# Finding Maximum Flows in Undirected Graphs Seems Easier than Bipartite Matching

David R. Karger* and Matthew S. Levine*

**Abstract**

Consider an $n$-vertex, $m$-edge, undirected graph with maximum flow value $v$. We give a method to find augmenting paths in such a graph in amortized sub-linear ($O(n\sqrt{v})$) time per path. This lets us improve the time bound of the classic augmenting path algorithm to $O(m + nv^{3/2})$ on simple graphs. The addition of a blocking flow subroutine gives a simple, deterministic $O(nm^{2/3}v^{1/6})$-time algorithm. We also use our technique to improve known randomized algorithms, giving $\tilde{O}(m+nv^{5/4})$-time and $\tilde{O}(m+n^{11/9}v)$-time algorithms for capacitated undirected graphs. For simple graphs, in which $v \leq n$, the last bound is $\tilde{O}(n^{2.\bar{2}})$, improving on the best previous bound of $O(n^{2.5})$, which is also the best known time bound for bipartite matching.

## 1 Introduction

In this paper we consider the problem of finding maximum flows in undirected graphs with small flow values. Traditionally, only a special case of this problem has been considered: unit-capacity graphs with no parallel edges (called *simple* graphs). Until recently, the best known algorithm for this special case used the blocking flow method of Dinitz [2], which Karzanov [15] and Even and Tarjan [3] showed runs in $O(m\min\{n^{2/3}, m^{1/2}, v\})$ time. Here $n$ is the number of nodes, $m$ is the number of edges, and $v$ is the value of the maximum flow. Note that for graphs with no parallel edges $m \leq n^2$ and for simple graphs $v \leq n$, so the above bound is $O(n^{8/3})$. In an exciting new result, Goldberg and Rao [7] extended Dinitz's algorithm to *capacitated* graphs, achieving Even-Tarjan-like bounds of $\tilde{O}(m\min\{n^{2/3}, m^{1/2}, v\}) = \tilde{O}(n^{8/3})$ time[1] on graphs whose edge capacities are polynomially bounded.

Recently, several algorithms have been developed that exploit the special properties of *undirected* graphs to get better time bounds for finding small flows. Karger [12, 14], has given several randomized algorithms culminating in an $\tilde{O}(v\sqrt{mn}) = \tilde{O}(n^{5/2})$ time bound. Note that Karger's latest algorithms do apply to graphs with capacities, although they are only useful when $v$ is small. At the same time, Goldberg and Rao [8] gave a blocking-flow based algorithm that runs in $O(n\sqrt{mn}) = O(n^{5/2})$ time on simple graphs.

The main result of this paper is inspired by the simple-graph algorithm of Goldberg and Rao [8]. They use bounds on the residual flow in a graph and a sparsification technique due to Nagamochi and Ibaraki [17] to throw away edges that need not be used by a maximum flow. We use a related idea, showing that we can find augmenting paths in $O(n\sqrt{v})$ amortized time per path by putting aside most of the edges and only bringing them back when necessary. Our approach is different from theirs in that they always keep enough edges to find all of the flow, reducing when possible, whereas we only ever work with enough edges to find a few augmenting paths, adding when necessary.

As a first application, we get simple deterministic algorithms that are faster than all previous ones for the most difficult values of $m$ and $v$ on simple graphs. First, we can find flow by augmenting paths in $O(m+nv^{3/2})$ time (substituting $O(n\sqrt{v})$ for $m$ in the classic $O(mv)$-time algorithm). Second, by incorporating a blocking flow subroutine, we can find flow in $O(nm^{2/3}v^{1/6})$ time. The first algorithm is the best known deterministic algorithm for dense graphs with small $v$; the second algorithm is the best known deterministic algorithm for dense graphs with large $v$. The second time bound is also at least as good as the Goldberg-Rao time bound of $O(n^{3/2}m^{1/2})$ for all values of $m$ and $v$. Both algorithms are clearly practical to implement, so only experiments can tell what is actually best for practical purposes. The first algorithm works for the capacitated case as well, running in $\tilde{O}(m+nv^{3/2})$ time.

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

[1] $f(n) = \tilde{O}(g(n))$ if $\exists c$ such that $f(n) = O(g(n)\log^c n)$

| Source | Year | Time bound | Capacities? | Directed? | Deterministic? |
|---|---|---|---|---|---|
| Ford-Fulkerson [4] | 1956 | $O(mv)$ | $\surd$ | $\surd$ | $\surd$ |
| Even-Tarjan [3] | 1975 | $O(m\min\{n^{2/3}, m^{1/2}\})$ | | $\surd$ | $\surd$ |
| Karger [13] | 1997 | $\tilde{O}(m^{2/3}n^{1/3}v)$ | | | |
| Goldberg-Rao [7] | 1997 | $\tilde{O}(m\min\{n^{2/3}, m^{1/2}\}\log v)$ | $\surd$ | $\surd$ | $\surd$ |
| Goldberg-Rao [8] | 1997 | $O(n\sqrt{nm})$ | | | $\surd$ |
| Karger [14] | 1998 | $\tilde{O}(v\sqrt{nm})$ | $\surd$ | | |
| this paper | 1998 | $O(m+nv^{3/2})$ | | | $\surd$ |
| this paper | 1998 | $O(nm^{2/3}v^{1/6})$ | | | $\surd$ |
| this paper | 1998 | $\tilde{O}(m+nv^{3/2})$ | $\surd$ | | $\surd$ |
| this paper | 1998 | $\tilde{O}(m+nv^{5/4})$ | $\surd$ | | |
| this paper | 1998 | $\tilde{O}(m+n^{11/9}v)$ | $\surd$ | | |

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

We also extend Karger's most recent algorithm [14], getting two Las Vegas randomized algorithms with expected running times of $\tilde{O}(m + nv^{5/4})$ and $\tilde{O}(m + n^{11/9}v)$. The latter time bound is $\tilde{O}(n^{2.\overline{2}})$ in the worst case for simple graphs, which is better than $O(n^{2.5})$, the best bound previously known [8, 14]. These algorithms are complicated, so likely not practical, but they do demonstrate that $O(n^{2.5})$ is not the right time bound for maximum flow in undirected simple graphs. Both of these algorithms also work for the capacitated case.

