SIAM J. COMPUT.
Vol. 18, No. 5, pp. 939-954, October 1989

IMPROVED TIME BOUNDS FOR THE MAXIMUM FLOW PROBLEM*

RAVINDRA K. AHUJA†‡, JAMES B. ORLIN†, AND ROBERT E. TARJAN§

Abstract. Recently, Goldberg proposed a new approach to the maximum network flow problem. The approach yields a very simple algorithm running in $O(n^3)$ time on n-vertex networks. Incorporation of the dynamic tree data structure of Sleator and Tarjan yields a more complicated algorithm with a running time of $O(nm \log(n^2/m))$ on m-arc networks. Ahuja and Orlin developed a variant of Goldberg's algorithm that uses scaling and runs in $O(nm + n^2 \log U)$ time on networks with integer arc capacities bounded by U. In this paper possible improvements to the Ahuja-Orlin algorithm are explored. First, an improved running time of $O(nm + n^2 \log U/\log\log U)$ is obtained by using a nonconstant scaling factor. Second, an even better bound of $O(nm + n^2(\log U)^{1/2})$ is obtained by combining the Ahuja-Orlin algorithm with the wave algorithm of Tarjan. Third, it is shown that the use of dynamic trees in the latter algorithm reduces the running time to $O(nm \log((n/m)(\log U)^{1/2} + 2))$. This result shows that the combined use of three different techniques results in speed not obtained by using any of the techniques alone. The above bounds are all for a unit-cost random access machine. Also considered is a semilogarithmic computation model in which the bounds increase by an additive term of $O(m \log_n U)$, which is the time needed to read the input in the model.

Key words. maximum flow, network algorithm, combinatorial optimization, scaling

AMS(MOS) subject classifications. 68Q20, 68Q25, 90C35

1. Introduction. We consider algorithms for the classical maximum network flow problem [5], [6], [13], [15], [20]. We formulate the problem as follows. Let $G = (V, E)$ be a directed graph with vertex set V and arc set E. The graph G is a *flow network* if it has two distinct distinguished vertices, a *source* s and a *sink* t, and a nonnegative real-valued *capacity* $u(v, w)$ on each arc $(v, w) \in E$. We assume that G is *symmetric*, i.e., $(v, w) \in E$ if and only if $(w, v) \in E$. We denote by n, m, and U the number of vertices, the number of arcs, and the maximum arc capacity, respectively. For ease in stating time bounds, we assume $m \geqq n$ and $U \geqq 4$. Bounds containing U are subject to the assumption that all arc capacities are integral. All logarithms in the paper are base two unless an explicit base is given.

A *flow* f on a network G is a real-valued function f on the arcs satisfying the following constraints:

(1) $f(v, w) \leqq u(v, w)$ for all $(v, w) \in E$ (capacity constraint);

(2) $f(v, w) = -f(w, v)$ for all $(v, w) \in E$ (antisymmetry constraint);

(3) $\sum_{(v,w)\in E} f(v, w) = 0$ for all $w \in V - \{s, t\}$ (conservation constraint).

* Received by the editors November 25, 1987; accepted for publication (in revised form) September 26, 1988.

† Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The research of these authors was supported by a Presidential Young Investigator Fellowship for the National Science Foundation, contract 8451517 ECS, grant AFOSR-88-0088 from the Air Force Office of Scientific Research, and grants from Analog Devices, Apple Computer Inc., and Prime Computer.

‡ This work was done while the author was on leave from the Indian Institute of Technology, Kanpur, India.

§ Department of Computer Science, Princeton University, Princeton, New Jersey 08544, and AT&T Bell Labs, Murray Hill, New Jersey 07974. The research of this author was partially supported by National Science Foundation grant DCR-8605962 and Office of Naval Research contract N00014-87-K-0467.

The *value* $|f|$ of a flow f is the net flow into the sink:

$$|f| = \sum_{(v,t)\in E} f(v, t).$$

A *maximum flow* is a flow of maximum value. The *maximum flow problem* is that of finding a maximum flow in a given network.

