Using Random Sampling to Find Maximum Flows in Uncapacitated Undirected Graphs

David R. Karger*

Abstract

We present new algorithms, based on random sampling, that find maximum flows in undirected uncapacitated graphs. Our algorithms dominate augmenting paths over all parameter values (number of vertices and edges and flow value). They also dominate blocking flows over a large range of parameter values. Furthermore, they achieve time bounds on graphs with parallel (equivalently, capacitated) edges that previously could only be achieved on graphs without them.

The key contribution of this paper is to demonstrate that such an improvement is possible. This shows that augmenting paths and blocking flows are non-optimal, and reopens the question of how fast we can find a maximum flow. We improve known time bounds by only a small (but polynomial) factor, and the complicated nature of our algorithms suggests they will not be practical.

A new idea of our algorithm is to find flow by *diminishing cuts* instead of augmenting paths. Rather than finding a way to push flow from the source to the sink, we identify and delete edges that are not needed in a maximum flow. When no more edges can be deleted, we know that every remaining edge must be saturated to give a maximum flow.

1 Introduction

Random sampling has been a useful tool for solving cut problems in undirected graphs. In previous work [Kar94], 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 [Kar94] and exactly finding [Kar96] global min-cuts in near-linear time with high probability. Benczur 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.

In this work, we show that sampling can speed up the fastest known algorithms for finding s-t max-flows (and thus exact s-t min-cuts) in uncapacitated undirected graphs. Until now, the best algorithms have used *augmenting paths* or *blocking flows* [Tar83, AMO93]. Augmenting paths can be used to find a flow of value v in $O(mv) = O(n^3)$ time on an m-edge, n-vertex graph. Blocking flows can be used to find a max-flow in $O(m^{3/2})$ time [Eve79]. Thus augmenting paths dominate for small values of v, while blocking flows dominate when v is large (greater than $\sqrt{m}$). If we restrict the graph to have no parallel edges, blocking flows achieve a time bound of $\tilde{O}(n^{2/3}m) = \tilde{O}(n^{8/3})$ [Eve79]. Together, augmenting paths and blocking flows achieve the best known time bounds for max-flows in uncapacitated graphs.

Our algorithmic technique can be applied to both of these classical methods. Modifying augmenting paths, we give an algorithm that runs in $\tilde{O}(m^{2/3}n^{1/3}v)$ time with high probability, improving the previously given running times by a factor of roughly $(m/n)^{1/3}$ while preserving the efficiency of the algorithm for small v. Modifying blocking flows, we give an algorithm with a (high probability) running time of $\tilde{O}(m^{5/6}n^{1/3}v^{2/3})$, which improves on blocking flows whenever $v \leq m/\sqrt{n}$. Both of our algorithms apply to graphs with or without parallel edges. Compared to blocking flows on graphs without parallel edges, our bound is better when $v^2 \leq n\sqrt{m}$. Taking $m = n^2$ and $v = n$ gives us a single-parameter time bound (for both algorithms) of $\tilde{O}(n^{8/3})$, previously achieved (but not bettered) only on graphs without parallel edges.

A new idea of our algorithm is to find flow by *diminishing cuts* instead of augmenting paths. Rather than identifying empty source-sink paths and increasing flow on them, we identify and delete edges that are not needed in a maximum flow. When no more edges can be deleted, we know that every remaining edge must be saturated to give a maximum flow. Deciding whether an edge can be deleted requires determining whether it crosses a minimum cut. Thus, we can use cut computations to search for the maximum flow.

Our sampling-based method is intuitively straightforward, but its implementation is quite complicated. Although diminishing cuts is a simple idea, our implementation of it appeals to a min-cost flow subroutine. Thus the algorithms presented here are unlikely to be practical. Their main virtue is to present a new technique, and demonstrate that neither augmenting

*MIT Laboratory for Computer Science, Cambridge, MA 02138.
Supported by NSF contract CCR-9624239.
email: karger@lcs.mit.edu.
URL: http://theory.lcs.mit.edu/ karger

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

paths nor blocking flows are optimal, thus reopening the question of how fast we can find a maximum flow.

All algorithms discussed in this paper are randomized. They are also all Las Vegas algorithms, meaning the algorithm is guaranteed to be correct but the running time only holds with high probability. Time in this abstract should always be taken as a measure of the high probability running time, rather than a deterministic quantity.

1.1 Background

Our approach was motivated by the results of [BK96]. That paper showed how sampling could be used to find approximate s-t min-cuts much faster than we know how to find s-t max-flows. In this section, we discuss this approach, highlighting features we will need for our flow algorithm. 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 [Kar94], 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 min-cuts in G by finding them in $G(p)$. Unfortunately, there is some variance in the random samples. However, [Kar94] proved that if G has connectivity c and $p > 8(\ln n)/\epsilon^2 c$, then all cuts are within $(1 \pm \epsilon)$ of their expected values. This led to algorithms for approximating global min-cuts quickly. It also led to the following result which we will use later:

Theorem 1.1 *In a graph with connectivity c, a max-flow of value v can be found in $\tilde{O}(mv/\sqrt{c})$ time.*

The requirement that the sampling probability exceed $\tilde{O}(1/c)$ for small error meant that this approach was only useful for graphs with large min-cuts. Benczur and Karger [BK96] dealt with this problem by showing that any graph with many edges has well-connected regions inside of which the sampling probability can be lowered safely. In particular:

Theorem 1.2 ([BK96]) *For any n-vertex m-edge graph G, and any k,*

1. *There is a set of at most kn edges whose removal partitions G into k-connected subgraphs.*

2. *In $\tilde{O}(m)$ time, we can find a set of $O(kn)$ edges whose removal partitions G into components, each of which is contained in a k-connected subgraph of G.*

The difference between items (1) and (2) above will cause some trouble later. One of our contributions here is to blend the two results, yielding the following theorem:

Theorem 1.3 *For any n-vertex m-edge graph G and any k, in $\tilde{O}(m)$ time, we can find a set of $O(kn \log n)$ edges whose removal partitions G into components, each of which is a k-connected subgraph of G.*

That is, we give up an $O(\log n)$ factor in the edge count compared to the previous construction, but in return we guarantee that each component is k-connected. We prove this theorem in Section 9 We refer to the partition constructed in this new theorem as a *k-strong partition*. This term was also used in [BK96] to describe the construction of item (2) in Theorem 1.2; note, however, that the components of that construction did not themselves need to be k-connected. In this paper, we use the term k-strong partition only in its new sense. We refer to the construction of [BK96] as a *k-strong refinement*. Despite the distinction, Theorem 1.2 was sufficient to prove the following:

Theorem 1.4 (Compression [BK96]) *For any n-vertex m-edge graph G, and for any ϵ,*

- *There exists an $\tilde{O}(n/\epsilon^2)$-edge* compressed graph *G' on the same vertex set such that every cut in G' has value in the range $(1 \pm \epsilon)$ times the value of the corresponding cut in G.*

- *There is an algorithm* `Compress` *that will construct graph G' in $\tilde{O}(m)$ time.*

We refer to the construction in this theorem as *graph compression*. It led to the following corollary:

Theorem 1.5 ([BK96]) *Given a graph with s-t min-cut v we can find a $(1 + \epsilon)$ times minimum s-t cut in $\tilde{O}(nv/\epsilon^2)$ time.*

Proof: If we compress our graph to yield G', any s-t min-cut in G' is a $(1 + \epsilon)$ times minimum s-t cut in G. Such a cut can be found quickly with an augmenting-path max-flow algorithm since G' has few edges. □

The construction used in the proof of Theorem 1.4 preserves the values of flows but distorts their shapes, so that only (approximate) s-t min-cuts, and not max-flows, can be extracted by examining the compressed graph. Furthermore, the algorithm is Monte Carlo, since the only way we know to verify min-cuts is to find max-flows.

