Random Sampling in Cut, Flow, and Network Design Problems

David R. Karger*

October 19, 2000

Abstract

We use random sampling as a tool for solving undirected graph problems. We show that the sparse graph, or *skeleton*, that arises when we randomly sample a graph's edges will accurately approximate the value of *all* cuts in the original graph with high probability. This makes sampling effective for problems involving cuts in graphs.

We present fast randomized (Monte Carlo and Las Vegas) algorithms for approximating and exactly finding minimum cuts and maximum flows in unweighted, undirected graphs. Our cut-approximation algorithms extend unchanged to weighted graphs while our weighted-graph flow algorithms are somewhat slower. Our approach gives a general paradigm with potential applications to any packing problem. It has since been used in a near-linear time algorithm for finding minimum cuts, as well as faster cut and flow algorithms.

Our sampling theorems also yield faster algorithms for several other cut-based problems, including approximating the best balanced cut of a graph, finding a k-connected orientation of a $2k$-connected graph, and finding integral multicommodity flows in graphs with a great deal of excess capacity. Our methods also improve the efficiency of some parallel cut and flow algorithms.

Our methods also apply to the *network design* problem, where we wish to build a network satisfying certain connectivity requirements between vertices. We can purchase edges of various costs and wish to satisfy the requirements at minimum total cost. Since our sampling theorems apply even when the sampling probabilities are different for different edges, we can apply *randomized rounding* to solve network design problems. This gives approximation algorithms that guarantee much better approximations than previous algorithms whenever the minimum connectivity requirement is large. As a particular example, we improve the best approximation bound for the minimum k-connected subgraph problem from 1.85 to $1 + O(\sqrt{(\log n)/k})$.

*MIT Laboratory for Computer Science, Cambridge, MA 02139. email: `karger@lcs.mit.edu`
URL: `http://theory.lcs.mit.edu/~karger`

1 Introduction

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.

We apply the concept of a representative sample to combinatorial optimization problems on graphs. Given an optimization problem, it may be possible to generate a small representative subproblem by random sampling. Intuitively, such a subproblem should form a microcosm of the larger problem. We can examine the subproblem and use it to glean information about the original problem. Since the subproblem is small, we can spend proportionately more time analyzing it than we would spend examining the original problem. Sometime, an optimal solution to the subproblem will 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.

We show this approach to be effective for problems involving cuts in graphs. A *cut* in an undirected graph is a partition of the graph's vertices into two nonempty sets. The *value* of the cut is the number, or for a weighted graph the total weight, of edges with one endpoint in each set. Cuts play an important role in determining the solutions to many graph problems. Most obviously, the connectivity of a graph is the minimum value of a cut in the graph. Similarly, the s-t maximum flow is determined by the smallest of all cuts that separate s and t—that is, the s-t minimum cut. In the $\mathcal{NP}$-complete *network design problem,* the goal is to build a graph that satisfies certain specified connectivity requirements by containing no small cuts. A special case is to find a minimum size (number of edges) k-connected subgraph of a k-connected graph. Other problems to which cuts are relevant include finding a minimum *balanced* cut (in which both sides of the cut are "large") and finding an orientation (assignment of directions) of the edges of an undirected graph that makes it k-connected as a directed graph. Cuts also play an important role in multicommodity flow problems, though the connection is not as tight as for the standard max-flow problem (Leighton and Rao 1988; Linial, London, and Rabinovich 1995; Aumann and Rabani 1998).

Random sampling helps us solve cut-dependent undirected graph problems. We define and use a *graph skeleton.* Given a graph, a skeleton is constructed on the same set of vertices by including a small random sample of the graph's edges. Our main result is that (with high probability) a skeleton accurately approximates all cut values in the original graph. This means random subgraphs can often be used as substitutes for the original graphs in cut and flow problems. Since the subgraphs are small, improved time bounds result.

In the most obvious application, by computing minimum cuts and maximum flows in the skeleton, we get fast algorithms for approximating global minimum cuts, s-t minimum cuts and maximum flows. For example, we give a near-linear-time algorithm for approximating the global minimum cut of a graph to within any constant factor with high probability. Furthermore, a randomized divide and conquer technique finds exact solutions more quickly than before. For example, we improve the time to find a minimum cut of value c in an m-edge unweighted (that is, with all edges having the same, unit, capacity) graph from $\tilde{O}(mc)$ (Gabow 1995) to $\tilde{O}(m\sqrt{c})$ (The notation $\tilde{O}(f)$ denotes $O(f \operatorname{polylog} f)$). This in turn yields faster algorithms for constructing the cactus representation of minimum cuts in a graph and for optimally augmenting graph connectivity. We improve the time to find a maximum flow of value v from $O(mv)$ to $\tilde{O}(mv/\sqrt{c})$. We improve the total work done by some parallel cut and flow algorithms. We also give applications to balanced cuts and orientations and to integral multicommodity flows.

While this work can stand independently, perhaps its greater value is in proving results on sampling that have since found several applications. The major improvement has been to eliminate the dependence on the minimum cut c appearing in this paper's results. Benczúr and Karger (1996) extend the sampling construction to weighted graphs, showing how to approximate s-t minimum cuts with high probability in $\tilde{O}(n^2)$ time. This author used sampling in an algorithm to find an *exact* minimum cut in any (weighted or unweighted) undirected graph with high probability in $\tilde{O}(m)$ time (Karger 1996). More recently, this author gave a faster, sampling-based algorithm that finds a maximum flow of value v in $\tilde{O}(\sqrt{mnv})$ time with high probability (Karger 1998a). Karger and Levine (1998) gave an even faster $\tilde{O}(nv^{5/4})$-time algorithm for simple graphs. All of these new results rely directly on this paper's sampling theorems and algorithms.

Our approach to maximum flows and minimum cuts exemplifies a natural random-sampling approach to *packing problems* in which the goal is to find a maximum collection of feasible subsets of some input universe. In the s-t maximum flow problem the universe is the graph's edges and the feasible sets are s-t paths. A different (tree-) packing problem corresponds to global minimum cuts. In a different paper (Karger 1998b), we show that the paradigm also applies to the problem of packing bases in a matroid.

Our approach also applies to certain *covering* problems. From random sampling, it is a small step to show that *randomized rounding* (Raghavan and Thompson 1987) can be effectively applied to graphs with fractional edge weights, yielding integrally weighted graphs with roughly the same cut values. This makes randomized rounding a useful tool in *network design* problems. In these $\mathcal{NP}$-complete problems, the goal is to construct a minimum-cost network satisfying certain connectivity demands (for example, the Steiner tree problem asks for the minimum cost subgraph connecting a certain set of vertices). For the version where edges can be reused, we give a polynomial time approximation algorithm with an approximation bound of $1 + O(\sqrt{(\log n)/f_{\min}})$, where $f_{\min}$ is the connectivity (minimum cut) of the optimum solution (and thus at least the minimum connectivity demand between two vertices). Previous approximation algorithms had bounds depending on the *maximum* connectivity demands $f_{\max}$, the best being $O(\log f_{\max})$ for a large class of problems (Agrawal, Klein, and Ravi 1995) . We get a $1 + \tilde{O}(1/\sqrt{k})$ bound for the *minimum k-connected subgraph problem* (where edges cannot be reused, all connectivity demands are k, and edge costs are 1 or infinity). For sufficiently large k this improves on a previous approximation ratio of 1.85 (Khuller and Raghavachari 1995). We also improve bounds for various other single-edge-use problems.

All of our techniques apply only to undirected graphs, as cuts in directed graphs do not appear to have the same predictable behavior under random sampling.

Preliminary versions of this work appeared in conference proceedings (Karger 1994a; Karger 1994c). A more extensive treatment is provided in the author's dissertation (Karger 1994b).

The remainder of this introduction includes a more detailed description of our results as well as a comparison to previous and subsequent work, followed by some definitions. Section 2 then presents our main theorem on cuts in sampled graphs. The paper then splits into two parts that can be read independently. In the first part, we show how to accelerate algorithms for computing s-t maximum flows and minimum cuts (Section 3) and global minimum cuts (Section 4) in unweighted graphs, with extensions to weighted graphs (Section 5). Section 6 describes applications to other cut problems. The second part of the paper discusses applications of the sampling theorem and randomized rounding to network design problems. In Section 7, we lay the groundwork and address the version where edges can be reused. In Section 8 we discuss the harder case in which edges can only be used once.

1.1 Definitions

We make the following definitions. Consider a statement that refers to a variable n. We say that the statement holds *with high probability (w.h.p.) in n* if for any constant d, there is a setting of constants in the statement (typically hidden by O-notation) such that the probability the statement fails to hold is $O(n^{-d})$.

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. We say an event regarding the algorithm occurs *with high probability (w.h.p.)* if it occurs with high probability in the problem size (that is, with probability at least $1-n^{-d}$ on problems of size n) and with low probability if the complementary event 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.

Finally, we remark that all logarithms in the paper are base 2 and recall that $\tilde{O}(f)$ denotes $O(f \operatorname{polylog} n)$.

1.2 Cuts and Flows

In the first part of this paper we present algorithms for approximating and for exactly finding s-t and global minimum cuts and maximum flows. To this end, we make the following definition:

Definition 1.1. An *α-minimum cut* is a cut whose value is at most α times that of the (global) minimum cut. An *α-minimum s-t cut* is defined similarly. An *α-maximum s-t flow* is an s-t flow whose value is at least α times the optimum.

We show that if we pick a small random sample of a graph's edges, then we get a graph whose minimum cuts correspond (under the same vertex partition) to $(1+\epsilon)$-minimum cuts of the original graph. Therefore, we can approximate minimum cuts by computing minimum cuts in a sampled graph. These cuts are found using *augmenting path* algorithms whose running times increase with both the size of the graph and the value of the output cut. Both of these quantities are smaller in the sampled graph, so we get a speedup for two different reasons. We extend these ideas to find approximately maximum flows by randomly partitioning the graph's edges and finding flows separately in each resulting edge group. Finally, we find exact flows by using augmenting path algorithms to "repair" the errors introduced by the approximation algorithms. Since the error is small, the repair takes little time.

Throughout this paper, we focus attention on an n vertex, m edge graph with minimum cut c and s-t minimum cut v. We give randomized Monte Carlo (MC) and Las Vegas (LV) algorithms to find the following objects in unweighted, undirected graphs:

- A global minimum cut in $\tilde{O}(m\sqrt{c})$ time (LV),
- A $(1+\epsilon)$-minimum cut in $\tilde{O}(m+n/\epsilon^3)$ time (MC) or $\tilde{O}(m/\epsilon)$ time (LV),
- An s-t maximum flow in $\tilde{O}(mv/\sqrt{c})$ time (LV),
- A $(1+\epsilon)$-minimum s-t cut in $O(m+n(v/c)^2\epsilon^{-3}) = O(mv/\epsilon^3 c^2)$ time (MC) or $\tilde{O}(mv/\epsilon c)$ time (LV),
- A $(1-\epsilon)$-maximum s-t flow in $\tilde{O}(mv/\epsilon c)$ time (LV).

Our cut approximation algorithms extend to weighted graphs with roughly the same time bounds. The flow approximation algorithms and exact algorithms use a "scaling" technique that, for a given maximum edge weight U, increases the time bounds of the flow algorithms by a factor of $\sqrt{U}$ rather than the naive factor of U.

Our approximation algorithms are in fact meta-algorithms: for example, given any algorithm to find an s-t minimum cut in time $T(m,n,v)$, we can approximate the cut in time $T(m/c,n,v/c)$. Previously, the best time bound for computing maximum flows in unweighted graphs was $O(m \cdot \min(v, n^{2/3}, \sqrt{m}))$, achieved using blocking flows (cf. Tarjan (1983, Ahuja, Magnanti, and Orlin (1993)). In the *unit graphs* that arise in bipartite matching problems, a running time of $O(m\sqrt{n})$ is known. Our exact algorithm improves on these bounds whenever $v/\sqrt{c}$ is small, and in particular when c is large. We are aware of no previous work on approximating s-t minimum cuts or maximum flows, although blocking flows can be used to achieve a certain large absolute error bound.

This work relates to several previous algorithms for finding minimum cuts. The Contraction Algorithm (Karger and Stein 1993) runs in $O(n^2 \log^3 n)$ time on undirected (weighted or unweighted) graphs. Gabow's Round Robin Algorithm (Gabow 1995) runs in $O(mc \log(n^2/m))$ time on unweighted (directed or undirected) graphs.

Matula (1993) gave a deterministic linear-time algorithm for finding a $(2+\epsilon)$-minimum cut in unweighted, undirected graphs. It is easily extended to run in near-linear time on weighted graphs (Karger 1994b).

