Max flows in $O(nm)$ time, or better

James B. Orlin*

Revised: July 25, 2012

Abstract

In this paper, we present improved polynomial time algorithms for the max flow problem defined on a network with n nodes and m arcs. We show how to solve the max flow problem in $O(nm)$ time, improving upon the best previous algorithm due to King, Rao, and Tarjan, who solved the max flow problem in $O(nm \log_{m/(n \log n)} n)$ time. In the case that $m = O(n)$, we improve the running time to $O(n^2/\log n)$.

We further improve the running time in the case that $U^* = U_{\max}/U_{\min}$ is not too large, where $U_{\max}$ denotes the largest finite capacity and $U_{\min}$ denotes the smallest non-zero capacity. If $\log(U^*) = O(n^{1/3} \log^{-3} n)$, we show how to solve the max flow problem in $O(nm/\log n)$ steps. In the case that $\log(U^*) = O(\log^k n)$ for some fixed positive integer k, we show how to solve the max flow problem in $\widetilde{O}(n^{8/3})$ time. This latter algorithm relies on a subroutine for fast matrix multiplication.

1 Introduction

Network flow problems form an important class of optimization problems and are central problems in operations research, computer science and combinatorial optimization. A special network flow problem, the max flow problem, has been widely investigated since the seminal research of Ford and Fulkerson in the 1950s. The max flow problem has applications in transportation, logistics, telecommunications, and scheduling. Numerous efficient algorithms for this problem exist including [7] and [4]. A comprehensive discussion of such algorithms and applications can be found in [1].

We consider the max flow problem on a directed graph with n nodes, m arcs, and integer valued arc capacities (possibly infinite), in which the largest finite capacity is bounded by U. The fastest strongly polynomial time algorithm is due to King et al. [21]. Its running time is $O(nm \log_{m/(n \log n)} n)$. When $m = \Omega(n^{1+\epsilon})$ for any positive constant ϵ, the running time is $O(nm)$. When $m = O(n \log n)$, the running time is $O(nm \log n)$. The fastest weakly polynomial time algorithm is due to Goldberg and Rao [16]. Their algorithm solves the max flow problem as a sequence of $O(\log U)$ scaling phases, each of which transforms a Δ-optimal flow into a $\Delta/2$-optimal flow. The running time per scaling phase is $O(\Lambda m \log(n^2/m))$, where $\Lambda = \min\{n^{2/3}, m^{1/2}\}$.

Our contribution. We show that the max flow problem can be solved in $O(nm + m^{31/16} \log^2 n)$ time. When $m = O(n^{(16/15)-\epsilon})$, this running time is $O(nm)$. Because the algorithm by King et al. [21] solves the max flow problem in $O(nm)$ time for $m > n^{1+\epsilon}$, our improvement establishes that the max flow problem can be solved in $O(nm)$ time for all n and m.

*Sloan School of Management and Operations Research Center, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, *E-mail:* jorlin@mit.edu

We also develop an $O(n^2/\log n)$ algorithm for max flow problems in which $m = O(n)$.

Our algorithm solves the max flow problem as a sequence of improvement phases, similar to the scaling phases in the Goldberg-Rao algorithm. We obtain a strongly polynomial time algorithm by replacing the residual network of the Δ-improvement phase by a more compact representation. The bottleneck operation for our algorithms is the creation of the compact representation. The other bottleneck operation is the transformation of flows in the compact network to flows in the residual network.

In addition, we present improved polynomial time algorithms for the max flow problem under several different parameter settings. Let $U^* = (U_{\max}/U_{\min})$, where $U_{\max}$ is the largest finite capacity, and $U_{\min}$ is the smallest non-zero capacity. If U^* is not too large (e.g., $\log U^* = O(n^{1/3}/\log^3 n)$), then one can solve the max flow problem in strongly polynomial time by first using the Goldberg-Rao algorithm to obtain a Δ-optimal flow for $\Delta = U_{\min}/2$, and subsequently using our strongly polynomial time algorithm to transform the Δ-optimal flow into an optimal flow. Suppose that we let $T(n, m)$ denote the running time to find an optimal flow starting with the Δ-optimal flow. We show that

1. $T(n, m) = O(nm/\log n)$ for all n and m.
2. $T(n, m) = \widetilde{O}(n^{17/12}m^{5/8} + n^{1+2\omega/3}) = \widetilde{O}(n^{8/3})$.

Our algorithm relies on fast matrix multiplication, which runs in $O(n^\omega)$ time, where ω is 2.3727. This bound is due to Williams [24], which improves upon the previous bound due to Coppersmith and Winograd [8]. The time to find the Δ-optimal flow is $\widetilde{O}(n^{2/3}m)$. The time it takes to find an optimal flow is $\widetilde{O}(n^{8/3})$. (The $\widetilde{O}$ bounds ignore factors that are polynomial in $\log n$.) In the case that $m = \Omega(n^2)$, this bound is a factor $n^{1/3}$ faster than the best previous strongly polynomial time max flow algorithm.

Our paper is organized as follows. In Section 2, we provide preliminary notation and definitions. In Section 3, we describe how the max flow problem is solved as a sequence of improvement phases. In Section 4, we describe the abundance graph and how contraction can speed up the algorithm. In Section 5, we show how abundant directed cycles as well as some other arcs may be contracted so as to result in a smaller max flow problem. In Section 6, we explain how nodes incident to only abundant arcs may be "compacted". Compaction is a concept that is new to this paper. In Section 7, we show how to run the improvement phase for the max flow problem on the compact network (when appropriate) rather than on the original network. We reduce the total running time to $O(nm + m^{31/16}\log^2 n)$, which is $O(nm)$ time if $m = O(n^{(16/15)-\epsilon})$. The bottleneck operation for our algorithm is the time it takes to maintain the transitive closure of the "abundance graph". In Section 8, we show how to solve the max flow problem in $O(n^2/\log n)$ time in the case that $m = O(n)$. In Section 9, we show how to speed up the algorithm further if $\log U^*$ is not too large.

2 Preliminary notation, definitions, and lemmas

We consider a directed graph $G = (N, A)$ with *node set* N and *arc set* A. We let $n = |N|$ and we let $m = |A|$. Each arc $(i, j) \in A$ has an associated integral *capacity* u_{ij}. We permit some of the arc capacities to be infinite. We let $U_{\max}$ denote the maximum of the finite arc capacities. We let $U_{\min}$ denote the minimum of the non-zero arc capacities.

There are two distinguished nodes in N: a *source* s and a *sink* t. A single commodity must be routed through G from s to t. The arcs incident to s or t are referred to as *external arcs*. The remaining arcs are called *internal arcs*. A node i is *internal* if $i \neq s$ and $i \neq t$. To simplify notation, we assume without loss of generality that whenever an internal arc (i, j) is in A, arc (j, i) is also in A. For every internal node i, we assume that (s, i) and (i, t) are in A, possibly with a capacity of 0.

To *contract* an arc (i, j) is to replace the nodes i and j by a single new node, referred to as the *contracted node*. Any arc that was formerly incident to node i or j before contraction is incident to the contracted node subsequently. Contraction is a standard operation in graph and network algorithms.

A *flow* is a function $x : A \to \mathbb{R}_+ \cup \{0\}$ that satisfies the *flow conservation constraints*; that is,

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0 \text{ for all } i \in N \setminus \{s, t\}.$$

A flow x is called *feasible* if it obeys the *capacity constraints*, that is, $x_{ij} \leq u_{ij}$ for each arc $(i, j) \in A$. We refer to x_{ij} as the *flow* on arc (i, j). The *value* of a flow x is the net flow out of the source, which is equal to the net flow into the sink. In a *max flow problem*, one seeks a feasible flow whose value is maximum.

Suppose that x is a feasible flow. For each internal node i, the *residual capacity* of arc (s, i) is $r_{si} = u_{si} - x_{si}$. The residual capacity of arc (i, t) is $r_{it} = u_{it} - x_{it}$. For each internal arc $(i, j) \in A$, $r_{ij} = u_{ij} + x_{ji} - x_{ij}$. The residual capacity expresses how much additional flow can be sent from i to j, starting with the flow x. We let $r[x]$ denote the vector of residual capacities. Often, we will denote the residual capacities more briefly as r. The residual network is denoted $G[r]$. The arcs (i, s) and (t, i) are not present in G and they also not present in $G[r]$.

An *s-t cut* is a partition of the node set N into two parts, S and T, such that $s \in S$ and $t \in T$. The *capacity* of the cut (S, T) is $u(S, T) = \sum_{i\in S, j\in T} u_{ij}$. If r is the vector of residual capacities and if (S, T) is an *s-t* cut, then the residual capacity of the cut (S, T) is $r(S, T) = \sum_{i\in S, j\in T} r_{ij}$. The following is the max-flow min-cut theorem of Ford and Fulkerson [11], as applied to the residual network.