1.2 A New Approach

Given that we can find (approximate) min-cuts faster than max-flows, an obvious thought is to use min-cut computations in a max-flow algorithm. We now describe such an approach. It is based on a divide-and conquer scheme. Suppose that we could quickly find an s-t min-cut in G. We know that all edges of this s-t min-cut are saturated by any max-flow. This knowledge allows use to solve G by solving independent max-flow problems on A and on B: the (A, B) min-cut edges define the sinks for a max-flow from s to the boundary of A, and the sources for a max-flow from the boundary of B to t. This sets up a natural divide and conquer algorithm for the max-flow, where we solve the two problems separately and then "patch" them through their interface at the (A, B) cut. We detail this method in Section 2. If we could somehow guarantee that the two subproblems were *balanced*, e.g. that each had at most $2n/3$ vertices, then our divide and conquer algorithm would yield a recurrence in which the time to find s-t min-cuts was the dominant factor.

Of course, at present the only known way to find an s-t min-cut is to compute a max-flow. However, the graph compression technique of Section 1.1 can be applied to find an *approximate* s-t min-cut quickly. We show in Section 3 that dividing and conquering on such an approximate s-t min-cut will yield an approximately maximum s-t flow that can be "cleaned up" via a small number of augmenting path computations.

The problem with all this is that a graph might have no balanced s-t min-cuts. The leads us to our scheme of *diminishing cuts* which we introduce in Section 4. We find any balanced partition of G into two groups and consider the set of edges cut by the partition. We delete edges from it until every edge that remains is necessary for carrying flow. This means that every remaining edge crossing the balanced cut is also crossing an s-t min-cut. We show that this implies the existence of a balanced s-t min-cut, which lets us apply the divide and conquer scheme of the previous paragraph.

Deciding whether an edge can be deleted or not amounts to deciding whether it crosses an s-t min-cut. As before, determining this fact exactly requires computing an s-t max-flow. However, we show in Section 5 that compression can be used to give us an "approximate" answer. Although we might delete a few vital edges, our deletions do not dramatically reduce the s-t min-cut/max-flow value. After we divide and conquer to find a flow with this value, a small number augmenting paths will increase it to a max-flow.

The time to solve the diminishing cut problem depends on the number of edges in the cut we want to diminish (it will amount to the time to find a max-flow in a graph with this many edges). We therefore aim in Section 6 to find a starting cut that is balanced but small. We use the k-strong partition defined in Theorem 1.3 as our starting point. If this partition has no large component, then we can divide its components into two balanced groups to get a balanced 2-way partition with only $\tilde{O}(kn)$ crossing edges to which we can quickly apply the diminishing cuts scheme. If there is one large component, then as we discuss in Section 6 and Section 7, we can divide into this component and its complement. The (small) complement can be solved recursively, while the large component, being k-connected, can be solved by the fast algorithm of Theorem 1.1.

For simplicity, we phrase this presentation in terms of an augmenting path algorithm; in Section 8 we discuss the changes that occur when we use blocking flows instead. Finally, in Section 9, we give an algorithm that proves Theorem 1.3.

2 Using a Cut Oracle to Find Flows

It is a classic result [FF56, FF62] that the value v of an s-t max-flow is equal to the value of the s-t min-cut. A corollary is that a max-flow saturates every edge crossing an s-t min-cut. This scheme is typically used to find cuts by computing flows. Here we show how it can turned around, finding flows by computing cuts.

Suppose that we are given an s-t min-cut (A, B) of value v with $s \in A$ and $t \in B$. We will use this cut to subdivide our original problem into two smaller max-flow problems. Contract A to a single vertex a, yielding a graph denoted G/A. Find a max-flow with source a, which we will call an (A, t)-flow. Since the degree of A is equal to the value of cut (A, B), namely v, we see that this max-flow has value at most v. On the other hand, if we start with an s-t flow of value v, the contraction of A turns it into an A-t flow of value v. Thus, the A-t max-flow is exactly v, and saturates all edges of cut (A, B). Similarly, if we contract B to a vertex, yielding a graph G/B, then the s-B max-flow in G/B is equal to v.

Lemma 2.1 (Patching) *Let (A, B) be an s-t min-cut. Given an A-t max-flow and an s-B max-flow, we can construct an s-t max-flow in linear time.*

Proof: (Sketch.) The only sets of edges the two flows share are those crossing cut (A, B). Both flows saturate all these edges, so they are consistent on them. So just give every edge of G the flow value it had in G/A or in G/B. □

Definition 2.2 *We will refer to the process described in Lemma 2.1 as* patching *the two flows.*

The above argument suggests the divide and conquer algorithm of Figure 2 for computing an s-t max-flow. All time other than the cut computation is linear.

<u>**Algorithm**</u> `Flow`(G, s, t)

if G has two vertices **then**
  saturate their connecting edges and return
$(A, B) \leftarrow$ an s-t min-cut in G
$f_A \leftarrow$`Flow`$(G/A, A, t)$
$f_B \leftarrow$`Flow`$(G/B, s, B)$
patch f_A and f_B in G to an s-t flow f (Lemma 2.1)

Figure 1: Divide and Conquer for Flows

3 Using Approximate Minimum Cuts

Of course, at present the only known algorithms for finding s-t min-cuts are max-flow algorithms. However, Benczur and Karger [BK96] have shown how to find *approximately* minimum s-t cuts quickly. As discussed in Theorem 1.5, a cut of value at most $(1 + \epsilon)$ times the min-cut can be found in $\tilde{O}(nv/\epsilon^2)$ time. We can adapt our divide and conquer strategy to make use of such approximate min-cuts. However, the use of a shared *approximate* min-cut gives the two subproblems options about which of the shared edges they use; they may make different choices, which makes patching the flows somewhat harder. To do so, we use the following technical lemma:

Lemma 3.1 (Flow Decomposition) *There is an $O(m)$-time algorithm that decomposes any integral s-t flow of value v into a set of v edge-disjoint s-t paths.*

Proof: Add v copies of edge (t, s), each carrying a unit of flow, and delete all edges not carrying flow. By flow conservation, we are left with an Eulerian graph. Find an Eulerian tour through it. Deleting the v copies of (t, s) breaks this tour into v disjoint paths as desired. □

Lemma 3.2 (Approximate Patching) *Let (A, B) be any $(1 + \epsilon)$-minimum s-t cut in G. Given an integral A-t max-flow in G/A and an integral s-B max-flow in G/B, an algorithm* `PatchApprox` *can construct an s-t max-flow in G in $O(m + \epsilon mv)$ time.*

Proof: Let f_A and f_B be integral max-flows in G/A and G/B respectively. Since contractions never reduce cut values, f_A and f_B have value at least v. Since the flows are integral, it follows that at least v of the $(1 + \epsilon)v$ edges crossing (A, B) are saturated by flow f_A, meaning at most ϵv cut edges are not saturated by f_A. Similarly, at most ϵv cut edges are not saturated by flow f_B.