As mentioned above, since this work appeared, Benczúr and Karger (1996) have given an $\tilde{O}(n^2)$ time algorithm for approximating s-t minimum cuts, Karger (1996) has given an $\tilde{O}(m)$ time algorithm for finding an exact minimum cut, and Karger and Levine (1998) have given an $O(nv^{5/4})$-time algorithm for finding a flow of value v, all regardless of c.

1.3 Network Design

In the second part of our paper we discuss the *network design problem.* We start with a set of vertices and "purchase" various edges in order to build a graph satisfying certain connectivity demands between the vertices. Each edge has an associated cost, and our goal is to meet the demands at minimum total cost. The minimum spanning tree problem is a special case where the "demand" is that all vertices be connected. Network design also covers many other classic problems, some $\mathcal{NP}$-complete, including perfect matching, minimum cost flow, Steiner tree, and minimum T-join. It also captures the *minimum cost k-connected subgraph problem,* where the goal is to build a minimum cost graph with minimum cut k. The minimum cost 1-connected subgraph is just the minimum spanning tree, but for larger values of k the problem is $\mathcal{NP}$-complete even when all edge costs are 1 or infinity (Eswaran and Tarjan 1976).

Agrawal, Klein, and Ravi (1995) studied a special case of network design called the *generalized Steiner problem*, first formulated by Krarup (see Winter (1987)). In this version, the demands are specified by giving a minimum connectivity d_{ij} that the output graph must satisfy between each pair of vertices i and j (setting all $d_{ij}=k$ gives the minimum cost k-connected subgraph problem). Assuming edges can be used repeatedly, they gave an $O(\log f_{\max})$-approximation algorithm, where $f_{\max}$ is the maximum demand across any cut (*i.e.* $\max d_{ij}$). This extended previous work (Goemans and Bertsimas 1993) on the special case where $d_{ij}=\min(d_i,d_j)$ for given "connectivity types" d_i. Aggarwal and Garg (1994) gave an algorithm with performance ratio $O(\log k)$, where k is the number of sites with nonzero connectivity demands.

A pair of papers (Williamson, Goemans, Mihail, and Vazirani 1993; Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson 1994) extended the $O(\log f_{\max})$ bound of Agrawal, Klein, and Ravi (1995) to the harder case where edges can be used only once, and extended the approximation technique to a larger class of network design problems. They also noted that for a wide range of problems (including all those just mentioned) a *fractional* solution can be found in polynomial time by using the ellipsoid algorithm.

Our graph skeleton construction can sample edges with different probabilities. This lets us apply Raghavan and Thompson's(1987) *randomized rounding* technique to the fractional solutions and get good approximation ratios, despite the fact that the rounding must simultaneously satisfy exponentially many constraints. Rounding a fractional solution gives an integral one whose cuts are all approximately equal to their fractional values (which were constrained to exceed the corresponding demands). The only complication is in the possibility that the rounded values might be slightly below the demands. When edges can be reused, this is easy: we simply increase each fractional weight slightly before rounding. This yields an approximation algorithm with a ratio of $1+O(\sqrt{(\log n)/f_{\min}}+(\log n)/f_{\min})$ for arbitrary edge costs, where $f_{\min}$ is the minimum demand across a cut.

When edges cannot be reused, increasing the fractional weights may not be possible. However, some more complicated techniques can often be applied instead. For the minimum k-connected subgraph problem with $k \geq \log n$, we give an approximation algorithm with performance ratio $1 + O(\sqrt{(\log n)/k})$. For any $k \gg \log n$, this improves on the previous best known approximation factor of 1.85 (Khuller and Raghavachari 1995). For general network design problems, we extend the Williamson et al. bound of $O(\log f_{\max})$ to $O\left(\log\left(f_{\max}\frac{\log n}{f_{\min}}\right)\right)$.

1.4 Related Work

Random sampling is a powerful general tool in algorithm design. It appears in a fast and elegant algorithm for finding the median of an ordered set (Floyd and Rivest 1975). It has many applications in computational geometry (Clarkson 1987; Clarkson and Shor 1987) and in particular in fixed-dimension linear and integer programming (Clarkson 1995). Random sampling drives the first linear-time minimum spanning tree algorithm (Karger, Klein, and Tarjan 1995). This author (Karger 1998b) shows how it can speed up algorithms for matroid optimization and for packing matroid bases.

Skeletons are conceptually related to *sparse graph certificates.* Certificates apply to any monotone increasing property of graphs—one that holds for G if it holds for some subgraph of G. Given such a property, a sparse certificate for G is a sparse subgraph that has the property, proving that G has it as well. The advantage is that since the certificate is sparse, the property can be verified more quickly. For example, sparsification techniques improve the running times of dynamic algorithms for numerous graph problems such as connectivity, bipartitioning, and and minimum spanning trees (Eppstein, Galil, Italiano, and Nissenzweig 1992). The skeleton is a kind of sparse *approximate* certificate.

A sparse certificate particularly relevant to this paper is the *sparse k-connectivity certificate*. For any graph, a sparse k-connectivity certificate is a kn-edge subgraph of G such that all cuts of value at most k in G have the same value in the subgraph. This differs from our skeleton in that cuts of value less than k have their values preserved exactly, but cuts of greater value are not preserved at all. Nagamochi and Ibaraki (1992b) give an algorithm that takes a graph and a parameter k and returns a sparse k-connectivity certificate. It runs in $O(m)$ time on unweighted graphs. In weighted graphs, where the resulting certificate has total *weight* kn and preserves cuts of value up to k, the running time increases to $O(m + n \log n)$ (Nagamochi and Ibaraki 1992a).

If we are looking for cuts or flows of value less than k, we can find them in the certificate, taking less time since the certificate has fewer edges. For example a sparse certificate can be constructed before Gabow's (1995) minimum cut algorithm is executed; this improves the algorithm's running time from $\tilde{O}(mc)$ to $\tilde{O}(m+nc^{3/2})$. Like Gabow's, all of our cut and flow algorithms can use this preprocessing step. As a result, m can be replaced by nc in all the bounds for our min-cut algorithms and min-cut approximation algorithms (since if we find a $2cn$-connectivity certificate, it will have the same minimum cuts and approximate minimum cuts as the original graph). Similarly, m can be replaced by nv in all of our s-t cut and flow algorithms since a v-certificate preserves all flows of value v. However, it clarifies the presentation to keep m in the time bounds and leave the obvious substitution to the reader.

2 Randomly Sampling Graph Edges

Our algorithms are all based upon the following model of random sampling in graphs. We are given an unweighted graph G with a *sampling probability* p_e for each edge e, and we construct a random subgraph, or *skeleton,* on the same vertices by placing each edge e in the skeleton independently with probability p_e. Let $\hat{G}$ denote the weighted graph with the vertices and edges of G and with edge weight p_e assigned to edge e, and let $\hat{c}$ be the minimum cut (by weight) of $\hat{G}$. Note that $\hat{G}$ is not the skeleton (a random object), but is rather an "expected value" of the skeleton, since the value of a cut in $\hat{G}$ is the expected value of the corresponding cut in the skeleton. The quantity $\hat{c}$ is the minimum expected value of any cut, though not necessarily the expected value of the minimum cut. Our main theorem says that so long as $\hat{c}$ is sufficiently large, every cut in the skeleton takes on roughly its expected value.

Theorem 2.1. *Let $\epsilon = \sqrt{3(d+2)(\ln n)/\hat{c}}$. If $\epsilon \leq 1$ then, with probability $1 - O(1/n^d)$, every cut in the skeleton of G has value between $1-\epsilon$ and $1+\epsilon$ times its expected value.*

To see the tightness of this theorem, note that if $\epsilon = \sqrt{3(d)(\ln n)/\hat{c}}$ then the Chernoff bound (below) only gives a $1/n^d$ bound on the probability that one particular minimum cut diverges by ϵ from its expected value. By changing d to $d+2$, we extend from the minimum cut to all cuts. To prove this theorem, we require two lemmas.

Lemma 2.2 (Karger and Stein (1996)). *In an undirected graph, the number of α-minimum cuts is less than $n^{2\alpha}$.*

Proof. A proof appears in the appendix. It is a minor variant of one that appeared previously (Karger and Stein 1996). A quite different proof has also been developed (Karger 1996). □

Lemma 2.3 (Chernoff (1952), cf. Motwani and Raghavan (1995)). *Let X be a sum of independent Bernoulli (that is, 0/1) random variables with success probabilities $p_1, \ldots, p_n$ and expected value $\mu = \sum p_i$. Then for $\epsilon \leq 1$*

$$\Pr[|X - \mu| > \epsilon\mu] \quad \leq \quad 2e^{-\epsilon^2\mu/3}$$

Lemma 2.2 applied to $\hat{G}$ states that the number of cuts with expected value within an α factor of the minimum less that $\alpha\hat{c}$ increases exponentially with α. On the other hand, Lemma 2.3 says that the probability that one such cut diverges too far from its expected value decreases exponentially with α. Combining these two lemmas and balancing the exponential rates proves the theorem. There is a simple generalization to the case $\epsilon > 1$ that we omit since it will not be used in the paper.

Proof of Theorem 2.1. Let $r = 2^n - 2$ be the number of cuts in the graph, and let $c_1, \ldots, c_r$ be the expected values of the r cuts in the skeleton listed in nondecreasing order so that $\hat{c} = c_1 \leq c_2, \cdots \leq c_r$. Let p_k be the probability that the k^{th} cut diverges by more than ϵ from its expected value. Then the probability that some cut diverges by more than ϵ is at most $\sum p_k$, which we proceed to bound from above.

Note that the (sampled) value of a cut is a sum of Bernoulli variables, so the Chernoff bound says that $p_k \leq e^{-\epsilon^2 c_k/3}$. Note that we have arranged that $e^{-\epsilon^2\hat{c}/3} = n^{-(d+2)}$. We now proceed in two steps. First, consider the n^2 smallest cuts. Each of them has $c_k \geq \hat{c}$ and thus $p_k \leq 2n^{-(d+2)}$, so that

$$\sum_{k \leq n^2} p_k \leq (n^2)(2n^{-(d+2)}) = 2n^{-d}.$$

Next, consider the remaining larger cuts. According to Lemma 2.2, there are less than $n^{2\alpha}$ cuts of expected value less than $\alpha\hat{c}$. Since we have numbered the cuts in increasing order, this means that $c_{n^{2\alpha}} \geq \alpha\hat{c}$. In other words, writing $k = n^{2\alpha}$,

$$c_k \geq \frac{\ln k}{2\ln n} \cdot \hat{c},$$

and thus

$$p_k \leq 2k^{-(d+2)/2}.$$

It follows that

$$\begin{aligned}
\sum_{k>n^2} p_k &\leq \sum_{k>n^2} 2k^{-(d+2)/2} \\
&\leq \int_{n^2}^{\infty} 2k^{-(d+2)/2} \\
&= \left.\frac{4}{d}k^{-d/2}\right|_{n^2}^{\infty} \\
&= O(n^{-d})
\end{aligned}$$

2.1 Constructing p-Skeletons

In the first part of this paper, we will generally fix some value p and set $p_e = p$ for all e. We call the resulting sample a *p-skeleton* of G and denote it $G(p)$. To avoid making exceptions for a special case let us define $G(p) = G$ for $p > 1$. We have the following immediate corollary to our sampling theorem.

Corollary 2.4. *Let G be any graph with minimum cut c and let $p = 3(d+2)(\ln n)/\epsilon^2 c$. Then the probability that the value of some cut in $G(p)$ has value more than $(1+\epsilon)$ or less than $(1-\epsilon)$ times its expected value is $O(n^{-d})$.*

Proof. Note that the minimum expected cut is $\hat{c} = pc$ and apply Theorem 2.1. □

Lemma 2.5. *A p-skeleton of an unweighted graph can be constructed in $O(m)$ time.*

Proof. To generate a skeleton we can flip an appropriately biased coin for each edge. In some models of computation, this is treated as a unit cost operation. If we want to be stricter, we can use the weaker model in which only an unbiased random bit can be generated in unit time. This would most obviously imply an $O(m \log 1/p)$ time bound for generating a skeleton. However, even in this model, it is possible to perform the m biased coin flips in $O(m)$ time with high probability (Knuth and Yao 1976), cf. Karger (1994b). □

2.2 Determining the Right p

Our approximation algorithms are based upon constructing p-skeletons. In these algorithms, given a desired approximation bound ϵ, we will want to sample with the corresponding $p = \Theta((\ln n)/(\epsilon^2 c))$ of Corollary 2.4 in order to ensure that in the skeleton no cut diverges in value by more than ϵ times its expectation. This would appear to require knowledge of c. However, it is sufficient to have a constant-factor underestimate c' for the minimum cut. If we use this underestimate to determine a corresponding sampling probability $p' = 3(d+2)(\ln n)/\epsilon^2 c'$, then we know that p' is larger than the correct p, so that ϵ remains an upper bound on the likely deviation in cut values. At the same time, since p' exceeds the correct p by only a constant factor, the expected number of edges in our sample will be of the same order as the number of edges using the correct p. These two properties are sufficient to guarantee the correctness and time bounds of our algorithms.