Lemma 1. (Max residual flow, min residual cut). *Suppose that r is a vector of residual capacities and (S, T) is an s-t cut. Then $r(S, T)$ is an upper bound on the maximum amount of flow that can be sent from source to sink in the residual network $G[r]$. Moreover, the maximum flow with respect to r is the minimum residual capacity of an s-t cut.*

We let $A(j)$ denote the subset of arcs incident to node j. We say that $A'(j)$ is an *anti-symmetric* subset of $A(j)$ if for every arc $(i, j) \in A(j)$, either $(i, j) \in A'(j)$ or $(j, i) \in A'(j)$ but not both. Note that if $A'(j)$ is antisymmetric, then $(s, j) \in A'(j)$ and $(j, t) \in A'(j)$.

Lemma 2. (Anti-symmetry lemma). *Suppose that $A'(j)$ is an anti-symmetric subset of $A(j)$. Suppose further that x is a feasible flow, and $r = r[x]$. Then*

$$\sum_{(i,j)\in A'(j)} r_{ij} - \sum_{(j,i)\in A'(j)} r_{ji} = \sum_{(i,j)\in A'(j)} u_{ij} - \sum_{(j,i)\in A'(j)} u_{ji}.$$

Proof. We note that $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. The lemma is true because of the following.

$$\sum_{(i,j)\in A'(j)} (u_{ij} - r_{ij}) - \sum_{(j,i)\in A'(j)} (u_{ji} - r_{ji})$$
$$= \sum_{(i,j)\in A'(j)} (x_{ji} - x_{ij}) + \sum_{(j,i)\in A'(j)} (x_{ji} - x_{ij}) = \sum_{(i,j)\in A(j)} (x_{ji} - x_{ij}) = 0.$$

□

Suppose that P is a directed path in G from node i to node j, and suppose that $(i,j) \in A$. Suppose further that $|P| \geq 2$. To *transfer δ units of capacity* from path P to arc (i,j) is to reduce $u_{k\ell}$ by δ of each arc (k,ℓ) of P and to increase u_{ij} by δ.

Lemma 3. (Capacity transfer lemma). *Let P be a path in G from node i to node j. Let (S,T) be an s-t cut. Suppose that u' is obtained from u by transferring δ units of capacity from P to arc (i,j). Then $u'(S,T) \leq u(S,T)$.*

Proof. Except in the case that $i \in S$ and $j \in T$, the lemma is trivially true because $u'_{k\ell} \leq u_{k\ell}$ unless $i = k$ and $j = \ell$. Suppose now that $i \in S$ and $j \in T$. Let ℓ be the first node of P that is in T, and let k be the preceding node of P. Then $u(S,T) - u'(S,T) \leq u_{k\ell} - u'_{k\ell} + u_{ij} - u'_{ij} = \delta - \delta = 0$. □

In general, transferring capacity from paths to arcs decreases the amount of flow that can be sent from source to sink. In essence it requires that δ units of the capacity of each of the arcs in P be reserved for sending flow from i to j. In Section 6, we will see an important special case of transferring capacity from a path to an arc that does not result in a decrease in the max flow.

3 Improvement phases

Our algorithm solves the max flow problem as a sequence of improvement phases. The input for an improvement phase is a flow x, a vector $r = r[x]$ of residual capacities, and an s-t cut (S,T). We typically denote the input for an improvement phase as the triple (r,S,T). We refer to the phase as the Δ-*improvement phase*, where $\Delta = r(S,T)$.

The output of the Δ-improvement phase is a flow x', a vector $r' = r[x']$ of residual capacities and an s-t cut (S',T') such that $r'(S',T') \leq \Delta/(4m)$.

We will run the improvement phase either on the network G or on a "compact network" G^c, described later.

4 Δ-abundant arcs and the abundance graph

Let (r,S,T) be the input for an improvement phase, and let $\Delta = r(S,T)$. An arc (i,j) is called Δ-*abundant* if $r_{ij} \geq 2\Delta$. We sometimes refer to it as *abundant* if Δ is obvious from context. The change in flow in any arc is at most Δ during an improvement phase. Therefore, the following lemma is true.

Lemma 4. *Suppose that (r,S,T) is the input at the beginning of the Δ-improvement phase, where $\Delta = r(S,T)$. If arc (i,j) is Δ-abundant at the beginning of the Δ-improvement phase, then (i,j) remains abundant at all subsequent improvement phases.*

Abundant arcs play two roles in the speed-up of our max flow algorithm. (1) Directed cycles of abundant arcs are "contracted" into a single node. The contracted arcs are expanded subsequent to the algorithm identifying an optimal flow in the contracted graph. (2) A node can be "compacted" if every incident arc is either abundant or it has very small capacity. Compacted nodes are not present in the compact network. We describe compaction in Section 6.

The *abundance graph* is the graph with node set N and whose arc set is the set of abundant arcs. We denote it as G^{ab}. By Lemma 4, once an arc becomes abundant, it remains abundant. The abundance graph increases dynamically over time.

An arc (i, j) is in the *transitive closure* of G^{ab} if there is a directed path in G^{ab} from node i to node j. Our algorithm maintains the transitive closure of G^{ab} over all iterations. This may be accomplished in $O(nm)$ time using Italiano's [18] algorithm for dynamically maintaining the transitive closure of a graph.

If there is an abundant path from node i to node j, we denote it as $i \Rightarrow j$. The transitive closure algorithm maintains a path from node i to node j whenever $i \Rightarrow j$. If there is more than one path, it will maintain the first path it determines. It maintains paths implicitly by using a matrix $\mathbf{M}$. If there is a path from node i to node j in G^{ab}, then $\mathbf{M}_{ij}$ is the node that precedes j on the path in G^{ab} from i to j. The time it takes to reconstruct a path P from the matrix $\mathbf{M}$ is $O(|P|)$.

The transitive closure algorithm is valid even if G^{ab} contains directed cycles. However, as we will see in Section 5, our algorithm contracts any abundant directed cycles. Contraction of abundant cycles does not increase the time needed to maintain the dynamic transitive closure.

5 Contraction of abundant cycles

If the abundance graph contains the internal arcs (i, j) and (j, i), then we can contract nodes i and j into a single node, and find an optimal flow in the contracted graph. After obtaining an optimal flow in this contracted graph, one can then expand the contracted node into their original pair of arcs (i, j) and (j, i). The flow in the expanded graph can be made feasible by sending flow on (i, j) or (j, i), whichever is needed in order to balance the flow in nodes i and j.

We illustrate this contraction on arcs (5, 6) and (6, 5) in Figures 1 and 2. After contraction, the total change in flow in each arc is less than 2Δ. When the node labeled 5-6 is ultimately expanded, it is possible that the flow conservation constraints are violated for nodes 5 and 6, but by an amount that is less than 2Δ. By sending flow from 5 to 6 or from 6 to 5, the conservation of flow constraints are reestablished.

It is also possible to contract abundant external arcs. We illustrate this type of contraction on arcs $(s, 1)$ and $(3, t)$ in Figures 3 and 4. When the nodes labeled s-1 and 3-t are ultimately expanded, it is possible that the flow conservation constraints are not satisfied at nodes 1 or 3. Flow conservation can be reestablished by sending flow in $(s, 1)$ and $(3, t)$.

In addition, one can contract any abundant cycle of internal arcs.

The total time for contraction in an improvement phase is $O(m)$. The time for expansion of contracted cycles is also $O(m)$. For more details on contraction and expansion of cycles, see Goldberg and Rao [16].

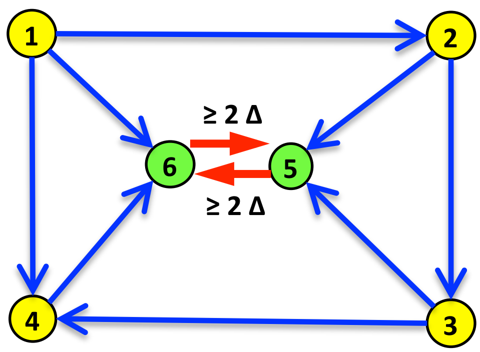

Figure 1. A residual network in which arcs (5, 6) and (6, 5) are both abundant.