Decompose flows f_A and f_B into paths using the algorithm of Lemma 3.1. Consider the (A, B) cut edges saturated by flow f_A. Of these (in fact, of all (A, B) cut edges) at most ϵv are not saturated by f_B. Delete every flow path in f_A that terminates at a cut edge not saturated by f_B. This leaves us with a flow f'_A of at least $(1 - \epsilon)v$ paths. Do the same with f_B, deleting each of the (at most ϵv) flow paths that start at a cut edge not saturated by f_A. This leaves us with two flows f'_A and f'_B of value $(1 - \epsilon)v$ that saturate the same $(1 - \epsilon)v$

edges. It follows as in Lemma 2.1 that these two flows can be patched to yield a flow in G of value at least $(1-\epsilon)v$.

Finally, perform an additional ϵv augmenting path computations to augment our flow to a max-flow. $\square$

We can use this lemma to adapt our divide and conquer algorithm `Flow` from Figure 2. We use the algorithm of [BK96] (Theorem 1.5) to identify a $(1+\epsilon)$-minimum s-t cut (A, B) in $\tilde{O}(nv/\epsilon^2)$ time. We recursively compute an A-t max-flow and an s-B max-flow. Using Algorithm `PatchApprox` of Lemma 3.2, we patch and augment these flows to a max-flow in $O(m + \epsilon mv)$ time. This yields a recurrence

$$\begin{aligned} T(m,n) &= T(m_1,n_1) + T(m+O(v)-m_1, n-n_1) \\ &\quad + \tilde{O}(nv/\epsilon^2) + O(m+\epsilon mv). \end{aligned}$$

(The $O(v)$ term arises from the shared cut edges that appear in both subproblems). Let us make the optimistic assumption that the approximately minimum s-t cuts we find are *balanced* so that, say, $n/4 \le n_1 \le 3n/4$. Under this assumption, we find that the recursive work is negligible and the running time is simply $\tilde{O}(nv/\epsilon^2 + m + \epsilon mv)$. Choosing $\epsilon = (n/m)^{1/3}$ to balance terms yields a running time of $\tilde{O}(m^{2/3}n^{1/3}v)$ as compared to the $O(mv)$ running time of standard augmenting paths.

4 Diminishing Cuts

Unfortunately, our optimism in the previous section is unjustified. There is no reason to suspect that we can always find a *balanced* approximately minimum s-t cut. In an expander graph, for example, all approximately minimum cuts are unbalanced.

We therefore show in this section how to *create* a balanced s-t min-cut by *deleting* edges from the graph until a balanced min-cut exists. So long as we do not delete any s-t min-cut edge, our new graph will have the same max-flow value as the original graph. Our deletions will create a new, balanced s-t min-cut on which we can divide and conquer as the previous section assumes.

Define a *β-balanced cut* to be one with at least βn vertices on each side. Suppose that we have a balanced but non-minimum cut (A, B) that we wish to refine to a balanced s-t min-cut. We do so by identifying a minimum set of its edges needed to carry an s-t max-flow, and deleting all of its other edges. Our approach will initially seem harder than our original problem. Suppose we assign cost 1 to the (A, B) edges, cost 0 to the other edges, and find a *minimum cost* maximum s-t flow. Such a flow will use a minimum set of (A, B) edges; we show this lets us construct the balanced min-cut we want and apply the previous section's divide and conquer scheme. We begin with a lemma characterizing the s-t min-cuts in a graph.

Definition 4.1 *Let f be any s-t max-flow in G. The* residual graph *of f is a directed graph that consists of all edges of G that are not carrying flow together with an edge (v, u) for every edge (u, v) carrying flow.*

Definition 4.2 *Let R be any directed graph. An* ideal *in R is a set of vertices such that for any edge (u, v), if $u \in S$ then $v \in S$ (in other words, an ideal is a set with no outgoing edges).*

Lemma 4.3 ([PR75]) *Every ideal in the residual graph of an s-t max-flow forms one side of an s-t min-cut.*

Proof: From the fact that f is a max-flow it follows that s is in every nonempty ideal, while the only ideal containing t is the entire vertex set.

Consider any nontrivial ideal Y. Since $s \in Y$ and $t \notin Y$, by conservation of flow, there must be a net of v units of flow leaving Y. Now consider an edge (x, y) of G with $x \in Y$ and $y \notin Y$. By the definition of an ideal (x, y) cannot be a residual edge, meaning that (x, y) must carry a unit of flow. Similarly, (y, x) cannot be carrying a unit of flow. In other words, all edges cut by Y must carry a (net) unit of flow out of Y. It follows that exactly v edges are cut by Y. In other words, Y is an s-t min-cut. $\square$

We now how to diminish a given cut (A, B) to an s-t minimum cut. It may not actually be possible to delete all but v edges from the cut (A, B), since it may be necessary for flow paths to travel back and forth from A to B several times. However, we can create an s-t min-cut that is "consistent" with (A, B) in a sense to be defined below.

Lemma 4.4 *Suppose we have an s-t max-flow f that uses a minimal set of edges from a cut (A, B). Suppose we delete every (A, B) edge not carrying flow. Then in the resulting graph, every strongly connected component in the residual graph of f will be contained in A or B.*

Proof: Let C be a strongly connected component in the residual graph of f after the edge deletions. Suppose that $C \cap A$ and $C \cap B$ are both nonempty. Since C is strongly connected, it follows that there is a cycle in the residual graph containing an edge crossing from A to B. By pushing flow around this residual cycle, we can remove flow from this A-B edge. (Recall that every remaining (A, B) edge is carrying flow, so the residual edge must point opposite the flow being carried. Thus augmenting on the residual edge cancels the flow through the edge.) This violates our hypothesis that our flow uses a minimal set of A-B edges, a contradiction. $\square$

Lemma 4.5 *Suppose we have an s-t max-flow that uses a minimal set of edges from a β-balanced cut (A, B). If we delete every (A, B) edge not carrying flow, then the resulting graph will have a $\beta/2$-balanced s-t min-cut that we can find in linear time.*

Remark: It should be noted that the original balanced cut need not separate s and t. If so, there might be no flow crossing the β-balanced cut, meaning that when we delete the (all unused) edges, s and t end up in a small connected component. This can only help us, since the large deleted component is unnecessary for the flow and can be ignored by our algorithms. Slightly abusing our definition, we will refer to the s-t min-cut we find in the component containing s and t as $\beta/2$-balanced. This might violate our definition if, for example, the component containing both s and t has few vertices; however, this can only improve the performance of the algorithms that rely on this lemma. $\square$

Proof: After the deletions, let R be the residual graph for the flow. In order to prove our theorem, we need only prove that R contains an ideal with between $\beta n/2$ and $(1-\beta/2)n$ vertices, and apply Lemma 4.3.