One way to get this constant factor approximation is to use Matula's (1993) linear-time min-cut approximation algorithm to find a 3-approximation to the minimum cut. Another approach is to initially guess a known upper bound on c (say $c' = n$ in unweighted graphs) and then repeatedly halve the value of the guess until we confirm that our approximation algorithms have run correctly. Since our algorithm's running times are proportional to the sample size, and thus inversely proportional to our guess c', this repeated halving will increase the running time of our algorithms by only a constant factor.

Thus, we will assume for the rest of this paper that the correct p for a given ϵ is known to us, so that given ϵ we can construct a corresponding p-skeleton in linear time.

3 s-t Min-Cuts and Max-Flows

We now show how the skeleton approach can be applied to minimum cuts and maximum flows. 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 (Ford and Fulkerson 1962) 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. Tarjan (1983, Ahuja, Magnanti, and Orlin (1993)). Given a graph and an s-t flow of value f, a linear-time 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. We now show how random sampling can be used to speed up such augmenting path algorithms. We have the following immediate extension of Corollary 2.4:

Theorem 3.1. *Let G be any graph with minimum cut c and let $p = \Theta((\ln n)/\epsilon^2 c)$ as in Corollary 2.4. 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 3.2. *Assuming $\epsilon < 1/2$, the s-t min-cut in $G(p)$ corresponds to a $(1+4\epsilon)$-minimum s-t cut in G with high probability.*

Proof. Assuming that Theorem 3.1 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 Theorem 3.1) this minimum cut corresponds to a cut of value at most $(1+\epsilon)c/(1-\epsilon) < (1+4\epsilon)c$ when $\epsilon < 1/2$. □

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. We lead into `DAUG` with some more straightforward algorithms.

3.1 Approximate s-t Minimum Cuts

The most obvious application of Theorem 3.1 is to approximate minimum cuts. We can find an approximate s-t minimum cut by finding an s-t minimum cut in a skeleton.

Lemma 3.3. *In a graph with minimum cut c, a $(1+\epsilon)$-approximation to the s-t minimum cut of value v can be computed in $\tilde{O}(mv/\epsilon^3 c^2)$ time (MC).*

Proof. Given ϵ, determine the corresponding $p = \Theta((\log n)/\epsilon^2 c)$ from Theorem 3.1. Assume for now that $p \leq 1$. Construct a p-skeleton $G(p)$ in $O(m)$ time. Suppose we compute an s-t maximum flow in $G(p)$. By Theorem 3.1, $1/p$ 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, the skeleton has $O(pm)$ edges with high probability. Also, by Theorem 3.1, 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 $O((pm)(pv)) = O(mv \log^2 n/\epsilon^4 c^2)$ time. Our improved augmenting paths algorithm DAUG in Section 3.4 lets us shave a factor of $\Theta(\sqrt{pc/\log n}) = \Theta(1/\epsilon)$ from this running time, yielding the claimed bound.

If $p \geq 1$ because $c = O((\log n)/\epsilon^2)$, then $\epsilon^3 c^2 = \tilde{O}(\sqrt{c})$, so our theorem is proved if we give a running time of $\tilde{O}(mv/\sqrt{c})$. This is the time bound of algorithm DAUG in Section 3.4. □

3.2 Approximate Maximum Flows

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

Lemma 3.4. *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 (MC).*

Proof. Given p as determined by ϵ, randomly partition the graph's edges into $1/p$ groups, creating $1/p$ graphs (this takes $O(m \log(1/p))$ time). Each graph looks like (has the distribution of) a p-skeleton, and thus with high probability has a maximum flow of value at least $pv(1-\epsilon)$ 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 $p \geq 1$ then the argument still holds since this implies $\epsilon^2 c \leq \log n$. If we use our improved augmenting path algorithm DAUG, we improve the running time by an additional factor of $\Theta(1/\epsilon)$, yielding the claimed bound. □

3.3 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 their error probabilities 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.

Corollary 3.5. *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 (LV).*

Proof. Run both the approximate min-cut and approximate max-flow algorithms, obtaining a $(1-\epsilon/2)$-maximum flow of value v_0 and a $(1+\epsilon/2)$-minimum cut of value v_1. We know that $v_0 \leq v \leq v_1$, so to verify the correctness of the results all we need do is check that $(1+\epsilon/2)v_0 \geq (1-\epsilon/2)v_1$, 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). Since the first attempt succeeds with high probability, the expected running time is as claimed. □

3.4 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 3.6. *In a graph with minimum cut value c, a maximum flow of value v can be found in $O(mv\min(1, \sqrt{(\log n)/c}))$ time (LV).*

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 suggests a general approach for solving packing problems with an augmentation algorithm (including packing bases in a matroid (Karger 1998b)). 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 1, 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 independently 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 (or blocking flows) to increase f to a maximum flow.

Figure 1: 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. Note also that unlike our approximation algorithms, this algorithm requires no prior guess as to the value of c. 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 expect G_1 to have $m/2$ edges. Also, we can apply Theorem 3.1 to deduce that with high probability the s-t maximum flow in G_1 is $(v/2)(1-\tilde{O}(\sqrt{1/c}))$ 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 $\tilde{O}(v/\sqrt{c})$. By deleting isolated vertices as they arise, we can ensure that every problem instance has more edges than vertices. Thus each augmentation takes $O(m')$ time on an m'-edge graph. Intuitively, this suggests the following sort of 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 values of cuts in and sizes of the subproblems are random variables not guaranteed to equal their expectations. Actually proving the result requires some additional work.

We consider the tree of recursive calls made by our algorithm. Each *node* of this tree corresponds to an invocation of the recursive algorithm. We can then bound the total running time by summing the work performed at all the nodes in the recursion tree. We first show that it is never worse than the standard augmenting paths algorithm, and then show that it is better when c is large.

Lemma 3.7. *The depth of the computation tree is $O(\log m)$ (w.h.p.).*

Proof. The number of computation nodes at depth d is 2^d. Each edge of the graph ends up in exactly one of these nodes chosen uniformly and independently at random from among them all. Thus, the probability that two different edges both end up in the same node at depth $3\log m$ is (summing over pairs of edges) at most $\binom{m}{2}/m^3$, which is negligible. But if there is only one edge, the base case applies with no further recursion. □

Lemma 3.8. `DAUG` *runs in $O(m\log m + mv)$ time (LV).*

Proof. First we bound the non-augmenting-path work (i.e. the work of building and reassembling the subproblems) in Steps 1–3. Note that at each node in the computation tree, the amount of such work performed, not including recursive calls, is linear in the size (number of edges) of the node (since we delete isolated vertices as they arise, there are always fewer vertices than edges). At each level of the recursion tree, each edge of the original graph is located in exactly one node. Therefore, the total size of nodes at a given level is $O(m)$. Since there are $O(\log m)$ levels in the recursion, the total work is $O(m \log m)$.

Next we bound the work of the augmenting path computations. Note first that the algorithm performs one "useless" augmenting path computation at each node in order to discover that it has found a maximum flow for that node. Since the work of this augmentation is linear in the size of the node, it can be absorbed in the $O(m \log m)$ time bound of the previous paragraph.

It remains to bound the time spent on "successful" augmentations that increase the flow at their node by one. We claim that the number of successful augmentations, taken over the entire tree, is v. To see this, telescope the argument that the number of successful augmentations at a node in the computation tree is equal to the value of the maximum flow at that node minus the sum of the maximum flows at the two children of that node. Since each successful augmentation takes $O(m)$ time, the total time spent on successful augmentations is $O(mv)$. □

Lemma 3.9. *When $c \geq \log n$,* `DAUG` *runs in $O(m \log m + mv\sqrt{\frac{\log n}{c}})$ time (LV).*

Proof. We improve the previous lemma's bound on the work of the successful augmentations that add a unit of flow at a node. The number of such augmentations is equal to the difference between the maximum flow at the node and the sum of the children's maximum flows. Consider a node N at depth d. Each edge of the original graph ends up at N independently with probability $1/2^d$. Thus, the graph at N is a (2^{-d})-skeleton.

First consider nodes at depths exceeding $\log(c/\log n)$. Each of these nodes has $O(m(\log n)/c)$ edges w.h.p. By the same argument as the previous lemma, there are only v successful augmentations performed at these nodes, for a total work of $O(mv(\log n)/c)$, which is less than the claimed bound if $c \geq \log n$.

At depths less than $\log(c/\log n)$, the minimum expected cut at a node N is large enough to apply the sampling theorem. This proves that the maximum flow at N is $2^{-d}v(1 \pm O(\sqrt{\frac{2^d \log n}{c}}))$ w.h.p. Now consider the two children of node N. By the same argument, each has a maximum flow of value $2^{-(d+1)}v(1 \pm O(\sqrt{\frac{2^{d+1} \log n}{c}}))$ (w.h.p.). It follows that the total number of augmentations that must be performed at N is

$$\frac{v}{2^d}\left(1 \pm O\left(\sqrt{\frac{2^d \log n}{c}}\right)\right) - 2 \cdot \frac{v}{2^{d+1}}\left(1 \pm O\left(\sqrt{\frac{2^{d+1} \log n}{c}}\right)\right) = O\left(v\sqrt{\frac{\log n}{2^d c}}\right).$$

By the Chernoff bound, each node at depth d has $O(m/2^d)$ edges with high probability. Thus the total amount of augmentation work done at the node is $O(m/2^d)$ times the above bound. Summing over the 2^d nodes at depth d gives an overall bound for the work at level d of

$$O\left(mv\sqrt{\frac{\log n}{2^d c}}\right).$$

We now sum this bound over all depths d to get an overall bound of $O(mv\sqrt{\frac{\log n}{c}})$. □

Combining this result with the previous one gives a bound of $O(m \log m + mv \min(1, \sqrt{(\log n)/c}))$. This time bound is still not quite satisfactory, because the extra $O(m \log m)$ term means the algorithm is slower than standard augmenting paths when v is less than $\log m$. This is easy to fix. Before running `DAUG`, perform $O(\log m)$ augmenting path computations on the original graph, stopping if a maximum flow is found. This guarantees that when $v = O(\log m)$, the running time is $O(mv)$. This completes the proof of the section's main theorem.

4 Global Minimum Cuts

We now show how sampling can be used for global minimum cuts. We improve an algorithm of Gabow (1995) that finds minimum cuts in $O(mc \log(n^2/m))$ time. This section is devoted to proving the following theorem. Some additional ramifications are discussed at the end.

Theorem 4.1. *A graph's minimum cut c can be found in $\tilde{O}(m\sqrt{c})$ time (LV). It can be approximated to within $(1+\epsilon)$ in $\tilde{O}(m)$ time (LV).*

We therefore improve Gabow's algorithm's running time by a factor of roughly $\sqrt{c}$ in the exact case and give a roughly linear-time algorithm for the approximate case. We have recently developed a near-linear time exact algorithm (Karger 1996), but it is Monte Carlo. These are the fastest known Las Vegas algorithms.

Our proof of Theorem 4.1 is the same as the one presented previously for finding maximum flows. The change is that instead of using the standard augmenting paths technique to pack paths, we use a matroid augmentation technique developed by Gabow (1995) to pack arborescences—that is, directed spanning trees. We must revise the analysis slightly because the time for a single "augmenting path" computation is not linear.

Gabow's algorithm is designed for directed graphs and is based on earlier work of Edmonds (1965). In a directed graph, a minimum cut is a vertex partition (S, T) that minimizes the number of edges directed from S to T. Given a particular vertex s, a *minimum s-cut* is a partition of the vertices into nonempty sets S and T such that $s \in S$ and the number of directed edges crossing from S to T is minimized. Since the minimum cut in a graph is a minimum s-cut in either G or G with all edges reversed, finding a global minimum cut in a directed graph reduces to two iterations of finding a minimum s-cut. Gabow's algorithm does so by packing *s-arborescences.* An s-arborescence in G is a spanning tree of directed edges that induce indegree exactly one at every vertex other than s. In other words, it is a spanning tree with all edges directed away from s. Edmonds (1965) gave the following characterization of minimum cuts:

> The minimum s-cut of a graph is equal to the number of disjoint s-arborescences that can be packed in it.

