Max Flows in $O(nm)$ Time, or Better

James B. Orlin
Sloan School of Management and Operations Research Center
Massachusetts Institute of Technology, Cambridge, MA 02139
jorlin@mit.edu

ABSTRACT

In this paper, we present improved polynomial time algorithms for the max flow problem defined on sparse networks with n nodes and m arcs. We show how to solve the max flow problem in $O(nm + m^{31/16} \log^2 n)$ time. In the case that $m = O(n^{1.06})$, this improves 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. This establishes that the max flow problem is solvable in $O(nm)$ time for all values of n and m. In the case that $m = O(n)$, we improve the running time to $O(n^2/\log n)$.

Categories and Subject Descriptors

F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity

General Terms

Algorithms, Theory

Keywords

max flows, maximum flow problem

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 [4] and [3]. 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 u_{ij} (possibly infinite), in which the largest finite capacity is bounded by U. The fastest strongly polynomial time algorithm is due to King et al. [9]. 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 [7]. 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. [9] 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. 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. In the case that $\log U \leq m^{7/16}$, the Goldberg-Rao algorithm already runs in $\widetilde{O}(m^{31/16})$ time. (The $\widetilde{O}$ notation ignores log factors of m and n). In the case that $\log U > m^{7/16}$, we reduce the time at an improvement phase by running the Goldberg-Rao scaling phase on a smaller network called the "compact network", where the average number of nodes per improvement phase is $C = O(m/\log U)$. The time to run the Goldberg-Rao scaling phase on a network with at most C nodes and $O(C^2)$ arcs is $\widetilde{O}(C^{7/3})$ time. A first-order approximation in the case that $\log U \geq m^{7/16}$ implies that the total running time over all phases is $\widetilde{O}(C^{7/3} \log U) = \widetilde{O}(m^{7/3} \log^{-4/3} U) = \widetilde{O}(m^{31/16})$.

The compact network is obtained from the original network through two operations: contraction and compaction. Contraction is a standard operation used in strongly polynomial time (and other) algorithms. One can contract a directed cycle if each of its arcs has a residual capacity larger than the max residual flow in the network. "Compaction" is new to this paper. The idea underlying compaction is the following. Suppose that every arc incident to node j has infinite (or sufficiently large) capacity. One can then eliminate node j and replace each pair of arcs (i, j) and (j, k) by an arc (i, k) with infinite capacity. Every flow in the compact

STOC'13, June 1-4, 2013, Palo Alto, California, USA.

network has a corresponding flow with the same flow value in the original graph.

In order to create the compact networks efficiently, one needs to dynamically maintain the transitive closure of the subnetwork of G containing arcs with large residual capacity. We rely on Italian's [8] algorithm in order to maintain the dynamic transitive closure in $O(nm)$ time. This is a bottleneck operation for our algorithm. When $m = O(n)$, dynamic transitive closure can be replaced by static dynamic closures, and the running time can be improved by a factor of $\log n$.

Our paper is organized as follows. In Section 2, we provide preliminary notation and definitions. Section 3 reviews how to solve the max flow problem as a sequence of improvement phases. Section 4 reviews contraction. In Sections 5 and 6, we define the compact network and analyze its properties. In Section 7, we show how to find the max flow in sparse networks in $O(nm)$ time. Section 8 shows how to improve the running time by a factor of $\log n$ in the case that $m = O(n)$. The appendices justifies the running times of the procedures that rely on the dynamic trees data structure.

2. PRELIMINARIES

We consider the max flow problem in a network $G = (N, A)$. 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, possibly with a capacity of 0. For every internal node i, we assume that (s, i) and (i, t) are in A.

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 are 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 $t \in T$. We denote the cut as $[S, T]$. Arc (i, j) is a *forward arc* of $[S, T]$ if $i \in S$ and $j \in T$. It is a *backward arc* of $[S, T]$ if $i \in T$ and $j \in S$. The forward arcs of the cut will be denoted as (S, T). The *capacity* of the cut $[S, T]$ is $u(S, T) = \sum_{i\in S, j\in T} u_{ij}$, that is, the sum of capacities of the forward arcs. 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 [5], 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.*

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 $\Delta = r(S, T)$. Thus Δ is an upper bound on the maximum residual flow from s to t. We refer to Δ as the *flow bound* for the improvement phase.

Associated with the Δ-improvement phase is the "compactness parameter" Γ, where $\Gamma \leq \Delta$. We explain this parameter in Section 6. 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 $r'(S', T') \leq \Gamma/(8m)$. At the next improvement phase, the flow bound $\Delta' = r'(S', T')$. Often, $\Gamma = \Delta$. Even in this case, $\Delta' \leq \Delta/(8m)$. That is, the flow bound improves by at least a factor of $8m$ at each phase.

4. ABUNDANT ARCS AND CONTRACTION

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 the Δ-improvement phase, and thus its ending residual capacity is at least Δ, which in turn is greater than $2\Delta'$, where $\Delta' = r'(S', T')$, and (r', S', T') is the input for some subsequent improvement phase. Therefore, the following lemma is true.

LEMMA 2. *Suppose that (r, S, T) is the input at the beginning of an improvement phase and $\Delta = r(S, T)$. If arc (i, j) is Δ-abundant at the beginning of the Δ-improvement phase, then (i, j) is also Δ'-abundant at the beginning of the Δ'-improvement phase if $\Delta' < \Delta$.*

Each improvement phase begins and ends with a flow on the original network. At the beginning of an improvement phase, the original network may be replaced by a smaller network called the "compact network." This compact network is expanded at the end of the improvement phase. Abundant arcs play an important role in contraction, as described in this section, and in compaction, as described in Sections 5 and 6.

The *abundance graph* is a subgraph of G. Its node set is N and its arc set is the set of abundant arcs. We denote

it as G^{ab}. By Lemma 2, 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 [8] algorithm for dynamically maintaining the transitive closure of a graph. This algorithm does not require the original graph to be acyclic.

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 matrix $\mathbf{M}$. If $i \Rightarrow j$, then $\mathbf{M}_{ij}$ is the node that precedes j on some path in G^{ab} from i to j. The time it takes to reconstruct a path P from the matrix $\mathbf{M}$ is $O(|P|)$.

At the beginning of an improvement phase, the algorithm contracts directed cycles of the abundance graph. It also contracts abundant external arcs. At the end of the improvement phase, the algorithm expands these contracted cycles and contracted external arcs. Any feasible flow in the contracted graph can be expanded to a flow in the original graph with the same flow value.

The total time for contraction in an improvement phase is $O(m)$. The time for expansion of contracted cycles is also $O(m)$. We will show in Section 7 that the number of improvement phases is bounded by $O(m^{2/3})$. Therefore, contraction of cycles and their expansion will not be a bottleneck operation. For more details on contraction and expansion of cycles, see Goldberg and Rao [7].