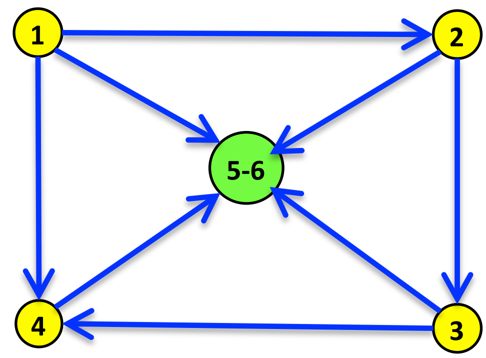

Figure 2. The residual network after contracting the arcs (5, 6) and (6, 5).

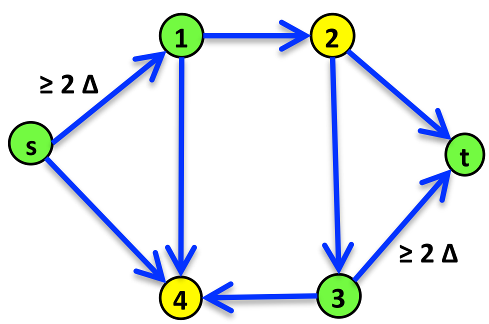

Figure 3. A residual network in which arcs (s, 1) and (3, t) are both abundant.

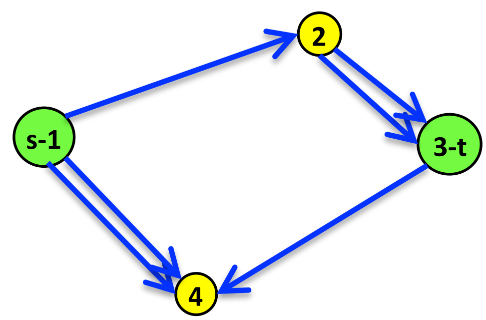

Figure 4. The residual network after contracting (s, 1) and (3, t).

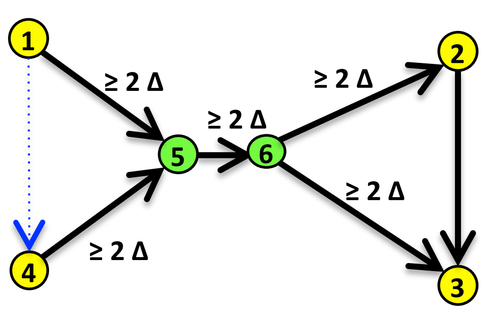

Figure 5. Part of a network in which nodes 5 and 6 are both strongly compactible. The solid arcs are all abundant.

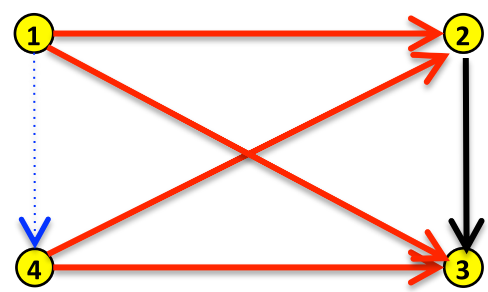

Figure 6. The subgraph of the strongly compact network obtained from Figure 5.

6 The compact network

In this section, we will define compact networks. We start by showing how to construct an intermediate version of the compact network that we refer to as the "strongly compact network". This network is essentially the same as the residual network except that we will eliminate any node for which all incident arcs are abundant. Subsequently, we will define the compact network and show how to construct it.

Algorithm 1. A procedure for creating the strongly compact network $G^{sc} = (N^{sc}, A^{sc})$.

Step 1. Iteratively contract abundant cycles. Iteratively contract abundant external arcs. Let (r, S, T) denote the input after contraction of the abundant cycles.

Step 2. Let N^{sc} be the subset of nodes incident to a non-abundant arc whose residual capacity is positive. A node in $N \backslash N^{sc}$ is referred to as *strongly compactible.*

Step 3. $A^{sc} = A^1 \cup A^2$, where $A^1 = \{(i,j) \in A : i \in N^{sc}$ and $j \in N^{sc}\}$, and $A^2 = \{(i,j) : i \in N^{sc}$ and $j \in N^{sc}$ and $i \Rightarrow j\}$. An $(i,j) \in A^1$ is referred to as an *original arc* and its capacity is r_{ij}. An arc $(i,j) \in A^2$ is referred to as a *pseudo-arc* and its residual capacity is 2Δ.

Algorithm 1 runs in $O(m + |A^{sc}|)$ time. We can create each pseudo-arc in O(1) time because we are maintaining the transitive closure of the abundance graph. (We will only create the compact networks in cases in which $|A^{sc}| = O(m^{9/8})$, a bound that arises from the analysis.)

We illustrate this construction in Figures 5 and 6. In Figure 6, nodes 5 and 6 do not appear because neither node was incident to a non-abundant arc with residual capacity. Figure 6 includes pseudo-arcs from nodes 1 and 4 to nodes 2 and 3.

Theorem 1. *Let v^* be the max flow in the residual network $G[r]$, and let v^{sc} be the max flow in the strongly compact network G^{sc}. Then $v^{sc} = v^*$.*

Proof. Consider first a flow in G^{sc}. This can be transformed into a flow in $G[r]$ by replacing the flows in pseudo-arcs of G^{sc} by flows on the corresponding paths in $G[r]$.

Now consider a flow x in $G[r]$. One can use flow decomposition (see, e.g., Ahuja et al. [1]) to represent x as the sum of flows on paths from s to t. For each path P in the flow decomposition, we carry out the following additional operations. Subdivide P into the union of subpaths, where each subpath begins and ends at a node in N^{sc}, but the other nodes of the subpath are in $N \backslash N^{sc}$. If P' is a subpath of P from node i to node j, then replace the flow on the subpath P' by flow in the corresponding pseudo-arc $(i,j) \in A^{sc}$. Repeating this process yields a flow decomposition in G^{sc}, which, in turn, can be expressed as a flow in G^{sc}. □

We next describe how to obtain the Δ-compact network, which is similar to the strongly compact network, but which may contain far fewer nodes. Our $O(nm)$ algorithm for the max flow problem exploits the fact that the running time to find an approximate max flow in the Δ-compact network may be less than the time it takes to find the approximate max flow in the original network.

An arc (i,j) is said to have *small capacity* with respect to Δ if $r_{ij} + r_{ji} < \Delta/(64m^2)$. An arc (i,j) is said to have *medium capacity* with respect to Δ if $r_{ij} \geq \Delta/(64m^2)$ and if $r_{ij} + r_{ji} < 4\Delta$.

An internal arc (i,j) is referred to as *anti-abundant* if $r_{ij} < 2\Delta$ and $r_{ji} \geq 2\Delta$. An external arc (s,j) or (j,t) is referred to as *anti-abundant* if it is not abundant.

The arcs of the Δ-compact network will consist of original arcs and pseudo-arcs. The abundant pseudo-arcs will be created in the same manner as in Algorithm 1. There will also be non-abundant pseudo-arcs that will be created by transferring capacity from a path P of anti-abundant arcs. (Transferring capacity was described in Section 2.) After transferring flows on paths, our construction of the compact network is very similar to our construction of the strongly compact network.

Our analysis relies on a potential function. For a given node $j \in N$, vector r of residual capacities and for a subset $\widetilde{A}$ of arcs, we define the potential function $\Phi(j, r, \widetilde{A})$ as follows.

$$\Phi(j, r, \widetilde{A}) = \sum_{(i,j)\in\widetilde{A}} r_{ij} - \sum_{(j,i)\in\widetilde{A}} r_{ji}. \tag{1}$$

Let $A'(\Delta)$ denote the set of anti-abundant arcs at the Δ-improvement phase. In the next lemma, the potential function is defined on $\widetilde{A} = A'$. We rely on other choices of $\widetilde{A}$ in the proof of Lemma 2.

Lemma 5. *Let (S,T) be an s-t cut in G with $r(S,T) \leq \Delta$, and let $A' = A'(\Delta)$. Suppose that $P \subseteq A'$ is a path from node i to node j, and suppose that $(i,j) \in A'$. Let r' be obtained from r by transferring δ units of residual capacity from path P to arc (i,j). Then (1) $\Phi(k, r', A') = \Phi(k, r, A')$ for each $k \in N$, and (2) $r'(S,T) = r(S,T)$.*

Proof. The proof of statement (1) is straightforward. We now consider statement (2). Statement (2) is trivially true if $|P| = 1$. Assume that $|P| \geq 2$, and let $P = i_1, i_2, \ldots, i_k$. The lemma is clearly true in the case that every node of P is in S or if every node of P is in T. So, we consider the case in which at least one node of P is in S and at least one node of P is in T. Since the reversal of each arc of (P) is abundant, and since $r(S,T) \leq \Delta$, there must be an index $\ell \in [1, k-1]$ such that (a) $i_j \in S$ for $j \leq \ell$, and (b) $i_j \in T$ for $j > \ell$. Under these circumstances, one can verify that $r'(S,T) = r(S,T)$. $\square$

We say that a node j is Δ-*compactible* if $|\Phi(j, r', A')| < \Delta/(16nm)$ and if j is not incident to a medium capacity arc. We say that node j is *very Δ-compactible* if it is Δ-compactible and if every incident anti-abundant arc has capacity less than $\Delta/(16nm)$. The proof of the following lemma is immediate from our definition of the potential function Φ.

