Better Random Sampling Algorithms for Flows in Undirected Graphs

David R. Karger*

November 3, 1997

Abstract

We present better random sampling algorithms for maximum flows in undirected graphs. Our algorithms apply to capacitated or uncapacitated graphs, and find a maximum flow of value v in $\tilde{O}(\sqrt{mnv})$ time. This improves on a previous bound of $\tilde{O}(m^{2/3}n^{1/3}v)$ given by the author recently, which in turn improved on the $O(mv)$ time bound for a typical augmenting path algorithm. In uncapacitated graphs without parallel edges, the bound is no worse than $\tilde{O}(n^{5/2})$. We give another algorithm that finds a $(1-\epsilon)$ times maximum flow in time $\tilde{O}(m\sqrt{n}/\epsilon)$, regardless of v.

1 Introduction.

Random sampling has been a useful tool for solving cut problems in undirected graphs. In previous work [Kar97a], this author showed that randomly choosing edges from a graph yields a sampled graph in which every cut is close to its expected value with high probability. This led to algorithms for approximating [Kar97a] and exactly finding [Kar96] global min-cuts in near-linear time with high probability. Benczúr and Karger [BK96] extended these results, showing how to find a $(1+\epsilon)$ times minimum s-t cut in $\tilde{O}((n/\epsilon)^2)$ time[1] on n-vertex capacitated graphs. However, these schemes could not find flows or exact s-t min-cuts, even in uncapacitated graphs. Karger [Kar97b] exploited the ability to find approximate s-t minimum cuts in an algorithm that finds a maximum flow of value v in an undirected uncapacitated graph in $\tilde{O}(m^{2/3}n^{1/3}v)$ time.

We make two significant improvements on the previous work. Our first improvement is a general *smoothing* procedure that allows us to apply sampling-based *flow* algorithms to *capacitated* graphs (the previous sampling algorithm of [BK96] for capacitated graphs only found cuts). In particular, we can speed up the new $\tilde{O}(m^{3/2})$-time flow algorithm of Goldberg and Rao [GR97a] to find a $(1-\epsilon)$ times optimum flow in any capacitated undirected graph in $\tilde{O}(m\sqrt{n}/\epsilon)$ time.

Our second improvement is upon the work of [Kar97b]. We develop a new partitioning-based scheme that is simpler and more powerful than that paper's. The previous algorithm applied only to uncapacitated graphs. Furthermore, it used a minimum cost flow routine as part of its solution. Our new algorithm replaces the minimum-cost flow routine with a standard maximum flow routine, which can of course be solved recursively by our improved maximum flow algorithm. This lets us improve the time to find a flow of value v from $\tilde{O}(m^{2/3}n^{1/3}v)$ to $\tilde{O}(\sqrt{mnv})$. Unlike the previous paper's bounds, ours also applies to capacitated graphs (though the dependence on v means it is only useful for small flow values).

These results improve on all known flow algorithms on uncapacitated graphs. They always dominate (to within logarithmic factors) the standard *augmenting path* algorithm that runs in $O(mv)$ time. They also dominate *blocking flow* based algorithms over an interesting range of parameters. On *simple* uncapacitated graphs—those without parallel edges—blocking flow based algorithms have long been known to run in $O(m^{3/2})$ and in $O(mn^{2/3})$ time [Eve79]. Recently, Goldberg and Rao [GR97b] gave a blocking flow based algorithm that runs in $O(\sqrt{m}nn)$ time. Of these three bounds, the first dominates when $n \le m \le n^{4/3}$, the second when $n^{4/3} \le m \le n^{3/2}$, and the last when $n^{3/2} \le m$. We dominate the first bound whenever $v \le m/\sqrt{n}$ and the second whenever $v \le m^{1/2}n^{1/6}$. Since a simple graph has $v \le n$, our algorithm is always within a logarithmic factor of the third, Goldberg-Rao bound, and is faster for $v = n/\log^{O(1)} n$.

In simple graphs, we can upper bound $m = O(n^2)$ and $v = O(n)$. The best of the past algorithms is then the Goldberg-Rao algorithm, whose bound simplifies to $O(n^{5/2})$. Our algorithm also achieves an $\tilde{O}(n^{5/2})$ running time, but applies to non-simple or capacitated graphs as well.

The algorithms discussed in this paper are all randomized Las Vegas algorithms. This means that the algorithms are guaranteed to be correct but the running times only holds with high probability. Time in this abstract always refers to the expected (and high probability) running time, rather than a deterministic quantity.

1.1 Background.

This paper reunifies two themes that were explored separately in previous work [Kar97a, BK96].

*MIT Laboratory for Computer Science, Cambridge, MA 02138. Supported by NSF contract CCR-9624239 and an Alfred P. Sloane Foundation Fellowship.
email: `kargerlcs.mit.edu.`
URL: `http://theory.lcs.mit.edu/~/karger`

[1] $\tilde{O}(f)$ denotes $f \log^{O(1)} n$

In this section, we provide background from that work which will be used in the current paper. A great deal of our discussion relies on a graph's *connectivity*. In this paper, connectivity always refers to *edge connectivity*, that is, the value of a minimum edge cut in the graph.

In earlier work [Kar97a], this author showed how to apply random sampling to cut and flow problems. If we choose every edge of a graph G independently with probability p, then we get a subgraph $G(p)$ each of whose cuts has expected value exactly p times the value of the corresponding cut in G (the *corresponding cut* is the one defined by the same vertex partition). If this correspondence were exact, we could, for example, find approximate min-cuts in G by finding minimum cuts in $G(p)$. Unfortunately, there is some variance in the random samples. However, we showed that this variation was small if G had a large minimum cut c: the relative errors were bounded by $\tilde{O}(1/\sqrt{pc})$. This gave fast algorithms for approximating minimum cuts and flows in such graph. A related divide and conquer scheme found a flow of value v in $\tilde{O}(mv/\sqrt{c})$ time.

The result had two limitations. The first is that it worked only for uncapacitated graphs. The second was the requirement that the sampling probability exceed $\tilde{O}(1/c)$ for small error: this meant that the approach was only useful for graphs with large min-cuts. Benczúr and Karger [BK96] dealt with these problems by showing that any graph with many edges has well-connected regions inside of which the sampling probability can be lowered safely. This led to a *graph compression* scheme that worked for *capacitated* graphs with *arbitrary* connectivities. Compression yields a graph with $\tilde{O}(n/\epsilon^2)$ edges in which all cuts are within $(1 \pm \epsilon)$ of their values in G. We can find approximately minimum cuts in G by finding minimum cuts in the compressed graph. Since the compressed graph is small, finding minimum cuts (via maximum flows) is easy.