Topologically sort the strongly connected components of R (that is, contract each strongly connected component and topologically sort the resulting DAG). Since the component containing s clearly has no outgoing residual edges, we can take this to be the first component C_1. Similarly, since the component containing t has no incoming residual edges, we can take this to be our last component C_k. Call the resulting sequence of connected components $C_1, \ldots, C_k$. Note that

$k > 1$ since our max-flow saturates some min-cut. Furthermore, every prefix $C_1, \ldots, C_r$ of this sequence with $r \le k$ is an ideal of R. It therefore suffices to prove that some prefix contains between $\beta n/2$ and $(1-\beta/2)n$ vertices.

To do so, start with the C_k and add one component at a time in order, keeping a running total of the number of vertices added. The previous lemma proves that every strongly connected component in R is contained in either A or B. Thus, each C_i has at size most $(1-\beta n)$. It follows that as we add components, our running total never increases by more than $(1-\beta)n$ in one step. Thus at some point, we will have between $\beta n/2$ and $(1-\beta/2)n$ vertices in the prefix, yielding our balanced ideal. This construction can actually be carried out in linear time, proving that we can identify the claimed cut. □

5 Fast Approximate Diminishing Cuts

Of course, finding an s-t max-flow is exactly what we wanted to do in the first place, so it is unrealistic to assume that we can use one to delete unnecessary edges. However, we will make use of an approximate s-t max-flow to achieve the same objective. Consider Algorithm `ApproxDiminish` in Figure 2. This algorithm uses a call to a subroutine `MinCostFlow` that finds a minimum cost flow of maximum value in a graph.

Algorithm `ApproxDiminish`(G, A, B)

input: Graph G with min-cut v
β-balanced cut (A, B) with cut edges S.
output: graph $H \subseteq G$ with min-cut $(1-\epsilon)v$
$\beta/2$-balanced $(1+\epsilon)$-minimum cut (X, Y)

$A' \leftarrow \texttt{Compress}(A)$ (Theorem 1.4)
$B' \leftarrow \texttt{Compress}(B)$
$G' \leftarrow A' \cup B' \cup S$
Assign cost 1 to all edges of S, cost 0 to others
$f \leftarrow \texttt{MinCostFlow}(G')$
$D \leftarrow$ edges of S without flow
find a balanced s-t min-cut (X, Y) in $G' - D$
(using Lemma 4.5)
return $G - D$ with cut (X, Y)

Figure 2: Constructing a balanced min-cut from a balanced cut

Lemma 5.1 *Given an r-edge β-balanced cut of a graph G, Algorithm* `ApproxDiminish` *runs in $\tilde{O}(v(r + n/\epsilon^2))$ and deletes edges from the balanced cut so that*

- *G still has s-t minimum cut at least $(1-\epsilon)v$, and*
- *G has a $\beta/2$-balanced s-t cut of value at most $(1+\epsilon)v$*

Proof: The algorithm's basic approach is to compress both sides of the cut, find a min-cost flow in the resulting graph, and delete balanced-cut edges that the flow does not use. As argued in Lemma 4.5, this leaves behind a minimum set of edges in the compressed graph; we rely on the cut-approximating properties of graph compression to prove that it leaves a near-minimum set of edges behind in the original graph as well.

Consider the algorithm of Figure 2. Given our balanced cut (A, B) (which we refer to as a separator), we compress the induced subgraphs A and B using the techniques of [BK96] as discussed in Theorem 1.4. This construction yields, in $\tilde{O}(m)$ time, two graphs A' and B', each with $\tilde{O}(n/\epsilon^2)$ edges, that approximate all cuts of A and B respectively to within $(1 \pm \epsilon)$. We replace A and B in G with A' and B', yielding graph G'. This does not touch the separator edges S, but leaves us with a graph containing $\tilde{O}(r + n/\epsilon^2)$ edges. Furthermore, since each cut of G involves a cut of A, a cut of B, and some separator edges, it is easy to see that G' approximates all cuts of G to within $(1 \pm \epsilon)$. In particular, the s-t min-cut in G' is at least $(1-\epsilon)v$, and any s-t min-cut in G' corresponds to an at most $(1+\epsilon)^2$-minimum s-t cut in G.

We now apply our diminishing cut construction to graph G'. As in Lemma 4.5, we assign cost 1 to the separator edges, compute a min-cost flow, and delete all the separator edges not carrying flow. Let us call the resulting reduced graph H', with remaining separator edges $S' = S - D$. Lemma 4.5 shows that we can immediately identify a $\beta/2$-balanced s-t min-cut in H'.

In this reduced graph H', replace A' by A and B' by B. Since the separator edges of G' (and H') are all edges of G, this leaves us with a subgraph $H \subseteq G$. Relying once again on the fact that A and A' agree to within ϵ on cut values, as do B and B', we deduce that H has s-t minimum cut at least $(1-\epsilon)v$ and that the $\beta/2$-balanced s-t min-cut we found in H' has value at most $(1+\epsilon)v$ in H.

To prove the time bound, note that the only non-linear time computation is of the min-cost flow. On an m-edge uncapacitated graph, a min-cost augmenting path algorithm will find such a flow in $O(mv)$ time [Tar83, AMO93]. The graph we work with has $\tilde{O}(n/\epsilon^2)$ edges in A' and B' and r edges crossing (A', B'), so the claim follows. □

5.1 Graphs with a Separator Oracle

We can use the above algorithm directly to find flows in any family of *graphs with a small separator oracle*. This is a family, closed under taking minors (that is, deleting and contracting edges) such that every graph of size n has a β-balanced cut of value $S(n)$ (which we call call a separator) that can be identified in $O(T_S(n))$ time. Note that in contrast to our invalid balanced min-cut assumption of the last section, there actually are interesting graph families, such as planar graphs, that have fast small separator oracles [GHT84, ASR90, KRP93, PRS94] (though the ones listed here have so few edges as to make our sampling scheme pointless).

Suppose we have a family of graphs with size-$S(n)$ separators that can be found in $O(T_S(n))$ time. Consider Algorithm `Flow` of Figure 3. After finding a β-balanced cut, the algorithm uses Algorithm `ApproxDiminish` of Lemma 5.1 to create a balanced approximate min-cut, recursively finds flows on the two sides of this min-cut, and then patches them as in Lemma 3.2.