Lemma 6. *Suppose that node j is Δ-compactible. Suppose further that there is no pair of anti-abundant arcs (i,j) and (j,k) with positive residual capacity. Then j is very Δ-compactible.*

Prior to constructing the compact network at the Δ-improvement phase, our algorithm will iteratively transfer the residual capacity of a path to a pseudo-arc. (Step 3A of Algorithm 2). The transfer of capacities will not affect the maximum flow value. Each transfer of capacities will eliminate at least one anti-abundant arc from the network. The transfer of capacities will continue until every Δ-compactible node becomes very Δ-compactible.

We say that a path P has *transferrable residual capacity* if (i) $|P| \geq 2$, (ii) $r(P) > 0$, and (iii) each arc of P is anti-abundant. If P is a path from node i to node j such that P has transferrable capacity, then to *transfer the residual capacity of* P is to transfer $r(P)$ units of capacity from P to (i,j). If arc (i,j) were not in A, we would add (i,j) as an anti-abundant pseudo-arc prior to transferring the capacity.

We say that a node j is Δ-*critical* if it is not Δ-compactible. The set of nodes in the Δ-compact network are the Δ-critical nodes. We next bound the total number of nodes in compact networks.

Theorem 2. *The total number of Δ-critical nodes over all improvement phases is $O(m)$.*

Proof. Let Δ_k denote the parameter for the k-th improvement phase. Our improvement algorithm ensures that for each k, $\Delta_{k+1} \le \Delta_k/(4m)$.

We first consider nodes in compact networks that are incident to arcs of medium capacity. Let r denote the residual capacities after Step 1 of Algorithm 1 at Improvement Phase k. Let r' denote the residual capacities at the beginning of Phase $k+4$. If (i,j) or (j,i) is of medium capacity with respect to Δ_k, then $\Delta_k/(64m^2) \le r_{ij} + r_{ji} \le 4\Delta_k$. Then $r'_{ij} + r'_{ji} = r_{ij} + r_{ji} > 4\Delta_{k+3}$. It follows that each arc is of medium capacity for at most four improvement phases. And the total number of medium arcs over all improvement phases is $O(m)$.

We next consider the remaining Δ-critical nodes, which we refer to as Δ*-special nodes.* If node j is Δ-special, then $|\Phi(j,r,A')| \ge \Delta/(16nm)$ and there are no medium capacity arcs incident to j.

We claim the following: if node j is Δ-special, then within four more improvement phases, node j will be on an abundant directed cycle, and will thus be contracted. If the claim is true, then we will have shown that the number of Δ-special nodes over all improvement phases is $O(n)$, which will complete the proof that the total number of Δ-critical nodes is $O(m)$. (We assume that $m \ge n$.)

Suppose that node j is Δ-special. Let $A'(j)$ be the set of arcs arcs incident to node j that are anti-abundant at Phase k. Let $A''(j)$ consist of arc (s,j) plus all of the arcs directed out of node j that are neither abundant nor anti-abundant at Phase k. (All of these arcs have small capacity with respect to Δ_k). Then $A'(j) \cup A''(j)$ is an anti-symmetric subset of $A(j)$. By Lemma 2 and by Lemma 5, $\Phi(j,r,A'(j)\cup A''(j)) = \Phi(j,r',A'(j)\cup A''(j))$. Therefore,

$$\begin{aligned} |\Phi(j,r',A'(j))| &\ge |\Phi(j,r,A'(j))| - |\Phi(j,r,A''(j))| - |\Phi(j,r',A''(j))| && (2)\\ &\ge \Delta_k/(16nm) - 2n\Delta_k/(64m^2) && (3)\\ &\ge \Delta_k/(32nm) > 4n\Delta_{k+4}. && (4)\end{aligned}$$

Inequality (3) relies on the fact that there at most n arcs in $A''(j)$, each with capacity less than $\Delta_k/(64m^2)$. Since $A'(j)$ has fewer than $2n$ arcs, inequality (4) implies that there must be an arc $a \in A'(j)$ such that $r'_a \ge 2\Delta_{k+4}$. Since the reversal of arc a is also abundant, it follows that arc a and its reversal is an abundant cycle, and thus arc a would be contracted. □

We next show how to create the Δ-compact network $G^c = (N^c, A^c)$. The node set N^c is the set of Δ-critical nodes. To obtain the arc set A^c, one first transfers capacities on paths whose arcs are anti-abundant.

Algorithm 2. A procedure for creating the Δ-compact network $G^c = (N^c, A^c)$.

Step 1. Iteratively contract abundant cycles. Iteratively contract abundant external arcs.

Step 2. Let N^c be the set of Δ-critical nodes.

Step 3A. If there is a Δ-compactible node with an entering anti-abundant arc and a leaving anti-abundant arc, find a path P with transferrable residual capacity such that the first and last nodes of P are in N^c and every other node of P is in $N \backslash N^c$. (Such a path will exist). Transfer the residual capacity from this path to a pseudo-arc. Continue finding paths with transferrable residual capacity until there is no Δ-compactible node with an entering anti-abundant arc and a leaving anti-abundant arc. (In Section B, we will show how to implement Step 3A efficiently using dynamic trees.)

Step 3B. Let r' denote the residual capacities of arcs and pseudo-arcs after Step 3A. Let $A^c = A^1 \cup A^2$, where $A^1 = \{(i,j) : i \in N^c, j \in N^c$, and $r'_{ij} > 0\}$, and $A^2 = \{(i,j) : i \in N^c$ and $j \in N^c$ and $i \Rightarrow j\}$. An arc $(i,j) \in A^1$ has capacity r'_{ij}. An arc in A^2 is referred to as a *pseudo-arc* and has residual capacity 2Δ.

Theorem 3. *Let v^* be the max flow in the residual network $G[r]$, and let v' be the max flow in the Δ-compact network G^c . Then $v' \leq v^* < v' + \Delta/8m$.*

Proof. Contracting abundant cycles or abundant external arcs does not affect the max flow value, nor does it increase the number of arcs.

We now consider Step 3A. Lemma 5 shows that transferring flow from transferrable paths to pseudo-arcs does not affect the value of the max flow, nor does it affect the potential function Φ.

We now consider Step 3B. If the only arcs incident to the Δ-compactible nodes were abundant arcs, then by Theorem 1, the max flow in G^c would be the same as the max flow in $G[r]$. However, in creating G^c, we also eliminate all small capacity arcs incident to Δ-compactible nodes as well as the anti-abundant arcs incident to the Δ-compactible nodes after the transfer of capacity from paths. There are at most m arcs with small capacity, and the sum of their capacities is less than $m(\Delta/(64m^2)) = \Delta/(64m)$. The total capacity of anti-abundant arcs incident to Δ-compactible nodes after Step 3A is less than $n\Delta/(16nm) = \Delta/(16m)$. We conclude that $v' \leq v^* \leq v' + \Delta/(64m) + \Delta/(16m) < v' + \Delta/(8m)$. □

Theorem 3 establishes that the transformation used to create the Δ-compact network decreases the maximum amount of flow by at most $\Delta/(8m)$.

7 Maximum flows in $O(nm)$ time

In this section, we show that the running time for our max flow algorithm is $O(nm)$, and the bottleneck is due to the maintenance of the transitive closure of G^{ab}.

The procedure *improve-approx-2* finds an approximately optimal flow in an improvement phase by considering three different cases. Let c denote the number of Δ-critical nodes. (i) If $c > m^{9/16}$, then the procedure finds a Δ'-optimal solution on $G[r]$, where $\Delta' = \Delta/(4m)$. (ii) If $m^{1/3} \leq c < m^{9/16}$, then the procedure finds a $\Delta'/2$-optimal solution on G^c, and transforms this into a Δ'-optimal solution on $G[r]$. If $c < m^{1/3}$, then the procedure first chooses a parameter Γ, where $\Gamma < \Delta'$. It then determines an optimal flow on what is referred to as the "(Δ, Γ)-compact network", and transforms this flow into a Γ-optimal flow for G. We will show that this third case occurs $O(m^{2/3})$ times, and the running time for the flow subroutines on each of these instances is $O(m)$.

