SIAM J. COMPUT.
Vol. 44, No. 2, pp. 320–339

FAST AUGMENTING PATHS BY RANDOM SAMPLING FROM RESIDUAL GRAPHS[*]

DAVID R. KARGER[†] AND MATTHEW S. LEVINE[‡]

David Karger wishes to dedicate this work to the memory of Rajeev Motwani. His compelling teaching and supportive advising inspired and enabled the line of research [12, 19, 13, 16, 20] *that led to the results published here.*

Abstract. Consider an n-vertex, m-edge, undirected graph with integral capacities and max-flow value v. We give a new $\tilde{O}(m + nv)$-time maximum flow algorithm. After assigning certain special sampling probabilities to edges in $\tilde{O}(m)$ time, our algorithm is very simple: repeatedly find an augmenting path in a random sample of edges from the residual graph. Breaking from past work, we demonstrate that we can benefit by random sampling from *directed* (residual) graphs. We also slightly improve an algorithm for approximating flows of arbitrary value, finding a flow of value $(1-\epsilon)$ times the maximum in $\tilde{O}(m\sqrt{n/\epsilon})$ time.

Key words. minimum cut, maximum flow random graph, random sampling, connectivity, cut enumeration, network reliability

AMS subject classifications. 05C21, 05C40, 05C80, 68W25, 68W40, 68Q25, 05C85

DOI. 10.1137/070705994

1. Introduction.

In this paper we consider the problem of finding maximum flows in undirected graphs with small integral edge capacities. We give an extremely simple algorithm that finds a maximum flow of value v in $O(m+nv)$ time by repeatedly sampling from and finding augmenting paths in the residual graph; our approach gives the first demonstration of the benefits for flow problems of random sampling from *directed* (residual) graphs. Our techniques also offer some small improvement to approximate max-flow algorithms in arbitrary-capacity graphs.

1.1. Background.

The study of algorithms for graphs with small integer capacities is as old as the study of the general maximum flow problem. In fact, the original $O(mv)$-time Ford–Fulkerson algorithm [6] is still the best known and most efficient deterministic algorithm for sparse graphs with sufficiently small flows. Here m is the number of edges, v is the value of the maximum flow, and n is the number of nodes. The algorithm works by repeatedly finding *augmenting paths* from source to sink in the *residual graph* defined by an existing, nonmaximal flow and adding flow along those paths in order to increase the value of the existing flow. Finding one augmenting path takes time $O(m)$ and increases the flow value by one; the overall $O(mv)$-time bound follows.

The dominant factor in the Ford–Fulkerson algorithm is the linear-time search for an augmenting path. Once the path is found, actually modifying the flow along its (at most n) edges is relatively quick. In one attempt to take advantage of this difference,

[*]Received by the editors October 21, 2007; accepted for publication (in revised form) December 13, 2013; published electronically March 31, 2015. A preliminary version of this work appeared previously as [18].
http://www.siam.org/journals/sicomp/44-2/70599.html

[†]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge MA 02139 (karger@mit.edu, http://people.csail.mit.edu/∼karger). The work of this author was supported in part by a grant from the National Science Foundation.
[‡]Akamai Technologies, 8 Cambridge Center, Cambridge, MA 02139 (mslevine@akamai.com).

TABLE 1
Summary of algorithms. The long history of $\tilde{\Omega}(mn)$-time algorithms, which are still best for large v, has been compressed.

Source	Year	Time bound	Capacities	Directed	Determ.
Ford–Fulkerson [6]	1956	$O(mv)$	$\surd$	$\surd$	$\surd$
Even–Tarjan [5]	1975	$O(mn^{2/3})$		$\surd$	$\surd$
		$O(m^{3/2})$		$\surd$	$\surd$
Karger [14]	1997	$\tilde{O}(m^{2/3}n^{1/3}v)$			
Goldberg–Rao [9]	1997	$\tilde{O}(mn^{2/3}\log v)$	$\surd$	$\surd$	$\surd$
		$\tilde{O}(m^{3/2}\log v)$	$\surd$	$\surd$	$\surd$
Goldberg–Rao [8]	1997	$O(n\sqrt{nm})$			$\surd$
Karger [15]	1998	$\tilde{O}(v\sqrt{nm})$	$\surd$		
Karger–Levine [17]	1998	$O(nm^{2/3}v^{1/6})$			$\surd$
		$\tilde{O}(m+nv^{3/2})$	$\surd$		$\surd$
		$\tilde{O}(m+nv^{5/4})$	$\surd$		
		$\tilde{O}(m+n^{11/9}v)$	$\surd$		
Karger–Levine [18] (detailed here)	2001	$\tilde{O}(m+nv)$	$\surd$		
Kelner et al. [21], Sherman [26]	2013	$\tilde{O}(mv^{2/3})$	$\surd$		
Lee–Rao–Srivastava [22]	2013	$\tilde{O}(m^{5/4}v^{1/4})$			
Madry [24]	2013	$\tilde{O}(m^{10/7})$		$\surd$	
Lee–Sidford [23]	2014	$\tilde{O}(m\sqrt{n}\log^2 U)$	$\surd$	$\surd$	

the technique of *blocking flows* [4] was developed; it uses a single linear-time scan to find and augment multiple paths simultaneously, thus amortizing the cost of the scans. This approach yields faster algorithms for moderately large flow values [27, 1]. However, in the worst case for small flows it might find only one augmenting path path per blocking phase.

Karger [13] developed a different approach for *undirected* graphs. He showed that random samples from the edges of an undirected graph have nice connectivity properties, so that most of the necessary augmenting paths can be found by scanning small random subsets of the edges in $o(m)$ time. Applying this idea in a series of increasingly complex algorithms yielded a sequence of peculiar time bounds: $\tilde{O}(mv/\sqrt{c})$-time for graphs with global minimum cut c [13], then $\tilde{O}(m^{2/3}n^{1/3}v)$-time for simple (unit-capacity, without parallel edges) graphs [14], then $\tilde{O}(m^{1/2}n^{1/2}v)$-time [15]. Later Karger and Levine [17] achieved time bounds of $\tilde{O}(m+nv^{5/4})$ and $\tilde{O}(m+n^{11/9}v)$. See Table 1 for a history of algorithms for small flow values.

In order to show which algorithms have the best performance for different values of m and v relative to n, we have drawn figures: one for deterministic algorithms only (Figure 1) and one including randomized algorithms (Figure 2). A point in the figure represents the value of m and v relative to n. Specifically, (a, b) represents $v = n^a$, $m = n^b$. Each region is labeled by the best time bound that applies for values of m and v in that region. Note that the region $m > nv$ is uninteresting because the sparsification algorithm of Nagamochi and Ibaraki [25] can always be used to make $m \leq nv$ in $O(m)$ time. The shaded region in Figure 2 corresponds to the algorithm given in this paper.

In Figure 1, note that the $O(nm^{2/3}v^{1/6})$-time deterministic algorithm (which is the fastest algorithm for the region surrounded by a dashed line) is the only determin-

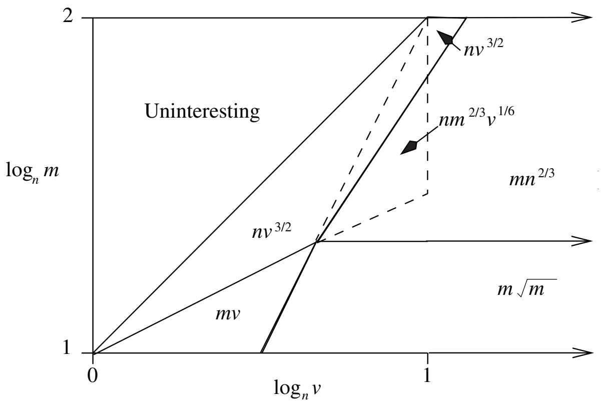

FIG. 1. *Pictures of the best deterministic bounds (see text for explanation).*

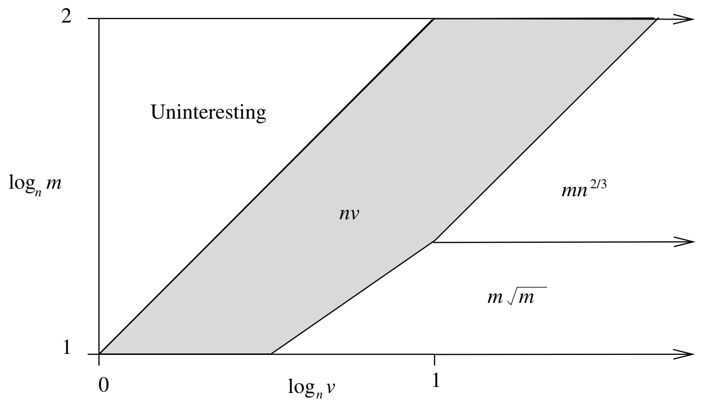

FIG. 2. *Pictures of the best randomized bounds (see text for explanation).*

istic one in the picture that cannot handle capacities or parallel edges. If capacities are being considered, then this algorithm should be removed from the picture; if only simple graphs are being considered, then the picture should end at $v = n$. In a similar vein, in Figure 1 dashed lines outline areas covered by two randomized algorithms [24, 22] that apply only to unit-capacity graphs.