Algorithm `Flow`(G, s, t)

$(A, B) \leftarrow \texttt{Separator}(G)$
$(H, X, Y) \leftarrow \texttt{ApproxDiminish}(G, A, B)$
$f_X \leftarrow \texttt{Flow}(H/X, X, t)$
$f_Y \leftarrow \texttt{Flow}(H/Y, s, Y)$
$\texttt{PatchApprox}(f_A, f_B)$ (using Lemma 3.2)

Figure 3: A flow by separators

We now analyze the running time of this algorithm. We find an $S(n)$-edge separator in time $T_S(n)$. Algorithm `ApproxDiminish` runs in $\tilde{O}(nv/\epsilon^2 + S(n)v)$ time. We thus have a recurrence in which it is easy to prove that the recursive calls have negligible cost:

$$\begin{aligned} & T(m,n) \\ & \quad \leq \quad T_S(n) + \tilde{O}(n/\epsilon^2 + S(n)v + \epsilon mv) \\ & \qquad\quad +T(m_1,n_1) + T(m+S(n)-m_1, n-n_1) \\ & \qquad\qquad ((\beta/2)n \leq n_1 \leq (1-\beta/2)n) \qquad (1) \\ & \quad = \quad \tilde{O}(T_S(n) + nv/\epsilon^2 + S(n)v + \epsilon mv) \\ & \quad = \quad \tilde{O}(T_S(n) + S(n)v + n^{2/3}m^{1/3}v) \end{aligned}$$

if we balance for ϵ.

6 Finding a Starting Cut

Our previous algorithm relied on the existence of small separators (and an oracle for finding them). In this section, we show that assumption was unnecessary. We give an algorithm that, for any graph, finds either the separator we relied on or an equally useful alternative structure. We start by setting $k = 1/\epsilon^2$ and constructing (in $\tilde{O}(m)$ time) the *k-strong partition* guaranteed by Theorem 1.3. This gives us a multiway partition of G such that $O(kn \log n)$ edges cross the partition.

If we are fortunate, the largest component of the partition has less than $n/2$ vertices. If so, then it is easy to divide the components into two groups such that the number of vertices in each group is at most $3n/4$. This gives us a $\frac{1}{4}$-balanced cut with $O(kn \log n) = \tilde{O}(n/\epsilon^2)$ edges crossing it and allows us to apply the divide and conquer scheme of the previous section.

We now deal with the possibility that the largest component has size greater than $n/2$. One particularly easy case occurs when there a single component of size n. In this case, we know that our graph is in fact k-connected. As noted in Theorem 1.1, an algorithm of [Kar94] uses random edge partitioning to find a flow of value v in any k-connected graph in $\tilde{O}(mv/\sqrt{k}) = \tilde{O}(\epsilon mv)$ time.

A greater challenge arises when there is more than one component but some component has size greater than $n/2$. In this case we will proceed with our divide step as before, but will then perform a recursive call only on the smaller side. We will handle the larger side using the algorithm for k-connected graphs just mentioned. (It is this step, which requires that the large side be k-connected and not just a subset of a k-connected graph, that forces us to develop a different k-strong partition algorithm than that of [BK96].) Since we do not recurse on the larger side, the divide and conquer recurrence will still have good time bounds.

Let us suppose that the k-strong partition gave us components $S_0, \ldots, S_r$, with $|S_0| \geq n/2$. Our approach begins as before. We try to use Algorithm `ApproxDiminish`$(G, S_0, \overline{S}_0)$ to construct a balanced approximately minimum cut of G. Recall that this algorithm finds a flow and deletes edges such that the strongly connected components $\{C_i\}$ of its residual graph are contained in S_0 or $\overline{S}_0$. Recall how we used this graph to define a balanced approximately minimum s-t cut: we topologically sorted the strongly connected components into a sequence $C_1, \ldots, C_k$ and chose a balanced prefix of the topological sort. If this method succeeds in our current graph, we can continue exactly as we did in the previous section. However, since we did not necessarily start with a balanced partition, we cannot assume that we succeed in getting a balanced prefix.

6.1 A Large Residual Component

Observe that the only way `ApproxDiminish` can fail to find a balanced cut is if one of the residual strongly connected components, say C_j, has size exceeding, say, $3n/4$. Since $|S_0| > n/2$, we must have $C_j \subseteq S_0$.

We now proceed slightly differently depending on which (if either) of s or t is in C_j. We first consider an easy case, where $s \in C_j$ (so $j = 1$) while $t \notin C_j$ (the general case will be discussed in the next section). In this case, we use C_j and its complement as our divide and conquer partition (note that $C_j = C_1$ is an ideal and thus defines a min-cut). We aim to find flows in G/C_j and $G/\overline{C}_j$ and patch them using Algorithm `PatchApprox`.

Since $|C_j| > n/2$ by hypothesis, the first recursive subproblem G/C_j has less than $n/2$ vertices, making a recursive solution by our current divide and conquer method cheap. On the other hand, C_j may be so large that the subproblem $G/\overline{C}_j$ seems expensive to solve. However, we show that this subproblem is essentially attacking a graph of connectivity k, allowing us to solve it in $\tilde{O}(mv/\sqrt{k}) = \tilde{O}(\epsilon mv)$ time (Theorem 1.1).

Suppose first that S_0 intersects $\overline{C}_j$. Then since $C_j \subseteq S_0$, contracting $\overline{C}_j$ to a single vertex leaves us with a graph that is in fact a contracted version of S_0. Since contraction can only increase connectivity, it follows that $G/\overline{C}_j$ is k-connected and can be solved in $\tilde{O}(mv/\sqrt{k}) = \tilde{O}(\epsilon mv)$ time (Theorem 1.1).

So now suppose that S_0 does not intersect $\overline{C}_j$. It follows that in fact $C_j = S_0$ and is k-connected. We also know that $G/\overline{C}_j$ has s-$\overline{C}_j$ flow $\Theta(v)$, which certainly implies that $\overline{C}_j$ has degree $\Theta(v)$ in $G/\overline{C}_j$. In other words, our contracted graph consists of a k-connected graph C_j to which we have appended a vertex of degree v. Such a graph has connectivity at least $\min(k, v)$.

We therefore consider two cases. If $v \geq k/\log n$, we deduce that our graph is $k/\log n$-connected. It follows that the algorithm of Theorem 1.1 finds a max-flow in this graph in $\tilde{O}(mv/\sqrt{k}) = \tilde{O}(\epsilon mv)$ time. If $v \leq k/\log n$, then our graph is v-connected, so Theorem 1.1 shows that the flow can be found in $\tilde{O}(m\sqrt{v})$ time. However, we can do better. [Kar97] shows that if we randomly partition the edges of C_j into $O(k/\log n)$ sets, then each contains a spanning tree of C_j with high probability. Thus we can use one spanning tree to route each of our desired $v \leq k/\log n$ units of flow. So we can solve this flow problem in $\tilde{O}(m)$ time. In either case, we are finding the flow in $\tilde{O}(m + \epsilon mv)$ time.