It is obvious that every tree in the packing must use at least one edge of any s-cut; the other direction of the inequality is harder. This characterization corresponds closely to that for maximum flows. Just as the minimum s-t cut is equal to the maximum number of disjoint paths directed from s to t, the minimum s-cut is equal to the maximum number of disjoint spanning trees directed away from s. Each arborescence can be thought of as directing a unit of flow from s to all other vertices simultaneously. Intuitively, the bottleneck in this flow is the vertex to which s can send the least flow—namely, one on the opposite side of the minimum s-cut.

Gabow's min-cut algorithm uses a subroutine that he calls the *Round Robin Algorithm* (`Round-Robin`). This algorithm takes as input a graph G with an arborescence packing of value k. In $O(m \log(n^2/m))$ time it either returns an arborescence packing of value $(k+1)$ or proves that the minimum cut is k by returning a cut of value k. `Round-Robin` can therefore be seen as a cousin of the standard augmenting-path algorithm for maximum flows: instead of augmenting by a path, it augments by a spanning tree that sends an extra unit of flow to *every* vertex. Like many flow algorithms, Gabow's algorithm does not explicitly partition his current flow into arborescences ("paths"). Rather, it maintains an edge set (called a complete intersection) that can be so partitioned. Actually carrying out the partition seems to be somewhat harder than finding the edge set.

Gabow's algorithm for finding a minimum cut is to repeatedly call `Round-Robin` until it fails. The number of calls needed is just the value c of the minimum cut; thus the total running time of his algorithm is $O(cm\log(n^2/m))$. Gabow's algorithm can clearly be applied (with the same time bounds) to undirected graphs as well: simply replace each undirected edge with two directed edges: one in each direction.

We can improve this algorithm as we did the max-flow algorithm. Use `DAUG`, but replace the augmenting path steps with calls to `Round-Robin`.

Lemma 4.2. `DAUG` *finds a global minimum cut in* $O(m\min(c,\sqrt{c\log n})\log n)$ *time.*

Proof. Reuse the proof for the maximum flow analysis as if we were looking for a flow of value c. The only change is that a single application of `Round-Robin` on a graph with m' edges takes $O(m'\log(n^2/m')) = O(m'\log n)$ time. Since each augmentation anywhere in the analysis is $O(\log n)$ times slower than for flows, the overall time bound is $O(\log n)$ times greater. □

We can improve the last logarithmic factor with a more careful algorithm and analysis. Before running `DAUG`, approximate the minimum cut to within some constant factor (using Matula's (1993) algorithm or the skeleton approach). Then, at depth $\log(c/\log n)$ in the recursion, when the incoming graph has minimum cut $O(\log n)$, run Gabow's original algorithm instead of recursing. This immediately proves Theorem 4.1 for $c = O(\log n)$. We now prove the other case to finish the proof of the theorem.

Lemma 4.3. *For* $c \geq \log m$*, the modified* `DAUG` *algorithm runs in* $O(m\sqrt{c\log m}\log(n^2/m))$ *time.*

Proof. Since the computation stops recursing when the depth reaches $\log(c/\log n)$, the recursion tree has depth $\log(c/\log n)$. As with the flow analysis, the overhead in setting up the subproblems at all levels is then $O(m\log(c/\log n))$, which is negligible. Since the time per augmentation is no longer linear, we must change the analysis of work performed during augmentations.

Consider first the "unsuccessful" augmentations that identify maximum arborescence packings. The algorithm performs one at each node in the recursion tree. The total work over all 2^d nodes at each depth d is thus

$$\begin{aligned} O\left(\sum_{d=1}^{\log(c/\log n)} 2^d(m/2^d)\log(2^d n^2/m)\right) &= O\left(\sum md + \sum m\log(n^2/m)\right) \\ &= O(m\log^2(c/\log n) + m\log(c/\log n)\log(n^2/m)), \end{aligned}$$

which is less than the specified bound since $\log^2(c/\log n) = o(\sqrt{c\log n})$.

Now consider the "successful" `Round-Robin` calls that actually augment a packing. We analyze these calls as in the maximum flow case. Comparing the minimum cuts of a parent node and its children, we see that at depth d, each of the 2^d nodes has $O(m/2^d)$ edges and requires $O(\sqrt{c(\log n)/2^d})$ `Round-Robin` calls for total of $O(m\sqrt{c(\log n)/2^d}\log(2^d n^2/m))$ work at depth d. Summing over all depths gives a total work bound of $O(m\sqrt{c\log n}\log(n^2/m))$.

Finally, consider the work in the calls to Gabow' algorithm at the leaves of the recursion. At depth $d = \log(c/\log n)$, there will be 2^d such calls on graphs with minimum cut $O(\log n)$, each taking $O((m/2^d)(\log n)(\log(n^2c/m\log n)))$ time. Since by assumption $c > \log n$, this is dominated by the time bound for successful augmentations. □

Remark. An alternative to running a separate approximation algorithm for the minimum cut is to modify `DAUG` so that before it recurses, it makes $O(\log n)$ calls to `Round-Robin` and halts if it finds a maximum packing. This causes the recursion to terminate at the same point as before while increasing the work at each recursion-tree node by at most a constant factor.

The improved time for packing arborescences has other ramifications in Gabow's (1991) work. He gives other algorithms for which computing an arborescence packing is the computational bottleneck. He gives an algorithm for computing a compact *m-tree* representation of all minimum cuts, and shows how this representation can be converted to the older $O(n)$-space cactus representation (Dinitz, Karzanov, and Lomonosov 1976) in linear time. He also gives an algorithm for finding a minimum set of edges to add to augment the connectivity of a graph from c to $c+\delta$. In both of these algorithms, computing an arborescence packing forms the bottleneck in the running time.

Corollary 4.4. *The cactus and m-tree representations of all minimum cuts in an undirected graph can be constructed in $\tilde{O}(m\sqrt{c})$ time (LV).*

Corollary 4.5. *A minimum set of edges augmenting the connectivity of a graph from c to $c+\delta$ can be computed in $\tilde{O}(m + n(c^{3/2} + \delta c + \delta^2))$ time (LV).*

4.1 Approximating the Minimum Cut

Just as with maximum flows, we can combine a minimum cut algorithm with random sampling to develop Monte Carlo and Las Vegas algorithms for finding *approximate* minimum cuts. Previously, Matula (1993) gave a linear-time deterministic $(2+\epsilon)$-approximation algorithm; we use randomization to get better approximations with the same time bound.

Corollary 4.6. *A $(1+\epsilon)$-minimum cut can be found in $O\left(m + n\left(\frac{\log n}{\epsilon}\right)^3\right)$ time (MC).*

Proof. Given an m edge graph, build a p-skeleton using the p determined by ϵ, and use the previous min-cut algorithm to find a minimum cut in it. Assume $p < 1$. Then the running time is $O(m(\log^3 n)/(\epsilon^3 c))$. Now note that before we run the approximation algorithm, we can use Nagamochi and Ibaraki's sparse certificate algorithm (discussed in Section 1.4) to construct (in $O(m)$ time) an $O(nc)$-edge graph with the same approximately minimum cuts as our starting graph. This reduces the running time of the sampling algorithm to the stated bound.

If $p > 1$, meaning that $\epsilon^2 = \tilde{O}(1/c)$, then the claimed running time is $\tilde{O}(nc^{3/2})$, which is achieved by running DAUG on the nc-edge sparse certificate. □

Corollary 4.7. *A $(1+\epsilon)$-minimum cut and $(1-\epsilon)$-maximum arborescence packing can be found in $O(m(\log^2 n)/\epsilon)$ time (LV).*

Proof. Recall from above that an arborescence-packing of value k certifies that the minimum cut is at least k. Given ϵ and its corresponding p, divide the graph in $1/p$ pieces, find a maximum arborescence packing in each of the pieces independently, and union the packings. The analysis proceeds exactly as in the approximate max-flow algorithm of Section 3.2. As in Corollary 3.5, the combination of a cut of value $(1+\epsilon/2)c$ and a $(1-\epsilon/2)c$-packing brackets the minimum cut between these two bounds. □

5 Weighted Graphs

We now describe the changes that occur when we apply our cut and flow algorithms to weighted graphs. We model an edge of weight w as a collection of w unweighted edges. This creates problems in applying the undirected graph algorithms. For the approximation algorithms, the time to construct a skeleton becomes proportional to the total edge weight. For the divide and conquer algorithms, the time for augmentations becomes large for the same reason.

Improved methods for weighted graphs have recently been developed for both cuts (Benczúr and Karger 1996) and flows (Karger 1998a; Karger and Levine 1998).

5.1 Constructing Skeletons

The first problem we face is constructing a skeleton. The number of edges implicitly represented by edge weights can be too large to let us take time to sample each individually. To speed our skeleton construction, we use the following alternative approach.

Lemma 5.1. *Let G be any unweighted graph with minimum cut c and let $p = 3(d+2)(\ln n)/\epsilon^2 c$. Let H be constructed from G by choosing $\lceil pm \rceil$ edges from G at* *random. Then the probability that some cut of value v in G has value more than $(1+\epsilon)pv$ or less than $(1-\epsilon)pv$ in H is $O(n^{-d}\sqrt{pm})$.*

Proof. We could prove this corollary from first principles by reapplying the cut-counting theorem, but we take an easier approach. Let ERR denote the event that some cut diverges by more than ϵ from its expected value. We know that if we sample each edge with probability p, then $\Pr[ERR]$ is $O(1/n^d)$. Let S denote the number of edges actually chosen in such a sample. Note that S has the binomial distribution and that its so-called *central term* $\Pr[S = \lceil pm \rceil] = \Omega(1/\sqrt{pm})$ (cf. (Feller 1968)). We can evaluate ERR conditioning on the value of S:

$$\begin{aligned}
1/n^d &\geq \Pr[ERR] \\
&= \sum_k \Pr[S = k] \cdot \Pr[ERR \mid S = k] \\
&\geq \Pr[S = \lceil pm \rceil] \cdot \Pr[ERR \mid S = \lceil pm \rceil] \\
&= \Omega(\frac{1}{\sqrt{pm}}) \cdot \Pr[ERR \mid S = \lceil pm \rceil].
\end{aligned}$$

In other words, $\Pr[ERR \mid S = \lceil pm \rceil] = O(\sqrt{pm}/n^d)$. □

This corollary tells us that so long as the expected number pm of edges in the skeleton is polynomial, we can construct the skeleton by taking a fixed-size sample and get roughly the same result as in the original construction: all cut values will be within ϵ of their expectations with high probability. We can construct such a modified p-skeleton by making pm random selections from among the edges of the graph. In a weighted graph this corresponds to using biased selection: choose the edge with probability proportional to the weight of the edge. In a graph with total edge weight W, each such selection takes $O(\log W)$ time since we generate $\log W$ random bits in order to identify a particular edge. Thus, the total time is $O(pW \log W)$. In fact, this algorithm can be made strongly polynomial: we can arrange for each selection to take $O(\log m)$ amortized time, but the digression into the details would take us too far afield. A discussion can be found elsewhere (Knuth and Yao 1976; Karger and Stein 1996).

Lemma 5.2. *In a weighted graph with m edges of total weight W, a p-skeleton can be constructed in $O(pW \log m)$ time.*

The only other issue that needs to be addressed is the estimation of the correct sampling rates p for a given approximation bound ϵ. As with the unweighted case, we actually only need a constant factor estimate of the minimum cut. One way to get it is to generalize Matula's $(2+\epsilon)$-approximation algorithm to weighted graphs (see (Karger 1994b) for details). An alternative is to generalize the repeated doubling approach of Section 2.1. Unweighted graphs had minimum cuts bounded by n, so only $\log n$ repeated doubling trials were needed to get the estimate. For weighted graphs, we need a slightly more complex algorithm. We use the following scheme to estimate the minimum cut to within a factor of n^2, and then repeatedly double the estimate (halving the estimated sampling probability) until (within $O(\log n)$ attempts) the estimate is correct to within a factor of 2. Compute a maximum spanning tree of the weighted graph, and then let w be the weight of the minimum weight edge of this maximum spanning tree. Removing this edge partitions the maximum spanning tree into two sets of vertices such that no edge of G connecting them has weight greater than w (else it would be in the maximum spanning tree). Therefore, the minimum cut is at most $n^2 w$. On the other hand, the maximum spanning tree has only edges of weight at least w, so one such edge crosses every cut. Thus the minimum cut is at least w.

5.2 Cuts

Our cut approximation algorithms have roughly the same running time as in the unweighted case: the only change is that we use the $O(pW \log n)$-time weighted-graph skeleton construction.