The complexity of these diagrams and recent significant improvements in max-flow suggests that more progress can yet be made.

1.2. Our contribution. Our work closes a chapter in this line of research on sampling, finally achieving a natural $\tilde{O}(m + nv)$-time bound, or amortized $\tilde{O}(n)$-time per augmenting path. Since a flow path can have as many as $n - 1$ edges whose flow value must be updated, this is arguably the best result (up to logarithmic factors) that one can hope to achieve by edge sparsification alone.[1]

Our key advance is taking random samples of the edges of *residual* graphs. All the previous work built on the fact that many augmenting paths could be found by taking disjoint random samples from the original undirected graph edges. But once these

[1] On the other hand, it has been shown [17] that there exists a solution in which most augmenting paths are short.

paths are found and a nonzero flow established, we end up with a residual graph that is directed, so we can no longer apply undirected graph-sampling results. This paper shows that a residual graph remains similar enough to the original undirected graph to let sampling be applied iteratively. With the exception of computing an "importance" measure known as *edge strength* [2] in near-linear time at the beginning, our algorithm is just this simple iteration: sample edges according to their strengths and find an augmenting path, repeating until done.

Our approach has several additional applications. The factor of m represents a preprocessing step that need be applied only once; afterwards, any max-flow calculation runs in time $\tilde{O}(nv)$. Using this fact, we show that a Gomory–Hu tree representing all s-t min-cuts in a graph, which can be constructed using n max-flow computations, can be built using our algorithm in $\tilde{O}(mn)$ time on unweighted undirected graphs (the Gomory–Hu algorithm computes a sequence of flows of value v_i such that $\sum v_i = O(m)$; we show how using our flow algorithm takes time $O(m + \sum nv_i) = O(nm)$). We also generalize the Benczúr–Karger *smoothing* technique to residual graphs. This lets us find a flow of value $(1-\epsilon)$ times the maximum in $\tilde{O}(m\sqrt{n/\epsilon})$ on graphs with *arbitrary* edge capacities, improving on the Benczúr–Karger [2] result of $\tilde{O}(m\sqrt{n}/\epsilon)$. That in turn lets us slightly improve the ϵ-dependence for s-t min-cut approximation as well.

1.3. Outline. The rest of this paper is organized as follows. In section 2 we review some notation and basic definitions. In section 3 we give a simple algorithm (with a worse time bound) that demonstrates our key idea of sampling from residual graphs. In section 4 we incorporate the idea of nonuniform sampling based on *edge strengths* [2] and give the main algorithm and its running time analysis. In section 5 we prove the key sampling theorem on which the algorithm depends. In section 6 we generalize this approach to compute *approximate* max-flows on *capacitated* graphs. We conclude and discuss some open questions in section 7.

A preliminary version of this work was published previously [18]; here we flesh out details and intuition, give cleaner proofs, and add results on approximate max-flow.

2. Notation and definitions.

We use the following notation:

G	the graph
v	the value of a maximum flow
n	the number of nodes
m	the number of (undirected) edges
s	the source
t	the sink
f	a flow or its value, as determined by context
G_f	the residual graph of G with respect to f
e	an edge
u_e or $u(e)$	the capacity of edge e
$u_f(e)$	the residual capacity of edge e given flow f
k_e	the strength of edge e (defined below)
X	the set of edges that cross a given cut

It is common to define a cut as a nonempty proper subset of the vertices. For convenience, in our proofs we often use a shorthand of referring to a cut by the set of edges that cross the cut, for which we have used the symbol X. In particular, in several cases where we wish to sum over the edges that cross a cut, we write $\sum_{e \in X}$.

Note that the graph has m *undirected* edges, but that each undirected edge represents two directed ones.

For graphs G with edge weights x_e and H with edge weights y_e and scalar value α, we define $H + \alpha G$ to be the graph with edge weights $y_e + \alpha x_e$.

We say that an event occurs *with high probability* if the probability that it does not occur (when we are considering graphs of size n) is inverse polynomial in n; in general, the polynomial can be made as large as we like by appropriate choice of constants in the algorithm.

We recall the following classical result.

Lemma 2.1. *Given a graph G and s-t flow f, every s-t cut in G_f has capacity (from the s side to the t side) exactly f less than its value in G.*

Proof. The proof is immediate by flow conservation [27]. ∎

For our proofs, it will be helpful to consider relative rather than additive changes in cut values, as shown next.

Corollary 2.2. *If G has minimum s-t cut v, then every s-t cut in G_f has residual capacity at least* $(v - f)/v$ times *its value in G.*

Proof. If the original cut value was v', then, by Lemma 2.1, its residual value is $v' - f = v'(v' - f)/v' \geq v'(v - f)/v$ since $v' \geq v$. ∎

3. A simple algorithm.

We begin with a simple algorithm that demonstrates sampling from a residual graph. Karger [13] used undirected-graph sampling for an algorithm with running time $\tilde{O}(mv/\sqrt{c})$ in graphs with global minimum cut c; we show that by applying sampling to residual graphs we can improve this running time to $\tilde{O}(mv/c)$. To analyze this algorithm, we need a slight variant of the Reliability Lemma (Lemma 3.5) from [2].

Reliability Lemma 3.1 (after [2]). *Let G be an n-vertex weighted undirected graph, and let $c_1, c_2, \ldots$ be a list of the values of all the cuts, with $c = \min c_i$ the minimum cut. If $\alpha^{-c} \leq n^{-2}$, then*

$$\sum_i \alpha^{-c_i} = O(n^2 \alpha^{-c}).$$

Proof. This is a restatement of the reliability lemma from [2] with all $p_e = \alpha$, since in that case $\max_C \prod_{e \in C} p_e = \alpha^{-c}$. ∎

We now use the above lemma to prove a sampling theorem for residual graphs.

Theorem 3.2. *If G is a c-connected unweighted graph, f is a flow, and $(6mv/c)(\ln nv)/(v - f)$ edges of G_f are chosen at random, then with high probability there is an augmenting path in the sample.*

For simplicity, the theorem assumes sampling with replacement. Of course, a duplicate sample has no impact on whether the sampled edge set has an augmenting path.

Proof. There is an augmenting path in the sample if and only if every s-t cut has a sampled residual edge crossing it (from the s side to the t side). We show that the probability of any s-t cut *not* having an edge sampled is small. Consider any s-t cut of original value r. By Corollary 2.2 the cut has capacity $r(v - f)/v$ in G_f. The residual graph has at most $2m$ directed edges, so for each sampled edge the probability that we choose one of the cut edges is at least $r(v - f)/2mv$. With the given number of samples, the probability that no edge of the cut is sampled is therefore

$$\begin{aligned}\left(1 - \frac{r(v-f)}{2mv}\right)^{(6mv/c)(\ln nv)/(v-f)} &\leq \exp\left(\frac{-3r(\ln nv)}{c}\right) \\ &= (nv)^{-3r/c}.\end{aligned}$$

The sum of this quantity over all s-t cuts union-bounds the probability that any cut is empty. But that sum has the form of Reliability Lemma 3.1, with $\alpha \geq (nv)^{3/c}$. This means $\alpha^{-c} = (nv)^{-3}$, so the conditions of Lemma 3.1 apply. We conclude that the sum is $O(n^2\alpha^{-c}) = O((nv)^{-1})$, meaning that no cut is empty with high probability. ∎

Suppose for now that c and v are known. Consider the following algorithm:

```
initially $f$ is the empty flow
**repeat** $v$ times
    sample $(6mv/c)(\ln nv)/(v-f)$ edges from $G_f$
    find an augmenting path in the sample and augment it
```

Theorem 3.2 proves that this algorithm is correct with high probability: in each iteration, an augmenting path will exist in the sample with probability $1-(nv)^{-1}$, so after v iterations we will reach $f = v$ with probability $1 - 1/n$.