The (Δ, Γ)-*compact network* is created in the same way as the Δ-compact network with the following exception. For a node j to be (Δ, Γ)-critical, it is incident to an arc (i,j) such that (i) $\Gamma/(64m^2) < r_{ij} + r_{ji} < 4\Delta$ or (ii) $|\Phi(j, r, A')| \geq \Gamma/(16nm)$. As before, abundance and anti-abundance are still defined with respect to Δ.

The following procedure transforms a Δ-optimal solution into a $\Delta/(4m)$-optimal solution.

Procedure *improve-approx-2*(r, S, T);

$\Delta := r(S,T)$
let c be the number of Δ-critical nodes
if $c > m^{9/16}$ **then** find a $\Delta/(4m)$-optimal flow on the residual network $G[r]$
else, if $m^{1/3} \leq c < m^{9/16}$ **then**
    let G' denote Δ-compact network

06. find a $\Delta/(8m)$-optimal flow x' on G'
07. transform the flow x' into a $\Delta/(4m)$-optimal flow x^* on G[r]
08. **else, if** $c < m^{1/3}$ **then**
09. choose the minimum value Γ such that the number of nodes
10. in the (Δ, Γ')-compact network is less than $2m^{1/3}$
11. let G' denote (Δ, Γ)-compact network
12. find an optimal flow x' on G'
13. transform the flow x' into a Γ-optimal flow x^* on $G[r]$

In the following lemma, when we refer to the time to find flows in *improve-approx-2*, we are referring steps 3 and 6. We are not referring to the time to create the compact networks.

Lemma 7. *Let c be the number of Δ-critical nodes for the procedure improve-approx-2. If $c \geq m^{1/3}$, then the running time of the flow subroutine is $O(cm^{15/16} \log^2 n)$.*

Proof. If $c > m^{9/16}$, then the running time per improvement phase is $O(m^{3/2} \log^2 n)$. Multiplying the running time by $c/m^{9/16}$ shows that the running time is $O(cm^{15/16} \log^2 n)$. If $m^{1/3} < c < m^{9/16}$, then the running time for determining an approximate max flow on the compact network is $O(c^{8/3} \log n)$, which is $O(cm^{15/16} \log n)$. □

Lemma 8. *Let c be the number of Δ-critical nodes for the procedure improve-approx-2. The number of phases for which $c < m^{1/3}$ is $O(m^{2/3})$. The running time of the flow subroutines that find the optimal flow in the compact network is $O(m)$ for each of these phases.*

Proof. The final statement of the theorem is obvious. If the number of nodes is c and if $c < m^{1/3}$, then the running time per max flow is $O(c^3) = O(m)$.

We next prove that the number of improvement phases is $O(m^{2/3})$. We first note that the condition for a node to be (Δ, Γ)-critical is more restrictive than the condition that a node be Δ-critical. Therefore, by Theorem 2, the number of nodes in compact (Δ, Γ)-compact networks is $O(m)$ over all improvement phases.

Suppose that in the k-th improvement phase, the parameters are Δ_k and Γ_k and that there are additional improvement phases beyond Phase k. By our rule for choosing Γ_k, the number of $(\Delta_k, \Gamma_k/2)$-critical nodes is greater than $2m^{1/3}$. Using a similar argument to the one in the proof of Theorem 2, we will show that within four more scaling phases $m^{1/3}$ arcs of medium capacity will have large capacity, or $m^{1/3}$ nodes will be contracted or both. This will establish that there are $O(m^{2/3})$ such phases.

We have already established that $\Delta_{k+1} \leq \Gamma_k$. Any arc that is medium capacity with respect to $\Gamma_k/2$ will become abundant or anti-abundant within 4 additional improvement phases. Suppose that node j is $(\Delta_k, \Gamma_k/2)$-critical but is not incident to an arc that is medium capacity with respect to $\Gamma_k/2$. Then node k will be contracted within four more improvement phases. This shows that the number of phases for which $c < m^{1/3}$ is $O(m^{2/3})$. □

Theorem 4. *For $m = O(n^{16/15-\epsilon})$, the max flow problem is solvable in $O(nm)$ time by iteratively calling the procedure improve-approx-2.*

Proof. We first consider the time spent in flow operations; i.e., Steps 4, 7, and 13 of procedure *improve-approx-2*. The total number of Δ-critical nodes over all improvement phases is $O(m)$. By Lemma 7, the time to find the approximate max flows is $O(m^{31/16} \log^2 n)$ over all phases with at least $m^{1/3}$ Δ-critical nodes. By Lemma 8, the number of improvement phases is $O(m^{2/3})$. If there are fewer than $m^{1/3}$ Δ-critical nodes, the time for the flow operations is $O(m)$. Thus, the total time for finding flows in these improvement phases is bounded by $O(m^{5/3})$.

The remaining operations to consider are the following: (1) the time to contract abundant cycles and to expand them; (2) the time to create the abundant pseudo-arcs of the compact networks; (3) the time to transform flows in abundant pseudo-arcs of the compact networks into flows on paths in the original network; (4) the time to transfer capacities from paths of anti-abundant arcs to pseudo-arcs of the compact networks, and (5) the time to transform flows in the non-abundant pseudo-arcs of the compact networks into flows on paths in the original network.

We have already stated that the time to contract abundant cycles and expand them is $O(m)$ per improvement phase, which is $O(m^{5/3})$ time in total.

We now consider (2). If c is the number of nodes of the compact network, then the time to create the abundant pseudo-arcs is $O(c^2)$ plus the time required to maintain the transitive closure of G^{ab}. The bottleneck is the time for the dynamic transitive closure, which is $O(nm)$ in total using the algorithm of Italiano [18].

We now consider (3). Let x' denote the flow in the compact network. In principle, we could just convert x' to a spanning tree flow. However, in order to use our definition of abundant arc, we need to ensure that flows do not change by more than Δ in an improvement phase. So, instead we modify only the flows in the pseudo-arcs.

We transform x' by iteratively sending flow around (undirected) cycles consisting of pseudo-arcs. We continue until there is no cycle of pseudo-arcs with positive flow. The resulting flow x'' has at most c pseudo-arcs that have positive flow. This "cycle canceling" approach can be carried out in $O(c \log n)$ steps using the dynamic tree data structure as described in Goldberg and Tarjan [17]. If (i, j) is an abundant pseudo-arc, its corresponding path of abundant arcs can be determined in $O(n)$ time from the matrix $\mathbf{M}$. Thus, the positive components of x'' corresponding to abundant pseudo-arcs can be transformed into flows on paths in $O(nc)$ time. Since there are $O(m)$ critical nodes over all phases, the running time over all phases for these transformations is $O(nm)$, which is the same time bound as for maintaining the transitive closure.

We establish in Appendix B that the time for transferring capacities and creating non-abundant pseudo-arcs is $O(m \log n)$ per improvement phase, and $O(m^{5/3} \log n)$ in total. We establish in Appendix C that the time for transforming flows in non-abundant pseudo-arcs into flows on paths is $O(m \log n)$ per improvement phase, and $O(m^{5/3} \log n)$ in total. □

8 An $O(n^2/\log n)$ algorithm for max flows when $m = O(n)$

In this section, we describe how to solve the max flow problem in $O(n^2/\log n)$ time when $m = O(n)$. In this case, the number of Δ-critical nodes in all iterations is $O(n)$. In order to achieve the $O(n^2/\log n)$ running time, we need to create the compact network in $O(cn/\log n)$ time, where c is the number of its nodes. We also need to transform the flow in the compact network into a flow in the residual network in $O(cn/\log n)$ time. This latter problem is less straightforward than the problem of creating the compact networks.

If we were to use a standard graph search algorithm to determine the abundant pseudo-arcs of a compact network with c nodes, it would take $O(cm)$ time. We can gain a factor $\log n$ improvement by relying on a technique developed by Gabow and Tarjan [13] in the context of a set union data structure. They represented subsets of a ground set $\mathcal{S}$ for which $|\mathcal{S}| = .3 \log n$ using integers in the range $[0, n^{1/3}]$. They also create tables in $O(n)$ time so that operation on subsets of $\mathcal{S}$ takes $O(1)$ steps using table lookup. Our approach relies on the same framework. Here, we assume that $\mathcal{S} = \{1, 2, 3, ..., K\}$. We assume that every element $i \in \mathcal{S}$ has an associated value a_i.

Our algorithm relies (in principle) on six tables, each of which can be created in $O(n)$ time. The tables permit each of the following operations to be carried out in $O(1)$ steps for subsets S and T of $\mathcal{S}$.

1. (Union.) $W := S \cup T$.
2. (Intersection.) $W := S \cap T$.
3. (Set difference.) $W := S \backslash T$
4. (Subset sum.) $w := \sum_{i \in S} a_i$.
5. (First element.) First(S) is the first element of S. If $S = \emptyset$, then First$(S) = \emptyset$.
6. (Is an element of.) Element(S, x) = TRUE if $x \in S$; otherwise, Element(S, x) = FALSE.