Graph compression has a different drawback. The graph it builds has the same cut values as the original, but individual edge capacities bear no such relationship. Thus, although minimum cuts in the compressed graph yield approximately minimum cuts in G, it was not clear how to turn a maximum compressed-graph flow into a flow in the original graph. This problem was first addressed in [Kar97b], where a scheme based on vertex partitioning led to an algorithm running in $\tilde{O}(m^{2/3}n^{1/3}v)$ time. That algorithm applied only to uncapacitated graphs.

1.2 New Results. We make two improvements on the work just discussed. Our first improvement is to extend the basic graph sampling theorems to capacitated graphs. Picking each edge with probability p clearly yields a graph with fewer edges—a good start for flow algorithms. Unfortunately, the original sampling theorems cannot prove good bounds on cut values in the resulting sample. In uncapacitated graphs, a single edge makes up only a $1/c$ fraction of the capacity of any cut, so sampled cut values are "smooth"—the presence or absence of a single edge in the sample does not dramatically affect the value of the sample. If the edges of a graph are capacitated, there can be a large variance in sampling outcomes: for example, even if a capacitated graph has a large minimum cut, it might have a single bridge edge (of large capacity) connecting two otherwise disconnected components. If we randomly sample edges, there is a good chance that we will miss the bridge edge and thus disconnect the graph. So sampling this capacitated graph cannot preserve cut values.

In Section 3 we show how to apply a "smoothing" procedure to any graph, creating a graph in which sampling every edge with probability p gives the desired tight concentration of sampled cut values. We smooth the graph by identifying some of its "critical" high capacity edges and breaking them into parallel edges of smaller capacity. The smaller-capacity edges have less impact on the values of cuts they participate in, thus reducing the variance. Smoothing creates extra edges, but we show that the number of new edges is negligible. This immediately lets us extend all of the uncapacitated-graph bounds of [Kar97a] to capacitated graphs. We can also combine smoothing with the new $\tilde{O}(m^{3/2})$-time flow algorithm of Goldberg and Rao [GR97a] to find a $(1 - \epsilon)$ times optimum flow in *any capacitated* undirected graph in $\tilde{O}(m\sqrt{n}/\epsilon)$ time.

Our second improvement in this paper is upon the work of [Kar97b]. We develop a new partitioning scheme that is simpler and more powerful than that previous paper's. We use a *k-strong partition*. This is a partition of the graph vertices into groups, such that each group induces a k-connected subgraph, but $O(kn)$ edges connect different groups. The existence of such a partition, and construction of a weaker structure, was was shown in [BK96]. In [Kar97b], we showed how to build a k-strong partition in $\tilde{O}(m)$ time. After constructing such a partition, we can divide our flow task in two: we find a flow on the cross-partition edges, and then we find a flow on the edges inside each component of the partition.

There are few cross-partition edges, so finding a flow on them would be easy if we could avoid dealing with the large number of edges inside each component. Conveniently, the compression scheme of [BK96] can be applied to these in-component edges, reducing their number without significantly affecting the flow value. We quickly find a flow on this partially compressed graph, and from it extract flow values on the cross-partition edges. This step is described in Section 5.

The flow values just assigned to the cross-partition edges send some flow into and out of each of the components in the partition of G. Our next step is to route this flow demand through each of the components on edges of G. This reduces to solving a maximum flow problem on each component. These components have many edges. However, the partition is such that the components are well connected. Thus, we can apply the algorithm of [Kar97a] for graphs with large connectivities to quickly find the flow within each component. This step, which involves some additional complications, is described in Section 6.

By itself, our partition-based approach yields a flow algorithm with an $\tilde{O}(m^{2/3}n^{1/3}v)$ running time, matching that of [Kar97b]. But unlike that algorithm, this one can recurse. The subtasks of our algorithm involve finding flows in certain subsidiary graphs. In [Kar97b], there were two obstacles to finding these flows recursively. The first was that the subproblems were *min-cost* flow problems which we do not know how to solve using our techniques. The second was that the subproblems arose from compressed graphs, which are *capacitated* and thus were not amenable to the sampling techniques of that paper. Our new partitioning scheme lets us solve standard (non-costed) maximum flow problems in the recursive step. The graph smoothing techniques described earlier let us apply all of our results to capacitated as well as uncapacitated graphs, thus resolving the second problem and yielding the $\tilde{O}(\sqrt{mnv})$ time bound. We describe this step in Section 7.

2 A Local Connectivity Measure

The original work on sampling had results parameterized by the connectivity c of the graph being sampled. In this section, we discuss a localized connectivity measure that leads to more powerful results. We begin with some definitions and results from [BK96].

Definition 2.1. A graph is *k-connected* if the value of each cut in G is at least k. A *k-strong component* is a k-connected vertex inducted subgraph of a graph.

Definition 2.2. The *strong connectivity* of an edge e, denoted k_e, is the maximum value of k such that a k-strong component contains e. We say e is *k-strong* if its strong connectivity is k or more, and *k-weak* otherwise.

Lemma 2.3. *In a capacitated graph with edge capacities u_e and strengths k_e,*

$$\sum u_e/k_e \leq n-1.$$

Proof. Define the cost of edge e to be u_e/k_e. We show that the total cost of edges is at most $n-1$. Let C be any connected component of G and suppose it has connectivity k. Then there is a cut of value k in C. On the other hand, every edge of C is in a k-strong component of G (namely C) and thus has strength at least k. Therefore,

$$\begin{aligned} \sum_{e \text{ crossing } C} u_e/k_e &\leq \sum u_e/k \\ &= k/k \\ &= 1 \end{aligned}$$

Thus, by removing the cut edges, of total cost at most 1, we can break C in two, increasing the number of connected components of G by 1.

If we find and remove such a cut $n-1$ times, we will have a graph with n components. This implies that all vertices are isolated, meaning no edges remain. So by removing $n-1$ cuts of cost at most 1 each, we have removed all edges of G. Thus the total cost of edges in G is $n-1$. □

2.1 Measuring Edge Strengths. At present, the fastest way to exactly compute all strengths k_e is to perform n minimum cut computations. This takes $\tilde{O}(nm)$ time. However, we showed previously [BK96] that approximations to k_e can be found faster. In $\tilde{O}(m)$ time, we can compute lower bounds $k'_e \leq k_e$ such that $\sum u_e/k'_e = O(n)$. Such bounds are sufficient for all applications discussed in this paper, so for brevity we will assume that we are working with the actual values k_e.