Now we turn to the running time. If we store the edges in an array, it is straightforward to choose a random subset of them by probing random positions in the array, taking $O(1)$ time per sampled edge. Note that, as we augment, certain edges may "reverse direction" in the residual graph, but they can continue to occupy the same array locations for sampling. Using this sampling scheme, the time to sample and the time to compute (and augment) an augmenting path are both linear in the number of sampled edges. With this implementation, the running time of the algorithm becomes

$$\begin{aligned} O\left(\sum_{f=0}^{v-1}(mv/c)(\ln nv)/(v-f)\right) &= O\left((mv/c)(\ln nv)\sum_{f=0}^{v-1}1/(v-f)\right) \\ &= O(mv(\ln nv)(\ln v)/c) \\ &= O(mv(\log^2 nv)/c). \end{aligned}$$

The algorithm just given assumes a knowledge of v and c, but that is not needed. The number of edges that needs to be sampled in each iteration is clearly between n and m. Start by sampling n edges, and then repeatedly double the sample size until an augmenting path is found. Theorem 3.2 asserts that this will happen with high probability as soon as the sample size exceeds $(6v/c)(\ln nv)/(v-f)$, which means it will be at most twice that. Since sampling and augmenting takes time proportional to the number of sampled edges, this repeated doubling work is dominated by the final stage (with the "correct" sampling rate), whose running time has already been accounted for in the previous analysis.

4. A faster algorithm with nonuniform sampling.

The simple algorithm above used residual-graph sampling to improve max-flow computation from $\tilde{O}(mv/\sqrt{c})$ [13] to $\tilde{O}(mv/c)$. However, it still exhibits a dependence on c—if m is large and c is small, the running time is large. Benczúr and Karger [2] showed how to fix this problem, effectively replacing m/c by n. We combine that approach with our residual sampling ideas to get an $\tilde{O}(m+nv)$ running time for our exact algorithm.

Definition 4.1. *The* strength *of an edge e, denoted k_e, is the maximum value of k such that a k-edge-connected vertex-induced subgraph of G contains e. We say e is k-*strong *if its strength is k or more, and k-*weak *otherwise. The strength of a residual-graph edge is the strength of the underlying undirected edge of which it is a part. The connected components induced by the k-strong edges are the k-*strong components.

Benczúr and Karger [2] give an algorithm for estimating edge strengths in $O(m \log^2 n)$ time on unweighted graphs (and $O(m \log^3 n)$ given arbitrary graph weights). More precisely, they compute approximate lower bounds k'_e on those strengths that satisfy the key properties we will need.

Our algorithm is easy to state. With the possible exception of the first step, it ought to be correspondingly easy to implement. It is of course difficult to know whether it will perform well in practice without actually implementing it.

```
1. run the Benczúr–Karger algorithm [2] to compute good lower bounds
   $k'_e$ on the edge strengths.
2. $\alpha = 1$.
3. while $\alpha n < m$,
   (a) sampling according to weights $u_e/k'_e$, choose $\alpha n$ edges from
       $G_f$ (with replacement, but ignoring duplicates).
   (b) find and augment an augmenting path among the sampled edges
       (which is also an augmenting path in $G$).
   (c) if no path is found, double $\alpha$.
4. find and augment paths in the entire residual graph until none remain.
```

The correctness of the algorithm is obviously guaranteed by the last step. Analysis of the running time is not obvious; it is the subject of the remainder of the section.

4.1. How to sample.

4.1.1. Cumulative sums.

We must fill in some details on how to sample. Consider a set of m items, each with a weight w_e, with weights summing to a total W. We wish to choose a single item e with probability w_e/W. Order the items arbitrarily. Compute cumulative sums of prefixes of the order, producing $m+1$ cumulative sums. An item can be sampled by generating a random number in the range $[0, W]$ and finding its position between two adjacent values in the cumulative sum array. The most obvious way to find the position is through binary search; this selects an item in $O(\log m)$ time.

A faster sampling method precomputes the positions in the array of m evenly spaced *marker* values kW/m for $k = 1, \dots, m$ (i.e., the two array indices between which each of the marker values falls) and stores the corresponding indices in a separate size-m array. The markers divide the number range $[0, W]$ into m equal-sized *marker intervals* of length W/m that contain the cumulative sums; each marker interval is subdivided into smaller subintervals by the cumulative sums it contains. We can choose a random edge by first choosing a marker interval uniformly at random and then choosing a random offset uniformly from the range $[0, W/m]$ within the interval and locating the resulting quantity among the cumulative sums inside the selected interval. Assuming we can generate a random number in constant time (or using the rounding technique of the first method), we can choose a marker interval in constant time using the secondary array and then find a subinterval within it using binary search in time proportional to the log of the number of subintervals in that marker interval. Since there are m marker intervals and m cumulative sums, each interval contains on average one cumulative sum. So finding the right subinterval of an interval takes $O(1)$ time in expectation. Since the maximum time for a single sample is $O(\log m)$, a standard Chernoff bound shows that, over the large number of trials in this algorithm (at least n), the time will average to $O(1)$ per sample with high probability.[2]

4.1.2. Numerical precision. One might be concerned about the numerical computations involved in summing fractional edge weights. To address this, scale all the edge weights by $nm^2(m+nv)/\max k'_e$ so that the sum $W > nm^2(m+nv)$. Then round all fractions to the nearest integer. Now summing weights and sampling is straightforward, and we have introduced a perturbation of at most m to any sum. If we imagine the weights defining adjacent lengths on the real line, then these perturbations move the boundaries between the lengths. A particular sampling outcome is changed by these perturbations only if one of the boundaries is perturbed across the random number that defines the sample. Since there are only m boundaries and each is perturbed by at most m, only m^2 total length is traversed by the perturbed boundaries, so the probability that a particular sample hits some of this length is at most $m^2/nm^2(m+nv) = 1/n(m+nv)$.

We will soon prove that the algorithm takes only $\tilde{O}(m+nv)$ time. Thus the probability that any of the $\tilde{O}(m+nv)$ samples taken during the execution of the algorithm is different than it would have been using exact arithmetic is $\tilde{O}(1/n)$, so this possibility can be ignored.

We can reduce the probability of a "faulty" sample further, to any polynomially small value, by increasing the scaling factor by a corresponding polynomial amount.

4.1.3. Sampling the residual graph. Our algorithm needs to choose *residual* graph edges according to weights u_e/k_e. This would not seem to fit the algorithm just described, since the set of residual edges changes each time. Note, however, that when we augment flow through an edge, we simply move some of the capacity from an edge to its reverse edge. Thus, the total capacity on the two directions of an edge, which we can associate with the undirected edge between the same endpoints, does not change. Similarly, the quantity $\sum u_e/k_e$, which serves as W, does not change. We can therefore use the above scheme to choose from among the *undirected* edges; we then choose a direction for that edge by flipping a coin biased according to the current residual capacity in the two directions along that edge. Alternatively, when we select an undirected edge, we may simply include both directions of it in the sample—this only doubles the number of edges taken, and only increases the probability of any edge being chosen, so our runtime analysis still applies.

4.2. The runtime analysis.

The foundation of our analysis is the following theorem, which extends the main theorem of Benczúr and Karger [2] to residual graphs.

THEOREM 4.2. *Consider graph G with s-t max-flow v and some s-t flow f. Let $\beta = \frac{32v\ln nv}{v-f}$. Given the estimated edge strengths k'_e, if a sample of βn residual-graph edges are chosen according to weights u_e/k'_e, then with high probability there is an augmenting path in the sample.*

This result holds whether we sample with or without replacement. In our algorithm for simplicity we sample with replacement. This means we may get the same edge multiple times. But we ignore duplicate samples without affecting the presence of an augmenting path.