Remark. We assume that all arc capacities are finite. If some arc capacities are infinite but no path of infinite-capacity arcs from s to t exists, then each infinite capacity can be replaced by the sum of the finite capacities without affecting the problem. $\square$

The maximum flow problem has a long, rich history, and a series of faster and faster algorithms for the problem has been developed. (See [10] for a brief survey.) Until now the fastest known algorithms have been the one of Goldberg and Tarjan [9], [10] (running in time $O(nm \log (n^2/m))$) and the one of Ahuja and Orlin [1] (running in time $O(nm + n^2 \log U)$). Both of these algorithms are refinements of a generic method proposed by Goldberg [8], which we shall call the *preflow algorithm.* For networks with $m = \Omega(n^2)$, the Goldberg-Tarjan bound is $O(n^3)$, which matches the bound of several earlier algorithms [12], [14], [16], [21]. For networks with $m = O(n^{2-\varepsilon})$ for some constant $\varepsilon > 0$, the Goldberg-Tarjan bound is $O(nm \log n)$, which matches the bound of the earlier Sleator-Tarjan algorithm [17], [18]. Under the *similarity assumption* [7], namely, $U = O(n^\gamma)$ for some constant γ, the Ahuja-Orlin bound beats the Goldberg-Tarjan bound unless $m = O(n)$ or $m = \Omega(n^2)$.

The Goldberg-Tarjan and Ahuja-Orlin algorithms obtain their speed from two different techniques. The former uses a sophisticated data structure, the dynamic tree structure of Sleator and Tarjan [18], [19], [20], whereas the latter uses scaling. In this paper we explore improvements in the Ahuja-Orlin algorithm obtained by incorporating other ideas, including the use of dynamic trees. We begin in § 2 by reviewing the generic preflow algorithm [8], [9], [10]. In § 3, we develop a version of the Ahuja-Orlin algorithm that uses a stack-based vertex selection rule and a nonconstant scaling factor to obtain a time bound of $O(nm + n^2 \log U/\log \log U)$. In § 4, we describe an even faster variant that uses a constant scaling factor but combines the scaling idea with the *wave algorithm* of Tarjan [21]. This algorithm has a running time of $O(nm + n^2 (\log U)^{1/2})$. In § 5, we add dynamic trees to the method of § 4, thereby obtaining a running time of $O(nm \log ((n/m)(\log U)^{1/2} + 2))$. The results in §§ 3-5 rely on the assumption that pointer manipulations and arithmetic operations on integers of magnitude U take $O(1)$ time. In § 6, we consider the effect on our time bounds of a weaker assumption, namely, that arithmetic on integers of magnitude n takes $O(1)$ time. The extra time needed by our algorithms in this *semilogarithmic computation model* is an additive term of $O(m \log_n U)$, which is the same as the time needed to read the input. We conclude in § 7 with some final remarks.

2. The preflow algorithm.

In contrast to the classical *augmenting path method* of Ford and Fulkerson [6] that moves flow along an entire path from s to t at once, the preflow method moves flow along a single arc at a time. The key concept underlying the algorithm is that of a *preflow*, introduced by Karzanov [12]. A preflow f is a real-valued function on the arcs satisfying constraints (1), (2), and a relaxation of (3). For any vertex w, let the *flow excess* of w be $e(w) = \sum_{(v,w)\in E} f(v, w)$. The required constraint is the following:

$$e(w) \geqq 0 \quad \forall w \in V - \{s\} \text{ (preflow constraint).} \tag{4}$$

We call a vertex v *active* if $v \neq t$ and $e(v) > 0$. Observe that the preflow constraint implies that $e(s) \leqq 0$.