5. COMPACTIBLE AND CRITICAL NODES

Our algorithm's improved running time is achieved by finding flows on the Γ-compact network, which we describe in this section and the next. We also show that the number of nodes in Γ-compact networks over all improvement phases is $O(m)$.

Recall that $r_{ij} + r_{ji} = u_{ij} + u_{ji}$. Suppose that (r, S, T) is the input at the beginning of the Δ-improvement phase, immediately subsequent to contracting abundant cycles and abundant external arcs.

We then partition the set of arcs into four subsets, which are defined with respect to the *compactness parameter* Γ. The algorithm selects the parameter Γ as part of the max flow algorithm in Section 7. In each improvement phase, $\Gamma \leq \Delta$, which is the flow bound.

An arc (i, j) is said to have Γ-*small capacity* if $u_{ij} + u_{ji} < \Gamma/(64m^3)$. Arc (i, j) is said to have Γ-*medium capacity* at the Δ-improvement phase if $\Gamma/(64m^3) \leq u_{ij} + u_{ji}$, and neither (i, j) nor (j, i) is Δ-abundant.

The third subset of arcs is the set of abundant arcs, which we denote as A^{ab}. Because we have contracted abundant cycles, if $(i, j) \in A^{ab}$, then $(j, i) \notin A^{ab}$

Finally, we say that an arc (i, j) is called *anti-abundant* at the Δ-improvement phase if (j, i) is abundant (and thus (i, j) is not abundant). We let A^{-ab} denote the set of anti-abundant arcs at the beginning of the Δ-improvement phase.

We let the terms $\hat{r}_{out}$ and $\hat{r}_{in}$ refer to the total residual capacity out of a node and the total residual capacity into a node as restricted to the anti-abundant arcs.

$$\hat{r}_{out}(j) = \sum_{(j,k) \in A^{-ab}} r_{jk}.$$

$$\hat{r}_{in}(j) = \sum_{(i,j) \in A^{-ab}} r_{ij}.$$

We say that a node j is Γ-*critical* if it is incident to a Γ-medium arc or if $|\hat{r}_{out}(j) - \hat{r}_{in}(j)| > \Gamma/(16m^2)$. If a node is not Γ-critical, we refer to it as Γ-*compactible*.

The algorithm will create a "compact network" in which every node is Γ-critical, and thus all of the compactible nodes have been eliminated. We will later show how a nearly optimal flow on this compact network can be transformed into a nearly optimal flow on the original network. We first bound the number of Γ-critical nodes over all improvement phases.

Let Δ and Γ denote the flow bound and compactness parameter at an improvement phase. Recall that $\Gamma \leq \Delta$. Let Δ' be the flow bound at the end of the improvement phase (and thus at the beginning of the next phase). Our algorithm satisfies the following important property at each improvement phase: $\Delta' \leq \Gamma/(8m)$. We refer to this as the *improvement property*, and we prove that it is always satisfied in Section 6.

THEOREM 1. *Suppose that each improvement phase satisfies the improvement property. Then the number of* Γ*-critical nodes over all improvement phases is* $O(m)$.

PROOF. We first claim that the number of Γ-medium capacity arcs over all improvement phases is $O(m)$. In fact, we will show that an arc can have medium capacity for at most 3 consecutive phases. Suppose that arc (i, j) has Γ-medium capacity at an improvement phase. Then $u_{ij} + u_{ji} \geq \Gamma/(64m^3)$. Let Δ' be the flow bound at the next phase. Then $u_{ij} + u_{ji} \geq \Delta'/(8m^2)$. If Δ^* is the flow bound two phases after Δ', then $u_{ij} + u_{ji} \geq 8\Delta^*$. In this case (i, j) or (j, i) is Δ^*-abundant, and (i, j) is no longer of medium capacity.

We next consider the remaining critical nodes, which we will refer to as "special nodes". Let Γ be the compactness parameter at the Δ-improvement phase. Let Δ^* be flow bound four phases later. We say that j is Γ-*special* if it is Γ-critical and if it is not incident to an arc that has Γ-medium capacity. If node j is Γ-special, we will show that there is some node k such that (j, k) and (k, j) are both Δ^*-abundant, and hence will be contracted within four phases. This will complete the proof that there are $O(m)$ Γ-critical nodes over all improvement phases.

If (j, k) and (k, j) are both abundant, we say that (j, k) (and also (k, j)) is *doubly-abundant*. Let r^* and y be the residual capacities and flows at the beginning of the Δ^*-improvement phase. We assume that the flow vector y is expressed with respect to the residual capacities r. That is, $0 \leq y_{ik} \leq r_{ik}$ for all arcs $(i, k) \in A$, and $r^*_{ik} = r_{ik} - y_{ik} + y_{ki}$. By Lemma 2, any Δ-abundant arc is also Δ^*-abundant. Also, if $r^*_{ik} > \Gamma/(64m^3)$, then $r^*_{ik} > 8\Delta^*$, and (i, k) is Δ^*-abundant.

We now consider the case that there is some Δ-abundant arc (j, k) such that $y_{jk} > \Gamma/(64m^3)$. In this case, since $r^*_{kj} \geq y_{jk}$, (j, k) and (k, j) are both Δ^*-abundant. Similarly, if (k, j) is Δ-abundant and if $y_{kj} > \Gamma/(64m^3)$, then (j, k) and (k, j) are both Δ^*-abundant.

The remaining case to consider is the case in which there is no Δ-abundant arc incident to node j that has flow greater

than $\Gamma/(64m^3)$. By assumption,

$$|\hat{r}_{out}(j) - \hat{r}_{in}(j)| > \Gamma/(16m^2).$$

We consider the case that $\hat{r}_{out}(j) - \hat{r}_{in}(j) > \Gamma/(16m^2)$. The other case can be proved similarly. Then

$$\begin{aligned}
\sum_{(j,k)\in A^{-ab}} y_{jk} &\le \sum_{(j,k)\in A} y_{jk} = \sum_{(i,j)\in A} y_{ij} \\
&< \sum_{(i,j)\in A^{-ab}} y_{ij} + \sum_{(i,j)\in A^{ab}} y_{ij} \ + \ m\Gamma/(64m^3) \\
&< \hat{r}_{in}(j) + 2m\Gamma/(64m^3) \\
&< (\hat{r}_{out}(j) - \Gamma/(16m^2)) + \Gamma/(32m^2) \\
&< \hat{r}_{out}(j) - \Gamma/(32m^2) \\
&= \sum_{(j,k)\in A^{-ab}} r_{jk} - \Gamma/(32m^2).
\end{aligned}$$

The inequality of the second line follows from the fact that every arc incident to node j is either anti-abundant, abundant, or of small capacity. The first term in the third line is true because $y_{ij} \le r_{ij}$. The second term in the third line is true because we have assumed that $y_{ij} \le \Gamma/(64m^3)$ for $(i,j) \in A^{ab}$.