We will prove the theorem in section 5. Given the theorem, we add up the times

[2]More generally, one can prove that the number of (random) bits needed to sample from a distribution is equal to the entropy of this distribution in expectation; since our distribution has only m elements, its entropy is at most $\log m$; i.e., a single random "machine word" suffices to choose one edge.

for each step of our algorithm to get the total running time. The first step, computing edge strengths, requires $O(m \log^2 n)$ time, or $O(m \log^3 n)$ time if the input graph has superpolynomial capacities [2]. Then, in each iteration of the loop we need to select αn edges and search for an augmenting path.

As discussed in section 4.1, the time to sample edges will, after $O(m)$ preprocessing, average to $O(1)$ per edge with high probability. Thus, the time to sample as well as the time to seek an augmenting path is linear in the number of sampled edges. The quantity α is initially small and is doubled when we fail to find an augmenting path. Theorem 4.2 asserts that with high probability this doubling will happen only when $\alpha < \beta$ from Theorem 4.2. It follows that in each augmentation, $\alpha < 2\beta$. Thus, sampling and searching for an augmenting path takes $O(\alpha n) = O(\beta n)$ time. Since the amount of remaining flow is $v - f$, we can spend at most $O(\beta n(v - f)) = O(nv \log nv)$ time before we run out of augmenting paths and double α. We can double α only $\lg(m/n)$ times before $\alpha = m/n$ and we move to the final cleanup step of the algorithm, so the total time for the loop is $O(nv \log nv \log m/n)$. When $\alpha = m/n$, it must be the case that $\beta > m/2n$ so $v/(v - f) = \Omega(m/(n \log nv))$. In other words, $v - f = O((nv \log nv)/m)$, which means that the time for the cleanup step is only $O(m(v - f)) = O(nv \log nv)$. Therefore, the total running time is $O((m + nv) \log^2 nv)$ for unweighted graphs (with an additional $O(m \log^3 n)$ term for weighted graphs).

Note that, so long as there is a residual path ($f \leq v - 1$), we have $\beta \leq 16v \ln nv$. Thus, when sampling $16nv \ln nv$ edges, we will find an augmenting path with high probability. However, with the Nagamochi–Ibaraki preprocessing step [25], we can always arrange for $m \leq nv$, so we will never get to this high a sampling rate in our algorithm.

4.3. Algorithm variants. The algorithm as stated above is the cleanest to analyze; however, several variants of the algorithm can be considered that have interesting implications or may work better in practice. For those interested only in the $\tilde{O}(m + nv)$ bound, this section can be skipped.

4.3.1. Lazy sampling. The original algorithm explicitly maintains a sampling rate α, but this is not necessary. Consider the algorithm in Figure 3. This algorithm chooses exactly as many random edges as are needed to find an augmenting path. It can be seen as a lazy version of the previous algorithm that reveals the sampled edges one at a time and terminates the process as soon as the path is found. Therefore, it chooses no more edges than are chosen by the previous version of the algorithm, so the previous bound on edges sampled applies.

Since this variant just samples edges until an augmenting path is found, there is no need to explicitly maintain the quantity α that controls the size of the samples—the algorithm can simply keep augmenting until an augmentation step samples all edges without finding an augmenting path.

As for runtime, note that each vertex gets marked and each edge scanned when it can be reached from s. When t gets marked, the marks can be traced backward (since each node is marked with its predecessor) to find the augmenting path. Since each sampled edge is scanned only once, the total time is linear in the number of sampled edges. The runtime bound of the previous algorithm therefore applies here. In practice, this lazy version might check fewer edges and outperform the previous one.

4.3.2. Local termination. Alternatively, we can effect a more local search. In each iteration, perform a standard augmenting path search. But each time we

```
PROCEDURE scan(vertex $u$, vertex $v$, graph $H$).
    if $v$ is marked
        return
    else
        mark $v$ with $u$
        for each $(v, w)$ in $H$
            scan($v$,$w$,$H$)

PROCEDURE augment($G_f$).
    make an empty graph $H$ on the vertices of $G_f$
    unmark all vertices
    mark $s$
    repeat until $t$ is marked or all edges of $G_f$ are sampled
        choose a random edge $(v, w)$ from $G_f$ (biased according to $u_e/k'_e$)
        add $(v, w)$ to $H$
        if $v$ is marked then
            scan($v$, $w$, $H$)
```

FIG. 3. *A lazy sample-and-augment algorithm.*

encounter a new vertex, we flip coins to decide which of its outgoing edges are part of the current iteration's sample. We then limit the continuation of the augmenting path search to those edges that were chosen. Looked at this way, we can think of our algorithm as nothing more than an augmenting path search that chooses random outgoing neighbors and applies "early termination" of the search at each vertex. This will waste less time generating samples for vertices we don't visit.

4.3.3. Nonadaptive sampling. As was discussed in the original algorithm, we sample directed edges by sampling their underlying *undirected* edges and then choosing a direction based on the residual capacity in each direction. The u_e/k'_e sampling probabilities of the undirected edges do not change in the residual graphs. Thus, if we take *both* directions, which only doubles the sampled graph size, then our sample becomes independent of the particular residual graph and can be chosen without regard to it. Thus, one can plan out the sampling schedule (which edges are included in which iteration) before doing any of the augmentations. And this schedule can remain fixed even if multiple distinct max-flows are computed, as in the Gomory–Hu tree algorithm below.

4.4. Gomory–Hu trees. A particular application of our time bound is to Gomory–Hu trees. The Gomory-Hu tree [10] is a single tree that represents all $\binom{n}{2}$ terminal-pair min-cuts in a graph—each edge in the tree is given a weight, and the maximum flow from u to v is equal to the minimum weight on the path from u to v in the tree. A Gomory–Hu tree always exists and can be found by performing n max-flow calculations in the graph. Our algorithm can be applied to those max-flow calculations. Since on simple graphs the maximum flow is bounded by the maximum degree n, a bound of $\tilde{O}(m + n^3)$ is immediate for computing the n max-flows for such graphs. But an observation of Hariharan et al. [11] shows a stronger result, as follows.

LEMMA 4.3 (see [11]). *The sum of edge weights in the Gomory–Hu tree of an unweighted graph is at most* $2m$.

Proof. Root the Gomory–Hu tree anywhere. Observe that the edge from vertex u to its parent v has weight equal to the maximum flow from u to v, which is upper bounded by the degree of u. It follows that the sum of edge weights is upper bounded by the sum of vertex degrees, which is at most $2m$. ∎

COROLLARY 4.4. *In unweighted undirected graphs, a Gomory–Hu tree can be built in $\tilde{O}(mn)$ time with high probability.*

Proof. The Gomory–Hu tree construction algorithm computes a series of maximum flows in the graph. Each maximum flow computation discovers an edge of the Gomory–Hu tree T whose weight is equal to the value of the maximum flow computed. The total time to compute all maximum flows using our algorithm can be broken into the time to compute edge strengths in each of the n flow calculations, which is $\tilde{O}(mn)$, plus the time to use our algorithm to find each max-flow, which is $\tilde{O}(n \sum_{e \in T} w_e)$, where w_e is the weight of edge e in T. The previous lemma shows that $\sum w_e = O(m)$, and the result follows. ∎

We can simplify this algorithm (and shave some logarithmic factors) by observing that edge strengths need be computed only once, at the beginning of the algorithm, and can then be used in all phases. This is not immediate because the Gomory–Hu algorithm, each time it finds an edge of the Gomory–Hu tree, *contracts* the two endpoints of that edge into a single vertex. This can change edge strengths. Fortunately, contraction never decreases edge strengths (cf. [2]). Thus, the original k'_e are still valid-strength lower bounds, which is sufficient for use in sampling (as will be discussed in section 5).

Hariharan et al. [11] achieve the same $\tilde{O}(mn)$ time bound; however, their construction is based upon a Steiner connectivity calculation that uses tree packing. Our algorithm uses the traditional sequence-of-max-flows method, simply accelerating each max-flow calculation.