The *residual capacity* of an arc (v, w) with respect to a preflow f is $u_f(v, w) = u(v, w) - f(v, w)$. An arc is *saturated* if $u_f(v, w) = 0$ and *unsaturated* otherwise. (The capacity constraint implies that any unsaturated arc (v, w) has $u_f(v, w) > 0$.)

The preflow algorithm maintains a preflow and moves flow from active vertices through unsaturated arcs toward the sink, along paths estimated to contain as few arcs as possible. Excess flow that cannot be moved to the sink is returned to the source, also along estimated shortest paths. Eventually the preflow becomes a maximum flow.

As an estimate of path lengths, the algorithm uses a *valid labeling* that is a function d from the vertices to the nonnegative integers such that $d(s) = n$, $d(t) = 0$, and $d(v) \leqq d(w) + 1$ for every unsaturated arc (v, w). A proof by induction shows that, for any valid labeling d, $d(v) \leqq \min\{d_f(v, s) + n, d_f(v, t)\}$, where $d_f(v, w)$ is the minimum number of arcs on a path from v to w consisting of arcs unsaturated with respect to the flow f. We call an arc (v, w) *eligible* if (v, w) is unsaturated and $d(v) = d(w) + 1$.

The algorithm begins with an initial preflow f and a valid labeling d defined as follows:

$$f(v, w) = \begin{cases} u(v, w) & \text{if } v = s, \\ -u(w, v) & \text{if } w = s, \\ 0 & \text{if } v \neq s \text{ and } w \neq s, \end{cases}$$

$$d(v) = \min\{d_f(v, s) + n, d_f(v, t)\}.$$

The algorithm consists of repeating the following two steps, in any order, until no vertex is active:

Push(v, w).
Applicability: Vertex v is active and arc (v, w) is eligible.
Action: Increase $f(v, w)$ by $\min\{e(v), u_f(v, w)\}$. The push is *saturating* if (v, w) is saturated after the push and *nonsaturating* otherwise.

Relabel(v).
Applicability: Vertex v is active and no arc (v, w) is eligible.
Action: Replace $d(v)$ by $\min\{d(w) + 1 \mid (v, w) \text{ is unsaturated}\}$.

When the algorithm terminates, f is a maximum flow. Goldberg and Tarjan [10] derived the following bounds on the number of steps required by the algorithm:

LEMMA 2.1 [10]. *Relabeling a vertex v strictly increases $d(v)$. No vertex label exceeds $2n - 1$, and the total number of relabelings is $O(n^2)$.*

LEMMA 2.2 [10]. *There are at most $O(mn)$ saturating pushes and at most $O(n^2m)$ nonsaturating pushes.*

Efficient implementations of the above algorithm require a mechanism for selecting pushing and relabeling steps to perform. Goldberg and Tarjan proposed the following method: For each vertex, construct a (fixed) list $A(v)$ of the arcs out of v. Designate one of these arcs, initially the first on the list, as the *current arc* out of v. To execute the algorithm, repeat the following step until there are no active vertices:

Push/Relabel(v).
Applicability: Vertex v is active.
Action: If the current arc (v, w) of v is eligible, perform *push*(v, w). Otherwise, if (v, w) is not the last arc on $A(v)$, make the next arc after (v, w) the

current one. Otherwise, perform *relabel*(v) and make the first arc on $A(v)$ the current one.

With this implementation, the algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating push. This gives an $O(n^2m)$ time bound for any order of selecting vertices for push/relabel steps. Making the algorithm faster requires reducing the time spent on nonsaturating pushes. The number of such pushes can be reduced by selecting vertices for push/relabel steps carefully. Goldberg and Tarjan showed that first-in, first-out selection (first active, first selected) reduces the number of nonsaturating pushes to $O(n^3)$. Cheriyan and Maheshwari [3] showed that highest label selection (always pushing flow from a vertex with highest label) reduces the number of nonsaturating pushes to $O(n^2m^{1/2})$. (The latter rule was first proposed by Goldberg [8], who gave an $O(n^3)$ bound.) Ahuja and Orlin proposed a third selection rule, which we discuss in the next section.