It follows that there is some arc $(j,k) \in A^{-ab}$ with $y_{jk} < r_{jk} - \Gamma/(32m^3)$. Thus $r^*_{jk} \ge r_{jk} - y_{jk} > \Gamma/(32m^3) > 16\Delta^*$. Therefore, (j,k) is Δ^*-abundant. But $(j,k) \in A^{-ab}$; so (k,j) is also Δ^*-abundant, completing the proof. □

6. THE COMPACT NETWORK

In the case that the number of Δ-critical nodes is sufficiently small (fewer than $m^{9/16}$), our algorithm will replace the residual network by the "compact network" in which every node is Γ-critical. The Γ-compactible nodes do not appear in the compact network. In this section, we describe the arcs of the compact network, which we denote as A^c.

At the beginning of the improvement phase, the algorithm first contracts abundant cycles as well as the abundant external arcs. As before, we let (r, S, T) denote the input after contraction. We let N^c denote the set of Γ-critical nodes.

There are three types of arcs that comprise A^c, which we denote as $A^1 \cup A^2 \cup A^3$. We let r^c denote the vector of residual capacities of arcs in A^c. The arcs in A^1 are "original" arcs. Suppose that $i \in N^c$ and $j \in N^c$. If $(i,j) \in A$, then there is an arc $(i,j) \in A^1$ with residual capacity $r^c_{ij} = r_{ij}$.

The arcs in A^2 are *abundant pseudo-arcs*. If $i \in N^c$ and $j \in N^c$, and if $i \Rightarrow j$ (i.e., there is an abundant path from i to j), then there is an abundant pseudo-arc $(i,j) \in A^2$ with $r^c_{ij} = 2\Delta$.

The arcs in A^3 are the *anti-abundant pseudo-arcs*. These arcs are created through Procedure *transfer-capacity*(r, Γ), which we describe next. This procedure is very similar to procedures used to create flow decompositions. It works with a vector q of residual capacities, where initially $q = r$. At the end of the procedure, $q = 0$.

Suppose that P is a path in the residual network, and let $q(P)$ denote its (positive) residual capacity with respect to vector q. We say that P has *transferrable capacity* if the following is true: (i) the first and last nodes of P are Γ-critical, (ii) the remaining nodes of P are Γ-compactible, and (iii) all arcs of P are anti-abundant.

Procedure *transfer-capacity* iteratively identifies paths with transferrable capacity and then adds anti-abundant pseudo-arc to A^3. The capacity r^c_{ij} of the pseudo-arc (i,j) is $q(P)$. After creating the pseudo-arc (i,j), for each arc (k,ℓ) of P, $q_{k\ell}$ is reduced by r^c_{ij}. At the end of the procedure, all of the pseudo-arcs from node i to node j are aggregated (by summing capacities) into a single pseudo-arc (i,j).

Procedure *transfer-capacity*(r, Γ);
01. Initialize; let H be the subset of arcs of A^{-ab}
02. incident to Γ-compactible nodes;
03. **for each** $(i,j) \in H$, $q_{ij} := r_{ij}$;
04. **while** $H \neq \emptyset$ **do**
05. select a node i of H with no incoming arcs;
06. use depth first search to find a path P that
07. starts at node i and ends
08. at a node ℓ such that $\ell \in N^c$ or
09. ℓ has no outgoing arc (or both);
10. let $\delta = \min\{q_{jk} : (j,k) \in P\}$;
11. **if** $i, \ell \in N^c$, **then** $A^3 = A^3 \cup \{(i,\ell)\}$ and $r^c_{i\ell} = \delta$;
12. **for all** $(j,k) \in P$, $q_{jk} := q_{jk} - \delta$.
13. delete each arc (j,k) from H such that $q_{jk} = 0$.
14. for all pairs of nodes $i, j \in N^c$, aggregate all
15. arcs in A^3 from i to j into a single arc (i,j).

All paths in the procedure *transfer-capacity* consist entirely of anti-abundant arcs. In line 11, the procedure transfers residual capacity from a path P to a pseudo-arc in A^c provided that both endpoints of the path are critical. If both endpoints of P were not critical, then no pseudo-arc is created. In this case, we say that δ units of residual capacity were *lost*. We note that lines 14 and 15 are needed to ensure that $|A^3| = O(|N^c|^2)$.

We next prove four lemmas concerning the compact network followed by two theorems. The first theorem implies that an approximately optimal flow in the compact network induces an approximately optimal flow in the original network. The second theorem bounds the running time for creating the compact networks. In the following section, we analyze the time it takes to transform flows from the compact network to the original network.

The first lemma bounds the amount of lost capacity. The second lemma shows that a flow with value α in the compact network induces a flow with value α in the residual network. The third lemma is a technical lemma concerning the paths involved in the procedure *transfer-capacity*. This lemma is needed in the proof of the fourth lemma, which shows that for all $\beta < 2\Delta$, a cut of capacity at most β in the compact network induces a cut of capacity at most $\beta + \Gamma/(16m)$ in the residual network.

We let $\mathbf{P} \cup \mathbf{Q}$ denote the set of paths created in Procedure *transfer-capacity*(r, Γ). The set $\mathbf{P}$ is the subset of paths that begin and end at Γ-critical nodes. The set $\mathbf{Q}$ is the subset of paths that begin or end at Γ-compactible nodes. All the capacity from paths in $\mathbf{Q}$ is lost.

Each of the following lemmas restricts attention to s-t cuts $[S', T']$ such that no forward arc is abundant. The flow bound at the improvement phase is less than Δ. There is no need to consider s-t cuts with an abundant arc, which would have residual capacity greater than 2Δ.

If P is a path whose capacity was lost (and thus not transferred), then the residual capacity of a cut in G^c may be less than the residual capacity of the corresponding cut in $G[r]$. The next lemma bounds the lost capacity.

LEMMA 3. *The total amount of residual capacity that is lost when running procedure transfer-capacity(r, Γ) is less than $\Gamma/(16m)$.*

PROOF. Capacity is lost when the path created in lines 6 to 10 is in $\mathbf{Q}$, and thus the path begins or ends at a Γ-compactible node. We next bound the residual capacities of paths of $\mathbf{Q}$ beginning or ending at a Γ-compactible node j.