5. Proof of the residual sampling theorem.

It remains to prove Theorem 4.2. This is essentially a combination of the compression theorem from Benczúr and Karger [2, Compression Theorem 6.2] (on sampling *nonuniformly* using edge strengths from *undirected* graphs) with our Theorem 3.2 (on sampling *uniformly* from *directed* residual graphs). As in the proof of Theorem 6.2 of [2], we show that edges inside well-connected subgraphs can be sampled with lower probability without disrupting connectivity. As in Theorem 3.2 here, we show that increasing the sampling probability makes up for the capacity consumed by the existing flow.

Our analysis considers the assignment of sampling *weights* to the edges of G and the value of the minimum cut in the resulting weighted graph; care must be taken to distinguish between (original graph) capacitated cut values and what we shall hereafter refer to as *weighted cut values*.

5.1. Supporting lemmas.

Before we can prove the main theorem, we need some supporting lemmas from the companion article [2].

LEMMA 5.1 (Corollary 4.9 of [2]). *An n vertex graph has at most $n-1$ strong components.*

LEMMA 5.2 (Lemma 4.10 of [2]). *The graph in which edge e is given weight u_e/k_e has weighted minimum cut at least 1. Similarly, each of the strong components of the graph, considered on its own, has weighted minimum cut at least 1.*

LEMMA 5.3 (Lemma 4.11 of [2]). *In the undirected graph G the edge strengths k_e satisfy*

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

COROLLARY 5.4. *In the residual graph G_f the edge strengths satisfy*

$$\sum u_f(e)/k_e \leq 2n.$$

Proof. Total capacity is conserved in the residual graph since augmentations only change the direction of some capacity. Each undirected edge of G contributes its capacity in both directions initially, which gives the factor 2. $\square$

LEMMA 5.5 (Decomposition Lemma 5.3 of [2]). *Any graph G can be decomposed as a positive weighted sum of graphs $\sum \alpha_i G_i$, where each G_i is one of the at most n strong components of G with edge e given weight u_e/k_e.*

5.2. Proof of the main theorem. We combine all the above pieces to prove Theorem 4.2. To keep the proof clean, we start with the simplifying assumption that all edge strengths k_e are known exactly and used for sampling. Given this assumption, we set $\beta = 8v\ln(nv)/(v-f)$ (which is 1/4th of its value in Theorem 4.2). At the end of the analysis, we will show how relaxing the assumption costs only the constant factor 4.

We give a preliminary application of the above lemmas to prove our sampling theorem for the case $f = 0$ (so $\beta = 8\ln nv$). To model sampling with probabilities u_e/k_e, define the *weight* of edge e as $w_e = u_e/k_e$, so that we sample each edge with probability proportional to its weight.

Corollary 5.4 says that the total weight $\sum u_e/k_e \leq 2n$. Thus, when we choose a directed edge from the graph (using weights u_e/k_e), the probability that we choose an edge from a cut of weight r is at least $r/2n$. Thus, over $\beta n = 8n\ln nv$ choices, the probability that we fail to choose any edge from that cut is at most $(1-r/2n)^{8n\ln nv} \leq e^{-4r\ln nv} = (nv)^{-4r}$. Since by Lemma 5.2 the graph has minimum weighted cut 1, this quantity is at most $(nv)^{-4}$ for any cut. Summing this quantity over all cuts union-bounds the probability that we fail to sample an edge from some cut. Thus the conditions of Reliability Lemma 3.1 apply (with $\alpha = (nv)^{-4}$ and $c = 1$), implying that the sum over cuts is $O((nv)^{-2})$, meaning that every cut has a sampled edge with high probability. But this implies that there is an augmenting path.

This argument applies not only to the entire graph, but also to each of the (at most n by Lemma 5.1) strong components of the graph. In other words, every cut of each strong component will also have an edge sampled. This is immediate from Lemma 5.2 that each strong component has weighted minimum cut 1.

Let us now consider later iterations of the algorithm. The flow running through the graph can decrease the capacity of cuts, but we would like to argue that the increased sampling probability (by a factor of $v/(v-f)$) makes up for the lost capacity. This was relatively straightforward under the uniform-sampling approach of section 3, but the use of edge strengths complicates the issue. While we can bound the amount of lost capacity using Corollary 2.2, we cannot immediately bound the amount of lost sampling weight, i.e., strength, associated with the flow. It is possible that the flow consumes all of the low strength (and thus high sampling weight) edges. However, we will invoke Lemma 5.5, which writes G as a sum of well-connected graphs, and argue that moving flow around cannot make them all badly connected at the same time. Generalizing Corollary 2.2, we have the following result.

LEMMA 5.6. *Any s-t cut induces, in some strong component of G, a cut whose residual edges have total sampling weight at least $(v-f)/v$ times the original weight of that cut.*

Proof. Lemma 5.5 asserts that $G = \sum \alpha_i G_i$, where edge e in each G_i is given weight u_e/k_e. Thus $\sum_{G_i \ni e} \alpha_i = k_e$. Let G'_i denote the graph on strong component G_i in which edge e is given weight $u_f(e)/k_e$. Then $\sum_{G_i \ni e} \alpha_i u_f(e)/k_e = (u_f(e)/k_e) \sum_{G_i \ni e} \alpha_i = u_f(e)$ for every e, so $\sum \alpha_i G'_i = G_f$ and, in particular, the capacity of cut C in G_f is the α_i-weighted sum of the weights of C in the G'_i.

Suppose for contradiction that there is some cut C where every strong component G_i fails the claim of the lemma, meaning that the weight of C in G'_i is less than $(v - f)/v$ times its original (prior to subtracting flow) weight. Then the capacity of C in G_f, which is a weighted sum of those capacities in G'_i, must also be less than $(v - f)/v$ times its original capacity. This contradicts Corollary 2.2. ∎

We will refer to any cut of some k-strong component whose residual weight is at least $(v - f)/v$ times its original weight as a *heavy cut*; the lemma above states that any s-t cut induces a heavy cut in some strong component. It follows that if an s-t cut is left empty when we sample, the heavy cut it induces must also be left empty.

So consider any heavy cut with original sampling weight w. By definition its residual weight is at least $(v - f)w/v$. Thus, since the total graph weight is at most $2n$ by Corollary 5.4, the probability that a random edge is chosen from the cut when sampling by weight is at least $(v - f)w/2nv$. Theorem 4.2 considers sampling βn edges from the graph; thus the probability that it fails to choose an edge from the heavy cut is $(1 - (v - f)w/2nv)^{\beta n} \leq e^{-4w \ln nv} = (nv)^{-4w}$.

We now apply a union bound over all heavy cuts, which we compute by adding a union bound taken in each strong component separately. In a particular component, heavy cut i of original weight w_i contributes value $(nv)^{-4w_i}$ to the sum, so we can upper bound the sum by summing $(nv)^{-4w_i}$ over *all* cuts. (Note that we are summing based on original weights; summing all cuts based on residual weights would fail, as residual weights on nonheavy cuts can be very small.) By Lemma 5.2, the minimum w_i in the strong component is 1. Thus, by Reliability Lemma 3.1, the sum is at most $O(n^2(nv)^{-4}) = O((nv)^{-2})$. It follows that over all n components (Lemma 5.1) the probability of failing to sample from any heavy cut is $O((nv)^{-1})$.

This completes our proof of Theorem 4.2: every s-t cut induces a heavy cut in some strong component, and every heavy cut has an edge sampled with high probability; thus every s-t cut has an edge sampled with high probability.

One detail remains. We performed the entire analysis using exact edge strengths. We do not know how to compute exact edge strengths quickly. However, Benczúr and Karger [2] give an algorithm that computes *lower bounds* k'_e on the edge strengths such that $\sum 1/k'_e \leq 4n$. If we use the quantities $1/k'_e$ in our algorithm, then, since the total weight of edges is at most 4 times that claimed in Corollary 5.4, and since each edge weighs no less than assumed in the analysis above, we conclude that the probability of a given edge being chosen is at least 1/4 of the probability assumed in our analysis. Therefore, if we choose 4 times as many edges, the probability of any edge being chosen rises to match the value used in the analysis above. Increasing the constant from the 8 we used in our proof to 32 in Theorem 4.2 provides the needed factor of 4 increase.