It follows that when the divide and conquer recurrence of Equation 1 for the running time $T(m,n)$ does not apply because all the s-t min-cuts are unbalanced, then our running time satisfies a different quantity $T_1(m,n)$ given by

$$T_1(m,n) \leq T(m, n/2) + \tilde{O}(m + \epsilon mv). \qquad (2)$$

We have therefore argued that any problem can be broken up into subproblems such that either Equation 1 or Equation 2 holds. Since we are trying to upper bound the running time of our algorithm, we assume that an adversary chooses the outcome that gives us a larger bound. This gives us a running time recurrence upper bounding our algorithm:

$$
\begin{aligned}
&T(m, n) \\
&\quad \leq \max \quad T(m, n/2) + \tilde{O}(m + \epsilon m v), \\
&\qquad\qquad\quad T(m_1, n_1) + T(m + \tilde{O}(n/\epsilon^2) - m_1, n - n_1) \\
&\qquad\qquad\qquad +\tilde{O}((n/\epsilon^2)v + \epsilon m v + m) \\
&\qquad\qquad\quad (\beta/2n \leq n_1 \leq (1-\beta/2)n)
\end{aligned}
$$

The second term in the maximization is Equation 1, plugging in for the separator oracle the $\tilde{O}(m)$ time algorithm of Theorem 1.3 that gives a k-strong partition with $\tilde{O}(n/\epsilon^2)$ edges. Solving this recurrence, it turns out that it never helps the adversary to choose the first (unbalanced) option, so we are basically left with the recurrence of Section 5 that assumes the balanced case. We derive the same bound of

$$T(n) = \tilde{O}(n^{2/3} m^{1/3} v).$$

7 A Three Way Divide and Conquer

In Section 6.1, we assumed that the large component of the residual graph was C_1, Here we consider a more complicated case in which neither s nor t is in the large component C_j. Without loss of generality, this is the only case we need to consider. For if $s \in C_j$, we can create a new source s' connected to s by a saturated capacity v edge, which makes $\{s'\}$ a separate component of the residual graph. We can do the same for t.

When neither s nor t is in C_j, our two recursive subproblems are different. Let $A = \cup_{i<j} C_i$ and let $B = \cup_{i>j} C_i$. We recursively find:

- An s-t max-flow in G/C_j, and
- An A-B max-flow in $(G/A)/B$

The second of these requires some explanation. $(G/A)/B$ is a graph in which A has been contracted to a single vertex and B has been contracted to a different single vertex.

This is a slight generalization of our previous two-way divide and conquer scheme. Since A and $A \cup C_j$ are both ideals of the residual graph, we know that they both define compressed-graph s-t min-cuts, so that the compressed-graph flow that we found all travels from A to C_j to B (some may go directly from A to B). It follows that the two recursive flows we find use all but ϵv of the edges from A to C_j and from C_j to B. We can therefore patch the two sub-problems roughly as before. When we decompose the flow in G/C_j into flow paths, we will find that some of the paths go through C_j while others do not. The paths that do not traverse C_j are left alone. The paths that do pass through "vertex" C_j are split at C_j into a set of s-C_j paths and a set of C_j-t paths. We then take the flow paths found in C_j in $(G/A)/B$, and (as in Lemma 3.2) patch the s-C_j paths in A to the A-C_j paths in C_j, and patch the C_j-B paths in C_j to the C_j-t paths in B.

Since $|C_j| > n/2$, the first recursive subproblem (on $A \cup B$) has at most $n/2$ vertices, permitting the divide and conquer step. On the other hand, C_j may be large, so that the subproblem on $(G/A)/B$ is expensive to solve. However, as above, we show that this subproblem is essentially attacking a graph of connectivity k, allowing us to solve it in $\tilde{O}(mv/\sqrt{k}) = \tilde{O}(\epsilon m v)$ time (Theorem 1.1).

Suppose first that S_0 intersects A and B. Then contracting A and B to single vertices leaves us with a graph that is in fact a contracted version of S_0. Since contraction can only increase connectivity, it follows that $(G/A)/B$ is k-connected and the algorithm of Theorem 1.1 applies to solve it in $\tilde{O}(mv/\sqrt{k})$ time.

So now suppose that S_0 intersects neither A nor B (the case where it intersects exactly one is handled similarly). It follows that in fact $C_j = S_0$ and is k-connected. We also know that $(G/A)/B$ has A-B flow $\Theta(v)$, which certainly implies that A and B have degree $\Theta(v)$ in $(G/A)/B$. In other words, our contracted graph consists of a k-connected graph to which we have appended two vertices of degree v. Such a graph has connectivity at least $\min(k, v)$.

We now finish the argument as in Section 6.1.

8 Blocking Flows

Our discussion above only made use of augmenting paths. Our approach can also be used to speed up a blocking-flow based algorithm. Blocking flow algorithms compute a series of blocking flows instead of a series of augmenting paths. A single blocking flow can be found in $\tilde{O}(m)$ time on capacitated or uncapacitated graphs [Tar83]. It can be shown [Eve79] that in an m-edge *uncapacitated* graph, or equivalently in a capacitated graph of total edge capacity m, $O(\sqrt{m})$ blocking flow computations suffice to find a maximum flow.

It is implicit in the work of [GT89, GK94] that these algorithms can be used to find min-cost flows in the same time bounds. We can use this modified min-cost flow algorithm to construct the compressed-graph flow that identifies an approximately minimum cut in Algorithm `ApproxDiminish` in Lemma 4.5. Since the compressed graph that we work in has $\tilde{O}(n/\epsilon^2)$ edges, the running time of a blocking flow algorithm step is $\tilde{O}(n/\epsilon^2)$. Since the compressed graph has total edge capacity m [BK96], the number of blocking flow iterations needed is $O(\sqrt{m})$. Thus the total time to find a flow in the compressed graph is $\tilde{O}(n\sqrt{m}/\epsilon^2)$. This replaces the $\tilde{O}(nv/\epsilon^2)$ term in the recurrence for our overall running time, leading to a bound of

$$\tilde{O}(\epsilon m v + \tilde{O}(n\sqrt{m}/\epsilon^2))$$

Balancing for ϵ gives a running time of

$$\tilde{O}(m^{5/6} n^{1/3} v^{2/3}).$$

Observe that so long as $v = O(n)$ and $m = O(n^2)$ (always the case in graphs without parallel edges), the above time bound is actually $\tilde{O}(n^{8/3})$. While such a bound could previously be achieved by blocking flows on graphs with no parallel edges, our algorithm applies to graphs with or without parallel edges.

9 Finding a k-Strong Partition

In this section, we describe the algorithm used in Theorem 1.3. The goal of this algorithm is to find a partition of G into k-connected subgraphs with few edges cut by the partition. The algorithm is guaranteed to return components that are k-connected. The edge bound applies only with high probability, but we can rerun the algorithm until it is met.

The basic idea of our algorithm is simple. We attempt to find a cut of value less than k in our graph. If no such cut exists, we know the graph is k-connected. Otherwise, we delete all edges crossing this cut, leaving two graph components. Clearly, each k-strong component of G is contained in one of these two components. Thus, we recursively seek k-strong partitions in the two pieces of G. It is easy to prove that no edge in a k-strong component will ever be deleted, but that every edge connecting two k-strong components eventually will. Thus, the set of all edges deleted during the original and recursive calls forms the set of edges crossing between k-strong components of the final output. Furthermore, since we delete at most k edges each time we increase the number of components by 1, the total number of edges cut by the resulting partition is at most kn.