Let $q_{out(j)}$ (resp., $q_{in(j)}$) denote the residual capacity out of (resp., into) node j for capacity vector q at some iteration of the procedure *transfer-capacity*. Let $\Phi(j,q) = q_{out}(j) - q_{in}(j)$. At the beginning of the procedure, $|\Phi(j,q)| = |\hat{r}_{out}(j) - \hat{r}_{in}(j)| \leq \Gamma/(16m^2)$. At the end of the procedure $\Phi(j,q) = 0$ because $q_{out}(j) = q_{in}(j) = 0$.

Consider first the case that $\hat{r}_{out}(j) - \hat{r}_{in}(j) > 0$. Note that $\Phi(j,q)$ does not change when capacity is transferred from a path in $\mathbf{P}$. The node j cannot begin a path in line 5 of the procedure until $q_{in}(j) = 0$. When j is selected for the first time in line 5, $q_{out}(j) = \hat{r}_{out}(j) - \hat{r}_{in}(j) \leq \Gamma/(16m^2)$. Thus, the total amount of flow that is lost because of a path that begins at node j is at most $\Gamma/(16m^2)$. A similar argument shows that the total amount of flow that is lost because of a path that ends at a Γ-compactible node is at most $\Gamma/(16m^2)$. We conclude that the total lost flow is less than $n\Gamma/(16m^2)$, which is less than $\Gamma/(16m)$. □

The next lemma relies on a correspondence between flows in the compact network and flows in the residual network. Suppose y is a flow in the compact network. We transform y into a flow y' in $G[r]$ as follows. If $y_{ij} > 0$ and if $(i,j) \in A^1$, then $y'_{ij} = y_{ij}$. If $y_{ij} > 0$ and if $(i,j) \in A^2$, then the value y_{ij} is added to $y'_{k\ell}$ for each arc (k,ℓ) of the corresponding abundant path from i to j in $G[r]$. (This path is obtainable in $O(n)$ time from the matrix $\mathbf{M}$ maintained by the dynamic transitive closure algorithm.) Suppose now that $y_{ij} > 0$ and $(i,j) \in A^3$. Recall that in the last step of Procedure *transfer-capacity*, we aggregated one or more anti-abundant pseudo-arcs into a single arc (i,j). Thus y_{ij} is the sum of flows of one or more pseudo-arcs from i to j created in Procedure *transfer-capacity*. In order to transform y_{ij} into flows in $G[r]$, we apply an inverse version of *transfer-capacity*. This leads to sending flow on the union of those paths in $G[r]$ that led to the creation of anti-abundant pseudo-arcs. We provide implementation details on sending flow on the anti-abundant path(s) in Appendix B. We show that the running time for the anti-abundant paths is $O(m \log n)$ per improvement phase.

We refer to y' as the flow induced by y. The following lemma follows from the construction of the induced flows.

LEMMA 4. *Suppose that y is flow of α units from s to t in the compact network. Let y' be the flow induced by y. Then y' is a flow of α units from s to t.*

The following lemma states that each anti-abundant path P includes at most one forward arc of an s-t cut, assuming that no forward arc of the cut is abundant. If P had two forward arcs of the cut, then it would also contain a backward arc (j,k) of the cut, and (k,j) would be abundant.

LEMMA 5. *Suppose that $P \in \mathbf{P} \cup \mathbf{Q}$ is a path from node i to node j. Suppose further that $[S', T']$ is an s-t cut with no abundant forward arc. If $i \in S'$ and if $j \in T'$, then P has exactly one arc in (S', T'). If $i \notin S'$ or $j \notin T'$, then no arc of P is in (S', T').*

We will rely on Lemma 5 in proving Lemma 6, which is the next lemma.

We next define types of correspondences between cuts in the compact network and cuts in $G[r]$. Suppose that $[S', T']$ is an s-t cut of $G[r]$, and suppose that there is no abundant forward arc of the cut. The cut of G^c *induced by* $[S', T']$ is the cut $[S^c, T^c]$, where $S^c = S' \cap N^c$, and $T^c = T' \cap N^c$.

Similarly, suppose that $[S^c, T^c]$ is an s-t cut in G^c, and suppose that there are no abundant forward arcs. We can extend $[S^c, T^c]$ to an s-t cut $[S', T']$ of $G[r]$ by letting S' be the set of nodes of N that are reachable from a node in $S^c \subseteq N$ by an abundant path in $G[r]$. We refer to $[S', T']$ as the cut *induced by* $[S^c, T^c]$. We observe that if $[S', T']$ is induced by $[S^c, T^c]$, then $[S^c, T^c]$ is induced by $[S', T']$. (The converse is not true. Different cuts in $G[r]$ can induce the same cut in G^c).

LEMMA 6. *Suppose that $[S^c, T^c]$ is an s-t cut in G^c with no abundant forward arcs. Let $[S', T']$ be the induced cut of G. Then $r^c(S^c, T^c) \leq r(S', T') \leq r^c(S^c, T^c) + \Gamma/16m$.*

PROOF. Our proof partitions the set (S^c, T^c) of forward arcs according as to whether the arc is an original arc or not. The original arcs were in A^1, and they are also in (S', T'). The arcs of $(S^c, T^c) \setminus A^1$ are all anti-abundant pseudo-arcs that were created in the procedure *transfer-capacity*.

Let $D = r(S', T') - r^c(S^c, T^c)$. We will show that $0 \leq D \leq \Gamma/16m$. We first note that we the arcs in A^1 contribute the same amount to $r(S', T')$ and to $r^c(S^c, T^c)$. Accordingly,

$$D = \sum_{(k,\ell) \in (S', T') \setminus A^1} r_{k\ell} - \sum_{(i,j) \in (S^c, T^c) \setminus A^1} r^c_{ij}. \qquad (1)$$

We now consider the arcs in $(S', T') \setminus A^1$. Each of these are anti-abundant arcs whose capacity was transferred or lost in the procedure *transfer-capacity*. Let $\mathbf{P}_F$ and $\mathbf{Q}_F$ denote the subsets of paths of $\mathbf{P}$ and $\mathbf{Q}$ that contain an arc of (S', T'). By Lemma 5, each path of $\mathbf{P_F} \cup \mathbf{Q_F}$ contains exactly one arc of (S', T'). Therefore,

$$\sum_{(k,\ell) \in (S', T') \setminus A^1} r_{k\ell} = \sum_{P \in \mathbf{P_F} \cup \mathbf{Q_F}} \sum_{(k,\ell) \in P} \delta(P). \qquad (2)$$

Each path $P \in \mathbf{P_F}$ transferred its capacity to an anti-abundant pseudo-arc in (S^c, T^c). Therefore,

$$\sum_{P \in \mathbf{P_F}} \sum_{(k,\ell) \in P} \delta(P) = \sum_{(i,j) \in (S^c, T^c) \setminus A^1} r^c_{ij}. \qquad (3)$$