We suppose without loss of generality that the node set $\mathcal{S}$ is $\{1, 2, 3, ..., K\}$. We next show how to determine in $O(m)$ steps the set of pairs $\{i, j : i \in \mathcal{S}, j \in N, \text{ and } i \Rightarrow j\}$. We assume that the arc set A^{ab} has no directed cycles, or equivalently that we have already contracted the abundant directed cycles.

For each $j \in V$, we let $F(j) = \{k \in \mathcal{S} : k \Rightarrow j\}$. For each $k \in F(j)$, our algorithm will (implicitly) identify an abundant path $P_k(j)$. We let $F(i, j) = \{k \in \mathcal{S} : (i, j) \in P_k(j)\}$. Our algorithm adapts the standard graph search algorithm so that it can identify paths from the subset $\mathcal{S}$. After the sets $F(\cdot), F(\cdot, \cdot)$ are determined with respect to the set $\mathcal{S}$, one can add the $O(cK)$ abundant pseudo-arcs from $\mathcal{S}$ to other nodes in the compact network in $O(cK) = O(c \log n)$ time. We will ultimately run the procedure *forward-search* on $c/\log n$ different subsets for nodes of the compact network.

Procedure *forward-search*;
Initialize
  for each $i \in \mathcal{S}$, $F(i) := \{i\}$
  for each $j \in N \backslash \mathcal{S}$, $F(j) := \emptyset$
  for each $(i, j) \in A$, $F(i, j) := \emptyset$
scan nodes of N in topological order
for each node i and **for each** arc (i, j) **do**
  $F(i, j) := F(i) \backslash F(j)$
  $F(j) := F(i) \cup F(j)$

Procedure *forward-search* is run to determine the abundant pseudo-arcs of the Δ-compact network (or the (Δ, Γ)-compact network). One can easily verify that the algorithm correctly identifies the sets $F(\cdot), F(\cdot, \cdot)$ in $O(m)$ steps.

Subsequent to creating the compact network, the algorithm determines flows in the arcs of the compact network (Steps 6 or 12 of *improve-approx-2*). Let Q denote the set of abundant pseudo-arcs in the compact network. Consider the flow on these pseudo-arcs. As in the proof of Theorem 4, the algorithm then converts y into a spanning tree flow on Q by sending flow around cycles. Inn the resulting spanning tree flow, there are at most $c - 1$ arcs of Q with positive flow.

Let y be the vector of flows on the pseudo-arcs of Q after this post-processing. Let $K = \lfloor .3 \log n \rfloor$. We next transform y into a flow in the residual network by considering three different cases. In the first two of these cases, we transform K pseudo-arcs with flow at a time.

1. There is a node $i \in N$ that are incident to at least K pseudo-arcs with positive flow in y.

2. There are K independent arcs of Q with positive flow. (A set of arcs is independent if no two arcs have a node in common.)

3. The remaining cases.

We first consider Case 1. We develop a procedure for processing nodes i that are incident to at least $\log n$ arcs with positive flow in y. We illustrate with a node i for which there are at least $\log n$ arcs directed out with positive flow. Let $W(i) = \{j : y_{ij} > 0\}$. Using a graph search algorithm, determine a tree $T \subseteq G^{ab}$ directed out of node i and containing all nodes j such that $i \Rightarrow j$. Thus $W(i) \subseteq T$. Then convert the flows y_{ik} for $k \in W(i)$ into flows y' for G, by finding the unique flow y' in T such that (i) for each $k \in W$, the flow into node k is y_{ik}, and (ii) the flow out of node i is $\sum_{k \in W(i)} y_{ik}$. The time to carry out this procedure for node i is $O(m)$. Subsequent to carrying out this procedure on all arcs directed out of node i or directed into node i, we set $y_{ij} = 0$ and $y_{ji} = 0$ for all j.

We now consider Case 2. Assume for now that we have transformed flows for all nodes i that are incident to at least $\log n$ pseudo-arcs with positive flow in y. Subsequently, for any node i, there are fewer than $\log n$ pseudo-arcs in Q with positive flow with respect to y.

In this case, we use a greedy algorithm to determine K independent pseudo-arcs of Q with positive flow. The greedy algorithm runs in $O(m)$ time. If the greedy algorithm fails to find K independent arcs, our procedure moves on to Case 3.

We consider the case in which the greedy algorithm succeeded in obtaining K independent arcs with positive flow. We first relabel the nodes, so that the K psuedo-arcs are $(i, K+i)$ for $i = 1$ to K.

As before, for each $j \in V$, we let $F(j) = \{k \in [1, K] : k \Rightarrow j\}$. For each $k \in F(j)$, we will (implicitly) identify an abundant path $P_k(j)$. We let $F(i,j) = \{k \in \mathcal{S} : (i,j) \in P_k(j)\}$. We first use procedure *forward-search* to determine $F(\cdot), F(\cdot,\cdot)$. We now define sets $B(i,j)$ and $B(j)$ as follows: $B(i,j) = \{k \in [1,K] : (i,j) \in P_k(K+k)\}$; $B(j) = \{k \in [1,K] : j \in P_k(K+k)\}$. The procedure that determines $B(i,j)$ and $B(j)$ relies on the following recurrence relations.

1. The arc (i,j) is on path $P_k(K+k)$ if and only if $j \in P_k(K+k)$ and $(i,j) \in P_k(j)$.

2. If $i \in P_k(K+k)$, then $i = K+k$ or else there is some arc (i,j) that is on path $P_k(K+k)$.

We will determine $B(i,j)$ and $B(j)$ in Steps 1 to 8 of procedure *backward-search*. Step 9 of *backward-search* transforms the flows on the K pseudo-arcs into flows on abundant paths.

Procedure *backward-search*;
01. Initialize
02. **for each** $j \in [1, K]$, $B(j) := \{j + K\}$
03. **for each** $j \in N \backslash [1, K]$, $B(j) := \emptyset$
04. **for each** $(i,j) \in G^{ab}$, $B(i,j) := \emptyset$
05. scan nodes of G^{ab} in reverse topological order
06. **for each** node i and **for each** arc (i,j) **do**
07. $B(i,j) := B(j) \cap F(i,j)$
08. $B(i) := B(i) \cup B(i,j)$
09. **for each** arc $(i,j) \in G^{ab}$, $y'_{ij} := y'_{ij} + \sum_{k \in B(i,j)} y_{k,k+K}$
10. **for** $k = 1$ to K **do** $y_{k,k+K} := 0$

We note that Step 9 takes $O(m)$ time because it consists of m calls of the subset sum operation, each on a subset of $[1, K]$.

We have now completed Case 1 and Case 2. Eventually, there is an iteration in which no node is incident to K arcs with positive flow with respect to y, and the greedy algorithm fails to determine

K independent arcs of Q with positive flow. But since each node of Q is incident to $O(K)$ arcs with positive flow, it follows that $O(K^2)$ arcs remain with positive flow. These final arcs can be transformed one at a time in $O(nK^2) = O(n \log^2 n) = O(cn/\log n)$ time. (Recall that $c > n^{1/3}$).

9 Further improvements if $\log(U_{\max}/U_{\min}) = O(n^{1/3-\epsilon})$

The analysis of the previous section extends to the case in which $m' = O(n)$, where m' is the number of arcs with finite capacity. If $m' = O(n)$, the bottleneck operations are the creation of the abundant pseudo-arcs and the transformation of flows on these arcs to flows on paths in $G[r]$. Each of these two bottleneck operations runs in $O(nm/\log n)$ time. In this section, we will show how to solve the max flow problem faster in this case when $m > n^{1.7}$. Our approach relies on using fast matrix multiplication to create the compact networks and modify the flows.

The special case of $m' = O(n)$ arises in a variety of situations. Of special note is the case that one can efficiently obtain a Δ-optimal spanning tree flow for some $\Delta \leq U_{min}/2$. In such a case, the non spanning tree arcs are at their upper or lower bound. That is, only the spanning tree arcs can have a flow between 0 and 2Δ. All other arcs either have no capacity or are abundant.

Recall that $U^* = U_{\max}/U_{\min}$. If $\log U^* = O(n^{1/3-\epsilon})$, then the Goldberg-Rao algorithm determines a $(U_{\min}/2)$-optimal flow x' in $O(n^{2/3} m \log(n^2/m) \log U^*) = O(n^{1-\epsilon} m \log(n^2/m))$ time. The solution x' can be converted into a basic feasible solution x'' with the same flow value in $O(m \log n)$ additional steps using dynamic trees (see, e.g., Goldberg and Tarjan [17]). Thus, if $\log U^* = O(n^{1/3-\epsilon})$, then the max flow problem is solvable in $O(nm/\log n)$ time.