Even more notable, however, than the fact that $O(n^{2.5})$ is not the right time bound for flow, is the fact that $O(n^{2.\overline{2}})$ is better than the best known time bound for bipartite matching, which is $O(m\sqrt{n}) = O(n^{2.5})$. This suggests that we should be able to improve the time bound for bipartite matching! Unfortunately, the well known reduction from bipartite matching is to flow on a *directed* graph, and does not work if we try to make the graph undirected [6]. So we do not improve the time bound for bipartite matching, but this work suggests that it may be possible to do so.

Another way to look at our results is as follows. We prove that a flow of value $v$ never needs to use more than $O(n\sqrt{v})$ edges. This suggests that we should be able to restrict attention to these "important" edges, thereby effecting a replacement of $m$ by $O(n\sqrt{v})$ in the time bound of any flow algorithm. For example, our $\tilde{O}(m + nv^{5/4})$-time bound is achieved by applying this substitution to Karger's $\tilde{O}(v\sqrt{mn})$-time algorithm. Unfortunately, we do not know how to identify the right $O(n\sqrt{v})$ edges without finding a flow. Nevertheless, we devise methods to achieve all or part of this speedup on undirected graphs.

Note that Galil and Yu [5] previously proved that flows need only use $O(n\sqrt{v})$ edges on simple graphs, but they did not show how to exploit the fact. Their proof was also somewhat complex. Henzinger, Kleinberg, and Rao [9] independently simplified the proofs of Galil and Yu, using essentially the same argument we use. Our result is stronger: we show that *any* acyclic flow uses few edges, even on *capacitated* graphs.

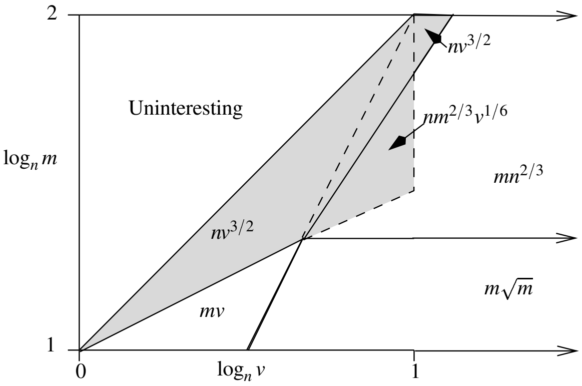


Figure 1: Pictures of the best deterministic bounds. (See text for explanation.)

In order to summarize the restrictions and performance of the various algorithms, we have done two things. Table 1 summarizes the history of the various algorithms we refer to in this paper. (The long history of $\tilde{\Theta}(mn)$-time algorithms, which are still best for large $v$, and were until recently [7] the only option for graphs with capacities, has been omitted.) Further, in order to show which algorithms have the best performance for different values of $m$ and $v$ relative to $n$, we have drawn pictures (Figures 1 and 2): one for deterministic algorithms only, and one including randomized algorithms. A point in the picture 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 [17] can always be used to make $m \le nv$ in $O(m)$ time. The shaded regions correspond to algorithms given in this paper. Note that the $O(nm^{2/3}v^{1/6})$-time algorithm (which is the fastest algorithm for the region surrounded by a dashed line) is the only one in the picture that cannot handle capacities or parallel edges, so the picture looks strange at $v = n$. 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 $v = n$. The complexity of these diagrams suggests that more progress can be made.

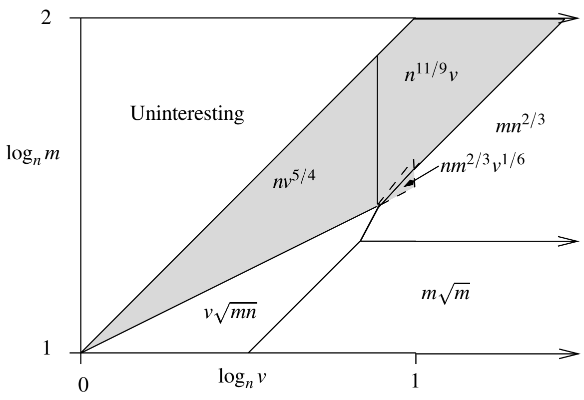


Figure 2: Pictures of the best randomized bounds. (See text for explanation.)

The rest of this paper is organized as follows. In Section 2 we review some notation and basic definitions. In Section 3 we give two algorithms for fast augmenting paths in simple graphs. In Section 4 we give two deterministic algorithms based on our fast augmenting paths subroutine. In Section 5 and 6, we apply fast augmenting paths to some randomized algorithms of Karger [12, 14]. In Section 7 we show how to extend fast augmenting paths to capacitated graphs, and discuss the implications for our other algorithms. We conclude and discuss some open questions in Section 8.

## 2 Notation and definitions

We use the following notation:

| | |
|---|---|
| $G$ | the graph |
| $s$ | the source |
| $t$ | the sink |
| $n$ | number of nodes |
| $m$ | number of edges |
| $v$ | value of a maximum flow |
| $f$ | a flow |
| $G_f$ | residual graph of $G$ with respect to $f$ |
| $d_f$ | $s$-$t$ distance in $G_f$ |
| $\lvert f\rvert$ | the value of flow $f$ |
| $E_f$ | the edges of $G$ carrying flow |

The only unusual item here is $E_f$. Two facts motivate this definition: 1) The residual graph $G_f$ is necessarily a directed graph, because flows are directed. 2) Undirected graphs have special properties that we can exploit. Since most flow algorithms work by repeatedly finding some flow and then restricting attention to the residual graph, it would seem that fact 1 renders fact 2 useless. However, the symmetry of an undirected graph is not entirely lost in $G_f$. In particular, since the capacity of a directed edge in $G_f$ is its capacity in $G$ minus the value of $f$ on that edge in that direction, it is only the edges with non-zero flow that "become directed". The unused edges still have the same capacity in both directions, so they may still be considered undirected. Therefore, in order to make good use of the properties of undirected graphs, we think of $G_f$ as having an undirected part, the unused edges, and a directed part, $E_f$.