6. Approximate flows by graph smoothing.

Benczúr and Karger [2] also developed a "smoothing" technique for approximating max-flows. They showed how to split apart high-strength edges so that a uniform random sample of the resulting graph preserved cut values. Our previous sampling theorem for residual graphs (Theorem 4.2) proves a weaker result—that the sample contains an augmenting path. In this section we show that it is also possible to smooth a residual graph so that the *values* of residual cuts, and not just their nonemptiness, is preserved. This result is stronger than what we stated above, but the proof is somewhat messier and not necessary for that algorithm. However, we can use this stronger result to give a faster approximation algorithm for max-flows of *arbitrary* value in *capacitated* graphs.

In terms of time bounds, this material is rendered obsolete by the recent work of Kelner et al. [21] and Sherman [26]. However, the techniques are entirely different and may find other applications.

6.1. Sampling from smooth graphs.

DEFINITION 6.1. *Given a graph G, let $G(p)$ denote a random graph constructed by choosing each edge independently at random with probability p.*

Note that if G is a capacitated graph, we choose (or don't) each edge as a whole; we don't treat it as a bundle of separately sampled unit-weight edges.

DEFINITION 6.2 (see [2]). *A graph G with edge capacities u_e and edge strengths k_e is* c-smooth *if, for every edge, $k_e \geq cu_e$.*

THEOREM 6.3. *Let G be a c-smooth graph with max-flow v, and let f be any flow on it. Let $p = \frac{2\rho_\epsilon}{c}\frac{v}{v-f}$, where $\rho_\epsilon = 16(\ln n)/\epsilon^2$ for $\epsilon < 1/2$. If $p < 1$, then with high probability, the minimum s-t cut in $G_f(p)$ exceeds $p(v-f)(1-\epsilon)$.*

This theorem generalizes [2, Smooth Sampling Theorem 7.2] to residual graphs. Our theorem scales up the sampling rate by a factor of $2v/(v-f)$ from the original.

Generalizing to residual graphs raises the same two issues as in Theorem 3.2. First, the flow of value f has subtracted f capacity from every s-t cut. This is easy to handle. If the original minimum cut was v, then as stated in Corollary 2.2 every residual cut has value at least a $(v-f)/v$ fraction of its original value. Thus, scaling up all sampling probabilities by $2v/(v-f)$ as specified in the theorem cancels out this decrease in capacity. The second problem is more complicated: the flow can move capacity around, taking it from edges of low strength (which are sampled with high probability) to edges of high strength (which are sampled less). This suggests that we may not be able to guarantee to sample enough capacity. However, aside from the already handled loss of f capacity, the capacity in a cut can only be moved, not destroyed. Thus, we can show that the necessary sampled capacity is coming from somewhere, even if it is not coming from the edges that originally had it. This idea is captured in the following lemma.

LEMMA 6.4. *Let G be any undirected graph with minimum cut c and edge capacities $u_e \leq 1$. Let G' be a (possibly directed) variant of G on the same edges but with capacities u'_e satisfying $u'_e \leq 2u_e$. Let $\epsilon \leq 1$. If we let $p = \rho_\epsilon/c$ as in* [2, Basic Sampling Theorem 3.1] *and construct $G'(p)$, then with high probability, each cut of value v in G and v' in G' will have value at least $pv'(1-\epsilon\sqrt{2v/v'})$ in $G'(p)$.*

Note that the capacity condition on G' is satisfied by any residual graph of G.

Proof. The theorem [2, Basic Sampling Theorem 3.1] was proved by bounding the probability that each cut diverged by ϵ from its expectation and summing those probabilities in a union bound. For this new lemma, we have chosen the error bound for each cut in $G'(p)$ so that the Chernoff bound on the probability of exceeding it is the same as the Chernoff bound on the probability that the same cut (vertex partition) in $G(p)$ exceeds its factor $(1-\epsilon)$ error bound in Theorem 3.1 of [2]. Thus, when we apply the union bound on $G'(p)$ we get the same sum as we did for $G(p)$ and can conclude that with high probability no cut exceeds its error bound in $G'(p)$.

In more detail, consider a cut of value v in G and v' in G'. The expected value of sampled edges in $G'(p)$ is pv'. We are setting a target error bound of $(1-\epsilon\sqrt{2v/v'}) = (1-\delta)$, where $\delta = \epsilon\sqrt{2v/v'}$. Since each $u'_e \leq 2u_e \leq 2$, we can scale all weights by $1/2$ (halving the expected value to $pv'/2$) and apply the Chernoff bound [3] for variables of maximum value 1 to conclude that the probability that the cut exceeds its error bound is $\exp(-\delta^2(pv'/2)/3) = \exp(-\epsilon^2 pv/3)$, which is same bound we gave on the divergence probability of the analogous cut by ϵ in $G(p)$ in the proof of Basic Sampling Theorem 3.1 of [2]. Thus, the union bound has the same small value. ∎

We now prove Theorem 6.3 by generalizing the above lemma on (residual) graphs of minimum-cut c to arbitrary c-smooth residual graphs, using the same graph-summation technique we used in Theorem 4.2.

Proof of Theorem 6.3. We first sketch the special case of the analysis done previously for undirected graphs [2]. It decomposes the graph G as a weighted sum of at most n graphs $G = \sum \alpha_i G_i$ such that each graph G_i has maximum edge weight $u_e/k_e \leq 1/c$ (for the smoothness parameter c) and minimum cut 1. We apply [2, Basic Sampling Theorem 3.1] to conclude that, when sampled with probability $p = \rho_\epsilon/c$, each graph G_i diverges by at most ϵ from its expectation with high probability. If no one graph diverges too much, then the sum does not diverge. In [2, Basic Sampling Theorem 3.1] the bound on the divergence probability for each graph was based on a union bound that summed the probability that each cut diverged from its expectation.

To generalize this sampling approach to residual graphs we decompose the graph in exactly the same way, assigning edge weight $u_f(e)/k_e$ in G_i to each k_i-strong edge present in G_i (where $u_f(e)$ is the residual capacity of e given flow f) and using the same α_i. Given the flow, the graphs G_i may not be k_i-strong using their residual connectivities. However, the residual capacities satisfy $0 \leq u_f(e) \leq 2u_e$—since G is undirected, the amount of residual capacity added to edge e can be at most u_e. We can therefore apply Lemma 6.4 to conclude a bound on the deviation probabilities of residual cuts in each G_i that can be summed to bound the cut in G_f.

In more detail, consider some particular s-t cut of value w in G and thus $w' = w - f$ in G_f. Its value is spread over the various graphs G_i; let w_i and w'_i be the value contribution to that cut from G_i and G'_i, respectively, meaning $\sum \alpha_i w_i = w$ and $\sum \alpha_i w'_i = w - f$. Lemma 6.4 asserts that the value of this cut in $G'_i(p)$ is at least $pw'_i(1 - \epsilon\sqrt{2w_i/w'_i})$ with high probability. It follows that with high probability the total value of this cut in $G_f(p)$ exceeds

$$\begin{aligned}\sum \alpha_i p w'_i \left(1 - \epsilon\sqrt{2w_i/w'_i}\right) &= \sum \alpha_i p w'_i - p\epsilon \sum \alpha_i \sqrt{2w_i w'_i} \\ &= pw' - p\epsilon \sum \sqrt{2\alpha_i w_i \cdot \alpha_i w'_i} \\ &\geq pw' - p\epsilon \sqrt{2\left(\sum \alpha_i w_i\right)\left(\sum \alpha_i w'_i\right)} \\ &= pw' - p\epsilon\sqrt{2ww'} \\ &= p(w-f)\left(1 - \epsilon\sqrt{2w/(w-f)}\right).\end{aligned}$$

The final quantity is an increasing function of w (because $w/(w-f)$ is decreasing); thus, since $w \geq v$ for any s-t cut, we can conclude that every cut has value at least $p(v-f)(1 - \epsilon\sqrt{2v/(v-f)})$.

At this point, we have proven that when $p = \rho_\epsilon/c$ as in Lemma 6.4, then with high probability the relative error is $\epsilon\sqrt{2v/(v-f)}$. It follows that if instead we multiply p by a factor of $2v/(v-f)$ to $p = (\rho_\epsilon/c)(v/(v-f))$, as is proposed in Theorem 6.3, then, since p is quadratic in the error ϵ, the error decreases by a factor of $\sqrt{2v/(v-f)}$ to ϵ. The completes the proof of Theorem 6.3. ∎

6.2. Application: Approximate max-flow. We show how that above sampling theorem can be used in an approximation algorithm for maximum flow in arbitrarily weighted graphs. The bound we achieve is essentially dominated by the new generation of $\tilde{O}(m/\epsilon^2)$-time approximation algorithms [26, 21]; however, it demonstrates a general technique for speeding up flow algorithms.