2.2 Strong Partitions The sum above can be used to bound the total capacity of edges of given strength. We can also say something about the structure of these edges.

Definition 2.4. A *k-strong partition* of a graph G is a partition of the vertices of G into r k-strong components such that the number of edges *crossing* the partition (that is, with endpoints in different sets) is $\tilde{O}(kr)$.

For any k, every graph has a k-strong partition (this is implicit in Lemma 2.3, or see a proof in [BK96]). We will see shortly that one can be found in $\tilde{O}(m)$ time. This allows for a useful division of labor in our flow algorithms. We run standard flow algorithms on the cross partition edges; this is cheap since there are few such edges. We run our high-connectivity flow algorithms on the components; this is cheap since each component is k-connected.

Slightly different definitions of k-strong partitions were used in [BK96] and [Kar97b]. Our definition here is the most convenient at present.

Lemma 2.5. *A k-strong partition can be found in $\tilde{O}(m)$ time.*

Proof. In [Kar97b], we gave an $\tilde{O}(m)$ time algorithm A that almost achieves what we want. It gives a partition into k-connected components, but the number of cross-partition edges is only bounded by $akn\log n$ for some constant a, even if the partition has few components. However, this algorithm is easily modified to meet our current needs. More precisely, we prove that at most $\log n$ calls to A are needed to get a k-strong partition, which proves that the total time is still $\tilde{O}(m)$.

Suppose that A returns a partition with $akn\log n$ crossing edges (for some constant a) when called on an n-vertex graph. Call it to get a partition of G. At most $akn\log n$ edges cross this partition. If the number of components exceeds $n/2$ then we are done: we have a k-strong partition with constant $2a$. Otherwise, contract each component of the k-strong partition to a vertex, yielding a graph G' with at most $n/2$ vertices. By induction, we can find a k-strong partition of this graph via $\log n/2 = \log n - 1$ calls to A. Each k-strong component in this partition is a made up of contracted groups from G; since each of these groups was k-connected, uncontracting them yields k-strong components in G. □

3 Sampling in Capacitated Graphs.

In this section we introduce *graph smoothing*—a technique that lets us apply random sampling to capacitated graphs, yielding fast algorithms for flows in such graphs. We show how to smooth any capacitated graph so that the sampling results of [Kar97a] can be applied to capacitated graphs. This yields the following results as easy corollaries:

Theorem 3.1. *In a capacitated undirected graph, a $(1-\epsilon)$-times-maximum flow can be found in $\tilde{O}(m\sqrt{n}/\epsilon)$ time.*

Theorem 3.2. *In an integer-capacitated graph with minimum cut c, a flow of value v can be found in $\tilde{O}(mv/\sqrt{c} + \sqrt{mnv})$ time.*

The first result follows from a trivial application of the new Goldberg-Rao flow algorithm [GR97a]. The second uses the divide and conquer methods of [Kar97a], and is crucial in the later part of this paper.

We extend the following theorem of [Kar97a]:

Theorem 3.3 ([Kar97a]). *If G has connectivity c and edge are sampled with probability $p > 8(\ln n)/\epsilon^2 c$, then with high probability all cuts in $G(p)$ are within $(1 \pm \epsilon)$ of their expected values.*

Since the sample had fewer edges than the original graph, flow algorithms ran faster on it. This led to algorithms for approximating global min-cuts quickly by finding minimum cuts in the sample. The sampling proof used a Chernoff bound, which relied on individual edges having only a small impact on the outcome of the experiment. In particular, since the graph had minimum cut c, and every edge was being chosen with probability p, every cut had expected value at least pc. Thus, the presence or absence of a single (capacity 1) edge could affect that value of a cut by at most a $1/pc$-fraction of its expected value. Sampling also led to the following result that we will use later:

Theorem 3.4. *In an uncapacitated graph with connectivity c, a max-flow of value v can be found in $\tilde{O}(mv/\sqrt{c})$ time. A $(1-\epsilon)$ times maximum flow can be found in $\tilde{O}(mv/\epsilon c)$, and a $(1+\epsilon)$ times minimum s-t cut can be found in $\tilde{O}(mv/\epsilon^2 c)$ time.*

Theorem 3.3 breaks down in capacitated graphs. A capacity-c edge might be the only edge crossing a cut of value c, meaning its presence or absence has a huge effect on the value of that cut in the sampled graph. One way to fix this problem is to replace a capacity-w edge by a collection of w parallel unit-capacity edges. Unfortunately this can increase the number of edges in the original, and thus the sampled, graph so much that sampling is worse than useless.

The idea of k-strong components lets us circumvent this problem. We know that a graph has well-connected subgraphs of connectivity $k \gg c$. Any edge in such a subgraph only crosses cuts of value exceeding k. Thus a Chernoff bound can apply so long as the capacity of the edge is small relative to k—even if it is large relative to c. In [BK96], this observation was used in algorithms for approximating cuts, but in a way that made it unable to approximate flows. We rectify this problem here.

3.1 Smooth Graphs.

We would like to sample a subset of edges from a *capacitated* graph without introducing large error in cut values. It is easy to extend the notation $G(p)$ to denote taking each capacitated edge with probability p, but somewhat harder to prove that sampling does the right thing. As discussed above, the problem is that a single capacitated edge might account for much of the capacity crossing a cut. The presence or absence of this edge has a major impact on the value of this cut in the sampled graph. However, the idea of edge strength described above gives us a useful bound on how much impact a given edge can have.

Definition 3.5. A graph G with edge capacities u_e and edge strengths k_e is *c-smooth* if for every edge, $k_e \geq cu_e$.

Note that a graph with integer edge weights and minimum cut c has smoothness at most c but possibly much less. We now argue that smoothness is the criterion we need to apply sampling to weighted graphs.