Corollary 5.3. *In a weighted graph, a $(1+\epsilon)$-minimum cut can be found in $O(m + n((\log n)/\epsilon)^3)$ time (MC).*

Proof. We have already discussed finding a rough approximation to c using, e.g., Matula's algorithm. Construct a sparse $3c$-connectivity certificate of total weight $O(nc)$ and proceed as in the unweighted graph algorithm. Regardless of the original graph weights, the skeleton will have $O(n(\log n)/\epsilon^2)$ edges and minimum cut $O((\log n)/\epsilon^2)$. □

Corollary 5.4. *In a weighted graph, a $(1+\epsilon)$-minimum s-t cut can be found in $\tilde{O}(m + n(v/c)^2\epsilon^{-3})$ time (MC).*

Proof. Suppose first that we knew v. Use Nagamochi and Ibaraki's (1992a) sparse certificate algorithm to construct a sparse $3v$-connectivity certificate of total weight $O(nv)$. Assuming $\epsilon < 1$, approximate cuts in the certificate are the same as those in the original graph. Construct a p-skeleton of the certificate using weighted selection from the certificate in $O(pnv \log m)$ time. Now proceed as in the unweighted graph case.

To make up for our ignorance of v, begin by estimating v to within a factor of n^2 as follows. Find (using an obvious variant of Dijkstra's shortest path algorithm) the path from s to t whose smallest edge weight w is maximized. It follows that every

s-t cut has weight at least w, since some edge on the found path is cut. However, if we remove all edges of weight w or less (a total of n^2w weight) then we disconnect s and t since every s-t path contains an edge of weight at most w. Therefore, v is between w and n^2w. Start by guessing $v = w$, and double it $O(\log n)$ times until the guess exceeds v, at which point the approach of the previous paragraph will yield the desired cut. □

The $\tilde{O}(mv/\epsilon^3c^2)$ bound of the unweighted case no longer follows, since it need no longer be the case that skeleton has only pm edges.

5.3 Flows

We can also adapt the max-flow algorithms. If we directly simulated the unweighted graph algorithm DAUG, we would partition the edges into two groups by generating a binomial distribution for each weighted edge in order to determine how much of its weight went to each of the two subgraphs. To avoid having to generate such complicated distributions, we return to Theorem 2.1 and use the following approach. If w is even, assign weight $w/2$ to each group. If w is odd, then assign weight $\lfloor w/2 \rfloor$ to each group, and flip a coin to decide which group gets the remaining single unit of weight. Since the minimum expected cut ($\hat{c}$ of Theorem 2.1) that results in each half is still $c/2$, we can deduce as in the unweighted case that little augmentation need be done after the recursive calls.

We have described the change in implementation, and correctness is clear, but we have to change the time bound analysis. It is no longer true that each new graph has half the edges of the old. Indeed, if all edge weights are large, then each new graph will have just as many edges as the old. We therefore add a new parameter and analyze the algorithm in terms of the number of edges m, the minimum cut c, the desired flow value v, and the *total weight* W of edges in the graph. Note the two subgraphs that we recurse on have total weight roughly $W/2$. In order to contrast with bit-scaling techniques, we also use the *average edge weight* $U = W/m$ which is no more than the maximum edge weight. The unweighted analysis suggests a time bound for minimum cuts of $\tilde{O}(W\sqrt{c}) = \tilde{O}(mU\sqrt{c})$, but we can show a better one:

Lemma 5.5. *A global minimum cut of value c can be found in $\tilde{O}(m\sqrt{cU})$ time (LV).*

Proof. We divide the recursion tree into two parts. At depths $d \leq \log(W/m)$, we bound the number of edges in a node by m. As in the unweighted analysis, we know each node at depth d has to perform $\tilde{O}(\sqrt{c/2^d})$ augmentations, each taking $\tilde{O}(m)$ time, so the total work at depth d is $\tilde{O}(2^d m\sqrt{c/2^d}) = \tilde{O}(m\sqrt{2^dc})$. Summing over $d \leq \log(W/m)$ gives a total work bound of $\tilde{O}(m\sqrt{Wc/m}) = \tilde{O}(m\sqrt{cU})$.

At depth $\log(W/m)$, we have W/m computation nodes, each with minimum cut $\tilde{O}(mc/W)$ (by the sampling theorem) and at most m edges. Our unweighted graph analysis shows that the time taken by each such node together with its children is $\tilde{O}(m\sqrt{mc/W})$. Thus the total work below depth $\log(W/m)$ is $\tilde{O}((W/m)(m\sqrt{mc/W})) = \tilde{O}(m\sqrt{cU})$. □

A similar result can be derived if we use the same algorithm to find flows, replacing Gabow's Round Robin Algorithm with standard augmenting paths.

Corollary 5.6. *A maximum flow of value v can be found in $\tilde{O}(mv\sqrt{U/c})$ time (LV).*

More recently (Karger 1998a) we introduced a *smoothing* technique that lets us avoid splitting large edges in two for the two recursive calls. Instead, after some preliminary splitting, we show that it is possible to assign the full weight of an edge randomly to one subproblem or the other, and still get the same accurate approximation of cut values. This lets us extend our unweighted-graph time bounds to weighted graphs as well.

6 Other Cut Problems

In this section, we discuss several other cut problems and algorithms and show how our sampling techniques can be applied to them.

6.1 Parallel Flow Algorithms

In the s-t min-cut problem the need for the final "cleanup" augmentations interferes with the development of efficient $\mathcal{RNC}$ `DAUG`-type algorithms for the problem, because there are no good parallel reachability algorithms for finding augmenting paths in directed graphs. However, we can still take advantage of the divide and conquer technique in a partially parallel algorithm for the problem. Khuller and Schieber (Khuller and Schieber 1991) give an algorithm for finding disjoint s-t paths in undirected graphs. It uses a subroutine that augments a set of k disjoint s-t paths to $k+1$ if possible, using $\tilde{O}(k)$ time and kn processors. This lets them find a flow of value v in in $\tilde{O}(v^2)$ time using vn processors. We can speed up this algorithm by applying the `DAUG` technique we used for maximum flows. Finding the final augmentations after merging the results of the recursive calls is the dominant step in the computation. It requires $\tilde{O}(v/c)$ iterations of their augmentation algorithm, each taking $\tilde{O}(v)$ time, for a total of $\tilde{O}(v^2/\sqrt{c})$ time using vn processors. Thus we decrease the running time of their algorithm by an $\tilde{O}(\sqrt{c})$ factor, without changing the processor cost.

6.2 Separators and Sparsest Cuts

The *edge separator* problem is to find a cut with a minimum number of edges that partitions a graph into two roughly equal-sized vertex sets. The *sparsest cut* problem is to find a cut (A, B) of value v minimizing the value of the quotient $v/(\|A\|\|B\|)$. These problems are $\mathcal{NP}$-complete and the best known approximation ratio is $O(\log n)$ (for separators, one has to accept a less balanced solution to achieve this bound). One algorithm that achieves this approximation for sparsest cuts is due to Leighton and Rao (Leighton and Rao 1988).

Klein, Plotkin, Stein, and Tardos (Klein, Plotkin, Stein, and Tardos 1994) give a fast concurrent flow algorithm which they use to improve the running time of Leighton and Rao's algorithm. Their algorithm runs in $O(m^2 \log m)$ time, and finds a cut with quotient within an $O(\log n)$ factor of the optimum. Consider a skeleton of the graph which approximates cuts to within a $(1 \pm \epsilon)$ factor. Since the denominator of a cut's quotient is unchanged in the skeleton, the quotients in the skeleton also approximate their original values to within a $(1 \pm \epsilon)$ factor. It follows that we can take $p = O(\log n/c)$ and introduce a negligible additional error in the approximation. By the same argument, it suffices to look for balanced cuts in a skeleton rather than the original graph.

Theorem 6.1. *An $O(\log n)$-approximation to the sparsest cut can be computed in $\tilde{O}((m/c)^2)$ time (MC).*

Benczur and Karger (Benczúr and Karger 1996) have improved this time bound to $\tilde{O}(n^2)$.

6.3 Orienting a Graph

Given an undirected graph, the *graph orientation problem* is to find an assignment of directions to the edges such that the resulting directed graph has the largest possible (directed) connectivity. Gabow (Gabow 1993) cites a theorem of Nash-Williams (Nash-Williams 1969) showing that a solution of (directed) connectivity k exists if and only if the input graph is $2k$-connected, and also gives a submodular-flow based algorithm for finding the orientation in $O(kn^2(\sqrt{kn} + k^2 \log(n/k)))$ time. We have the following result:

Lemma 6.2. *A $(k - O(\sqrt{k \log n}))$-connected orientation of a $2k$-connected graph can be found in linear time.*

Proof. Orient each edge randomly with probability $1/2$ in each direction. A minor adaptation of Theorem 2.1 shows that with high probability, for each cut, there will be at least $k - O(\sqrt{k \log n})$ edges oriented in each direction. In other words, every directed cut will have a value exceeding the claimed one. □

Using this randomly oriented graph as a starting point in Gabow's algorithm also allows us to speed up that algorithm by a factor of $\tilde{O}(\sqrt{k})$.

6.4 Integral Multicommodity Flows

Suppose we are given an unweighted graph G and a multicommodity flow problem with k source-sink pairs (s_i, t_i) and demands d_i. Let c_i be the value of the s_i-t_i minimum cut and suppose that $\sum d_i/c_i \leq 1$. Then it is obvious that there is a fractional solution to the problem: divide the graph into k new graphs G_i, giving a d_i/c_i fraction of the capacity of each edge to graph G_i. Then the s_i-t_i minimum cut of G_i has value exceeding d_i, so commodity i can be routed in graph G_i. There has been some interest in the question of when an *integral* multicommodity flow can be found (the problem is discussed in (Ford and Fulkerson 1962); more recent discussions include (Grötschel, Lovász, and Schrijver 1988, Section 8.6) and (Frank 1990)). Our sampling theorem gives new results on the existence of integral flows and fast algorithms for finding them. Rather than assigning a fraction of each edge to each graph, assign each edge to a graph G_i with probability proportional to d_i/c_i. We now argue as for the flow algorithms that, given the right conditions on c, each graph G_i will be able to integrally satisfy the demands for commodity i. Thus k max-flow computations will suffice to route all the commodities. In fact, in an unweighted graph, if m_i is the number of edges in G_i, we know that $\sum m_i = m$, so that the max-flow computations will take $O(\sum m_i n) = O(mn)$ time. Various results follow; we give one as an example:

Lemma 6.3. *Suppose that each $d_i \geq \log n$, and that $\sum d_i \leq c/2$ (where c is the minimum cut). Then an integral multicommodity flow satisfying the demands exists and can be found in $O(mn)$ time.*

Proof. Assign each edge to group i with probability proportional to d_i/c. Since $\sum d_i/c \leq 1/2$, this means the probability an edge goes to i is at least $2d_i/c$. Thus the minimum expected cut in G_i is at least $2d_i$, so the minimum cut exceeds d_i with high probability and that graph can satisfy the i^{th} demand. □

7 Network Design

We now turn to the *network design problem.* Here, rather than sampling as a preprocessing step to reduce the problem size, we sample as a postprocessing step to round a fractional solution to an integral one.

7.1 Problem Definition

The most general form of the network design problem is as a covering integer program with exponentially many constraints. We are given a set of vertices, and for each pair of vertices i and j, a *cost* c_{ij} of establishing a unit capacity link between i and j. For each cut C in the graph, we are given a *demand* d_C denoting the minimum number of edges that must cross that cut in the output graph. Since there are exponentially many cuts (in the number of vertices n), the demands must be specified implicitly if the problem description is to be of size polynomial in n. Our goal is to build a graph of minimum cost that obeys all of the cut demands, *i.e.* to solve the following integer program:

$$\begin{aligned} \text{minimize } \sum & c_{ij}x_{ij} \\ \sum_{(i,j) \text{ crossing } C} x_{ij} \;\geq\; & d_C \quad (\forall \text{ cuts } C) \\ x_{ij} \;\geq\; & 0 \end{aligned}$$

There are two variants of this problem: in the *single edge use* version, each x_{ij} must be 0 or 1. In the *repeated edge use* version, the x_{ij} can be arbitrary nonnegative integers.

There are several specializations of the network design problem (further details can be found in the paper by Agrawal, Klein, and Ravi (1995)):

The generalized Steiner problem specifies a connectivity demand d_{ij} for each pair of vertices i and j, and the demand across a cut C is just the maximum of d_{ij} over all pairs (i, j) separated by C. An early formulation is due to Krarup (see Winter (1987)).

The survivable network problem has $d_{ij} = \min(d_i, d_j)$ for certain "connectivity types" i and j. It was studied by Goemans and Bertsimas (1993).

The minimum k-connected subgraph problem is to find a smallest (fewest edges) k-connected subgraph of an input graph G. This is a network design problem in which all demands are k and all edges have cost 1 (present in G) or ∞ (not present).