We also use the following definitions:

**Definition 2.1** *A graph is* simple *if all edges have unit capacity and there are no parallel edges.*

**Definition 2.2** *A flow $f$ is* acyclic *if there is no directed cycle on which every edge has positive flow in the cycle direction.*

## 3 Finding augmenting paths quickly

In this section we show how to find augmenting paths in an undirected simple graph in $O(n\sqrt{v})$ amortized time per path. We focus on simple graphs, deferring discussion of graphs with capacities to Section 7. There are two facts that make our result possible. The first is that an acyclic flow in a simple graph uses only $O(n\sqrt{v})$ edges. The second is that in an undirected graph, a maximal spanning forest on the unused edges, together with the flow-carrying edges $E_f$, contains an augmenting path if there is one. So the basic idea is to maintain a maximal spanning forest $T$ of the undirected edges and use $T \cup E_f$ to search for an augmenting path in $O(n + n\sqrt{v}) = O(n\sqrt{v})$ time.

There are two ways to do this. The direct approach is to use a dynamic connectivity data structure to maintain a maximal spanning forest. The other possibility is to compute many spanning forests at once and use them for many paths, amortizing away the cost of finding the forests. We describe both approaches.

We begin by proving the structure theorems we need, and then give the details of the two approaches.

### 3.1 Structure theorems

#### 3.1.1 Flows use few edges

The first important theorem is that small flows in simple graphs use few edges:

**Theorem 3.1** *An acyclic flow $f$ in a simple graph uses at most $3n\sqrt{|f|}$ edges.*

Note that this theorem is very close to a theorem proved by Galil and Yu [5] and simplified by Henzinger, Kleinberg, and Rao [9] that says there exists a flow that uses only

$O(n\sqrt{v})$ edges. Our proof is very much the same as that of Henzinger, Kleinberg and Rao, although we proved it independently.

We use two lemmas to prove the theorem:

**Lemma 3.2** *[3] In a simple graph with a flow $f$, the maximum residual flow value is at most $2(n/d_f)^2$. (Recall that $d_f$ is the length of the shortest source-sink path in $G_f$.)*

**Proof.** Define the distance of a node to be the length of the shortest path (in the residual graph) from that node to the sink. Let $V_i$ be the set of nodes at distance $i$. Since $s$ is in $V_{d_f}$ and the sink is in $V_0$, the cut separating $\cup_{j\le i}V_j$ from $\cup_{j>i}V_j$ is an $s$-$t$ cut. Call this cut the *canonical cut* separating $V_i$ from $V_{i+1}$. Observe that a node in $V_{i+1}$ cannot have an edge to a node in $V_j$ for any $j<i$ since it would then be in $V_{j+1}$. So edges leaving $V_{i+1}$ can only go to $V_j$ with $j\ge i$. Since there are no parallel edges, the number of edges crossing the canonical cut separating $V_{i+1}$ from $V_i$ is at most $|V_{i+1}||V_i|$.

Now consider the $V_i$ in pairs: $V_0\cup V_1, V_2\cup V_3,\ldots$. There are $\lfloor (d_f+1)/2\rfloor$ such pairs, and they are vertex disjoint, so some pair has at most $2n/d_f$ vertices in it. The canonical cut separating this pair has at most $\max_x(x)(2n/d_f-x)=(n/d_f)^2$ edges crossing it. Each edge of the residual graph has capacity at most 2 (one original unit and possibly one more if it is carrying flow in the wrong direction), so the maximum residual flow value is $2(n/d_f)^2$. ■

**Lemma 3.3** *(Small modification to Theorem 6 in [3]) In a simple graph, if a flow $|f|$ is found by repeatedly finding and augmenting on a shortest path in $G_f$, then the total length of the paths is at most $3n\sqrt{|f|}$.*

**Proof.** Restating Lemma 3.2, we have that when $x$ flow remains in $G_f$, the length of the shortest source-sink path in $G_f$ is at most $n\sqrt{2/x}$. In the execution of any augmenting path algorithm, $x$ takes on each value from 1 to $|f|$ once, so if we always use the shortest augmenting path in $G_f$ we see that the total length of the paths is

$$\sum_{x=1}^{|f|}\frac{n\sqrt{2}}{\sqrt{x}}\le 3n\sqrt{|f|}$$

■

**Proof of Theorem 3.1.** Consider $E_f$. By definition, $f$ is an $s$-$t$ max-flow in $E_f$ of value $|f|$. Further, since $f$ is acyclic, there can be no residual cycle in $E_f$, so the maximum flow in $E_f$ is unique. That is, $f$ is the only max-flow in $E_f$. By Lemma 3.3, if we were to find a max-flow in $E_f$ by shortest augmenting paths, the total length of these paths would be at most $3n\sqrt{|f|}$, meaning that at most $3n\sqrt{|f|}$ edges were used. But this (unique) max-flow is $f$, so $f$ uses at most $3n\sqrt{|f|}$ edges. ■

Observe that Theorem 3.1 is tight up to constant factors. Figure 3 gives an example of a graph with an acyclic maximum flow that uses $\Theta(n\sqrt{v})$ edges.