Lemma 3.6. *Let G be a c-smooth graph. Let $p = \rho_\epsilon/c$ where $\rho_\epsilon = O((\log n)/\epsilon^2)$. Then with high probability, every cut in $G(p)$ has value in the range $(1 \pm \epsilon)$ times its expectation (which is p times its original value).*

Proof. (Sketch). The proof runs along the same lines as one in [BK96]. The crucial idea used in the construction there was to ensure that an edge that took on capacity w in the sampled graph was contained in a k-strong component for some $k \geq \rho_\epsilon w$. In other words, the presence or absence of the edge could affect the value of the sampled cut by at most a $1/\rho_\epsilon$ fraction of its expectation. Thus, a Chernoff bound showed that the relative deviation from expectation was bounded by ϵ. In that paper, it was always the case that the *expected* capacity of an edge was 1, but this was not actually crucial for the proof.

Smoothness achieves the same goal. Consider a c-smooth graph in which every edge is sampled with probability p. Consider some edge e of capacity u_e. By definition, this edge is in a component of strength at least cu_e. So the expected capacity of any cut this edge crosses is at least pcu_e.

Thus, the contribution of edge e to the value of the sampled cut is at most a $1/pc = 1/\rho_\epsilon$ fraction of its expected value. The Chernoff bound therefore shows that the relative deviation from expectation is bounded by ϵ with high probability.

Of course, this argument refers to one cut, while a graph has exponentially many. With so many cuts, some might be expected to suffer unusually large deviations from expectation. However, the proof from [BK96] shows that this is not a problem. □

3.2 Sampling Algorithms. We have argued that in any c-smooth graph, sampling with probability p produces a graph in which with high probability all cuts are within $(1 \pm \epsilon)$ of their expected values. An immediate consequence is that (so long as $p = \Omega(\log n/c)$) $G(p)$ is pc-smooth with high probability—if edge e originally had strength k_e, meaning that it was in a k_e connected subgraph, then that subgraph is $\Omega(pk_e)$-connected in $G(p)$, implying the claim.

These two facts (tight concentration of cut values, and its corollary on smoothness) are the only ones used in the uncapacitated graph algorithms of [Kar97a]. Therefore, those results immediately generalize to the smooth graphs defined here—we simply replace "minimum cut" with "smoothness" in all of those results. The generalizations are as follows:

Lemma 3.7. *Let $T(m, n, v, c)$ be the time to find a maximum flow in a (integer capacitated) graph with m edges, n vertices, flow v and smoothness c. Then for any ϵ, the time to find a flow of value $(1 - \epsilon)v$ on an m-edge, n-vertex, smoothness-c graph is*

$$\tilde{O}(\frac{1}{p}T(pm, n, pv, pc))$$

where $p = \Theta((\log n)/\epsilon^2 c)$.

Proof. (Sketch.) Divide the graph edges into $1/p$ random groups. Each defines a graph with pm edges and (by the sampling theorem for smooth graphs) flow $(1 - \epsilon)pv$. Find flows in each piece, and combine the results. □

Lemma 3.8. *On integer capacitated c-smooth graphs, a maximum flow can be found in $\tilde{O}(mv/\sqrt{c})$ time.*

Proof. (Sketch.) Divide the graph edges into two random groups. Recursively find maximum flows in each group. Combine the flows. This yields a flow of value $(1 - \tilde{O}(1/\sqrt{c}))v$ in the original graph. Use $\tilde{O}(v/\sqrt{c})$ augmenting paths to turn it into a maximum flow. □

3.3 Making Graphs Smooth. Since it is easy to find flows in smooth graphs, we now give algorithms for transforming any graph into a smooth one.

Lemma 3.9. *Given an m edge capacitated graph, a* smoothness parameter *r, and the strengths k_e of all edges, we can transform the graph into an $(m + O(rn))$-edge r-smooth graph in $\tilde{O}(m)$ time.*

Proof. Suppose we have an edge with capacity u_e and strength k_e such that $k_e < ru_e$. Divide the edge into $\lceil ru_e/k_e \rceil$ parallel edges, each of capacity less than k_e/r but with total capacity u_e. These edges remain k_e strong, but now satisfy the smoothness criterion.

It remains to prove that this division creates $O(nr)$ new edges. But the number of edges in our divided graph is simply

$$\begin{aligned}\sum \lceil ru_e/k_e \rceil &\leq m + \sum ru_e/k_e \\ &= m + r\sum u_e/k_e \\ &\leq m + rn\end{aligned}$$

where the last line follows from Lemma 2.3. □

We now argue that the values k_e are not actually needed to perform the construction.

Lemma 3.10. *Given an m-edge integer-capacitated graph and smoothness parameter r, in $\tilde{O}(m)$ time, we can transform it into an $m+O(rn)$ edge integer-capacitated r-smooth graph.*

Proof. In [BK96], we gave an $\tilde{O}(m)$-time algorithm that produced lower bounds $k'_e < k_e$ that still satisfy Lemma 2.3 to within a constant factor. Since these values k'_e are underestimates for the actual strengths, using them instead of the actual strengths can only make our resulting graph smoother. On the other hand, since Lemma 2.3 is still satisfied, the algorithm generates only rn edges. □

Corollary 3.11. *In $O(m)$ time we can transform any m-edge capacitated graph into an $O(m)$-edge capacitated (m/n)-smooth graph.*

The lemmas above do not guarantee that the edge capacities in the smooth graph we construct will be integers. For some of our applications below, this is a necessity.

Lemma 3.12. *Given an m edge capacitated graph with minimum cut at least c, in $\tilde{O}(m)$ time, we can transform an m-edge capacitated graph into an $m+O(cn)$ edge c-smooth graph with integer capacity edges.*

Proof. Simply note that when aiming for a smoothness equal to the minimum cut, all strengths k_e will exceed the smoothness parameter. Thus, we will never need to subdivide an edge of capacity 1. □

3.4 General graphs. We can apply the smooth-graph algorithms to arbitrary graphs by smoothing them first.

Lemma 3.13. *Consider any maximum flow algorithm that handles non-integer edge capacities with running time $T(m, n)$. For any ϵ, a $(1 - \epsilon)$ times optimum maximum flow can be found in*

$$\tilde{O}(\frac{\epsilon^2 m}{n}T(n/\epsilon^2, n))$$

time.

Proof. Start by converting the graph into an $r = m/n$-smooth graph with $2m$ edges. Define $p = 1/\epsilon^2 r$ and apply Lemma 3.7. $\square$

The new Goldberg-Rao [GR97a] algorithm gives $T(m, n) = \tilde{O}(m^{3/2})$ for the previous lemma. (Since we are already giving up a factor of ϵ, we can assume without loss of generality that all edge capacities are polynomial, thus eliminating the scaling term.) Plugging this in proves Theorem 3.1.