Even the minimum k-connected subgraph problem is $\mathcal{NP}$-complete, even for $k = 2$ (Eswaran and Tarjan 1976).

7.2 Past Work

Khuller and Vishkin (1994) gave a 2-approximation algorithm for the minimum cost k-connected graph problem and a 3/2-approximation for the minimum (unit cost) k-connected subgraph problem. Khuller and Raghavachari (1995) gave a 1.85-approximation for the minimum k-connected subgraph problem for any k.

Agrawal, Klein, and Ravi (1995) studied the repeated-edge-use generalized Steiner problem (with costs) and gave an $O(\log f_{\max})$ approximation algorithm, where $f_{\max}$ is the maximum demand across a cut, namely $\max d_{ij}$.

Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson (1994), extending work of Williamson, Goemans, Mihail, and Vazirani (1993)) have recently given powerful algorithms for a large class of network design problems, namely those defined by so-called *proper* demand functions (this category includes all generalized Steiner problems). Their approximation algorithm, which we shall refer to as the *Forest Algorithm*, finds a graph satisfying the demands of cost at most $O(\log f_{\max})$ times the optimum. It applies to both single and repeated edge-use problems. It can also be used to *augment* a given graph, adding edges so as to meet some proper demand function; the approximation ratio becomes the logarithm of the maximum *deficit*, *i.e.* difference between the demand across a cut and its starting capacity. The authors also note that a *fractional* solution, in which each edge is to be assigned a real-valued weight such that the resulting weighted graph satisfies the demands with a minimum total (weighted) cost, can be found in polynomial time by using the ellipsoid algorithm even though the number of constraints is exponential (Gabow, Goemans, and Williamson 1993). For example, given a generalized Steiner problem, a separation oracle can be implemented by computing all-pairs i-j minimum cuts in a candidate solution to see if some connectivity demand d_{ij} is not satisfied. If it is not, an i-j minimum cut gives a violated constraint.

7.3 Present Work

We use the fractional solution produced by the ellipsoid algorithm as the starting point in a *randomized rounding* based solution to network design problems. Randomized rounding (Raghavan and Thompson 1987) is a general technique developed to solve integral packing and covering problems problems.

Using randomized rounding, we give approximation algorithms whose bounds depend on $f_{\min}$, the minimum connectivity requirement between any pair of vertices. We begin by considering the version in which edges can be used repeatedly. If $f_{\min} \leq \log n$, randomized rounding leads to an approximation bound of $O((\log n)/f_{\min})$. If $f_{\min} \geq \log n$, our approximation bound is $1 + O(\sqrt{(\log n)/f_{\min}})$. This bound contrasts with a previous best bound of $O(\log f_{\max})$ (Agrawal, Klein, and Ravi 1995), providing significant improvements when the minimum connectivity demand is large.

We also give results for the single-edge-use case. For the k-connected subgraph problem, we give an approximation algorithm with performance ratio $1 +$

$O(\sqrt{(\log n)/k} + (\log n)/k)$. For any $k \gg \log n$, this improves on the previous best known approximation factor of 1.85 (Khuller and Raghavachari 1995). For more general problems, we give an approximation algorithm with ratio $\log(f_{\max}(\log n)/f_{\min})$, compared to the previous $O(\log f_{\max})$ bound (Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson 1994).

7.4 Randomized Rounding

The network design problem is a variant of the *set cover problem.* In this problem, we are given a collection of sets drawn from a universe, with each element of the universe possibly assigned a cost. We are required to find a minimum (number or total cost) collection of elements that intersects every set. The Steiner tree problem is an instance of set cover involving exponentially many sets. The universe is the set of edges, and each cut that separates two terminals corresponds to a set (the edges of the cut) that must be covered. An extension of this problem corresponding more closely to general network design is the *set multicover problem,* in which a demand d_S is specified for each set S and the covering set is required to contain d_S elements of S. The network design problem is an instance of set multicover in which the universe is the set of edges, and each cut induces a set consisting of the edges crossing it.

The set cover problem is easily formulated as an integer linear program, and its linear programming dual is what is known as a packing problem: find a maximum collection of sets that do not intersect. Raghavan and Thompson (1987) developed a technique called *randomized rounding* that can be used to solve such packing problems. The method is to solve the linear programming relaxation of the packing problem and then use the fractional values as probabilities that yield an integer solution by randomly setting variables to 0 or 1.

In the Raghavan-Thompson rounding analysis, the error introduced by rounding increases as the logarithm of the number of constraints. Thus, their approach typically works well only for covering problems with polynomially many constraints, while the network design problem has exponentially many. However, using Theorem 2.1, we prove that the special structure of graphs allows us to surmount this problem. This gives a simple approach to solving the multiple-edge-use versions of network design problems. A more complicated approach described in Section 8.2 gives us some weaker results for the single-edge-use version of the problem. We now describe the randomized rounding technique.

Consider a fractional solution to a network design problem (which has been found, for example, with the ellipsoid algorithm (Gabow, Goemans, and Williamson 1993)). Without loss of generality, we can assume every edge has fractional weight at most 1, since we can replace an edge of weight w by $\lfloor w \rfloor$ parallel edges of weight 1 and a single edge of weight $w - \lfloor w \rfloor$ without changing the solution value. Therefore, the weights on the edges can be thought of as sampling probabilities.

Suppose that we build a random graph by sampling each edge with the given probability. As a weighted graph, our fractional solution has minimum cut $f_{\min}$ and each cut C has weight at least equal to the demand d_C across it. Therefore, by Theorem 2.1, each cut C in the random graph has value at least $d_C(1 - \sqrt{(12 \ln n)/f_{\min}})$ with probability $1 - 1/n^2$. Now consider the cost of the random graph. Its expected value is just the cost b of the fractional solution, which is clearly a lower bound on the cost of the optimum integral solution. Therefore, by the Markov inequality, the probability that the random graph cost exceeds $(1 + 1/n)b$ is at most $1 - 1/n$. Therefore, if we perform the rounding experiment $O(n \log n)$ times, we have a high probability of getting one graph that satisfies the demands to within $(1 - \sqrt{(12 \ln n)/f_{\min}})$ at cost $(1 + 1/n)b$. To get our results, we need only explain how to deal with the slight under-satisfaction of the demands.

7.5 Repeated Edge Use

We first consider the repeated edge-use version of the network design problem. To handle the under-satisfaction of demands, we simply increase the weight of each edge slightly before we perform the randomized rounding.

Theorem 7.1. *The network design problem for proper demand functions with repeated edge use can be solved in polynomial time to within $1 + O(\sqrt{(\log n)/f_{\min}} + (\log n)/f_{\min})$ times optimum (LV).*

Proof. Assume first that $f_{\min} > 12 \ln n$. Before rounding the fractional solution, multiply each edge weight by $(1 + O(\sqrt{(\log n)/f_{\min}}))$. This increases the overall cost by the same factor. Now when we round, we get a graph with cut values $1 - \sqrt{(12 \ln n)/f_{\min}}$ times the *new* values (w.h.p.). Thus by suitable choice of constants we can ensure that the rounded value exceed the original fractional values w.h.p.

Now consider the case $f_{\min} < 12 \ln n$. The previous argument does not apply because $(1 - \sqrt{(12 \ln n)/f_{\min}}) < 0$ and we thus get no approximation guarantee from Theorem 2.1. However, if we multiply each edge weight by $O((\log n)/f_{\min})$, we get a graph with minimum cut $\Omega(\log n)$. If we round this graph, each cut gets value at least half its expected value, which is in turn $\Omega((\log n)/f_{\min})) \geq 1$ times its original value. $\square$

Remark. Note how the use of repeated edges is needed. We can constrain the fractional solution to assign weight at most 1 to each edge in an attempt to solve the single-edge-use version of the problem, but scaling up the fractional values in the solution could yield some fractional values greater than 1 that could round to an illegal value of 2. However, when $f_{\min} \geq \log n$, we will use every edge at most twice.

8 Single Edge-Use Network Design

The simple scaling up and rounding procedure that we applied for multiple-edge-use problems breaks down when we are restricted to use each edge at most once. We cannot freely scale up the weights of edges because some of them may take on values greater than one. Instead, we round the graph based on the original fractional weights and then "repair" the resulting graph. To characterize the necessary repairs, we make the following definition:

Definition 8.1. Given a network design problem and a candidate solution H, the *deficit* of a cut C in H is the difference between the demand across C and the value of C in H. The *deficit* of graph H is the maximum deficit of a cut in H.

8.1 Minimum k-connected Subgraph

A particularly easy case to handle is the minimum k-connected subgraph problem, where the best previous approximation value was 1.85 (Khuller and Raghavachari 1995).

Theorem 8.2. *For $k > \log n$, a $(1 + O(\sqrt{(\log n)/k}))$-approximation to the minimum k-connected subgraph can be found in polynomial time (LV).*

Proof. We exploit tight bounds on the optimum solution value. Consider any k-connected graph. It must have minimum degree k. and thus at least $kn/2$ edges. On the other hand, as discussed in Section 1.4, any sparse k-certificate of G will be k-connected if G is and will contain at most kn edges. Thus, the optimum solution has between $kn/2$ and kn edges (so a 2-approximation is trivial).

To get a better approximation, take the input graph G and find a fractional solution F using the ellipsoid algorithm (Gabow, Goemans, and Williamson 1993). By construction, F has (weighted) minimum cut k. Suppose the solution has total weight W. As discussed above, we know $kn/2 \leq W \leq kn$. Clearly W is a lower bound on the number of edges in the integral solution. Use randomized rounding to define a subgraph H. By the Chernoff bound, the number of edges in H is $W + O(\sqrt{W \log n})$ with high probability. Since F has minimum cut k, Theorem 2.1 says that H has minimum cut $k - O(\sqrt{k \log n})$ with high probability. That is, the deficit of H is $O(\sqrt{k \log n})$.

We now show how to remove this deficit. Consider the following procedure for reducing the deficit of H by one. Find a spanning forest of $G - H$, and add its edges to H. To see that this reduces the deficit of H, consider any cut of H that is in deficit. By definition less than k edges cross it. However, we know that at least k edges cross the corresponding cut in G (since by assumption G is k connected). It follows that one spanning-forest edge crosses this cut, and thus its deficit is decreased by one.

It follows that if we perform the deficit-reduction step $O(\sqrt{k \log n})$ times, then H will at the end have no deficit, and will therefore be k-connected. Now note that each deficit-reduction step adds one forest with at most n edges to H, so the total number of additions is $O(n\sqrt{k \log n})$. Since the optimum number of edges exceeds $W \geq kn/2$, we have that $n\sqrt{k \log n} = O(W\sqrt{(\log n)/k})$ and $\sqrt{W \log n} = O(W\sqrt{(\log n)/kn})$. Thus the total number of edges in our solution is $W + O(\sqrt{W \log n}) + O(n\sqrt{k \log n})$, which is $O(W(1 + \sqrt{(\log n)/k}))$. $\square$

Remark. It is not in fact necessary to perform the repeated deficit reduction steps. A more efficient approach is to use Nagamochi and Ibaraki's sparse certificate algorithm (discussed in Section 1.4). After deleting all the edges in H, build a sparse $O(\sqrt{k \log n})$-connectivity certificate C in the remaining graph. A modification of the above argument shows that $C \cup H$ is k-connected while C has $O(n\sqrt{k \log n})$ edges.

Corollary 8.3. *There is a $(1 + O(f_{\max}\sqrt{\log n}/f_{\min}^{3/2}))$-approximation algorithm for finding a smallest subgraph satisfying given connectivity demands.*

Proof. The minimum solution has at least $nf_{\min}/2$ edges. After rounding the fractional solution the maximum deficit is $O(f_{\max}\sqrt{(\log n)/f_{\min}})$ and can therefore be repaired with n times that many edges. $\square$

8.2 General Single-Edge Problems

We now consider more general single edge-use problems in which the demands can be arbitrary and the edges have arbitrary costs. As before, we solve the problem by first rounding a fractional solution and then repairing the deficits that arise. We can no longer use the deficit reduction procedure of the k-connected subgraph case, because there is no immediate bound relating the cost of a single forest to the cost of the entire solution. Instead, we use the Forest Algorithm of Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson (1994).

The Forest Algorithm can be used to solve *augmentation problems* that generalize network design problems. Namely, it attempts to find the minimum cost way to augment a graph H so as to satisfy a set of demands across cuts. If the maximum deficit in the augmentation problem is d, then the Forest algorithm finds a solution with cost $O(\log d)$ times the optimum.

8.2.1 Oversampling

Since the approximation factor of the Forest Algorithm worsens with the deficit, we first show how to modify the rounding step so as to keep the deficit small. We begin with a variant of the Chernoff bound that we can use when we are not allowed to scale weights above 1.