Goldberg and Rao [9] show how blocking flows can be applied to capacitated graphs, achieving a time bound of $\tilde{O}(m^{3/2})$ for maximum flow on any graph. Benczúr and Karger [2] use the Goldberg–Rao algorithm in an *approximation algorithm* for maximum flow; it finds a flow with $(1-\epsilon)$ of the maximum flow's value in time $\tilde{O}(m\sqrt{n}/\epsilon)$. We show how sampling from residual graphs can improve the algorithm's running-time dependence on ϵ next.

THEOREM 6.5. *In an undirected graph, a $(1-\epsilon)$-approximation to the maximum flow can be found in $\tilde{O}(m\sqrt{n/\epsilon})$ time.*

Proof. Let us evaluate the time to convert from a $(1-\delta)$ approximation to a $(1-\delta/2)$-approximation—that is, to reduce the residual flow from $v-f=\delta v$ to $\delta v/2$. Note that over the course of this augmentation we have $v/(v-f)\geq 2/\delta$.

Benczúr and Karger [2] show how to transform any graph into a $c=m/n$-smooth graph with $O(m)$ edges and the same cut and flow values in $\tilde{O}(m)$ time. Consider some flow f on the graph, and let us set $\epsilon=1/2$. Theorem 6.3 says that so long as there is $\delta v/2$ residual flow, there is a $p=\tilde{O}(nv/m(v-f))=\tilde{O}(n/\delta m)$ such that sampling every edge with probability p yields a graph that, with high probability, has residual flow at least $\frac{1}{2}p(v-f)$. We can find this flow using the blocking flow algorithm of Goldberg and Rao [9] for capacitated graphs, which runs in $\tilde{O}(m^{3/2})$ time on m-edge graphs. Suppose we repeat this whole process $3/p$ times. If we don't reduce to $\delta v/2$ residual flow, then, by the above argument, we will find at least $(3/p)(\frac{1}{2}p(v-f))>v-f$ additional flow, a contradiction. So within $3/p$ iterations we will reduce to $\delta v/2$ residual flow. The time taken for these $3/p$ iterations is $\tilde{O}((1/p)(pm)^{3/2})=\tilde{O}(\sqrt{p}m^{3/2})=\tilde{O}(m\sqrt{n/\delta})$.

Suppose now that we wish to find a flow within $(1-\epsilon)$ of the maximum flow. Start by setting $\delta=1/2$, meaning that we initially sample $\tilde{O}(n)$ edges and run $O(m\delta/n)=O(m/n)$ phases, and continue halving δ until it reaches the desired value ϵ. This will take $O(\log 1/\epsilon)$ phases of geometrically increasing runtime, dominated by the final phase's runtime of $\tilde{O}(m\sqrt{n/\epsilon})$. In other words, the overall time to find a $(1-\epsilon)$-approximate maximum flow will be $\tilde{O}(m\sqrt{n/\epsilon})$. ☐

Note that in the above algorithm v need not be known. The number of edges sampled in the δ-phase is $pm=\tilde{O}(n/\delta)$, independent of v. Thus the algorithm iterates purely by setting the density parameter δ.

The choice of $c=m/n$ for smoothing and $\epsilon=1/2$ in the inner phase may seem arbitrary. However, using a worse smoothness doesn't reduce the asymptotic number of edges, while achieving better smoothness (which would reduce the required sampling probability) requires that more edges be created, at a rate that exactly cancels out the benefit of the smaller sampling rate. Similarly, for a given smoothness, using a smaller ϵ can at best double the amount of residual flow found in a sample (since we already find half), but quadratically increases the size of the sample, for a net loss.

One might hope to apply this algorithm to get a faster exact algorithm, by first approximating the max-flow and then using augmenting paths to find the final residue; unfortunately, the best such algorithm (attained by balancing for δ) is no faster than the exact algorithms already developed. In particular, notice that although our randomized augmenting paths algorithm from section 4 has running time $O(nv)$ to find the entire flow, its (best currently provable) running time is also $\Omega(nv)$ for finding just the last $\sqrt{v}$ units of flow. Thus, to beat an $\tilde{O}(nv)$ running time, we need to find a way to reduce the residual flow below $\sqrt{v}$ before switching to the augmenting path cleanup. Unfortunately, achieving $\epsilon = 1/\sqrt{v}$ in our approximation algorithm takes time $\tilde{O}(mn^{1/2}v^{1/4})$, a time bound dominated by the existing ones.

6.2.1. Contrast to the method of Benczúr and Karger. Given that we are using a subroutine max-flow algorithm with no dependence on v, it may seem strange that we are first sampling sparsely and then densely—wouldn't the algorithm find the same approximate flow if we sampled densely at the beginning? This is the algorithm of Benczúr and Karger [2]. The key difference is that our sampling step preserves *residual* capacity, meaning that we can repeatedly sample more sparsely than Benczúr and Karger while still progressively finding more flow each time.

When targeting a $(1-\epsilon)$-approximate max-flow, after making the graph c-smooth for $c = m/n$, the approximation algorithm of Benczúr and Karger using their Smooth Sampling Theorem 7.2 from [2] needs to set a sampling rate p small enough to guarantee that the relative error introduced by sampling—which is $\tilde{\Omega}(1/\sqrt{pc})$—is smaller than the target ϵ, i.e., that $p = \tilde{O}(1/\epsilon^2 c)$. Our algorithm, instead of trying to achieve the ϵ-approximation in one leap, creeps up on it by repeatedly augmenting a partial flow. To do so, it only needs to choose a p large enough to ensure that *half* the current *residual* flow is preserved in the sample. In particular, when aiming for a $(1-\epsilon)$-optimal flow, we can choose $p = \tilde{O}(1/\epsilon c)$.

6.3. Approximating the *s*-*t* min-cut value. Our approximate max-flow algorithm can in turn slightly improve the performance of Benczúr and Karger's approximation algorithm for s-t minimum cuts. That algorithm finds a $(1+\epsilon)$ approximation to the s-t minimum cut by compressing the graph to $\tilde{O}(n/\epsilon^2)$ edges and finding a maximum flow and min-cut in that graph in $\tilde{O}(m^{3/2}) = \tilde{O}(n^{3/2}/\epsilon^3)$ time using the Goldberg–Rao algorithm [9]. We can instead use our approximate flow algorithm, with the same parameter ϵ, and find an approximate max-flow in $\tilde{O}((n/\epsilon^2)\sqrt{n/\epsilon}) = \tilde{O}(n^{3/2}/\epsilon^{5/2})$ time. That approximate max-flow will approximate the value of the s-t min-cut. Note, however, that it will give only the value and will not identify a cut of that value.

We can improve things yet further if we invoke our sampling approximation result for residual graphs. Suppose we want to find a $1+O(\epsilon)$-approximation to the minimum s-t cut of value v. As a first step, we can use (undirected) graph compression to reduce the number of edges to $m' = \tilde{O}(n/\epsilon^2)$ while sacrificing only an ϵ error factor in cut values. If we then find a $1+O(\epsilon)$-approximate s-t min-cut in the compressed graph, it will also be a $1+O(\epsilon)$-approximate s-t min-cut in the original graph. The key question, then, is how quickly we can find a $1+O(\epsilon)$-approximate s-t min-cut in a graph with $\tilde{O}(n/\epsilon^2)$ edges.

To do this we will find a $(1-\delta)$-approximate maximum s-t flow f, of value at least $(1-\delta)v$. (If the resulting value exceeds $(1-\delta)v$, we will reduce it to focus on this worst case.) In the resulting residual graph, the minimum s-t cut has value δv. We will use residual graph smoothing to find an ϵ/δ-approximation to this minimum residual s-t cut. In other words, we will find a cut whose value is at most $(1+\epsilon/\delta)$ times the minimum residual cut of value δv. It follows that the value of this cut in the original compressed graph is at most $(1-\delta)v + (1+\epsilon/\delta)\delta v \leq (1+\epsilon)v$, so we achieve our $1+O(\epsilon)$-approximation.