We can also use this lemma to prove Theorem 3.2. Start by making the graph c-smooth with $m + nc$ integer capacity edges, as described in Lemma 3.12. If $m > nc$, so that the number of edges in the smooth graph is less than $2m$, the algorithm of Lemma 3.8 runs in $O(mv/\sqrt{c})$ time. Otherwise, consider Lemma 3.7. Set $p = m/nc$ in that algorithm. This yields a flow of value $(1 - \epsilon)v$ where $\epsilon = \tilde{O}(1/\sqrt{pc})$. If we use the algorithm of Lemma 3.8 to take $T(m, n, v, c) = \tilde{O}(mv/\sqrt{c})$, then the running time of our approximation algorithm is

$$\tilde{O}(\frac{1}{p}p(m + nc)(pv)/\sqrt{pc}) = \tilde{O}(nv\sqrt{pc})$$

since $m \leq nc$. Since we have a flow of value $(1 - \epsilon)v$, an additional ϵv augmenting paths will find the maximum flow. Of course, these augmentations can be performed in the original graph which has only m edges, so they take time $\epsilon mv = \tilde{O}(mv/\sqrt{pc})$. So our overall time is

$$\tilde{O}(nv\sqrt{pc} + mv/\sqrt{pc}) = \tilde{O}(\sqrt{mn}v).$$

4 Inter-Component Flows.

Graph smoothing resolves the distinction between capacitated and uncapacitated graphs. However, the smoothing-based algorithms described above still suffer from their dependence on the minimum cut value c. When c is small, smoothing does not help. To solve this problem, we use the *k-strong partition* discussed in section 2.2, which shows that even a graph with small connectivity is well connected in most places. We use the partition as follows. First, we find how flow should be carried on cross-partition edges. Then, we find how the cross-partition flow should be routed inside the components of the partition. The first step is easy because there are few cross-partition edges; the second because each component is well connected (so we can use the algorithms of [Kar97a]).

To simplify our presentation, we spend the next few sections developing an $\tilde{O}(m^{2/3}n^{1/3}v)$ algorithm for *uncapacitated graphs*. While this bound is now better than the previous one [Kar97b], we will show in Section 7 how to adapt it to achieve the claimed bound.

4.1 Definitions. In this section, we give some of the technical definitions and lemmas that will be needed to make this approach work.

It is convenient to restrict attention to k-strong partitions that have the source and sink in singleton components of the partition. We can assume this is the case by adding a new source connected only to S by an edge of capacity v, and a new sink connected only to t by an edge of capacity v, and placing each of these new vertices in its own singleton component. It is easy to see that if the original partition was k-strong, the new partition is still k-strong.

Consider a partition P of G. Certain edges have endpoints in two different components of P; we refer to these as the *edges of* P or the P*-edges*. Other edges have both endpoints within the same component C; we refer to these as C-edges. If an edge has exactly one endpoint in C, we refer to it as *incident on* C. The set of edges incident on C forms the *boundary* of C. Note that every edge is in the boundary of 0 or 2 components. All boundary edges are P-edges and vice versa. We also define the *degree* of C to be the number of edges incident on C.

Consider a flow f in G. This flow assigns a legal (within capacity) flow value to every edge in G. In particular it assigns flow value to every P-edge. A set of assignments of flow values to P-edges will be called a P*-flow*, denoted f_P. We call the P-flow *legal* if it is consistent with some flow on G—that is, can be extended to a flow on G by assigning legal flow values to the non-P-edges of G such that flow is conserved (that is, the same amount of flow enters as exits) at every vertex but the source and sink. Stating things differently, we can define a legal P-flow to be the set of values given to P-edges by some legal flow in G. The *value* of a legal P-flow f_P is the value of any s-t flow that legally extends f_P to a flow. To see that this value is well defined, recall our assumption that the source s is a singleton vertex of the component. Therefore f_P assigns a flow value to every edge incident on s. This determines the net flow out of s, which gives the value of the flow.

Besides defining a P-flow f_P in G, a flow in G also assigns flow value to every edge inside a given component C. We refer to an assignment of flow values to edges of C as a C*-flow*. It is a *legal* C-flow if it is the set of values assigned to edges of C by a legal s-t flow. If we consider a legal P-flow and legal C-flow that both arise from the same underlying s-t flow, we observe that the P-flow and C-flow are *consistent* in the following sense: the flow values that f_P assigns to the boundary edges of C, along with the values of the C-flow, obey flow conservation at all vertices of C (this is trivial from the fact that the P-flow and C-flow are projections of a flow).

Given a legal P-flow f_P, the task of extending it to a legal s-t flow (of the same value) is the task of assigning values to the non-P-edges of G so that flow conservation is achieved. In light of the previous paragraph, this can be rephrased as follows: for each component C of P, find a legal C-flow that is consistent with f_P.

This turns out to be an instance of the *feasible flow problem* [AMO93, Section 6.2] on C. To see how it can be solved, consider a P-flow f_P and a particular component C. Define an auxiliary *demand graph* C_{ST} for C as follows. Add a *virtual source vertex* S and *virtual sink vertex* T to C. For every P-edge (v, w) with $w \in C$ and $f_P(v, w) > 0$ (that is, an edge carrying flow *into* C), add an edge (S, w) of capacity $f_P(v, w)$. For each P edge (v, w) with $v \in C$ and $f_P(v, w) > 0$ (that is, an edge carrying flow *out of* C), add an edge (v, T) in C_{ST} with capacity $f_P(v, w)$. Consider a legal C-flow that is consistent with f_P. It is easy to verify that if we assign its flow-values to C_{ST} and saturate every edge leaving S and entering T, we get a legal (maximum) S-T flow in C_{ST}. Conversely, it is easily verified that any S-T max-flow in C_{ST} yields a legal C-flow that is consistent with f_P. The value of this max-flow is just the sum of capacities of edges leaving S. We will refer to this quantity as the amount of flow that f_P *routes through* C.

The above discussion can be summarized as follows. Given a partition P, if we want to find a max-flow of value v, then it suffices to find a legal P-flow of value v and then, for each component C, to find a legal C-flow consistent with f_P.