If $m = O(n^{4/3})$ and if $\log U^* = O(n^{1-\epsilon}/m^{1/2})$, then then the Goldberg-Rao algorithm determines a $(U_{\min}/2)$-optimal flow x' in $O(m^{3/2} m \log n \log U^*) = O(n^{1-\epsilon} m \log n)$ time. In this case, it can determine an optimal flow in $O(nm/\log n)$ time.

If $m' = O(n)$, fast matrix multiplication can be used to obtain the transitive closure in $O(n^\omega)$ time, where $\omega = 2.3727$. This running time was developed by Williams [24]. In the case that $m' = O(n)$, using fast matrix multiplication, our algorithms solve the max flow problem in $O(T(n,m))$ time , where

$$T(n,m) = \begin{cases} O(nm/\log n) & \text{if } m \leq n^{2\omega/3}, \\ \widetilde{O}(n^{1+2\omega/3}) & \text{if } n^{2\omega/3} \leq m \leq n^{(16\omega/15)-2/3}, \\ \widetilde{O}(n^{17/12} m^{5/8}) & \text{if } n^{(16\omega/15)-2/3} \leq m \leq n^2. \end{cases} \tag{5}$$

We note that $n^{1+2\omega/3} = O(n^{2.582})$, and that for $m = O(n^2)$, $n^{17/12} m^{5/8} = O(n^{8/3})$.

In the following algorithm, we let $\alpha = \log_n m$, and we let $\beta = (2+3\alpha)/8$. The value β was selected so as to minimize the running time. We apply the following procedure if $m > n^{2\omega/3}$, and if the number of non-abundant arcs is $O(n)$.

Procedure *improve-approx-3*(r, S, T)
01. $\Delta := r(S,T)$
02. let c be the number of Δ-critical nodes
03. **if** $c \geq n^\beta$ **then** find a $\Delta/(4m)$-optimal solution on the residual network $G[r]$
04. **else, if** $n^{\omega/3} < c < n^\beta$ **then**
05. find a $\Delta/(8m)$-optimal flow x' on the Δ-compact network

transform x' into a $\Delta/(4m)$-optimal flow x'' on the residual network $G[r]$
else, if $c < n^{\omega/3}$ **then**
choose Γ minimal so that the number of (Δ, Γ)-critical nodes is less than $2n^{\omega/3}$
find an optimal flow x' on the (Δ, Γ)-compact network
transform x' into a Γ-optimal flow x'' on the residual network $G[r]$

In the following theorem, we use $\widetilde{O}$ notation, which ignores terms that are polynomial in $\log n$.

Theorem 5. *Suppose that $m' = O(n)$ and that the max flow problem is solved by iteratively calling improve-approx-3. Then the running time is $\widetilde{O}(n^{1+2\omega/3} + n^{17/12}m^{5/8})$, which is $\widetilde{O}(n^{5/3})$.*

Proof. We first consider all phases in which $c \geq n^{\beta}$. In this case, the running time per phase is $\widetilde{O}(n^{2/3}m) = \widetilde{O}(n^{\alpha+2/3})$. By Theorem 2, the number of these phases is $O(n/n^{\beta})$. Thus the total running time of these phases is $\widetilde{O}(n^{\alpha-\beta+5/3})$.

We next consider phases in which $n^{\omega/3} < c < n^{\beta}$. During these phases, we find the Δ-optimal solution on the compact network in $O(c^{8/3})$ time, and we determine the compact network using fast matrix multiplication in $O(n^{\omega})$ time. The number of these phases is $O(n/n^{\omega/3})$, and so the worst case running time for the fast matrix multiplication over all of these phases is $O(n^{1+2\omega/3})$. We next consider the running time due to the flow subroutines only. The worst case running time for the flow subroutines over all phases with$n^{\omega/3} \leq c \leq n^{\beta}$ occurs when $c = n^{\beta}$. This running time is $\widetilde{O}((n/n^{\beta})n^{8\beta/3}) = \widetilde{O}(n^{1+5\beta/3})$.

If we balance the running times of the flow procedures in these phases with those of the phases in which $c \geq n^{\beta}$, we let $\beta = 1/4 + 3\alpha/8$. Then the total running time of the flow subroutines is $\widetilde{O}(n^{(17/12)+(5\alpha/8)}) = \widetilde{O}(n^{17/12}m^{5/8})$.

Finally, we consider phases in which $c < n^{\omega/3}$. In these phases, the bottleneck operation is the fast matrix multiplication, which takes $O(n^{\omega})$ time. The number of these phases is $O(n/n^{\omega/3})$. Thus, the total running time of these phases is $O(n^{1+2\omega/3})$. We conclude that the running time over all phases is $\widetilde{O}(n^{1+2\omega/3} + n^{17/12}m^{5/8})$. For $\alpha \leq 16\omega/15 - 2/3$, the first term dominates. Otherwise, the second term dominates. □

Acknowledgments The author thanks Ebrahim Nasrabadi for his help in improving the readability of this paper. The author also thanks him for many useful discussions on the technical results of this paper. The author also gratefully acknowledge support of this research through the Office of Naval Research grant N000141110056.

References

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

[2] R. K. Ahuja and J. B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37:748–759, 1989.

[3] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18:939–954, 1989.

[4] B.G. Chandran and D.S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations research*, 57(2):358, 2009.

[5] J. Cheriyan, T. Hagerup, and K. Mehlhorn. An $O(n^3)$-time maximum-flow algorithm. *SIAM Journal on Computing*, 45:1144–1170, 1996.

[6] B. V. Cherkasky. Algorithm for construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations. *Mathematical Methods of Solution of Economical Problems*, 17:112–125, 1977. (In Russian).

[7] B.V. Cherkassky and A.V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[8] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9, 1990.

[9] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[10] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

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

[12] H. N. Gabow. A data structure for dynamic trees. *J. Computer and System Sciences*, 31:148–168, 1985.

[13] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

[14] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Informatica*, 14(3):221–242, 1980.

[15] Z. Galil and A. Naamad. An $O(VE\log^2 E)$ algorithm for the maximal flow problem. *J. Computer and System Sciences*, 21:203–217., 1980.

[16] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45:783–797, 1998.

[17] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[18] G.F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(0):273–281, 1986.

[19] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[20] V. King, S. Rao, and R.E. Tarjan. A faster deterministic maximum flow algorithm. In *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 157–164, 1992.

[21] V. King, S. Rao, and R.E. Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 23:447–474, 1994.

[22] V. M. Malhotra, P. Kumar, and S. N. Maheshwari. An $O(V^3)$ algorithm for finding the maximum flows in networks. *Information Processing Letters*, 7:277–278, 1978.

[23] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Computer and System Sciences*, 24:362–391, 1983.

[24] V. V. Williams. Breaking the Coppersmith-Winograd barrier. 2012.

A Running times of max flow algorithms

Table 1: Polynomial algorithms for the max flow problem

#	Due to	Year	Running Time
1	Ford & Fulkerson [11]	1956	$O(nmU)$
2	Edmonds and Karp [10]	1972	$O(nm^2)$
3	Dinic [9]	1970	$O(n^2m)$
4	Karzanov [19]	1974	$O(n^3)$
5	Cherkasky [6]	1977	$O(n^2\sqrt{m})$
6	Malhotra, Kumar & Maheshwari [22]	1977	$O(n^3)$
7	Galil [14]	1980	$O(n^{5/3}m^{2/3})$
8	Galil & Naamad [15]	1980	$O(nm\log^2 n)$
9	Sleator & Tarjan [23]	1983	$O(nm\log n)$
10	Gabow [12]	1985	$O(nm\log U)$
11	Goldberg & Tarjan [17]	1988	$O(nm\log(n^2/m))$
12	Ahuja & Orlin [2]	1989	$O(nm + n^2\log U)$
13	Ahuja, Orlin & Tarjan [3]	1989	$O(nm\log(n\sqrt{U}/(m+2))$
14	King, Rao & Tarjan [20]	1992	$O(nm + n^{2+\epsilon})$
15	King, Rao & Tarjan [21]	1994	$O(nm\log_{m/nlogn} n)$
16	Cheriyan, Hagerup & Mehlhorn [5]	1996	$O(n^3/\log n)$
17	Goldberg & Rao [16]	1998	$O(\min\{n^{2/3}, m^{1/2}\}m\log(n^2/m)\log U)$
18	Orlin [this paper]	2012	$O(nm)$
19	Orlin [this paper]	2012	$O(n^2/\log n)$ if $m = O(n)$

In Table 1, we summarize the ruining time of the polynomial algorithms for solving the max flow problem with n nodes and m arcs. This table is essentially the same as the one provided in [16]. Those algorithms whose running times are a function of U assume integral capacities whose values are bounded by $U = U_{\max}$.