For the approximate residual cut computation, we make the graph $m'/n = \tilde{O}(1/\epsilon^2)$-smooth and plug $v/(v-f) = 1/\delta$ into Theorem 6.3. We conclude that, using

$p = \tilde{O}((n/\delta^2 m)(1/\delta)) = \tilde{O}(n/\delta^3 m)$, we will get a sample of $\tilde{O}(n/\delta^3)$ edges where all residual cut samples are within δ of their expected values, as required. Finding a minimum *s*-*t* cut in the sample (by finding a maximum flow) thus yields the requisite $1+O(\delta)$-optimal *s*-*t* cut in the *residual* graph. This takes time $\tilde{O}((n/\delta^3)^{3/2})$ using the Goldberg–Rao flow algorithm. Since the first, approximate, flow computation takes time $\tilde{O}((n/\epsilon^2)\sqrt{n/\delta})$, balancing the two terms yields $\delta = \sqrt{\epsilon}$ and an overall runtime of $\tilde{O}(n^{3/2}/\epsilon^{9/4})$.

We note that it is also possible to prove a *compression theorem* for a residual graph along the lines of the compression theorem of Benczúr and Karger [2, Compression Theorem 6.2]—that sampling with probabilities proportional to $1/k_e$ and multiplying up the capacity of each sampled edge e by k_e preserves all residual cut *values* reasonably well—but we have found no algorithmic use for this result, as flows in this graph are not feasible for the original graph.

7. Conclusion.

For unweighted graphs, our result of $\tilde{O}(m+nv)$ time is a natural stopping point, but it is not necessarily the end of progress on algorithms for small maximum flows in undirected graphs. For one thing, it is randomized, so there is still the question of how well a deterministic algorithm can do. Perhaps there is some way to apply Nagamochi–Ibaraki sparse certificates [25] in a residual graph. For another, Galil and Yu [7] showed there always exists a max-flow using $O(n\sqrt{v})$ edges on simple graphs. Therefore, while some augmenting paths can require $n-1$ edges, most of them are much shorter. Thus $\tilde{O}(m+n\sqrt{v})$ would be another natural time bound to hope to achieve. And of course there is no evidence ruling out linear time.

Another open question is whether it is possible to give a faster algorithm for small flows in general directed graphs. In sampling from residual graphs, we have shown that random sampling in directed graphs is not entirely hopeless. Perhaps there is a suitable replacement for edge strength in a directed graph that would allow random sampling to be applied to arbitrary directed graphs. One discouraging observation is that even a half-approximation algorithm for flows in directed graphs can be transformed into an exact algorithm with little change in time bound, since we can repeatedly find and augment half the flow in the residual graph. So approximation is little easier than exact solution.

Finally, it is still an open question whether our sampling techniques can be applied to finding *exact* flows in undirected graphs with large flow values. One obvious goal would be to replace m by n in the currently best $\tilde{O}(m^{3/2})$-time exact algorithm [9]. There also remains the question of min-cost flows—even in simple undirected graphs, none of our random sampling techniques have yet been applied successfully to that problem.

Note added in proof. Subsequent to the original publication of this work [18], algorithms for max-flow have continued to progress, and there also has been an explosion of results on approximate max-flow. For undirected graphs (the focus of this paper), Lee, Rao, and Srivastava [22] give an algorithm with runtime $\tilde{O}(m^{5/4}v^{1/4})$. Madry [24] gives a max-flow algorithm for unit-capacity directed graphs with runtime $\tilde{O}(m^{10/7})$. Lee and Sidford [23] solve max-flow even on capacitated graphs in $\tilde{O}(m\sqrt{n}\log^2 U)$ time. For approximate flows, both Kelner et al. [21] and Sherman [26] show how to find an ϵ-approximate max-flow in undirected graphs in $\tilde{O}(m/\epsilon^2)$ time. Setting $\epsilon = v^{-1/3}$ yields a flow of value $v - v^{2/3}$ in $\tilde{O}(mv^{2/3})$ time. The resulting flow may be nonintegral but can be converted to an integral flow in $\tilde{O}(m)$ time [22]. Then $v^{2/3}$ augmenting path steps can raise this to a max-flow in $O(mv^{2/3})$ time. This yields an $\tilde{O}(mv^{2/3})$-time max-flow algorithm for integer capacities.

REFERENCES

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

[2] A. A. Benczúr and D. R. Karger, *Randomized approximation schemes for cuts and flows in capacitated graphs*, SIAM J. Comput., 44 (2015), pp. 290–319.

[3] H. Chernoff, *A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Stat., 23 (1952), pp. 493–509.

[4] E. A. Dinitz, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[5] S. Even and R. E. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[6] L. R. Ford, Jr., and D. R. Fulkerson, *Maximal flow through a network*, Canadian J. Math., 8 (1956), pp. 399–404.

[7] Z. Galil and X. Yu, *Short length versions of Menger's theorem (extended abstract)*, in Proceedings of the 27th ACM Symposium on Theory of Computing, ACM, New York, 1995, pp. 499–508.

[8] A. Goldberg and S. Rao, *Flows in undirected unit capacity networks*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1997, pp. 32–35.

[9] A. Goldberg and S. Rao, *Beyond the flow decomposition barrier*, J. ACM, 45 (1998), pp. 783–797.

[10] R. E. Gomory and T. C. Hu, *Multi-terminal network flows*, J. Soc. Indust. Appl. Math., 9 (1961), pp. 551–570.

[11] R. Hariharan, T. Kavitha, D. Panigrahi, and A. Bhalgat, *An O(mn) Gomory-Hu tree construction algorithm for unweighted graphs*, in Proceedings of the 39th Annual ACM Symposium on Theory of Computing, ACM, New York, 2007, pp. 605–614.

[12] D. R. Karger, *Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm*, in Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 1993, pp. 21–30.

[13] D. R. Karger, *Random sampling in cut, flow, and network design problems*, Math. Oper. Res., 24 (1999), pp. 383–413.

[14] D. R. Karger, *Using random sampling to find maximum flows in uncapacitated undirected graphs*, in Proceedings of the 29th ACM Symposium on Theory of Computing, ACM, New York, 1997, pp. 240–249.

[15] D. R. Karger, *Better random sampling algorithms for flows in undirected graphs*, in Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 1998, pp. 490–499.

[16] D. R. Karger, *A randomized fully polynomial approximation scheme for the all-terminal network reliability problem*, SIAM J. Comput., 29 (1999), pp. 492–514; corrected version in SIAM Rev., 43 (2001), pp. 499–522.

[17] D. R. Karger and M. Levine, *Finding maximum flows in simple undirected graphs seems faster than bipartite matching*, in Proceedings of the 29th ACM Symposium on Theory of Computing, ACM, New York, 1998, pp. 69–78.

[18] D. R. Karger and M. S. Levine, *Random sampling from residual graphs*, in Proceedings of the 33rd ACM Symposium on Theory of Computing, ACM, New York, 2002, pp. 63–66.

[19] D. R. Karger and C. Stein, *An $\tilde{O}(n^2)$ algorithm for minimum cuts*, in Proceedings of the 25th ACM Symposium on Theory of Computing, Alok Aggarwal, ed., ACM, New York, 1993, pp. 757–765.

[20] D. R. Karger and C. Stein, *A new approach to the minimum cut problem*, J. ACM, 43 (1996), pp. 601–640.

[21] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, *An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations*, in Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, Philadelphia, 2014, pp. 217–226.

[22] Y. T. Lee, S. Rao, and N. Srivastava, *A new approach to computing maximum flows using electrical flows*, in Proceedings of the 45th ACM Symposium on Theory of Computing (STOC), ACM, New York, 2013, pp. 755–764.

[23] Y. T. Lee and A. Sidford, *An $\tilde{o}(m\sqrt{(n)})$ algorithm for the minimum cost flow problem*, arXiv:1312.6713, 2013.

[24] A. Madry, *Navigating central path with electrical flows: From flows to matchings, and back*, in Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 2013, pp. 253–262.

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

[26] J. Sherman, *Nearly maximum flows in nearly linear time*, in Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 2013, pp. 263–269.

[27] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Reg. Conf. Ser. Appl. Math. 44, SIAM, Philadelphia, 1983.