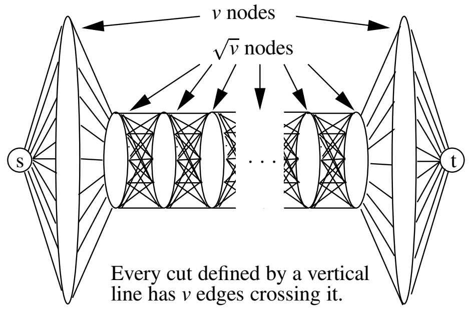


Figure 3: A graph with an acyclic flow that uses $O(n\sqrt{v})$ edges

### 3.1.2 Unused edges can be reduced to a spanning forest

The second important structure theorem is that the flow-carrying edges together with a maximal spanning forest of unused edges have an augmenting path if and only if $G_f$ does:

**Theorem 3.4** *Let $T$ be any maximal spanning forest of $G_f - E_f$. Then $T\cup E_f$ has an augmenting path if and only if $G_f$ does.*

**Proof.** Let $G'=T\cup E_f$. Since $G'$ is a subgraph of $G_f$, it is clear that if $G'$ has an augmenting path then $G_f$ does. For the other direction, suppose that there is an augmenting path in $G_f$, but not in $G'$. By the max-flow min-cut theorem, we can restate this condition as follows: there is an $s$-$t$ cut $C$ that has a residual edge $e$ crossing it (from the $s$ to the $t$ side) in $G_f$, but no edges crossing it in $G'$. If $e$ is in $E_f$, then it is in $G'$, a contradiction. So $e$ must be in $G_f-E_f$. But $T$ is a maximal spanning forest of $G_f-E_f$, which means that it contains an edge from every nonempty cut of $G_f-E_f$. Since $C$ is nonempty in $G_f-E_f$ ($e$ crosses it) some edge of $T$, and thus of $G'$, crosses $C$. This contradicts our (restated) original assumption. ■

With these two results, we can now give some algorithms.

## 3.2 An algorithm based on a dynamic connectivity data structure.

In this section we show how to exploit Theorem 3.4 in the most literal way: by maintaining an acyclic flow $E_f$ and a maximal spanning forest of $G_f-E_f$. The most important piece of this implementation is a data structure for dynamic connectivity:

**Lemma 3.5** *[10] It is possible to maintain a maximal spanning forest of an undirected graph under edge insertions and deletions in $O(\log^2 n)$ amortized time per operation.*

We also need to worry about whether our flow is acyclic, because Theorem 3.1 only applies if it is. Fortunately, using a procedure due to Sleator and Tarjan [18], it is easy to remove all cycles from a flow (we will refer to this procedure as *decycling*). Since we are largely concerned with the simpler case of unit-capacity graphs, we observe that their algorithm minus the dynamic trees works a little faster in a unit-capacity graph:

**Lemma 3.6** *In a unit-capacity graph, it is possible to take a flow $f$ and find an acyclic flow $f'$ of the same value ($|f'| = |f|$) in $O(|E_f|)$ time.*

**Proof.** Do a depth first search from the source on edges carrying flow. Whenever the sink is reached, retreat. Whenever a back edge (an edge leading to vertex already on the current depth-first search path) is found, we have found a cycle. Delete the cycle and continue the search from the head of the back edge. (The head of the back edge is the node furthest from the source and still on the current depth-first search path.) Deleting a cycle leaves a flow of the same value. The search only advances over each edge once, and only deletes each edge once, so it takes $O(|E_f|)$ time. Note that it is easy to show by contradiction that there are no cycles left in $E_f$ when this procedure terminates. ■

We can now give the basic algorithm for fast augmenting paths:

```
SparseAugment1(G, f)

Input: Graph G, flow f
Output: maximum flow in G

  insert all edges of G that are not carrying flow
    into a dynamic connectivity data structure,
    and use it to maintain a maximal spanning forest T
  repeat:
    look for an augmenting path in E_f ∪ T
    if no such path exists
      return f
    else
      augment f using the path
      f ← decycle(f)
      update the connectivity structure as appropriate
```

Note that in practice we might decycle the flow only when it has many edges. To show that this algorithm is correct, we just need to know that $G'$ contains an augmenting path if and only if $G_f$ does. This result is immediate by Theorem 3.4. It remains to analyze the running time.

**Theorem 3.7** *SparseAugment1 runs in $O((m+rn)\log^2 n + rn\sqrt{v})$ time, where $r = v - |f|$ is the number of augmenting paths that need to be found.*

**Proof.** For now ignore the dynamic connectivity operations. Since we decycle the flow in each iteration, every augmenting path search takes place in a graph with $O(n\sqrt{v})$ edges and therefore takes $O(n\sqrt{v})$ time. Similarly, every decycling takes $O(n\sqrt{v})$ time. Since there are $r$ iterations, the total time is $O(rn\sqrt{v})$.

It remains to account for the dynamic connectivity operations. First consider deletions. An edge is deleted from the data structure when we place flow on it. This happens to at most $n$ edges in any one augmenting path, for a total of $nr$ deletions taking $O(nr\log^2 n)$ time. Now consider insertions. Initially, we insert all edges in the structure in $O(m\log^2 n)$ time. Later, edges are inserted in the data structure when flow is removed from them. Note, however, that flow cannot be removed from an edge until flow has been added to the edge, We have already counted the cost of deleting edges when we add flow to them; this cost can also absorb the equal cost of inserting those edges when the flow is removed. ■

### 3.3 An algorithm based on sparse connectivity certificates

Another way to exploit Theorem 3.4 is to find several spanning forests at once and use them to find several augmenting paths, thus achieving the same average time per augmenting path. To do this, we use an idea and algorithm given by Nagamochi and Ibaraki [17]:

**Definition 3.8** *For an undirected graph $G = (V,E)$, a* sparse connectivity certificate *is a partition of $E$ such that $E_i$ is a maximal spanning forest in $G - E_1 \cup E_2 \cup \cdots \cup E_{i-1}$, for $i = 1,2,\ldots,|E|$, where possibly $E_i = E_{i+1} = \cdots = E_{|E|} = \emptyset$ for some $i$.*

**Definition 3.9** *A* sparse $k$-certificate *is the subgraph $G_i = (V, E_1 \cup E_2 \cup \cdots \cup E_k)$ derived from a sparse connectivity certificate.*

**Lemma 3.10** *[17] The value of a minimum s-t cut in a sparse $k$-certificate $G_k$ of $G$ is equal to the smaller of $k$ and the value of the minimum s-t cut in $G$.*

**Lemma 3.11** *[17] In an undirected graph with unit capacity edges, it is possible to construct a sparse connectivity certificate in $O(m)$ time.*

Notice that one easy application of this construction is to reduce $m$ to $nv$. By Lemma 3.10, using a sparse $nv$-certificate does not reduce the value of any $s$-$t$ cut below $v$, so a maximum flow in the certificate is a maximum flow in the original graph. This gives an $O(m+nv^2)$-time flow algorithm using standard augmenting paths.