4.2 The Algorithm Using the previous definitions, we can give a more detailed picture of our planned algorithm. We start working from a k-strong partition P. We aim to find a legal P-flow and then extend it to the components. The P-flow will be easy to find because there are few P-edges. The components will be easy to solve because each of them has high connectivity. The next two sections are devoted to proving the following two lemmas:

Lemma 4.1. *Given a graph G with maximum s-t flow v, and a k-strong partition of G, a P-flow of value $(1-1/\sqrt{k})v$ can be found in $\tilde{O}(knv)$ time.*

Note that this time bound is the one needed to find a flow of value v in a graph with $\tilde{O}(kn)$ edges. In other words, this lemma succeeds in ignoring the intra-component edges of G. This lemma is proved in Section 5.

Lemma 4.2. *If P is a k-strong partition, we can extend a P-flow of value v to an s-t flow of the same value in G in $\tilde{O}(mv/\sqrt{k} + m\sqrt{k})$ time.*

Note that this time bound is the one needed to find a flow of value v in a k-connected graph. In other words, this lemma succeeds in ignoring the cross-partition edges of G. This lemma is proved in Section 6.

We can use these two lemmas to find a maximum flow as follows.

1. Find a k-strong partition P of G for some k.
2. Find a P-flow of value $(1 - 1/\sqrt{k})v$ using the method of the previous section.
3. Extend the P-flow to G using the algorithm of this section.
4. Use $\tilde{O}(v/\sqrt{k})$ augmenting paths to raise the flow value to v.

Since the final augmenting paths take $\tilde{O}(mv/\sqrt{k})$ time, the time of this algorithm is dominated by the time to find a P-flow in G', namely $\tilde{O}(knv)$, plus the time to extend it to G, namely $\tilde{O}(mv/\sqrt{k} + m\sqrt{k})$. This gives a total running time of

$$\tilde{O}(mv/\sqrt{k} + m\sqrt{k} + knv).$$

We can arrange for the $m\sqrt{k}$ term to be negligible. Nagamochi and Ibaraki [NI92] give a *sparse certificate algorithm* that reduces the number of edges in G to at most nv without affecting the flow value. If we build the certificate before we run our algorithm, we can assume that $m \leq nv$. Thus we know that $m\sqrt{k} \leq knv$, implying that our overall time bound is

$$\tilde{O}(mv/\sqrt{k} + knv).$$

Setting $k = (m/n)^{2/3}$ gives a bound of $\tilde{O}(m^{2/3}n^{1/3}v)$. A recursive application of the above algorithm described in 7 will yield the $\tilde{O}(\sqrt{mnv})$ time bound.

5 Finding a Legal P-flow.

Our first algorithmic step is finding a legal P-flow. We prove Lemma 4.1. Our approach is to apply the compression scheme of [BK96] described above. However, since we aim to discover what flow values to give to edges crossing P, we do not want to compress the entire graph—a flow in this graph would tell us nothing about flow on edges of P. We therefore take a different approach. We separately compress the subgraph induced by each component of P. This yields a new graph G' that shares with G all the edges that cross partition P. We will show that the resulting graph has few edges and accurately approximates G. We might therefore hope to find a max-flow in this compressed graph and use the values it assigns to P-edges to define a P-flow f_P in G.

Unfortunately, this need not yield a legal P-flow in G. The compression step introduces small errors in cut values and, in particular, may give a component C a larger connectivity in the compressed graph than it has in G. Thus the fact that we can route f_P through C in the compressed graph does not guarantee that we can do so in G. We therefore introduce a *subcompression* scheme that ensures that any flow that routes through a compressed component can also route through the original component. This means that an assignment of flow-values to P-edges in the compressed graph will define a legal P-flow in the original graph. Subcompression is described in the following lemma:

Lemma 5.1. *Given a graph (or in our case a component) C with n_C vertices and m_C edges, and $\epsilon < 1/2$, in $\tilde{O}(m_C)$ time we can build a* compressed graph *C' such that*

- *C' has $\tilde{O}(n_C/\epsilon^2)$ edges*
- *no cut value in C' exceeds the corresponding cut value in C*
- *every cut value in C' is at least $(1-\epsilon)$ times the corresponding cut value in C.*

In other words, we get the same behavior as in the compression scheme of [BK96], but all errors in cut values are underestimates.

Proof. The construction in [BK96] assigns a *compression probability* p_e to each edge e, builds the compressed graph by including each edge with probability p_e, and giving it weight $1/p_e$ if it is included. Observe that every edge therefore has expected value 1 in the compressed graph. The analysis then shows that every cut has value in the range $(1 \pm \epsilon)$ times its expectation. We simply modify the construction to include every edge with probability $(1-\epsilon)p_e$, and give it weight $1/p_e$ if it is included. The analysis goes through unchanged, showing that every cut has value $(1 \pm \epsilon)$ times its expectation. But now the expected value of each edge is $(1-\epsilon)$, meaning that the expected value of each cut is $(1-\epsilon)$ times it original value. Thus, the upper bound on the compressed cut value is $(1+\epsilon)(1-\epsilon) < 1$ times the original value, while the lower bound is $(1-\epsilon)(1-\epsilon) > 1-2\epsilon$ time the original value. The lemma follows if we substitute $\epsilon/2$ for ϵ. □

Lemma 5.2. *Let P be a k-strong partition of a graph G with s-t max-flow v. We can find a legal P-flow of value $(1-\epsilon)v$ in $\tilde{O}(nv(k+1/\epsilon^2))$ time.*

Proof. As a first step, apply the subcompression algorithm of Lemma 5.1 to each component to yield a graph G'. This takes total time $\tilde{O}(\sum m_C) = \tilde{O}(m)$ and produces a graph with $\tilde{O}((\sum n_C)/\epsilon^2) = \tilde{O}(n/\epsilon^2)$ edges inside the components, so that G' has $\tilde{O}(n(k+\epsilon^{-2}))$ edges including the P-edges. It does not increase the value of any component cut and decreases any cut value by at most $(1-\epsilon)$. Since the value of any cut in G is just the sum of the P-edges it cuts and the component edges it cuts, it follows that every cut in G' is at least $(1-\epsilon)$ times the value of the corresponding cut in G. Thus the minimum s-t cut in G' has value at least $(1-\epsilon)v$. We can therefore find a max-flow f' of this value by using $v(1-\epsilon)$ augmenting paths in a time bound of $\tilde{O}(nv(k+1/\epsilon^2))$. This max-flow assigns a flow value to all P-edges; we claim this assignment is in fact a legal P-flow of value $(1-\epsilon)v$.