B Transferring residual capacities from paths

In this section, we show how to transfer flow from paths whose arcs are anti-abundant. In the subsequent section, we will show how transform the flow in these arcs of the compact network into a flow in the residual network.

The algorithm for creating the pseudo-arcs is a variant of flow decomposition. (See, for example, [1].) Our variant of flow decomposition transforms residual capacities on paths into residual capacity on pseudo-arcs. In order to create the pseudo-arcs sufficiently quickly, we rely on the dynamic trees data structure, which was developed by Sleator and Tarjan [23].

We use dynamic trees once again in order to transform flows on pseudo-arcs into flows on the corresponding paths. The dynamic trees data structure is a remarkably efficient data structure for carrying out flow operations and other tree-based operations on a forest. Each tree of the forest has a root node. The root node for the tree containing node i is denoted as root(i). The node that follows node i on the path from i to root(i) is the *parent* of node i and it is denoted as $p(i)$. We let Path(i) denote the path from node i to root(i). We consider Path(i) to include both nodes and arcs. Associated with each non-root node i is a real number value(i), which refers to a real number associated with arc $(i, p(i))$. In the algorithms of this section, value(i) refers to the residual capacity of $(i, p(i))$, and it will be called *res-cap(i)* in the procedures. In the next section, it refers to the flow on $(i, p(i))$, and we will call it *flow(i)* in the procedures.

Dynamic trees support various operations, each with an amortized time complexity of $O(\log n)$ per operation. That is, over a sequence of $K > n$ consecutive operations on the dynamic trees, the running time is $O(K \log n)$. We next list a collection of dynamic tree operations that are sufficient for our purposes.

(i) *create-tree*. This operation initializes an empty dynamic tree.

(ii) *link*(i, j). This operation assumes that i and j belong to two different trees. It merges the tree containing node i with the tree containing node j, lets $p(i) = j$, and sets the root of the merged tree to root(j). It sets value(j) to $r'_{j,p(j)}$. (In the next section, it sets value(j) to $y'_{j,p(j)}$.)

(iii) *cut*(j). This operation breaks the dynamic tree containing node j into two trees by deleting the arc $(j, p(j))$. Node j becomes the root of its tree. It lets $r'_{j,p(j)} :=$ value(j). (In the next section, it lets $y'_{j,p(j)} :=$ value(j).)

(iv) *add-value*(i, val). Value(i) := Value(i) + val for all nodes of Path(i).

(v) *find-value*(i). Returns Value(i) ;

(vi) *find-root*(i). Finds the root of node i.

(viii) *find-min*(i). This operation finds argmin$\{$ value(i) : $i \in$ Path(i)$\}$.

The residual capacities in the reversal of arcs of dynamic trees are also updated appropriately, but we omit the details. The dynamic tree data structure can efficiently support other operations as well, but the above operations are sufficient for our purposes. Initially, the residual capacities are denoted by the vector r. As residual capacities are transferred, we let r' denote the modified capacities.

We say that an arc (i, j) is *admissible* with respect to r' if $r'_{ij} > 0$ and at most one of the nodes i and j are in N^c. All arcs in the dynamic trees of the following algorithms will be admissible. We say that a node i is a *valid initial node* if $i \in N^c$ and if there is some admissible arc emanating from i. A path P is *admissible* with respect to r' if (i) it has positive residual capacity, (ii) its first and last nodes are in N^c, and (iii) no other node of P is in N^c. OpList is an array of all links and cuts that are carried out by the four procedures of this section that are described below. We keep track of the links and cuts for use in the procedure transform-flows, which is presented in the next section. OpList(k) is k-th operation on the dynamic tree data structure, as restricted to links and cuts.

The procedure *find-admissible-path* determines an admissible path P starting with a valid initial node. The procedure *transfer-capacity* transfers capacity from path P to a pseudo-arc. The procedure *prune-tree* eliminates from the dynamic tree any arc (j, k) of P if r'_{jk} became 0. The procedure *transfer-all-capacities* puts it all together.

Procedure *find-admissible-path*(r', i);
    $j :=$ *find-root*(i)
    while $j \notin N^c \backslash \{i\}$ **do**
        select an arc (j, k) with $r'_{jk} > 0$
        link(j, k)
        $K := K + 1$
        OpList$(K) :=$ ("link", j, k)
        $j :=$ *find-root*(k)

Procedure *transfer-capacity*(r', i);
    $p :=$ *find-min*(i)
    $\delta :=$ *find-value*(p)
    $j :=$ *find-root*(i)
    $A^c := A^c \cup \{(i, j)\}$
    $r'_{ij} := r'_{ij} + \delta$
    add-value$(i, -\delta)$
    // Keep additional records of this path for later use //
        $\gamma_K := \delta$
        $v_K := i$
        $w_K := j$

Procedure *prune-tree*(r', i);
    $p :=$ *find-min*(i)
    $\delta :=$ *find-value*(p)
    while $\delta = 0$ **do**
        cut(p)
        $K := K + 1$
        OpList$(K) :=$ ("cut", p)
        $p :=$ find-min(i)
        $\delta :=$ *find-value*(p)

Procedure *transfer-all-capacities*(r');
    // initialize //
        create an empty dynamic tree
        $K := 0$; OpList $:= \emptyset$
        $\gamma := 0$; $v := \emptyset$; $w := \emptyset$
    while there is a valid initial node **do**
        select a valid initial node i
        find-admissible-path(r', i)
        transfer-capacity(r', i)
        prune-tree(r', i)

Incidentally, it is possible that a pseudo-arc (i, j) will end up with capacity greater than 2Δ in

Step 5 of *transfer-capacity* because its capacity may be increased multiple times. In principle, if there are several different paths with the same endpoints i and j, then these paths correspond to different pseudo-arcs, each of which is non-abundant. We do treat the paths differently in Steps 7 to 9 of *transfer-capacity*, where we keep additional records of each path on which capacity is transferred. We will use the additional information in procedure *transform-flows*, which is described in the next section. Accordingly, we still refer to the pseudo-arcs of G^c as non-abundant even if the capacity created in Step 5 is more than 2Δ.

Theorem 6. *The procedure transfer-all-capacities creates the non-abundant arcs of the compact network. Its running time is $O(m \log n)$ per improvement phase.*

C Transforming flows on non-abundant pseudo-arcs into flows on paths

After finding a flow x^c in the compact network G^c, one needs to be able to transform x^c into a flow on the residual network from which G^c was derived. In this section, we describe the transformation.

There are two issues that need to be considered. First of all, there is no efficient way of storing all of the paths in dynamic trees that were determined using the procedures *tree-advance* and *transfer-capacity*. That is, there is no way of storing them so that they can are available for random access. Instead, we recreate the dynamic trees sequentially using the information stored in OpList. Second, there may be more than one path from node i to node j that had its residual capacity transferred. We consider these paths one at a time in the procedure *transform-flows*.

Let k denote the number of elements in OpList at the beginning of some iteration of procedure *transfer-capacity*. The path from i to its root is uniquely determined by the k elements in OpList. We will denote the path as P_k. In Step 8 of transfer-capacity $v_k := i$ and in Step 9, $w_k := j = \text{root}(i)$. There were γ_k units of capacity that were transferred from P_k (step 7 of *transfer-capacity*). In order to send flow on P_k in the procedure *transform-flows*, we recreate the dynamic trees that were used in transfer-all-capacities.

The following procedure transforms the flows of x^c into flows in the residual network from which G^c was created. The description is limited to non-abundant pseudo-arcs of G^c. We let y denote the flow prior to transformation. We let y' be the flow after transformation. In this procedure, the dynamic trees operate on the flow vector y' rather than on the residual capacities r'.

Procedure *transform-flows*(x^c);
```
    y := x^c; y' := 0;
    create an empty dynamic tree
    K := number of elements of OpList
    for k = 1 to K do
        carry out the k-th operation of OpList on the dynamic tree
        if γ_k > 0, then δ := min{γ_k, y_{v_k,w_k}}
        if δ > 0 then do
            add-value(v_k, δ);    // This increases by δ the flow on the arcs P_k //
            y'_{v_k,w_k} := y'_{v_k,w_k} − δ
```

Theorem 7. *The procedure transform-flows modifies the flows in the non-abundant pseudo-arcs of G^c and a flow in the residual network from with G^c was derived. Its running time is $O(m \log n)$ per improvement phase.*