This construction turns out to be precisely what we want. We formalize this idea with the following generalization of Theorem 3.4:

**Theorem 3.12** *Let $G_k$ be a sparse $k$-certificate of $G_f - E_f$. Then $E_f \cup G_k$ contains $i < k$ augmenting paths if and only if $G_f$ has $i$ augmenting paths, and $E_f \cup G_k$ contains at least $k$ paths if $G_f$ contains at least $k$.*

**Proof.** The idea here is the same as that of Theorem 3.4, except that now we have several spanning forests instead of one. Again $G' = E_f \cup G_k$ is a subgraph of $G_f$, so can have no more augmenting paths than $G_f$. For the other direction, consider a minimum $s$-$t$ cut of $G'$. Suppose $G_f$ has more residual edges crossing this cut, that is, has an edge crossing the cut that is not in $G'$. It is impossible for this edge to be in $E_f$, because $G'$ contains all edges of $E_f$. So there must be more unused edges crossing the cut in $G_f - E_f$ than in $G_k$. But by Lemma 3.10, this can only happen if more than $k$ edges cross the cut in $G_f - E_f$, in which case at least $k$ edges must cross the cut in $G_k$. This completes the proof. ■

We now give the basic algorithm using sparse certificates:

```
SparseAugment2(G, f)

  k ← ⌈√(m/n)⌉
  repeat:
     f ← decycle(f)
     G_k ← a sparse k-certificate of unused edges of G
     G' ← E_f ∪ G_k
     run augmenting paths on G' until
        k paths are found or no more paths exist
     if the previous step found less than k paths
        return f
```

To show the correctness of this algorithm, we just need to know that when we find less than $k$ augmenting paths in $G'$, we have a maximum flow in $G$. This is immediate from Theorem 3.12. It remains to analyze the running time.

**Lemma 3.13** *The running time of SparseAugment2$(G,f)$ on a simple graph is $O(m + r(n\sqrt{v} + \sqrt{mn}))$, where $r$ is the number of augmenting paths that need to be found.*

**Proof.** By Lemma 3.11 and Lemma 3.6, the cost per iteration of the first two steps in the loop is $O(m)$. The cost of the augmenting paths step is $O(m'k)$, where $m'$ is the number of edges in $G'$. By definition of a sparse $k$-certificate and Theorem 3.1, $m' \le nk + n\sqrt{v} = \sqrt{mn} + n\sqrt{v}$. The number of iterations is $\lceil r/k \rceil$, so the total time is $O((m + m'k)\lceil r/k \rceil) = O(m + r(n\sqrt{v} + \sqrt{mn}))$. ■

This bound is somewhat unsatisfactory, in that the cost per augmenting path becomes $\sqrt{mn}$ when $m \ge nv$. But if we knew $v$ at the beginning, we could find a sparse $v$-certificate and ensure that we only worked with $nv$ edges for the rest of the algorithm. This would give the amortized $O(n\sqrt{v})$ time per path that we want. A complicated way to solve this problem is to use the graph compression technique of Benczúr and Karger [1] to get a 2-approximation to $v$ in $\tilde{O}(m + nv)$ time. A simpler approach is to simulate knowing $v$ by taking a small guess and doubling it until we are correct:

```
SparseAugment3(G, f)

  compute a sparse connectivity certificate of
     unused edges of G
  For any w, let G_w denote the first w forests of this
     sparse certificate (a sparse w-certificate)
  w ← |f|
  repeat:
     w ← min w' such that |G_w'| > 2|G_w|
     SparseAugment2(G_w, f), stopping when |f| ≥ w
  until |f| < w
  return f
```

Notice that $G_w \subset G_{2w}$, so we do not start over each iteration, we just continue with more of the edges from $G$. This is irrelevant to the time bound, but seems likely to yield better constant factors in practice.

**Theorem 3.14** *The running time of SparseAugment3$(G,f)$ on a simple graph is $O(m + rn\sqrt{v})$, where $r$ is the number of augmenting paths that need to be found.*

**Proof.** The running time of the initial step is $O(m)$. The running time of the $i$th iteration is $O(m_i + r_i(n\sqrt{v} + \sqrt{m_i n}))$ by Lemma 3.13. (Here the notation $x_i$ is used to mean the value of $x$ in the $i$th iteration.) Since $m_i$ doubles with each iteration, the sum over iterations of the first term is $O(m)$. Let $k$ be the number of iterations. It must be the case that $w_{k-1} \le v$ in order for the $(k-1)^{st}$ iteration to not terminate. Thus $m_{k-1} \le nv$. Since we attempt to double $m_i$, ending up with at most one spanning forest too many, $m_k \le 2nv + n = O(nv)$. Since $\sum r_i = r$, the sum over iterations of the second term is $O(rn\sqrt{v})$. The total is $O(m + rn\sqrt{v})$. ■

## 4 Applications of fast augmenting paths

The main result of Section 3 can be used in several ways to give fast flow algorithms. Most obviously, direct application of SparseAugment3 gives a simple, deterministic $O(m + nv^{3/2})$-time flow algorithm. In the worst case, when $m = \Theta(n^2)$ and $v = \Theta(n)$, this gives an $O(n^{5/2})$ time bound, which is as good as all previous known algorithms'. For smaller $v$ this is the best deterministic algorithm known. Note that ours is the first deterministic algorithm to achieve this bound without blocking flows, and unlike previous blocking flow approaches it benefits from small $v$. If we do use blocking flows, we can do better for large $v$:

```
BlockThenAugment (G, k)

  f ← the result of computing blocking flows on
     shortest paths in G_f until d_f ≥ k
  return SparseAugment3(G, f)
```

**Theorem 4.1** *On an undirected simple graph BlockThenAugment $(G, nv^{1/6}/m^{1/3})$ runs in $O(nm^{2/3}v^{1/6}) = O(n^{5/2})$ time.*

**Proof.** Finding a blocking flow takes $O(m)$ time. We compute at most $k$ blocking flows, which takes $O(mk) = O(nm^{2/3}v^{1/6})$ time. We then have $d_f \geq k$, so by Lemma 3.2 the remaining flow is $O((n/k)^2)$. Thus the time for the second step is $O(n^3\sqrt{v}/k^2)$, which is also $O(nm^{2/3}v^{1/6})$. ∎

This algorithm also takes $O(n^{5/2})$ time in the worst case, but it is better when the graph is sparse but the flow value is large. It is always at least as good as the bound of $O(n^{3/2}m^{1/2})$ given by Goldberg and Rao [8], and in general better by a factor of $(n^3/mv)^{1/6}$.