Definition 8.4. Consider a random sum $S = \sum_{i=1}^{n} X_i$ in which $X_i = 1$ with probability p_i and 0 otherwise. Define the *oversampling of S by α* as $S(\alpha) = \sum_{i=1}^{n} Y_i$, where $Y_i = 1$ with probability $\min(1, \alpha p_i)$ and 0 otherwise.

Note that $S(1) = S$.

Lemma 8.5. *Let $E[S] = \mu$. Then* $\Pr[S(1+\delta) < (1-\epsilon)\mu] < e^{-\epsilon\delta\mu/2}$.

Proof. Suppose $S = \sum X_i$. Write $S = S_1 + S_2$, where S_1 is the sum of X_i with $p_i \geq 1/(1+\delta)$ and S_2 is the sum of the remaining X_i. Let $\mu_1 = E[S_1]$ and $\mu_2 = E[S_2]$. Then $\mu = \mu_1 + \mu_2$, and $S(1+\delta) = S_1(1+\delta) + S_2(1+\delta)$.

Since the variables in S_1 have $p_i \geq 1/(1+\delta)$, $S_1(1+\delta)$ is not random: it is simply the number of variables in S_1, since each is 1 with probability one. In particular, $S_1(1+\delta)$ is certainly at least μ_1. It follows that $S(1+\delta) < (1-\epsilon)\mu$ only if $S_2 < (1-\epsilon)\mu - \mu_1 = \mu_2 - \epsilon\mu$.

The variables in S_2 have $p_i < 1/(1+\delta)$ so that the corresponding oversamplings have probabilities $(1+\delta)p_i$. It follows that $E[S_2(1+\delta)] = (1+\delta)\mu_2$. By the standard Chernoff bound, the probability that $S_2 < \mu_2 - \epsilon\mu$ is at most

$$\exp\left(-\frac{((1+\delta)\mu_2 - (u_2 - \epsilon\mu))^2}{2(1+\delta)\mu_2}\right) = \exp\left(-\frac{(\delta\mu_2 + \epsilon\mu)^2}{2(1+\delta)\mu_2}\right)$$

Our weakest bound arises when the above quantity is maximized with respect to μ_2. It is straightforward to show that the quantity is a concave function of μ_2 with its global maximum at $\mu_2 = \epsilon\mu/\delta$. However, μ_2 is constrained to be at least $\epsilon\mu$ (since otherwise $\mu_1 \geq (1-\epsilon)\mu$, immediately giving $S(1+\delta) \geq \mu_1$). We thus have two cases to consider. If $\delta < 1$, then $\epsilon\mu/\delta$ is a valid value for μ_2, and the corresponding bound is $\exp(2\epsilon\delta\mu/(1+\delta))$. If $\delta > 1$, then the bound is maximized at the smallest possible μ_2, namely $\mu_2 = \epsilon\mu$, in which case the bound is $\epsilon\mu(1+\delta)/2$. Over the given ranges of δ, each of these bounds is less than the bound given in the theorem. $\square$

Remark. The lemma easily extends to the case where the X_i take on arbitrary values between 0 and w. In this case, $e^{-\epsilon\delta\mu}$ bounds the probability that the deviation exceeds $\epsilon w\mu$ rather than $\epsilon\mu$.

8.2.2 Application

A combination of the above oversampling lemma with the proof of Theorem 2.1 yields the following:

Corollary 8.6. *Given a fractional solution f to a network design problem, if each edge weight w_e is increased to $\min(1, (1+\delta)w_e)$ and randomized rounding is performed, then with high probability no cut in the rounded graph will have value less than $(1-\epsilon)$ times its value in the original weighted graph, where $\epsilon = O(\log n/(\delta f_{\min}))$.*

We now combine Corollary 8.6 with the Forest Algorithm. Suppose we have fractionally solved a network design problem. Set $\delta = 2$ and apply Corollary 8.6, so that at cost twice the optimum we get a graph in which the maximum deficit is $O(f_{\max}(\log n)/f_{\min})$. Then use the Forest Algorithm (Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson 1994) to augment it to optimum. This yields the following:

Lemma 8.7. *There is an $O(\log \frac{f_{\max} \log n}{f_{\min}}))$ approximation algorithm for the single edge-use network design problem.*

This compares favorably with the Forest Algorithm's $O(\log f_{\max})$ bound whenever $f_{\min} > \log n$.

8.3 Fixed Charge Networks

Our algorithms also apply to the *fixed charge* problem in which each edge has a capacity of which all or none must be purchased. In this problem, the best currently known approximation ratio is a factor of $f_{\max}$ (Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson 1994). The introduction of large capacities increases the variances in our random sampling theorems. In particular, if we let U denote the maximum edge capacity, we have the following result based on a modification of Theorem 7.1:

Corollary 8.8. *There is a $(1+O((U \log n)/f_{\min} + \sqrt{(U \log n)/f_{\min}}))$-approximation algorithm for the fixed-charge network design problem with repeated edges.*

Proof. The Chernoff bound that we use for the randomized rounding proof applies if all random variables have maximum value at most 1. Take the fixed charge problem, and divide each demand and edge capacity by U. Now the original theorems apply, but new minimum demand is $f_{\min}/U$. □

Note that we can upper bound U by $f_{\max}$, since any edge with capacity exceeding $f_{\max}$ can have its capacity reduced to $f_{\max}$ without affecting the optimum solution.

Corollary 8.9. *There is a* $(1+O(\frac{f_{\max}\log n}{f_{\min}}))$*-approximation algorithm for the fixed charge network design problem with repeated edges.*

We also extend Theorem 2.1 as follows:

Corollary 8.10. *Given a fractional solution to* f*, if each edge weight* w_e *is increased to* $min(1,(1+\delta)w_e)$ *and randomized rounding is performed, then with high probability no cut in the rounded graph will have value less than its value in the original fractionally weighted graph, where* $\epsilon = O(U\log n/(\delta f_{\min}))$*.*

Corollary 8.11. *There is an* $O(\sqrt{\frac{Uf_{\max}\log n}{f_{\min}}})$*-approximation algorithm for the fixed-charge single-use network design problem.*

Proof. Apply oversampling with $\delta = \sqrt{(Uf_{\max}\log n)/f_{\min}}$. □

Corollary 8.12. *There is an* $O(f_{\max}\sqrt{\frac{\log n}{f_{\min}}})$*-approximation algorithms for the fixed charge single-use network design problem when* $f_{\min} \geq \log n$*.*

Corollary 8.13. *There is an* $O(\sqrt{k\log n})$*-approximation algorithm for the fixed-charge* k*-connected subgraph problem.*

9 Conclusion

This work has demonstrated the effectiveness of a sampling for solving problems involving cuts. We have shown how random sampling tends to "preserve" all cut information in a graph. This suggests that we might want to try to reformulate other problems in terms of cuts so that the random sampling methods can be applied to them.

One result of this approach has been to reduce large max-flow and min-cut problems on undirected graphs to small max-flow and min-cut problems on directed graphs. Our techniques are in a sense "meta-algorithms" in that improved cut or flow algorithms that are subsequently developed may well be accelerated by application of our technique. In particular, our exact algorithms' running times are dominated by the time needed to perform "cleaning up" augmenting path computations; any improvement in the time to compute a sequence of augmenting paths would translate immediately into an improvement in our algorithm's running time. We have achieved this objective for simple graphs (unweighted graphs without parallel edges) (Karger and Levine 1998). One way to get such an improvement on general graphs might be to generalize our sampling theorems to the case of directed graphs. Unfortunately, directed graphs do not have good cut-counting bounds like the ones we used here.

Our approach to cuts and flows, combining sampling with an augmentation algorithm, is a natural one for any problem of packing disjoint feasible sets over some universe. All that is needed for the approach to work is

1. a sampling theorem, showing that a sample from half the universe has a packing of about half the size, and

2. an augmentation algorithm that increases the size of the packing by one.

One additional domain where we have shown these two features apply is that of *matroids.* In particular, we show that the problem of packing matroid bases is susceptible to this approach (Karger 1998b).

Our work studies sampling from arbitrary graphs. A huge amount of work has gone into the study of sampling from *complete* graphs, yielding what are generally known as random graphs. Indeed, one of the very first results on random graphs was that their minimum cut was close its expected value (Erdös and Rényi 1961). Our results can be seen as generalizing those results, but (perhaps because of their generality) are not as tight. Perhaps our results can be tightened by considering special cases, and perhaps other results from random graphs can be extended to the study of sampling from arbitrary graphs.

Our randomized constructions show the *existence* of sparse subgraphs that accurately approximate cut values. A natural question is whether these subgraphs can be constructed deterministically in polynomial time. In the case of complete graphs, this has been accomplished through the deterministic construction of *expanders* (Gabber and Galil 1981). Indeed, just as the expander of (Gabber and Galil 1981) has constant degree, it may be possible to deterministically construct a $(1+\epsilon)$-accurate skeleton with a constant minimum cut, rather than the size $\Omega(\log n)$ minimum cut produced by the randomized construction.

A related question is whether we can derandomize the randomized rounding approach to network design problems. Raghavan (1988) uses the method of conditional expectations to derandomize the randomized-rounding algorithm for explicitly specified packing problems. However, this approach requires a computation for each constraint. This is not feasible for our problem with its exponentially many constraints.

A very general goal would be to reformulate other network problems in terms of cuts so that the sampling theorems could be applied.

A Counting Cuts

This section is devoted to proving a single theorem bounding the number of small cuts in a graph. This theorem is a slightly tightened version of one that appeared earlier (Karger and Stein 1996).

Theorem A.1 (Cut Counting). *In a graph with minimum cut c, there are less than $n^{2\alpha}$ cuts of value at most αc.*

We prove this theorem only for unweighted multigraphs, since clearly to every weighted graph there corresponds an unweighted multigraph with the same cut values: simply replace an edge of weight w with w parallel edges. To prove the theorem, we present an algorithm that selects a single cut from the graph, and show that the probability that a particular cut of value αc is selected is more than $n^{-2\alpha}$. It follows that there are less than $n^{2\alpha}$ such cuts.

A.1 The Contraction Algorithm

The algorithm we use is the *Contraction Algorithm* (Karger and Stein 1996). This algorithm is based on the idea of contracting edges. An efficient *implementation* is given by Karger and Stein (1996), but here we care only about the abstract algorithm.

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). Extending this definition, for an edge set F we will let G/F denote the graph produced by contracting all edges in F (the order of contractions is irrelevant up to isomorphism).

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. More formally, at any point in the algorithm, we can define $s(a)$ to be the set of original vertices contracted to a current metavertex a. Initially $s(v) = v$ for each $v \in V$, and whenever we contract (v, w) to create vertex x we let $s(x) = s(v) \cup s(w)$. We say a cut (A, B) in the contracted graph *corresponds to* a cut (A', B') in G, where $A' = \cup_{a \in A} s(a)$ and $B' = \cup_{b \in B} s(b)$. Note that a cut and its corresponding cut will have the same value. When the series of contractions terminates, yielding a graph with two metavertices a and b, we have a corresponding cut (A, B) in the original graph, where $A = s(a)$ and $B = s(b)$.

Lemma A.2. *A cut (A, B) is output by a contraction algorithm if and only if no edge crossing (A, B) is contracted by the algorithm.*

Proof. The only if direction is obvious. For the other direction, consider two vertices on opposite sides of the cut (A, B). If they end up in the same metavertex, then there must be a path between them consisting of edges that were contracted. However, any path between them crosses (A, B), so an edge crossing cut (A, B) would have had to be contracted. This contradicts our hypothesis. $\square$

We now give a particular contraction-based algorithm, and analyze it to determine the probability that a particular cut is selected. 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 2, 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 G

Figure 2: The Contraction Algorithm

Lemma A.3. *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 (A, B) with c crossing edges. We will use the term *minimum cut edge* to refer only to edges crossing (A, B). From Lemma A.2, we know that 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 (A, B), then the cut (A, B) 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 $2/r$. The probability that we never contract a minimum cut edge through all $n-2$ contractions is thus at least

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

□

A.2 Proof of Theorem

We now extend our analysis to prove the section's main theorem. To begin with, we have the following:

Corollary A.4. *The number of minimum cuts in a graph is at most* $\binom{n}{2}$.