To see this, consider a component C of P and the corresponding component C' in G'. Build the demand graph C'_{ST} discussed in Section 4.1. This graph has an S-T max-flow of some value r that is equal to the amount of flow that f' routes through C'. It follows that the S-T min-cut in C' has value at least r. Now suppose that we replace C' with C in the demand graph. This gives us the demand graph C_{ST} for the P-flow in G. Recall that by construction, every cut in C' is less than the corresponding cut in C. Thus, C_{ST} has S-T min-cut at least r, meaning that we can find an S-T flow of value r in C_{ST}. This is a C-flow consistent with our P-flow. If we carry out this construction for every component, we get a flow that extends the P-flow and thus has the same value, namely $(1-\epsilon)v$. □

Lemma 4.1 follows by taking $\epsilon = 1/\sqrt{k}$.

6 Extending the P-flow.

The above discussion shows it is relatively easy to find flow values for the cross partition edges. Now we turn to the problem of filling in flow values on the k-strong components. We prove Lemma 4.2.

6.1 Proof overview. As was observed earlier, the obvious thing to try is to use the flow on P-edges to define demands for a given component C, and solve a max-flow problem on the k-connected graph C to satisfy those demands. Since C is k-connected, the maximum flow problem can be solved quickly. The only problem is that the value of the flow may greatly exceed v. The value of the flow we need to compute (by reduction to the demand graph discussed earlier) is equal to the total amount of P-flow entering C. There is no *a priori* reason for this flow to be bounded by v.

To avoid this problem, we introduce techniques to ensure there will always be some component with little incident flow. Our basic algorithm is the following. Until our graph is k connected, we

Fill-in. Find a component of P that has $\tilde{O}(k)$ incident edges, and *fill in* the flow on this component to match the incident P-flow using the algorithm of Theorem 3.4 on the demand graph of the component.

Shortcut. For every vertex v in the component just filled in, *shortcut* v and remove it from the graph.

Repartition. If necessary, repartition G to ensure that some remaining component has degree $\tilde{O}(k)$.

We will show that the Fill-in step is applied at most once to each component of P. Since each component is k-connected and we find a flow of value $\tilde{O}(k)$, the total time will be $\tilde{O}(km/\sqrt{k}) = \tilde{O}(m\sqrt{k})$. We will see that the Shortcut step is applied at most once to each vertex, with work equal to the vertex degree, so that the total time spent is $O(m)$; and that Repartitioning occurs $O(\log n)$ times and takes $\tilde{O}(m)$ time, so that the total time spent on it is $\tilde{O}(m)$. Eventually, we will be left with a k-connected graph, at which point we use the algorithm of Theorem 3.4 to find the s-t flow in $O(mv/\sqrt{k})$ time.

We now describe the steps of the above algorithm in more detail.

6.2 Filling In. The existence of k-strong partitions lets us find components that we can fill in easily.

Lemma 6.1. *Half the components of a k-strong partition have $\tilde{O}(k)$ incident edges.*

Proof. Immediate from the fact that the average degree of a component is $\tilde{O}(k)$. □

Remark. It is this lemma that forced us to use a k-strong partition slightly different from that in [Kar97b].

Clearly, any component with degree $\tilde{O}(k)$ can have incident P-flow at most $\tilde{O}(k)$. It follows that if we are given a k-strong partition, then we can fill in at least half the k-strong components in $\tilde{O}(mk/\sqrt{k}) = \tilde{O}(m\sqrt{k})$ time using the algorithm Theorem 3.4 (for the uncapacitated case; we will address the capacitated case later).

6.3 Shortcutting. Of course, after we fill in a component, it still has degree $\tilde{O}(k)$ but is no longer a useful candidate for filling in. It is also possible that some of the degree-$\tilde{O}(k)$ components of P are singleton vertices, for which filling in is pointless. We need to eliminate such components and vertices in order to ensure that some of the unfilled-in components of the partition will have degree $\tilde{O}(k)$ so that we can fill them in.

To eliminate filled in components from consideration, consider the following *shortcutting* procedure. This procedure can be applied to any vertex at which flow is conserved (that is, that has the same amount of incoming and outgoing flow). Any vertex that is in a singleton component of G will fit this description, since all of its incident edges are P-edges. Similarly, all vertices in a component of the partition that has just been filled in will fit the description. Due to space limitations, we describe the algorithm for an *uncapacitated* graph, in which all edges carry 0 or 1 unit of flow. The generalization to capacitated graphs is straightforward.

To shortcut w, let $x_1, \ldots, x_j$ be the set of vertices for which there is a unit of flow on edge (x_i, w) from x_i to w, and let $y_1, \ldots, y_j$ be the set of vertices for which (w, y_i) is carrying a unit of flow from w to y_i. Delete every edge incident to w that is not carrying flow, and replace the edges (x_i, w) and (w, y_i) with an edge (x_i, y_i) carrying a unit of flow. Call the resulting graph G'. It is simple to see that this transformation isolates vertex w from the graph. However, if we consider the s-t flow corresponding to the P-flow, we see that this shortcutting procedure turns it into an s-t on G'. It follows that the P flow is still legal after we perform the shortcutting step.

Suppose we go on to fill in all the other compressed components in G', and end up with an (arbitrary) flow in G'. We can replace v in the graph, and thus get a flow of the same value in G, with the following simple procedure: replace the edges (x_i, y_i) with v and its original incident edges. Set the flow value in G on (x_i, w) and (w, y_i) equal to the flow value of (x_i, y_i) in G'. Clearly this preserves flow conservation.

6.4 Repartitioning. As we shortcut vertices, we eliminate some of the k-strong components in the k-strong partition P. Eventually, there might not be any low-degree components left to fill in and shortcut. We can fix this problem by repartitioning the graph. Temporarily contract each component of P to a single vertex, and find a sparse k-strong partition in the resulting graph. Now expand each contracted vertex. This gives us a k-strong partition P' of which P is a refinement. In particular, every P'-edge has a flow assignment corresponding to a legal flow, so we clearly have a legal P'-flow. Some P-edges may end up inside components of P', but we can just ignore their flow values.

6.5 Combining the Techniques. Combining the three techniques just described will let us solve the problem of converting a P-flow into a flow on G. We describe the algorithm for doing so recursively. The input is a graph with a k-strong partition P and a P-flow. We carry out the following steps.