Note that unlike Dinitz's algorithm, where the improved running time arose by changing the *analysis* of the algorithm to augmenting paths at a certain point, we must explicitly change the *execution* of the algorithm at a certain point to achieve our bounds. Since our algorithm must change its actions, we need to know what that point is. In particular, we need to know $v$ in order to achieve our bound. We can again get around this limitation by either estimating $v$ with another algorithm and computing a sparse certificate or using the iterative doubling trick of SparseAugment3.

## 5 New tricks for an old DAUG

Using our fast augmentation, we can also improve the running time of the "divide and augment" algorithm (DAUG) given by Karger [12]. This result is of relatively minor interest in itself, but we make good use of it in the next section.

The idea of DAUG is that if we randomly divide the edges of a graph into two groups, then about half of the flow can be found in each group. So we can recursively find a maximum flow in each half, put the halves back together, and use augmenting paths to find any flow that was lost because of the division. In the original version, the time spent finding augmenting paths at the top level dominated the running time, so it is natural to expect an improvement with faster augmentations. Here is the original algorithm:

```
DAUG(G)

    if G has no edges, return the empty flow
    randomly divide the edges of G into two groups,
        giving G1 and G2
    f1 ←DAUG(G1)
    f2 ←DAUG(G2)
    f ← f1 + f2
(*) use augmenting paths to turn f into a maximum flow
    return f
```

The key fact that makes DAUG work is that random sampling preserves cut values fairly well as long as all cuts are large enough:

**Definition 5.1** *A graph is $c$-*connected *if the value of each cut is at least $c$.*

**Theorem 5.2** *[12] If $G$ is $c$-connected and edges are sampled with probability $p$, then with high probability all cuts in the sampled graph are within $(1 \pm \sqrt{8\ln n/pc})$ of their expected values.*

Thus when we divide the edges into two groups (effecting $p = 1/2$ in each group), the minimum $s$-$t$ cut in each group is at least $\frac{v}{2}(1 - O(\sqrt{\log n/c}))$. So the flow in each half has at least this value, giving us a flow of value at least $v(1 - O(\sqrt{\log n/c}))$ when we put the two halves together. This leaves only $O(v\sqrt{\log n/c})$ augmenting paths to be found in Step (*). It turns out that this step is the dominant part of the running time (the time bound for DAUG is $O(mv\sqrt{\log n/c})$), so it makes sense to use SparseAugment. We refer to this new algorithm as newDAUG.

Now, by Theorem 3.14, the time to find the augmenting paths is $O(m + nv\sqrt{v\log n/c})$. So a recurrence for the running time of newDAUG is

$$T(m,v,c) = 2T(m/2, v/2, c/2) + O\left(m + nv\sqrt{v\log n/c}\right)$$

This solves to $\tilde{O}(m + nv\sqrt{v/c})$, but unfortunately, because of the randomization in the algorithm, the problem reduction is expected, not guaranteed, so solving this recurrence does not actually prove anything about the running time of newDAUG. We need to look at the recursion tree (See [12] for a full discussion). This proof is more technical than interesting, and goes the same way as in [12], so we just sketch it.

**Theorem 5.3** *The running time of newDAUG on a $c$-connected graph is $\tilde{O}(m + nv\sqrt{v/c})$.*

**Proof.** (Sketch) As in the original algorithm, the depth of the recursion tree is $O(\log m)$, and the time spent looking unsuccessfully for augmenting paths is $O(m\log m)$. It remains to bound the time spent in successful augmentations. Consider a recursion node $N$ at depth $d$. Each edge of the original graph ends up at $N$ independently with probability $2^{-d}$, so the graph at this node is equivalent to one obtained by sampling with probability $2^{-d}$.

Consider the nodes at depths exceeding $\log(c/\log n)$. By Theorem 5.2, at these nodes the flow is $\tilde{O}(v/c)$. So by Theorem 3.14, the total time spent on successful augmenting paths is $\tilde{O}(nv\sqrt{v/c})$. At the nodes at depth $d \leq \log(c/\log n)$, the argument from [12] continues to apply, showing that the number of augmenting paths that need to found is $O(v\sqrt{\log n/2^d c})$. Since the value of the flow is $O(v/2^d)$, the time taken is $\tilde{O}((v\sqrt{1/c})n\sqrt{v/2^d}) = \tilde{O}(nv\sqrt{v/c}/2^d)$. Adding this up over the whole recursion, we get the claimed bound. ∎

Note that this time bound is very good if $v$ is not much bigger than $c$. In particular, we get the following easy corollary:

**Corollary 5.4** *In a simple graph where $v = \tilde{O}(c)$, the running time of newDAUG is $\tilde{O}(m + nv) = \tilde{O}(m)$. (Note that $m \geq nc/2$ in a $c$-connected simple graph.)*

## 6 $\tilde{O}(m+nv^{5/4})$- and $\tilde{O}(m+n^{11/9}v)$-time algorithms

The algorithm of the previous section is only an improvement over the $O(m+nv^{3/2})$-time algorithm if $c$ is large. Nevertheless, we can take advantage of it by using ideas from [14]. In that paper, a number of ideas are put together to get a fast flow algorithm, CompressAndFill, that runs in $\tilde{O}(v\sqrt{mn})$ time on any undirected graph. For our purposes, that algorithm can be summarized with the following theorem:

**Theorem 6.1** *[14] Let $T(m,n,v,c)$ denote the time to find a maximum flow of value $v$ in a $c$-connected undirected graph with $m$ edges and $n$ nodes. Given flow algorithms $A_1$ and $A_2$, ($A_1$ must handle capacities), with running times $T_1$ and $T_2$ respectively, it is possible to define a flow algorithm $A_3$ with expected running time (up to log factors) given by*

$$\begin{aligned} T_3(m,n,v,c) \le T_1(nk,n,v,c) + T_2(m,n,v,k) + T_2(m,n,k,k) \\ + \text{ time to find } O(v/\sqrt{k}) \text{ augmenting paths} \end{aligned}$$

*(There is a technicality that the bound of $T_2$ must be "reasonable"—at least linear in $n$ or $m$— for this theorem to be true.)*

CompressAndFill results from picking $k \approx m/4n$, using CompressAndFill (recursively) for $A_1$, and using DAUG (with runtime $\tilde{O}(mv/\sqrt{k})$ for $A_2$. Thus the recurrence for the running time is

$$\begin{aligned} T(m,n,v,c) &\le T(m/2,n,v,c) + \tilde{O}(mv\sqrt{k}) \quad +\tilde{O}(m\sqrt{k}) + \\ &\qquad \tilde{O}(mv\sqrt{k}) \\ &\le T(m/2,n,v,c) + \tilde{O}(v\sqrt{mn}) \\ &\le \tilde{O}(v\sqrt{mn}) \end{aligned}$$

We improve on this algorithm by replacing the subroutines $A_1$ and $A_2$ and the augmenting path step appropriately. In particular, we use newDAUG instead of DAUG for $A_2$ and we find augmenting paths at the end with SparseAugment. We also consider two possibilities for $A_1$: CompressAndFill and the $\tilde{O}(mn^{2/3})$-time algorithm of Goldberg and Rao. Note that we investigated using a recursive strategy again, but we were unable to get an improvement that way.

**Theorem 6.2** *On undirected simple graphs, we can find a maximum flow in expected time $\tilde{O}(m+nv^{5/4})$.*

**Proof.** Use Theorem 6.1 with $A_1$ =CompressAndFill, $A_2$ = newDAUG, and SparseAugment to find the augmenting paths at the end. The resulting time bound is

$$\begin{aligned} \tilde{O}(v\sqrt{(nk)n}) + \tilde{O}(nv^{3/2}/k^{1/2}) + \tilde{O}(nv) + \tilde{O}(n\sqrt{v}\cdot v/\sqrt{k}) \\ = \tilde{O}(vn\sqrt{k} + nv^{3/2}/k^{1/2}) \end{aligned}$$

Picking $k = \sqrt{v}$ completes the proof. ∎

**Theorem 6.3** *On undirected simple graphs, we can find a maximum flow in expected time $\tilde{O}(m+n^{11/9}v)$.*

**Proof.** Use Theorem 6.1 with $A_1$ = the $\tilde{O}(mn^{2/3})$-time algorithm of Goldberg and Rao [7], $A_2$ = newDAUG, and SparseAugment to find the augmenting paths at the end. The time is

$$\begin{aligned} \tilde{O}((nk)n^{2/3}) + \tilde{O}(nv^{3/2}/k^{1/2}) + \sum \tilde{O}(nv) + \tilde{O}(n\sqrt{v}\cdot v/\sqrt{k}) \\ = \tilde{O}(kn^{5/3} + nv^{3/2}/k^{1/2}) \end{aligned}$$

Picking $k = v/n^{4/9}$ completes the proof. ∎

## 7 Extensions to graphs with capacities

In this section we show that much of what we have already shown for simple graphs actually applies to graphs arbitrary integer capacities. The key fact is that Theorem 3.1 continues to hold:

**Theorem 7.1** *An acyclic flow $f$ in a graph with integer capacities and no parallel edges uses at most $2n\sqrt{|f|}$ edges.*

Besides extending to capacitated graphs, this theorem yields better constants, even for the simple-graph case, than the similar theorems of Galil and Yu [5] and Henzinger et al [9]. The lower-bound example of Figure 3 shows that our bound is tight to within a factor of 2.

Notice also that restricting a capacitated graph to have no parallel edges is no restriction at all, because in time linear in the input we can merge parallel edges into one edge with capacity equal to the sum of the capacities of the edges that make it up, and at the end we can split the flow on such an edge among the edges that make it up.

Our proof of Theorem 3.1 bounded the number of edges used by a flow by breaking it down into augmenting paths and counting their total length. That argument does not work, because a single path of length $n$ and capacity $v$ would cause the total length of augmenting paths to be $nv$. However, a very similar argument does work. The problem is that one edge can be in many paths, so our old proof counts it many times. The idea of the new proof is to redefine the length of an edge so that the total length of augmenting paths gives a more accurate bound. Specifically, define the length of a residual edge to be 1 if it has unit capacity and 0 if its capacity is larger. Again we begin with a lemma:

**Lemma 7.2** *In a graph with flow $f$ that has no parallel edges, the maximum residual flow value is at most $(n/d_f)^2$, where $d_f$ is the length of the shortest (with respect to the length function defined above) source-sink path in $G_f$.*

**Proof.** The argument used in the proof of Lemma 3.2 continues to imply that there is a canonical cut with only $(n/d_f)^2$ edges crossing it. The only difference now is that the edges of $G_f$ are not limited to capacity 2. However, no length 0 edge can cross a canonical cut from the $s$ side to the $t$ side, because that would violate the definition of the cut. (A node $w$ at distance $i$ from the sink cannot possibly have a length-0 edge to a node at distance less than $i$, because $w$ would then be at distance less than $i$.) Therefore only length 1—that is, capacity 1—edges cross canonical cuts, so the residual flow value is at most $(n/d_f)^2$. ■

**Proof of Theorem 7.1.** As in the proof of Theorem 3.1, we consider finding a max-flow in $E_f$. To do this, define a graph $G' = (V, E_f)$ where the capacity of an edge is equal to the value of $f$ on it. Again, since $E_f$ has no cycles, the maximum flow in $G'$ is unique and therefore must use all the capacity of all the edges.