Proof. In analyzing the contraction algorithm, we showed that the probability a minimum cut survives contraction to 2 vertices is at least $\binom{n}{2}^{-1}$. Since only one cut survives these contractions, the survivals of the different minimum cuts are disjoint events. Therefore, the probability that some minimum cut survives is equal to the sum of the probabilities that each survives. But this probability is at most one. Thus, if there are k minimum cuts, we have $k\binom{n}{2}^{-1} \leq 1$. □

This corollary has been proven in the past (Dinitz, Karzanov, and Lomonosov 1976; Lomonosov and Polesskii 1971). This bound is tight. In a cycle on n vertices, there are $\binom{n}{2}$ minimum cuts, one for each pair of edges in the graph. Each of these minimum cuts is produced by the Contraction Algorithm with equal probability, namely $\binom{n}{2}^{-1}$. We now extend the analysis to *approximately* minimum cuts. No such analysis was previously known.

Lemma A.5. *For α a half-integer, the probability that a particular α-minimum cut survives contraction to 2α vertices exceeds* $\binom{n}{2\alpha}^{-1}$.

Proof. We consider the unweighted case; the extension to the weighted case goes as before. The goal is to reapply Lemma A.2. Let α be a half-integer, and c the minimum cut, and consider some cut of weight at most αc. Suppose we run the

Contraction Algorithm. If with r vertices remaining we choose a random edge, then since the number of edges is at least $cr/2$, we take an edge from a cut of weight αc with probability at most $2\alpha/r$. If we repeatedly select and contract edges until $r = 2\alpha$, then the probability that the cut survives is

$$(1-\frac{2\alpha}{n})(1-\frac{2\alpha}{(n-1)})\cdots(1-\frac{2\alpha}{(2\alpha+1)}) \quad = \quad \binom{n}{2\alpha}^{-1}$$

□

Remark. A cycle on n vertices again shows that this result is tight, since each set of 2α edges forms an α-minimum cut.

Corollary A.6. *For α a half-integer, the number of α-minimum cuts is at most $2^{2\alpha-1}\binom{n}{2\alpha} < n^{2\alpha}$.*

Proof. We generalize Corollary A.4. Suppose we randomly contract a graph to 2α vertices. The previous lemma lower bounds the survival probability of an α-minimum cut, but we cannot yet apply the proof of Corollary A.4 because with more than one cut still remaining the cut-survival events are not disjoint. However, suppose we now take a random partition of the 2α remaining vertices. This partition gives us a corresponding unique cut in the original graph. There are only $2^{2\alpha-1}$ partitions of the 2α vertices (consider assigning a 0 or 1 to each vertex; doing this all possible ways counts each partition twice). Thus, we pick a particular partition with probability $2^{1-2\alpha}$. Combined with the previous lemma, this shows that we select a particular unique α-minimum cut with probability exceeding $2^{1-2\alpha}\binom{n}{2\alpha}^{-1}$. Now continue as in Corollary A.4.

Th $n^{2\alpha}$ bound follows from the fact that $2^{2\alpha-1} \leq (2\alpha)!$. □

We can also extend our results to the case where 2α is not an integer. We use *generalized binomial coefficients* in which the upper and lower terms need not be integers. These are discussed in Knuth (1973, Sections 1.2.5–6) (cf. Exercise 1.2.6.45). There, the Gamma function is introduced to extend factorials to real numbers such that $\alpha! = \alpha(\alpha-1)!$ for all real $\alpha > 0$. Many standard binomial identities extend to generalized binomial coefficients, including the facts that $\binom{n}{2\alpha} < n^{2\alpha}/(2\alpha)!$ and $2^{2\alpha-1} \leq (2\alpha)!$ for $\alpha \geq 1$.

Corollary A.7. *For arbitrary real values $\alpha > 1$, there are less than $n^{2\alpha}$ α-minimum cuts.*

Proof. Let $r = \lceil 2\alpha \rceil$. Suppose we contract the graph until there are only r vertices remaining, and then pick one of the 2^{r-1} cuts of the resulting graph uniformly at random. The probability that a particular α-minimum cut survives contraction to r vertices is

$$\begin{aligned}(1-\frac{2\alpha}{n})(1-\frac{2\alpha}{(n-1)})\cdots(1-\frac{2\alpha}{r+1}) &= \frac{(n-2\alpha)!}{(r-2\alpha)!}\frac{r!}{n!} \\ &= \frac{\binom{r}{2\alpha}}{\binom{n}{2\alpha}}.\end{aligned}$$

It follows that the probability our cut gets picked is $2^{1-r}\binom{r}{2\alpha}\binom{n}{2\alpha}^{-1}$. Thus the number of α-minimum cuts is at most $2^{r-1}\binom{n}{2\alpha}\binom{r}{2\alpha}^{-1} < \binom{n}{2\alpha}$. □

References

Aggarwal, A. (Ed.) (1993, May). *Proceedings of the* 25^{th} *ACM Symposium on Theory of Computing*. ACM: ACM Press.

Aggarwal, M. and N. Garg (1994, January). A scaling technique for better network design. See Sleator (1994), pp. 233–240.

Agrawal, A., P. Klein, and R. Ravi (1995, June). When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing 24*(3), 440–456. A preliminary version appeared in Proceedings of the 23^{rd} ACM Symposium on Theory of Computingpp. 134–144.

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

Aumann, Y. and Y. Rabani (1998). An $O(\log k)$ approximate min-cut max-flow theorem and approximation algorithm. *SIAM Journal on Computing 27*(1), 291–301.

Benczúr, A. A. and D. R. Karger (1996, May). Approximate s–t min-cuts in $\tilde{O}(n^2)$ time. See Miller (1996), pp. 47–55.

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

Clarkson, K. L. (1987). New applications of random sampling in computational geometry. *Discrete and Computational Geometry 2*, 195–222.

Clarkson, K. L. (1995). Las Vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM 42*(2), 488–499.

Clarkson, K. L. and P. W. Shor (1987). Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry 4*(5), 387–421.

Dinitz, E. A., A. V. Karzanov, and M. V. Lomonosov (1976). On the structure of a family of minimum weighted cuts in a graph. In A. A. Fridman (Ed.), *Studies in Discrete Optimization*, pp. 290–306. Nauka Publishers.

Edmonds, J. (1965). Minimum partition of a matroid into independents subsets. *Journal of Research of the National Bureau of Standards 69*, 67–72.

Eppstein, D., Z. Galil, G. F. Italiano, and A. Nissenzweig (1992, October). Sparsification—a technique for speeding up dynamic graph algorithms. In *Proceedings of the* 33^{rd} *Annual Symposium on the Foundations of Computer Science*, pp. 60–69. IEEE: IEEE Computer Society Press.

Erdös, P. and A. Rényi (1961). On the strength of connectedness of a random graph. *Acta Mathematica Acad. Sci. Hungar. 12*, 261–267.

Eswaran, K. P. and R. E. Tarjan (1976). Augmentation problems. *SIAM Journal on Computing 5*, 653–665.

Feller, W. (1968). *An Introduction to Probability Theory and its Applications* (third ed.), Volume 1. John Wiley & Sons.

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

Ford, Jr., L. R. and D. R. Fulkerson (1962). *Flows in Networks*. Princeton, New Jersey: Princeton University Press.

Frank, A. (1990). Packing paths, circuits, and cuts—a survey. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver (Eds.), *Paths, Flows, and VLSI Layout*, Volume 9 of *Algorithms and Combinatorics*, Chapter 4. Heidelberg: Springer-Verlag.

Gabber, O. and Z. Galil (1981). Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Sciences 22*, 407–420.

Gabow, H. N. (1991, October). Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the* 32^{nd} *Annual Symposium on the Foundations of Computer Science*, pp. 812–821. IEEE: IEEE Computer Society Press.

Gabow, H. N. (1993, November). A framework for cost-scaling algorithms for submodular flow problems. In L. Guibas (Ed.), *Proceedings of the* 34^{th} *Annual Symposium on the Foundations of Computer Science*, pp. 449–458. IEEE: IEEE Computer Society Press.

Gabow, H. N. (1995, April). A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences 50*(2), 259–273. A preliminary version appeared in Proceedings of the 23^{rd} ACM Symposium on Theory of Computing.

Gabow, H. N., M. X. Goemans, and D. P. Williamson (1993). An efficient approximation algorithm for the survivable network design problem. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pp. 57–74.

Goemans, M. X. and D. J. Bertsimas (1993). Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming 60*, 145–166.

Goemans, M. X., A. Goldberg, S. Plotkin, D. Shmoys, É. Tardos, and D. Williamson (1994, January). Improved approximation algorithms for network design problems. See Sleator (1994), pp. 223–232.

Grötschel, M., L. Lovász, and A. Schrijver (1988). *Geometric Algorithms and Combinatorial Optimization*, Volume 2 of *Algorithms and Combinatorics*. Springer-Verlag.

Karger, D. R. (1994a, May). Random sampling in cut, flow, and network design problems. In *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing*, pp. 648–657. ACM: ACM Press. Mathematics of Operations Research, To appear.

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

Karger, D. R. (1994c, January). Using randomized sparsification to approximate minimum cuts. See Sleator (1994), pp. 424–432.

Karger, D. R. (1996, May). Minimum cuts in near-linear time. See Miller (1996), pp. 56–63.

Karger, D. R. (1998a, January). Better random sampling algorithms for flows in undirected graphs. In H. Karloff (Ed.), *Proceedings of the* 9^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 490–499. ACM-SIAM.

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

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

Karger, D. R. and M. Levine (1998, May). Finding maximum flows in simple undirected graphs seems faster than bipartite matching. In *Proceedings of the 29^{th} ACM Symposium on Theory of Computing*, pp. 69–78. ACM: ACM Press.

Karger, D. R. and C. Stein (1993, May). An $\tilde{O}(n^2)$ algorithm for minimum cuts. See Aggarwal (1993), pp. 757–765.

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

Khuller, S. and B. Raghavachari (1995, May). Improved approximation algorithms for uniform connectivity problems. In *Proceedings of the 27^{th} ACM Symposium on Theory of Computing*, pp. 1–10. ACM: ACM Press.

Khuller, S. and B. Schieber (1991, April). Efficient parallel algorithms for testing connectivity and finding disjoint s-t paths in graphs. *SIAM Journal on Computing 20*(2), 352–375.

Khuller, S. and U. Vishkin (1994, March). Biconnectivity approximations and graph carvings. *Journal of the ACM 41*(2), 214–235. A preliminary version appeared in Proceedings of the 24^{th} ACM Symposium on Theory of Computing.

Klein, P., S. A. Plotkin, C. Stein, and É. Tardos (1994). Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing 23*(3), 466–487. A preliminary version appeared in Proceedings of the 22^{nd} ACM Symposium on Theory of Computing.

Knuth, D. E. (1973). *Fundamental Algorithms* (2nd ed.), Volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company.

Knuth, D. E. and A. C. Yao (1976). The complexity of nonuniform random number generation. In J. F. Traub (Ed.), *Algorithms and Complexity: New Directions and Recent Results*, pp. 357–428. Academic Press.

Leighton, T. and S. Rao (1988, October). An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29^{th} Annual Symposium on the Foundations of Computer Science*, pp. 422–431. IEEE: IEEE Computer Society Press.

Linial, N., E. London, and Y. Rabinovich (1995). The geometry of graphs and some of its algorithmic applications. *Combinatorica 15*(2), 215–246. A preliminary version appeared in Proceedings of the 35^{th} Annual Symposium on the Foundations of Computer Science.

Lomonosov, M. V. and V. P. Polesskii (1971). Lower bound of network reliability. *Problems of Information Transmission 7*, 118–123.

Matula, D. W. (1993, January). A linear time $2+\epsilon$ approximation algorithm for edge connectivity. In *Proceedings of the 4^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 500–504. ACM-SIAM.

Miller, G. (Ed.) (1996, May). *Proceedings of the 28^{th} ACM Symposium on Theory of Computing*. ACM: ACM Press.

Motwani, R. and P. Raghavan (1995). *Randomized Algorithms*. New York, NY: Cambridge University Press.

Nagamochi, H. and T. Ibaraki (1992a, February). Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics 5*(1), 54–66.

Nagamochi, H. and T. Ibaraki (1992b). Linear time algorithms for finding k-edge connected and k-node connected spanning subgraphs. *Algorithmica 7*, 583–596.

Nash-Williams, C. S. J. A. (1969). Well-balanced orientations of finite graphs and unobtrusive odd-vertex-pairings. In W. T. Tutte (Ed.), *Recent Progress in Combinatorics*, pp. 133–149. Academic Press.

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

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

Sleator, D. D. (Ed.) (1994, January). *Proceedings of the 5^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM.

Tarjan, R. E. (1983). *Data Structures and Network Algorithms*, Volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM.

Williamson, D., M. X. Goemans, M. Mihail, and V. V. Vazirani (1993, May). A primal-dual approximation algorithm for generalized Steiner problems. See Aggarwal (1993), pp. 708–717.

Winter, P. (1987). Generalized Steiner problem in outerplanar networks. *Networks*, 129–167.