Each path $P \in \mathbf{Q_F}$ resulted in lost capacity. Thus by Lemma 3,

$$\sum_{P \in \mathbf{Q_F}} \sum_{(k,\ell) \in P} \delta(P) \leq \Gamma/16m. \qquad (4)$$

By (1) to (4), we conclude that $0 \leq D \leq \Gamma/16m$. □

THEOREM 2. *Let y be an α-optimal flow vector in the Γ-compact network G^c. Let (S^c, T^c) be a cut in G^c with $r(S^c, T^c) \leq v + \alpha$, where v is the flow value of y. Let y' and (S', T') be the induced flow vector and cut in G. Then y' has a flow of v and is α'-optimal, where $\alpha' = \alpha + \Gamma/(16m)$. Moreover, $r(S', T') \leq v + \alpha'$.*

PROOF. Lemma 4 establishes that the flow value of y' is v. Lemma 6 establishes that $r(S', T') \leq r(S^c, T^c) + \Gamma/(16m) = v + \alpha'$. □

THEOREM 3. *Suppose that the algorithm maintains the dynamic transitive closure of the abundance graph. Suppose further that each compact network $G^c = (N^c, A^c)$ has at most $n^{16/9}$ nodes. Then the time it takes to create G^c is $O(m^{9/8})$.*

PROOF. We assume that the parameter Γ is given, and we do not consider the time to compute Γ here. (We consider how to compute Γ in the next section.) The first step is to contract the abundant cycles. The strongly connected components of the abundance graph can be determined in $O(m)$ time, and thus the cycles can be contracted in $O(m)$ time. The contracted cycles can be expanded at the end of an improvement phase in $O(m)$ time.

The time it takes to determine the critical and compactible nodes is $O(m)$. The time it takes to determine the set A^1 of original arcs is $O(m)$. To determine the set A^2 of abundant pseudo-arcs, one can inspect $O(|N^c|^2)$ components of the matrix $\mathbf{M}$ in $O(|N^c|^2)$ time. Because $|N^c| < m^{16/9}$, this time is $O(m^{9/8})$. The time it takes to determine the set A^3 of anti-abundant pseudo-arcs is $O(m \log n)$ using dynamic trees. The proof of this latter result appears in Appendix B. □

7. MAXIMUM FLOWS IN $O(nm)$ TIME

In this section, we show that for $m < n^{1.06}$, the running time for our max flow algorithm is $O(nm)$. 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. The procedure considers three different cases according to the number of Δ-critical nodes.

Procedure *Improve-approx-2*(r, S, T);
01. $\Delta := r(S, T)$;
02. let C be the number of Δ-critical nodes;
03. **if** $C \geq m^{9/16}$ **then** let $\Gamma = \Delta$;
04. find a $\Gamma/(8m)$-optimal flow in $G[r]$;
05. **else, if** $m^{1/3} \leq C < m^{9/16}$ **then** let $\Gamma = \Delta$;
06. let G^c denote the Γ-compact network;
07. find a $\Gamma/(16m)$-optimal flow y on G^c;
08. let y' be the induced $\Gamma/(8m)$-opt flow on G[r];
09. update the residual capacities;
10. **else, if** $C < m^{1/3}$ **then**
11. choose the minimum value Γ such that
12. the number C of Γ-critical nodes in the
13. network is less than $m^{1/3}$;
14. let G^c denote the Γ-compact network;
15. find an optimal flow y on G^c;
16. let y' be the induced $\Gamma/(16m)$-opt flow in $G[r]$;
17. update the residual capacities;

The $\Gamma/(8m)$-optimality of the flow in line 08 follows from Theorem 2. Similarly, the $\Gamma/(16m)$-optimality of the flow in line 16 follows from Theorem 2. Accordingly, the improvement phases satisfy the improvement property described in Section 5.

If we run Procedure *improve-approx-2* at each improvement phase, it will eventually determine the optimum solution. We will show that the running time is $O(nm + m^{31/16} \log^2 n)$ over all improvement phases. Our proof relies on four lemmas that establish the following:

1. The number of improvement phases is $O(m^{2/3})$.
2. The time to create all of the compact networks is $O(nm + m^{43/24})$.
3. The time to find all of the approximately optimal or optimal flows is $O(m^{31/16} \log^2 n)$.
4. Given the flows in compact networks, the time it takes to find the induced flows over all iterations is $O(nm + m^{5/3} \log n)$.

We now address the subtlety that led us to consider the parameter Γ in the first place. The time for our procedure to create a compact network is bounded below by $m \log n$. To achieve a time bound of $O(nm)$, we need to bound the number of improvement phases. The choice of Γ in lines 11 to 13 ensure that the number of compact networks is $O(m^{2/3})$, as we state and prove in Lemma 7.

We also address a second subtlety. The induced flows (as described in the discussion following Lemma 3) includes a term that is n times the number of abundant pseudo-arcs with positive flow. To maintain $O(nm)$ as an upper bound on running time, we need to ensure that the number of abundant pseudo-arcs with positive flow is at most C, the number of Γ-critical nodes. So, prior to finding the flow induced by y, we transform y into an equivalent flow in which there are at most C abundant pseudo-arcs with positive flow. This can be accomplished efficiently using the dynamic tree data structure. We will mention this again later in this section, and will provide more detail in Appendix B.

LEMMA 7. *The number of improvement phases is $O(m^{2/3})$.*

PROOF. By Theorem 1, the number of Γ-critical nodes over all improvement phases is $O(m)$. The proof of Theorem 1 also shows that the number of $(\Gamma/2)$-critical nodes over all improvement phases is $O(m)$. In each case in Procedure *improve-approx-2*, the number of $(\Gamma/2)$-critical nodes is at least $m^{1/3}$. Therefore, the number of improvement phases is $O(m^{2/3})$. □

LEMMA 8. *The time to create all of the compact networks is $O(nm + m^{43/24})$.*

PROOF. One aspect of creating the compact network is the selection of the parameter Γ in lines 11-13. The parameter Γ can be chosen in $O(m + n \log n)$ time as follows. For each node j, we scan its incident arcs to compute the greatest value of Γ for which j is in the Γ-compact network. We then sort the nodes by these values and select the minimum value of Γ such that the compact network has at most $m^{1/3}$ nodes.

We now consider the remaining time to create all of the compact networks. By Theorem 3, the time to create each compact network is $O(m^{9/8})$ assuming that we maintain the transitive closure of the abundance graph. The time for maintaining the transitive closure is $O(nm)$. Because the number of distinct compact networks is $O(m^{2/3})$, the total time for maintaining the transitive closure and creating the compact networks is $O(nm + m^{43/24})$. □

The following lemma refers to the time it takes to find the approximately optimal flows in lines 04 and 07 and the optimal flows in line 15 of *improve-approx-2*. It does not refer to the time to find induced flows in the original network.