Consider finding a maximum flow in $G'$ by repeatedly finding and augmenting one unit of flow on a shortest path (with respect to the length function above) in $G'_f$. Lemma 7.2 tells us that the length of the path is at most $n/\sqrt{x}$. In the execution of any augmenting path algorithm, $x$ takes on each value from 1 to $|f|$ once, so if we always use the shortest augmenting path we see that the total length of the paths is

$$\sum_{x=1}^{|f|} \frac{n}{\sqrt{x}} \leq 2n\sqrt{|f|}$$

Since every edge is reduced to 0 capacity at the end, every edge has length 1 at least one of the times it is on an augmenting path. It follows that the total length of the augmenting paths is an upper bound on the number of edges used by $f$.

Note that during the augmenting path algorithm the lengths of edges can change in unpredictable ways, but this does not affect our analysis. All we care is that each edge has length 1 during at least one augmentation through it. ■

Given that $|E_f|$ is still small for a capacitated graph, we need to make sure that we can still decycle and that our methods to sparsify the unused edges still work. Fortunately, the original Sleator-Tarjan decycling algorithm [18] already takes care of capacitated graphs, and a later paper of Nagamochi and Ibaraki [16] says that we can still find sparse certificates quickly.

**Lemma 7.3** *[18] It is possible to take a flow $f$ and find an acyclic flow $f'$ of the same value ($|f'| = |f|$) in $O(|E_f| \log n)$ time.*

**Lemma 7.4** *[16] In an undirected graph, it is possible to construct a sparse connectivity certificate in $O(m + n \log n)$ time.*

It follows immediately that we can find augmenting paths in a capacitated graph in amortized $\tilde{O}(n\sqrt{v})$ time. Almost all of our simple-graph time bounds extend as easy corollaries.

**Theorem 7.5** *In an undirected graph, it is possible to find $r$ augmenting paths in $\tilde{O}(m + rn\sqrt{v})$ time.*

**Corollary 7.6** *A maximum flow in an undirected graph can be found in $\tilde{O}(m + nv^{3/2})$ time.*

**Corollary 7.7** *A maximum flow in an undirected graph can be found in $\tilde{O}(m + nv^{5/4})$ expected time.*

**Corollary 7.8** *A maximum flow in an undirected graph can be found in $\tilde{O}(m + n^{11/9}v)$ expected time.*

Notice that BlockThenAugment does not extend, because it relies on Lemma 3.2 to bound the remaining flow after several blocking flow computations. However, the remaining algorithms do extend. In [14], Karger shows how to extend DAUG to graphs with capacities. Ignoring the details, the bottom line is that $m$ has to be increased to $m + nc$. The time bound for newDAUG is independent of $m$, so it remains $\tilde{O}(nv\sqrt{v/c})$. CompressAndFill was originally designed to work with capacities, so given that newDAUG and fast augmenting paths continue to work with the same time bounds (up to logarithmic factors), our algorithms of Section 6 do as well.

## 8 Conclusion

We have given algorithms that improve the time bounds for maximum flow in undirected graphs. However, our results seem to open more questions than they resolve. By giving an algorithm that runs in $\tilde{O}(m + n^{11/9}v) = \tilde{O}(n^{2.\bar{2}})$ time, we show that $O(n^{2.5})$ is not the right time bound for maximum flow in undirected simple graphs. Further, for the case when $v = \tilde{O}(c)$, we give an algorithm that runs in $\tilde{O}(m)$ time. This reopens the question of what the right time bound is. The hope that the time bound has a simple form leads us to conjecture that it is possible to find flows in $\tilde{O}(m + nv)$ time, which is $\tilde{O}(n^2)$ on simple graphs.

We have also shown that maximum flow in undirected simple graphs can be found faster than bipartite matching. As discussed before, the standard reduction of bipartite matching to flows is to directed flows, so our techniques do not help. This opens the question of whether bipartite matching can be reduced to undirected flow or, more generally, whether the time for bipartite matching is really correct.

It is also natural to ask whether our techniques can be extended further. The best performance improvement we could hope for from our present techniques is reduction of $m$ to $n\sqrt{v}$; we achieve this reduction for augmenting paths, but only get part-way when blocking flows are involved. It would be nice to find a way to sparsify for a blocking flow computation. In particular, if we could achieve a full reduction to $n\sqrt{v}$ edges when blocking flows were involved, it would imply an $O(n\sqrt{v}n^{2/3}) = O(n^{2.1\bar{6}})$-time algorithm. Further, the structure theorem, that a flow does not use many edges, holds for directed graphs, but our sparsification techniques do not. It would be nice to close the gap between directed and undirected graphs.

## Acknowledgments

We thank Allen Knutson and Joel Rosenberg for assistance in proving Theorem 7.1.

## References


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

[2] E. A. Dinitz. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[3] S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM Journal on Computing*, 4:507–518, 1975.

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

[5] Z. Galil and X. Yu. Short length versions of Menger's theorem (extended abstract). In *Proceedings of the* $27^{th}$ *ACM Symposium on Theory of Computing*, pages 499–508. ACM, ACM Press, May 1995.

[6] A. Goldberg. Personal communication, Oct. 1997.

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

[8] A. Goldberg and S. Rao. Flows in undirected unit capacity networks. In *Proceedings of the* $30^{th}$ *Annual Symposium on the Foundations of Computer Science* [11], pages 32–35.

[9] M. R. Henzinger, J. Kleinberg, and S. Rao. Short-length Menger theorems. Technical Report 1997-022, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, 1997.

[10] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic graph algorithms I: Connectivity and minimum spanning tree. Technical Report DIKU-TR-97/17, University of Copenhagen, 1997. To appear in STOC 1998.

[11] IEEE. *Proceedings of the* $30^{th}$ *Annual Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, Oct. 1997.

[12] D. R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 1998. To appear. A preliminary version appeared in STOC 1994.

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

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

[15] A. V. Karzanov. O nakhozhdenii maksimal′nogo potoka v setyakh spetsial′nogo vida i nekotorykh prilozheniyakh. In *Matematicheskie Voprosy Upravleniya Proizvodstvom*, volume 5. Moscow State University Press, Moscow, 1973. In Russian; title translation: On Finding Maximum Flows in a Network with Special Structure and Some Applications.

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

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

[18] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.