM-SIAM, January 1997.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* SIAM, 1983.