LEMMA 9. *Procedure improve-approx-2 determines the optimal or approximately optimal flows in all of the compact networks in $O(m^{31/16}\log^2 n)$ time. In iterations in which the number of Δ-critical nodes is less than $m^{9/16}$, the time to find approximately optimal flows in the residual networks is $O(m^{31/16}\log n)$.*

PROOF. Let C denote the number of Γ-critical nodes in the network in one iteration of *improve-approx-2*. Let T denote an upper bound on the running time for the max flow subroutine of Procedure *improve-approx-2*. We will next bound the ratio T/C. If $C \geq m^{9/16}$, then $T = O(m^{3/2}\log^2 n)$, and $T/C = O(m^{15/16}\log^2 n)$. (Recall that an improvement phase requires $O(\log n)$ of the usual scaling phases of the Goldberg-Rao algorithm.) If $m^{1/3} \leq C < m^{9/16}$, then $T = O(C^{8/3}\log n)$. Therefore, $T/C = O(C^{5/3}\log n) = O(m^{15/16}\log n)$. If $C < m^{1/3}$, then $T = O(C^3)$, and $T/C = O(C^2) = O(m^{2/3})$. In all cases, the time for finding the flow is at most $O(m^{15/16}\log^2 n)$ per Γ-critical node. By Theorem 1, the number of Γ-critical nodes over all improvement phases is $O(m)$. Therefore, the total time for finding flows is $O(m^{31/16}\log^2 n)$. □

LEMMA 10. *The total time it takes to transform the flows in compact networks to the flows in the residual networks is $O(nm + m^{5/3}\log n)$.*

PROOF. Let $C = |N^c|$ be the number of nodes in the compact network at some phase. Let y be the flow in the compact network G^c. Recall that $A^c = A^1 \cup A^2 \cup A^3$. Let y' be the induced flow in the residual network. We obtain y' as follows. If $(i,j) \in A^1$, then $y'_{ij} = y_{ij}$.

If $(i,j) \in A^2$, we can determine the abundant path P from i to j in $G[r]$ in $O(n)$ time because we maintain the transitive closure of G^{ab}. We can then replace y_{ij} in G^c by increasing the flow in each arc of P by the amount y_{ij}. This approach takes $O(n)$ time for each arc of A^2. In Appendix B.3 we show how to reduce the number of arcs with positive flow in A^2 to fewer than C. The time for this procedure is $O(m\log n)$ per improvement phase, and thus $O(m^{5/3}\log n)$ over all phases.

Subsequently, there are fewer than C arcs of A^2 with positive flow. Computing the flow in G that is induced by these arcs takes $O(nC)$ time in an improvement phase and $O(nm)$ over all improvement phases.

We next consider arcs in A^3. If $(i,j) \in A^3$, we transform the flow y_{ij} into the flow on path(s), using dynamic trees. We describe this procedure in Appendix B.2. The time for this procedure is $O(m\log n)$ per improvement phase, and thus $O(m^{5/3}\log n)$ over all phases.

The time for maintaining the transitive closure of the abundance graph and the time for transforming flows from abundant pseudo-arcs is $O(nm)$. Thus the total time for this procedure over all phases is $O(nm + m^{5/3}\log n)$. □

We summarize the results of this section with a theorem.

THEOREM 4. *If the flow in each improvement phase is obtained using improve-approx-2, then the running time to find a maximum flow is $O(nm + m^{31/16}\log^2 n)$. If $m = O(n^{1.06})$, the running time is $O(nm)$.*

8. A SPEEDUP FOR SPARSE NETWORKS

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 Γ-critical nodes in all iterations is $O(n)$. In order to achieve the $O(n^2/\log n)$ running time, we need to create a compact network with C nodes in $O(Cn/\log n)$ time. We also need to transform the flow in the compact network into a flow in the residual network in $O(Cn/\log n)$ time.

To determine the abundant pseudo-arcs, our procedure determines all nodes of G^{ab} reachable from the C critical nodes using an abundant path. A standard implementation of a breadth first search takes $O(m)$ time per critical node and $O(Cm)$ time in total. We obtain a factor $\log n$ speedup using an approach due to Gabow and Tarjan [6] in the context of a set union data structure. (A related and more general approach is due to Blelloch et al. [2]).

Let $K = \lfloor(\log n)/3\rfloor$. In a similar manner to Gabow and Tarjan, we represent subsets of the ground set $\mathcal{S} = \{1, 2, 3, ..., K\}$ using integers in the range $[0, n^{1/3}]$. We consider the case in which every element $i \in \mathcal{S}$ has an associated value a_i. In this case, we create six tables in $O(n)$ time so that each operation on one or two subsets of $\mathcal{S}$ takes $O(1)$ time using table look-up, improving upon the usual running time by a factor of $\log n$.

Our algorithm relies on the following six operations.

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.

The procedure *forward-search* determines in $O(m)$ time the set of pairs $\{i, j : i \in \mathcal{S}, j \in N,$ and $i \Rightarrow j\}$. By carrying out this procedure from all nodes of N^c, K nodes at a time, one can obtain all of the abundant pseudo-arcs of A^c in $O(mC/\log n)$ time.

We assume that the arc set A^{ab} has no directed cycles, or equivalently that we have already contracted the abundant directed cycles. We then topologically order the nodes in $O(m)$ time so that if $(i,j) \in A^{ab}$, then $i < j$.

For each $j \in N$, 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 $F(i,j) = \{k \in \mathcal{S} : (i,j) \in P_k(j)\}$. The procedure *forward-search* determines $F(\cdot)$ and $F(\cdot,\cdot)$.

Procedure *forward-search*;
01. Initialize;
02. **for each** $i \in \mathcal{S}$, $F(i) := \{i\}$;
03. **for each** $j \in N\backslash\mathcal{S}$, $F(j) := \emptyset$;
04. **for each** $(i,j) \in A^{ab}$, $F(i,j) := \emptyset$;
05. scan nodes of N in topological order;
06. **for each** node $i \in N$ and **for each** $(i,j) \in A^{ab}$ **do**
07. $F(i,j) := F(i)\backslash F(j)$;
08. $F(j) := F(i) \cup F(j)$;

Line 07 identifies nodes of S that can reach node j using arc (i,j), and for which no previous path from that node to node j had been found. Line 08 updates $F(j)$.

Procedure *forward-search* correctly identifies the sets $F(\cdot)$ and $F(\cdot,\cdot)$ in $O(m)$ time. This procedure can be used to create the compact networks, and obviates a need for maintaining the transitive closure of G^{ab}.