The simplest way to implement this approach is to use a minimum cut algorithm to find a cut of value less than k if one exists. This gives an algorithm with running time $\tilde{O}(mn)$ if we use the min-cut algorithm of [Kar96].

To achieve a faster time bound, the algorithm of [BK96] gave something up. Although the cuts that it finds are small enough on *average* to ensure that the total number of edges deleted (and thus the number of edges cut by the partition it constructs) is $O(kn)$, the algorithm may occasionally delete a large cut, possibly separating a k-strong component of G into two or more non-k-connected pieces. Thus, the algorithm does not guarantee that the components it finds are k-connected; only that they are subsets of G's k-strong components. One might try to ensure the components are k-strong by recursively applying the algorithm to the components it picks the first time; unfortunately, it is possible that each recursive application will succeed in separating only one vertex of a component, implying a recursion depth of $\Omega(n)$ and thus a running time of $\Theta(mn)$.

Here we present a different approach that is more carefully designed to allow for recursive calls to clean up the components we find initially. It pays for this care by building a partition with a larger number (by a logarithmic factor) of cut edges than that produced by the algorithm of [BK96]. Our algorithm `StrongPartition` relies on a to-be-described subroutine `SmallOrStrong`. Algorithm `SmallOrStrong`(G, k) runs in $\tilde{O}(m)$ time and produces a partition of G with the following properties:

- $O(kn \log n)$ edges cross the partition
- any component of size exceeding $n/2$ in the partition is k-connected

Using `SmallOrStrong`, it is trivial to implement Algorithm `StrongPartition` as in figure 4.

Algorithm `StrongPartition` (G, k)

input: n vertex graph G, parameter k

$P \leftarrow$ `SmallOrStrong`(G, k)
Output any component of P of size exceeding $n/2$
Foreach component $C \in P$ of size less than $n/2$
    call `StrongPartition`(C, k)

Figure 4: Algorithm `StrongPartition`

Lemma 9.1 *Suppose that* `SmallOrStrong` *performs as claimed above. Then* `StrongPartition` *runs in* $\tilde{O}(m)$ *time and produces a partition of* G *such that every component is* k*-connected while* $O(kn \log^2 n)$ *edges are cut by the partition.*

Remark: A technique similar to one used in [BK96] reduces the number of cross-partition edges to $O(kn \log n)$. $\square$

Proof: Consider the recursion tree of calls to `StrongPartition`. After the call to `SmallOrStrong`, every component of size exceeding half the input is output immediately. Thus each recursive call is on a problem of at most half the size of the original. It follows that the depth of the recursion tree is $O(\log n)$.

Now note that the recursive calls partition the graph G. It follows by induction that the total size of problems at a given level of the recursion is at most $m + n$. Since all the work at a given level is done by `SmallOrStrong`, which runs in $\tilde{O}(m)$ time, the total work done at one level of the recursion tree is $\tilde{O}(m)$. Since there are $O(\log n)$ recursion tree level, the total work remains $\tilde{O}(m)$.

It remains to prove that the resulting partition has the desired properties. It is trivial that every output component is k-connected, We must also bound the number of edges cut by the partition. By induction, these are just the edges cut in the various calls to `SmallOrStrong`. Once again, consider the recursion tree. Since the total vertex-count of problems at each level is n, `SmallOrStrong` deletes $O(kn \log n)$ edges from G at each level. Since the number of levels is $O(\log n)$, the total number of edges deleted is $O(kn \log^2 n)$. $\square$

We now turn to the implementation of `SmallOrStrong`. This algorithm uses a parameter $r = O(k \log n)$ that will be specified later. It uses the *k-strong refinement* routine [BK96] described in Theorem 1.2. It also makes use of a to-be-described subroutine `BigStrong` that runs in $\tilde{O}(m)$ time and finds a k-connected subgraph of size exceeding $n/2$ in any graph with an r-connected component of size exceeding $n/2$.

Algorithm `SmallOrStrong` (G, k)

input: n vertex graph G, parameter k

Find an r-strong refinement R of G
        (using the algorithm of Theorem 1.2)
if no component of R has size greater than $n/2$ **then**
    output R
else [G has r-connected subgraph of size $> n/2$]
    $K \leftarrow$ `BigStrong`(G, k)
    while any vertex $v \notin K$ has $\geq k$ neighbors in K
        add v to K
    output a two-way partition: $(K, \overline{K})$

Figure 5: Algorithm `SmallOrStrong`

Lemma 9.2 *Let us suppose for now that* `BigStrong` *performs as claimed. Then Algorithm* `SmallOrStrong` *runs in* $\tilde{O}(m)$ *time and outputs a partition of* G *with* $O(kn \log n)$ *cut edges such that any component of size exceeding* $n/2$ *is* k*-connected.*

Proof: The running time follows from the fact that `BigStrong` and the algorithm of Theorem 1.2 run in $\tilde{O}(m)$ time. We need only argue that the while loop can be efficiently implemented. To do so, simply keep a queue of vertices not in K with more than k neighbors in K. When we remove a vertex v from the queue and place it in K, increment the number of neighbors each neighbor of v counts in K, adding vertices to the queue if their neighbor count rises to k. This is easily seen to take $O(m+n)$ time.

To prove that the partition has the claimed properties, consider two cases. If we return the k-strong refinement R then the edge-bound follows from Theorem 1.2 and the claim on large components is vacuously true. Otherwise, consider the k-connected subgraph K returned by `BigStrong`. It is straightforward that if v is a vertex with k neighbors in a k-connected subgraph K, then $K \cup \{v\}$ is also k-connected. This shows that the modified component K we build is k-connected. Since it is also the only component of size exceeding $n/2$ that we return, the claim on large components is satisfied. Now consider the number of edges crossing from (the final) K to its complement. There are at most $n/2$ vertices not in K, and each has less than k neighbors in K. Thus the number of edges crossing from K to its complement is at most kn, better than claimed. □

9.1 Finding a Large k-Strong Component

It remains to show how to implement `BigStrong`. Recall that `BigStrong` is called only if G has an r-connected component R of size exceeding $n/2$. Out algorithm works by deleting vertices from G until a large k-connected graph K remains. In particular, we find a k-connected subgraph that contains R. This subgraph clearly has size exceeding $n/2$, as required. Our plan for stripping away vertices is to repeatedly identify a cut of value less than k and delete the smaller side of such a cut. Since K is always on the large side, we will never delete vertices from K. When we can no longer find a cut of value less than k, we know the remaining graph is k-connected.

Our implementation of this idea is somewhat more complicated. We begin by partitioning the graph edges randomly into k groups, defining edge-subgraphs $G_1, \ldots, G_k$ of G. It follows that for any cut C of value less than k, some one of our subgraphs contains no edges of C and is thus disconnected at vertex C's vertex partition, in a sense revealing that C has value less than k. We will delete the all vertices on the smaller side of C from G and all the G_i. This may create some new cuts of value less than k in what remains of G. However, such a new cut will also have no edges in some G_i and will thus be revealed for deletion. We will repeat this process until G is empty or every G_i is a connected graph. At this point each G_i contains a spanning tree, so it follows that $\cup G_i$ contains k edge-disjoint spanning trees and is therefore a k-connected graph K. We proceed to show that $K \supseteq R$. This means that K has size exceeding $n/2$ and can therefore be returned by `BigStrong`.