First, we identify all components with degree $\tilde{O}(k)$. We fill in all such components using the algorithm of Theorem 3.4. If this is all the components, we are done and can return the result. Otherwise, we shortcut every vertex in the components we just filled in, as well as any singleton components of P. Then, we repartition P, yielding a k-strong partition P', and recursively convert P' to a flow. When we get back the flow from this recursive call, we patch in all the vertices we shortcutted (as described above) and return the resulting flow.

To bound the running time of this recursive algorithm, note that one recursive level takes $\tilde{O}(m\sqrt{k})$ time for filling in components (as discussed in Section 6.2) plus $\tilde{O}(m)$ time to shortcut and repartition the graph. Also note that the partition P' has only half as many components as P: this follows from the fact that we eliminate half the components of P by uncompressing and shortcutting them, along with the fact that P is a refinement of P'. We can therefore describe a recurrence in terms of the number r of components of P:

$$\begin{aligned} T(1) &= \tilde{O}(mv/\sqrt{k} + \sqrt{mn}v) \\ T(r) &\leq \tilde{O}(m\sqrt{k}) + T(r/2) \end{aligned}$$

The base case reflects the time to find a flow of value v in a k-connected graph using Theorem 3.2. Solving this recurrence proves the result claimed in Lemma 4.2.

7 A Recursive Algorithm.

Since finding the P-flow involves a maximum flow computation, it is natural to try to compute that maximum flow recursively in order to improve our time bounds. Consider the following algorithm. The input is a capacitated graph G with m edges. We use two previous observations: (1) for any k, we can find a k-strong partition with at most $akn \log n$ edges for some constant a, and (ii) for any ϵ, we can subcompress any n-vertex graph to have $bn(\log n)/\epsilon^2$ edges and cuts accurate to within ϵ.

1. Choose a minimum ϵ and maximum $k = 1/\epsilon^2$ for which $akn \log n \leq m/4$ and $bn(\log n)/\epsilon^2 \leq m/4$.
2. If the graph is k-connected, use the capacitated graph algorithm of Theorem 3.2 to find a flow in it and return that flow. Otherwise,
3. Find a k-strong partition P of G that has at most $m/4$ P-edges.
4. Subcompress all components of the partition with error ϵ so they have at most $m/4$ edges in total.
5. Recursively find a maximum flow (of value at most v) in the resulting $m/2$-edge graph. This gives a P-flow.
6. Extend the P-flow to G using the algorithm of Section 6.
7. Augment the resulting flow in G to a maximum flow.

Note that this recursive algorithm is designed for *capacitated* graphs. This is necessary, even if we initially aim to solve an uncapacitated graph, because subcompressing an uncapacitated graph produces a capacitated graph which is then solved recursively. We must make use of the capacitated graph smoothing techniques developed above in order to perform Step 6 in these capacitated graphs.

To determine the running time of this algorithm, note that the constraints on k and ϵ let us take $k = \tilde{\Omega}(m/n)$ and $\epsilon = \tilde{O}(\sqrt{n/m})$. Note also that as before, by applying Nagamochi and Ibaraki's sparse certificate algorithm first, we can assume that $m/n \leq v$. Therefore, when we apply the algorithm of Section 3 to find flows (of value $\tilde{O}(k)$) in the components (of connectivity k) in Step 6, the running time is

$$\begin{aligned} \tilde{O}(\sqrt{mn}k + mk/\sqrt{k}) &= \tilde{O}(\sqrt{mn}k + m\sqrt{k}) \\ &= \tilde{O}(\sqrt{mn}v + m\sqrt{m/n}) \end{aligned}$$

Finally, Step 7 performs at most ϵv augmenting paths, at a total cost of $\epsilon mv = \tilde{O}(\sqrt{mn}v)$. Therefore, the total running time of the algorithm is given by the recurrence

$$\begin{aligned} T(m) &= T(m/2) + \tilde{O}(\sqrt{mn}v + m\sqrt{m/n}) \\ &= \tilde{O}(\sqrt{mn}v + m\sqrt{m/n}) \end{aligned}$$

To simplify this bound, we again use the fact that, via Nagamochi and Ibaraki's sparse certificate, we can assume $m < nv$. This implies that $v > m/n$, which implies in turn that $\sqrt{mn}v > \sqrt{mn}m/n = m\sqrt{m/n}$, which means that the second term above is dominated by the first. Thus, our running time simplifies to $\tilde{O}(\sqrt{mn}v)$, exactly as claimed.

8 Conclusion.

We have given new algorithms for undirected graphs that find flows faster than traditional methods for uncapacitated graphs. The natural question is how much further these techniques can go. Since compression essentially lets us reduce the number of edges in G to $\tilde{O}(n)$, it seems plausible to argue that a time bound of $O(m + nv)$ might be achieved. More ambitiously, we might see if any of the techniques described here lead to fast strongly polynomial flow algorithms. The appropriate starting point appears to by the strongly polynomial $\tilde{O}(m\sqrt{n}/\epsilon)$ time bound for finding a $(1-\epsilon)$ times maximum flow.

References

[ACM96] ACM. *Proceedings of the 28^{th} ACM Symposium on Theory of Computing*. ACM Press, May 1996.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[BK96] András A. Benczúr and David R. Karger. Approximate s–t min-cuts in $\tilde{O}(n^2)$ time. In *Proceedings of the 28^{th} ACM Symposium on Theory of Computing* [ACM96], pages 47–55.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, Potomac, MD, 1979.

[GR97a] Andrew Goldberg and Satish Rao. Beyond the flow decomposition barrier. In *Proceedings of the $3*^{th}$ Annual Symposium on the Foundations of Computer Science* [IEE97], pages 2–11.

[GR97b] Andrew Goldberg and Satish Rao. Flows in undirected unit capacity networks. In *Proceedings of the $3*^{th}$ Annual Symposium on the Foundations of Computer Science* [IEE97], pages 32–35.

[IEE97] IEEE. *Proceedings of the $3*^{th}$ Annual Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, October 1997.

[Kar96] David R. Karger. Minimum cuts in near-linear time. In *Proceedings of the 28^{th} ACM Symposium on Theory of Computing* [ACM96], pages 56–63.

[Kar97a] David R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 1997. To appear.

[Kar97b] David R. Karger. Using random sampling to find maximum flows in uncapacitated undirected graphs. In *Proceedings of the 29^{th} ACM Symposium on Theory of Computing*, pages 240–249. ACM, ACM Press, May 1997.

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