We now consider the other bottleneck in the max flow algorithm, that of transforming flows on abundant pseudo-arcs in the compact network into flows on paths in $G[r]$. As in the proof of Lemma 10 and as described in Appendix B.3, we first find an equivalent flow in G^c with fewer than C abundant pseudo-arcs with positive flow. Let y^c denote the resulting flow in G^c, as restricted to the abundant pseudo-arcs. We find the flow in $G[r]$ induced by y^c in three stages, as described next.

1. In the first stage, there is a node $i \in N^c$ that is incident to at least K pseudo-arcs with positive flow in y^c.
2. In the second stage, one uses a greedy algorithm to determine K independent arcs with positive flow. (A set of arcs is independent if no two arcs have a node of N^c in common.)
3. The second stage ends and the third stage begins when the greedy algorithm fails to determine K independent arcs.

Consider the first of these stages. Let $i \in N^c$ be a node incident to at least K abundant pseudo-arcs with positive flow. Let $W = \{j : y^c_{ij} > 0\}$. We next show how find the flows induced by the arcs $(i,j) : j \in W$. (A similar procedure shows how to find the flows induced by arcs entering node i.)

Using breadth first search, determine a tree $T \subseteq G^{ab}$ directed out of node i and containing all nodes j such that $i \Rightarrow j$. Thus $W \subseteq T$. Then convert the flows y^c_{ik} for $k \in W$ into a flow y' for G as follows: y' is the unique flow 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 $\sum_{k \in W} y_{ik}$. The time to carry out this procedure for node i is $O(m)$.

Then we carry out in $O(m)$ time an analogous procedure for all arcs with positive flow directed into node i. Subsequently, we eliminate node i and all incident arcs from the compact network. We repeat this procedure until there is no node in G^c that has at least K incident abundant pseudo-arcs with positive flow. Then we go to the second stage.

In the second stage, we use a greedy algorithm to determine K independent pseudo-arcs with positive flow. We let y denote the flow as restricted to these K arcs. The greedy algorithm requires $O(m)$ time to identify these arcs. If the greedy algorithm fails to find K independent arcs, our procedure moves on to the third stage.

Suppose that the greedy algorithm succeeded in obtaining K independent arcs with positive flow. We next exploit the independence of the pseudo-arcs and we relabel the nodes so that the K pseudo-arcs are $(i, K+i)$ for $i = 1$ to K.

We then run *forward-search* to determine $F(\cdot)$ and $F(\cdot,\cdot)$, which are defined as above. This procedure implicitly determines the paths $P_k(j)$ for all $k \in S$ and $j \in N$. For each $k \in S$, we will be sending flow on the path $P_k(K+k)$, which is the abundant path corresponding to pseudo-arc $(k, K+k)$. And we will be sending all K flows in a total of $O(m)$ time.

Let $B(i,j) = \{k \in [1,K] : (i,j) \in P_k(K+k)\}$. Let $B(j) = \{k \in [1,K] : j \in P_k(K+k)\}$. The procedure *backward-search* determines $B(i,j)$ and $B(j)$ by relying 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 determine $B(i,j)$ and $B(j)$ in Lines 1 to 8 of procedure *backward-search*. Line 09 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 all** $(i,j) \in G^{ab}$, **do** $y'_{ij} := y'_{ij} + \sum_{k \in B(i,j)} y_{k,k+K}$
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 $[1,K]$.

Eventually, there is an iteration in Stage 2 in which the greedy algorithm fails to determine K independent arcs with positive flow. Because each node is incident to fewer than K arcs with positive flow (because Stage 1 has ended), and because the greedy algorithm failed, it follows that that there are fewer than $2K^2$ abundant pseudo-arcs remaining that have positive flow. These final arcs can be transformed iteratively in $O(mK^2) = O(m \log^2 n)$ time, which is not a bottleneck.

Summarizing the results of the three stages, when $m = O(n)$, we can find the flows induced by C abundant pseudo-arcs in $O(nC/\log n)$ time. This is $O(nm/\log n)$ time over all improvement phases.

9. 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.

10. 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] G. Blelloch, V. Vassilevska, and R. Williams. A new combinatorial approach for sparse graph problems. *Automata, Languages and Programming*, pages 108–120, 2008.

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

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

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

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

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

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

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

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

APPENDIX

A. DYNAMIC TREES

Sleator and Tarjan [10] developed the data structure dynamic trees, which supports various operations, each with an amortized time complexity of $O(\log n)$ per operation. That is, over a sequence of $q > n$ consecutive operations on the dynamic trees, the running time is $O(q \log n)$. In this section, we present a variant of dynamic trees in which the trees maintain both nodes and arcs. Each of the operations presented here can be simulated via $O(1)$ operations of the standard version of dynamic trees. In the next section, we apply dynamic trees to the procedures described earlier in this paper.

A dynamic tree is a data structure that represents a dynamically changing rooted forest T on a directed graph $D = (V, E)$. The root node in the component of T containing node i will be denoted as $root(i)$. For each non-root node j, we let $p(j)$ denote the node that follows node j on the path from j to $root(j)$. Associated with each arc $(i, j) \in E$ is a non-negative real number $value(i, j)$, which represents residual capacities in our applications of dynamic trees.

In the standard description of dynamic trees, the value associated with node i implictly represents a value on arc $(i, p(i))$. In the description here, if $i \in T$, then we associate values with arcs $(i, p(i))$ and $(p(i), i)$ rather than place values on the nodes. Our description can be implemented using the original data structure by maintaining two values for node i, one representing arc $(i, p(i))$ and one representing arc $(p(i), i)$.

If nodes i and j are in the same component of T, we let $Path(i, j)$ denote the unique path in T from node i to node j.

We next list eight dynamic tree operations that are sufficient for our purposes. We assume that the reader is familiar with the dynamic tree data structure, as described in [10].

(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).

(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.

(iv) *find-root*(i). Returns $root(i)$.

(v) *make-root*(j). This operation makes node j the root of its component in the dynamic tree.

(vi) *add-value*(i, j, val). This operation replaces $value(k, \ell)$ by $value(k, \ell) + val$ for all arcs (k, ℓ) of $Path(i, j)$. This operation is only applied if i and j are in the same component of T.

(vii) *find-min-value*(i, j). This operation finds $\min\{value(k, \ell) : (k, \ell) \in Path(i, j)\}$.

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

The operations (i), (ii), (iii), (iv) and (v) are all operations that are implemented in a standard version of dynamic trees. Operations (vi), (vii) and (viii) are new to this paper, but they are easily implemented. If $j = root(i)$, then the operations *add-value*(i, j, val), *find-min-value*(i, j), and *find-min*(i, j) are essentially the same as standard dynamic tree operations described in [10]. If $j \neq root(i)$, then the operations can be carried out by first making j the root of its component.