We prove that $K \supseteq R$ by proving that no vertex of R is ever deleted. Conceivably, an empty cut in G_i corresponds to a cut of value greater than k in G, and could in fact separate two vertices of R. We now argue that this is not the case. We use the following lemma from [Kar94].

Lemma 9.3 ([Kar94]) *Let G be an n-vertex graph from which every edge is selected independently with probability $1/k$. There is an $r = O(k \log n)$ such that if G is r-connected, the selected edges contain a spanning tree of G with high probability.*

We take r as in this lemma to define r in `SmallOrStrong` and `BigStrong`. This lets us prove the following.

Lemma 9.4 *No vertex of R is ever deleted by algorithm `BigStrong`. Thus, the algorithm returns a (k-connected) subgraph of size exceeding $n/2$.*

Proof: Take r as in the previous lemma. Consider the large r-strong component R. In `BigStrong` its edges are randomly distributed into the k graphs G_i. The set of edges in each G_i has the distribution we get by choosing edges independently with probability $1/k$. Therefore, by the above lemma, this edge set contains a spanning tree of R with high probability. Applying this argument to all k graphs G_i lets us deduce that with high probability, every G_i contains a spanning tree of R. In other words, every vertex of R is in a connected component of size exceeding $n/2$ in every G_i. It follows that no vertex in R will ever be selected for deletion by `BigStrong`. This in turn implies that the spanning trees of R in the G_i will always survive, yielding a proof by induction that no vertex of R is ever deleted. □

9.2 Efficient Data Structures

We have shown that `BigStrong` finds a subgraph with the desired properties. We now show how it can be implemented efficiently. We must implement an algorithm for identifying and deleting small connected components from the G_i. We use the dynamic connectivity data structure of Henzinger and King [HK95]. This data structure maintains a query structure for a graph under insertion and deletion of edges such that inserting edges, deleting them, and answering queries about whether two vertices are connected takes $O(\log n)$ time per operation. We actually use only deletions and queries, and can therefore also use Thorup's adaptation [Tho95] of the Henzinger-King algorithm to this case.

After partitioning our edges into the graphs G_i, we build the connectivity data structure for each. We also identify in each G_i the vertices not in the largest component and put them into a queue of to-be-deleted vertices. We now go through the queue, deleting vertices and possibly enqueuing new vertices for deletion, until the queue is empty. At this point, the undeleted vertices (if any exist) form the large k-connected component K.

As we delete vertices, we maintain the following invariant: every small connected component of each G_i (that is, every component of G_i with fewer than $n/2$ vertices) contains a vertex that has been queued for deletion. It follows that when the queue is empty, every G_i is connected. This invariant is clearly true when we initialize; we show how to maintain it. Suppose for the moment that we can answer queries about whether a vertex is in the large component of G_i in $\tilde{O}(1)$ time. To maintain the invariant when we delete a vertex v, we consider each G_i in turn. First, we delete all edges incident on v in G_i. Then, for each u that was a neighbor of v before the edge deletions, we test whether u is in the large component of G_i. If it is not, we mark it for deletion. It is easy to see that this maintains the invariant, since any small component formed by the deletion of v must contain some (ex-)neighbor of v.

Finally, it remains to show that we can answer queries about whether u is in the large component. To achieve this, before starting our algorithm, choose a vertex t at random. With probability at least $1/2$ we choose $t \in R$. Assume this is the case. Then at all times t is in the large component of all G_i. So we can test for membership in the large component by testing for connectivity to t. If we repeat the whole algorithm $O(\log n)$ times, then with high probability at least once we will get a $t \in R$; in this iteration our large component test will work correctly.

To measure the running time of this algorithm, note that we delete each vertex at most once and perform one vertex query and one edge deletion operation for every neighbor of each vertex we delete; thus, over the entire execution, the total number of queries and deletions is $O(m)$. Since each operation takes $\tilde{O}(1)$ time, the total work is $\tilde{O}(m)$ per iteration, and thus $\tilde{O}(m)$ over the $O(\log n)$ iterations we need to succeed with high probability.

10 Conclusion

The major contribution of this paper has been to show that neither augmenting paths nor blocking flows are optimal algorithms for finding flows in undirected uncapacitated graphs. We have given algorithms that do better than both by applying random sampling. The algorithms are too complicated to be practical but will hopefully stimulate research into simpler algorithms. Intuition also suggests that one can find algorithms asymptotically better than those above, as the time bounds we derive are certainly not natural stopping points.

There are two obvious ways to simplify the above algorithms. The first would be to give a better subroutine to identify a large, well connected component of a graph, eliminating the complicated reliance on dynamic connectivity algorithms. The other would be to somehow replace the min-cost flow calculations with appropriate max-flow calculations. This might allow the algorithm to be applied (edge)-recursively, leading to better bounds of $\tilde{O}(v\sqrt{mn})$.

Perhaps the most exciting extension would be to apply this scheme to devise algorithms with running time not dependent on v, breaking the longstanding $\tilde{O}(mn)$ bound for 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.

[ASR90] Noga Alon, Paul Seymour, and Thomas R. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the 22^{nd} ACM Symposium on Theory of Computing*, pages 293–299. ACM, ACM Press, May 1990.

[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.

[FF56] Lester R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

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

[GHT84] John R. Gilbert, Joan P. Hutchinson, and Robert E. Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5:375–390, 1984.

[GK94] A. V. Goldberg and R. Kennedy. Global Price Updates Help. Technical Report STAN-CS-94-1509, Department of Computer Science, Stanford University, 1994. To appear in SIAM J. on Discrete Math.

[GT89] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[HK95] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the 27^{th} ACM Symposium on Theory of Computing*, pages 519–527. ACM, ACM Press, May 1995.

[Kar94] David R. Karger. Random sampling in cut, flow, and network design problems. In *Proceedings of the 26^{th} ACM Symposium on Theory of Computing*, pages 648–657. ACM, ACM Press, May 1994. Submitted for publication..

[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.

[Kar97] David R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. *SIAM Journal on Computing*, 1997. To appear. A preliminary version appeared in STOC 1995.

[KRP93] Phil Klein, Satish Rao, and Serge Plotkin. Excluded minors, network decomposition, and multicommodity flow. In *Proceedings of the 25^{th} ACM Symposium on Theory of Computing*, pages 682–690. ACM, ACM Press, May 1993.

[PR75] J.C. Picard and H.D. Ratliff. Minimum cuts and related problems. *Networks*, 5:357–370, 1975.

[PRS94] Serge Plotkin, Satish Rao, and W. Smith. Shallow excluded minors and improved decompositions. In *Proceedings of the 5^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 462–470. ACM-SIAM, January 1994.

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

[Tho95] Mikkel Thorup. Dynamic decremental connectivity. In *Proceedings of the 6^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 305–313. ACM-SIAM, January 1995.