Each dynamic tree operation takes $O(\log n)$ time, including the time it takes to compute $value(i, p(i))$ whenever $i \in T$.

B. APPLICATIONS OF DYNAMIC TREES

In this section, we outline how to apply dynamic trees to three different procedures: (1) transferring residual capacities from paths to anti-abundant pseudo-arcs, (2) transforming flows in anti-abundant pseudo-arcs to flows in the original network, and (3) transforming a flow in abundant pseudo-arcs into an equivalent flow in a forest.

B.1 Transferring residual capacities

We first consider the procedure *transfer-capacity*. This procedure carries out a depth first search. Nodes and arcs of the depth first search path are added to the dynamic tree via *link* operations. This implies that each arc is added to the dynamic tree at most once and deleted at most once, resulting in $O(m)$ links and cuts in total. Removing the residual capacity of a path in line 12 relies on the operation *add-value*. Accordingly, the procedure *transfer-capacity* takes $O(m \log n)$ time per improvement phase using dynamic trees.

B.2 Flows induced by anti-abundant pseudo-arcs

After finding a flow y in the compact network G^c, one needs to be able to transform y into a flow on $G[r]$. We next describe the transformation. Let y^3 denote the flows in y as restricted to the anti-abundant pseudo-arcs of A^3.

There is no efficient way of storing all of the paths that were determined using the procedure *transfer-capacity*. Instead, we store the operations that were carried out on dynamic trees in the procedure *transfer-capacity*. We later reproduce these operations and recreate paths as needed.

We assume that the procedure *transfer-capacity* was enhanced so that it maintained a list of records that we refer to as *OpList*. Thus *OpList* contains a list of records of the

dynamic tree operation executed during Procedure *transfer-capacity*.

We let $op(k)$ denote the k-th dynamic tree operation. Thus, $op(k)$ is "$link$" or "cut" or "$find-root$", etc. If $op(k)$ = "*add-value*", then this corresponds to the case in which capacity was reduced in a path P. In this case, we let $\alpha(k)$ denote the first node of P. And we let $\gamma(k)$ denote the amount by which the residual capacity in P was reduced. The last node of P is $root(\alpha(k))$. All of these are part of the $k-$th record of *OpList* .

Recall that there may be more than one path from node i to node j that had its residual capacity transferred. These arcs in A^c were combined into a single pseudo-arc (i, j). We consider these paths one at a time in the procedure *transform-flows*.

The following procedure determines the flow induced by y^3. We initialize by letting $w = y^3$. We use the notation T' to represent the dynamic tree.

Procedure *transform-flows*(y^3);
01. $w := y^3$; $y' := 0$;
02. create an empty dynamic tree T';
03. $K :=$ number of records of OpList;
04. **for** $k = 1$ to K **do**
05. **if** op(k) $\neq$ "add-value", **then** reproduce
06. the k-th operation of OpList on T';
07. **else do**
08. $i := \alpha(k)$
09. $\delta := \min\{\gamma(k), w_{i,root(i)}\}$;
10. **if** $\delta > 0$ **then do**
11. *add-value*(i, δ);
12. // that is, $y'_{j\ell} := y'_{j\ell} + \delta$ for all $(j, \ell) \in P_k$ //
13. $w_{i,root(i)} := w_{i,root(i)} - \delta$;

There may be many pseudo-arcs that were aggregated into a single arc (i, j). Lines 7 to 12 ensure that the flow y^3_{ij} is transformed into a flow in $G[r]$ one path at a time, in the order that the paths were created in procedure *transfer capacity*. The flow added to path P in *transform-flows* is bounded by the residual capacity that was subtracted from P in *transfer capacity*.

THEOREM 5. *The procedure transform-flows determines the flows induced by the non-abundant pseudo-arcs of G^c in $O(m \log n)$ time per improvement phase.*

B.3 Adjusting flows in abundant pseudo-arcs

After finding a flow y in the compact network, one needs to ensure that there are at most C abundant pseudo arcs (that is, arcs of A^2) with positive flow, where $C = |N^c|$. This can be achieved by sending flow around cycles of abundant arcs. (To send δ units of flow around a cycle is to increase the flow in the forward arcs of the cycle by δ and to decrease the flow in the backward arcs of the cycle by δ.

The following procedure is well known in dynamic trees folklore, but this author was unable to locate the appropriate reference.

We let y^2 denote the flow y as restricted to arcs of A^2. We will transform y^2 into an equivalent flow y' in G^c such that the subset of arcs with positive flow in y' is a forest. We do so by sending flows around cycles. When we consider an arc (i, j) with $y^2_{ij} > 0$, we send flow around the cycle in $T' \cup \{(j, i)\}$. That is, we send flow around the cycle so as to reduce the flow in (i, j).

In this dynamic tree procedure, we first represent the flows in abundant arcs using a vector r' of residual capacities. If $y^2_{ij} > 0$, we let $r'_{ji} = y^2_{ij}$. Because (i, j) is abundant, we let $r'_{ij} = \infty$. Increasing the flow in (i, j) by δ is achieved by decreasing r'_{ij} by δ and increasing r'_{ji} by δ. The following procedure transforms y^2 into an equivalent flow on a forest in $O(m \log n)$ time. In the dynamic tree T, $value(i, j)$ refers to r'_{ij}.

Procedure *flows-around-cycles*(y^2);
01. $A' := \{(i, j) : y^2_{ij} > 0\}$;
02. **for all** $(i, j) \in A'$, $r'_{ij} := \infty$ and $r'_{ji} := y^2_{ij}$;
03. create an empty dynamic tree T;
04. for each arc $(i, j) \in A'$ **do**
05. **if** $root(i) \neq root(j)$, then $link(i, j)$;
06. **else do** //send flow around the cycle//
07. $\gamma :=$ *find-min-value*(i, j);
08. $\delta := \min\{r'_{ji}, \gamma\}$;
09. $r'_{ji} := r'_{ji} - \delta$; $r'_{ij} := r'_{ij} + \delta$;
10. *add-value*$(i, j, -\delta)$ and delete
11. any arc (k, ℓ) from T if $r'_{k\ell} = 0$.
12. *add-value*(j, i, δ);
13. **if** $root(i) \neq root(j)$, then $link(i, j)$;

The deletion in lines 10 and 11 can be achieved by using the operations *find-min* and *cut*. The number of links is $O(C^2)$. The number of all other dynamic tree operations is also $O(C^2)$. We conclude with the following theorem.

THEOREM 6. *Procedure flows-around-cycles(y^2) transforms a flow on abundant pseudo-arcs of A^2 into an equivalent flow in which there are fewer than C arcs with positive flow. The running time for the procedure is $O(C^2 \log n)$ per improvement phase.*