3. The scaling algorithm. The intuitive idea behind the Ahuja-Orlin algorithm, henceforth called the *scaling algorithm*, is to move large amounts of flow when possible. The same idea is behind the maximum capacity augmenting path method of Edmonds and Karp [4] and the capacity scaling algorithm of Gabow [7]. One way to apply this idea to the preflow algorithm is to always push flow from a vertex of large excess to a vertex of small excess, or to the sink. The effect of this is to reduce the maximum excess at a rapid rate.

Making this method precise requires specifying when an excess is large and when it is small. For this purpose the scaling algorithm uses an *excess bound* Δ and an integer *scaling factor* $k \geqq 2$. A vertex v is said to have *large excess* if its excess exceeds Δ/k and *small excess* otherwise. As the algorithm proceeds, k remains fixed, but Δ periodically decreases. Initially, Δ is the smallest power of k such that $\Delta \geqq U$. The algorithm maintains the invariant that $e(v) \leqq \Delta$ for every active vertex v. This requires changing the pushing step to the following:

Push (v, w).
Applicability: Vertex v is active and arc (v, w) is eligible.
Action: If $w \neq t$, increase $f(v, w)$ by $\min\{e(v), u_f(v, w), \Delta - e(w)\}$. Otherwise, $(w = t)$, increase $f(v, w)$ by $\min\{e(v), u_f(v, w)\}$.

The algorithm consists of a number of *scaling phases*, during each of which Δ remains constant. A phase consists of repeating push/relabel steps, using the following selection rule, until no active vertex has large excess, and then replacing Δ by Δ/k. The algorithm terminates when there are no active vertices.

Large excess, smallest label selection: Apply a push/relabel step to an active vertex v of large excess; among such vertices, choose one of smallest label.

If the edge capacities are integers, the algorithm terminates after at most $\lfloor \log_k U+1 \rfloor$ phases. After $\lfloor \log_k U+1 \rfloor$ phases, $\Delta < 1$, which implies that f is a flow, since the algorithm maintains integrality of vertex excesses. Ahuja and Orlin derived a bound of $O(kn^2 \log_k U)$ on the total number of nonsaturating pushes. We repeat the analysis here, since it provides motivation for our first modification of the algorithm.

LEMMA 3.1 [1]. *The total number of nonsaturating pushes in the scaling algorithm is* $O(kn^2(\log_k U+1))$.

Proof. Consider the function $\Phi = \sum_{v \text{ active}} e(v)\, d(v)/\Delta$. We call Φ the *potential* of the current preflow f and labeling d. Since $0 < e(v)/\Delta \leqq 1$ and $0 \leqq d(v) \leqq 2n$ for every active vertex v, $0 \leqq \Phi \leqq 2n^2$ throughout the algorithm. Every pushing step decreases Φ. A nonsaturating pushing step decreases Φ by at least $1/k$, since the push is from a vertex v with excess more than Δ/k to a vertex w with $d(w)=d(v)-1$, and $e(w)\leqq\Delta/k$ or $w=t$. The value of Φ can increase only during a relabeling or when Δ changes. A relabeling of a vertex v increases Φ by at most the amount $d(v)$ increases. Thus the total increase in Φ due to relabelings, over the entire algorithm, is at most $2n^2$. When Δ changes, Φ increases by a factor of k, to at most $2n^2$. This happens at most $\lfloor\log_k U+1\rfloor$ times. Thus the total increase in Φ over the entire algorithm is at most $2n^2\lfloor\log_k U+2\rfloor$. The total number of nonsaturating pushes is at most k times the sum of the initial value of Φ and the total increase in Φ. This is at most $2kn^2\lfloor\log_k U+3\rfloor$. □

Choosing k to be a constant independent of n gives a total time bound of $O(nm+n^2\log U)$ for the scaling algorithm, given an efficient implementation of the vertex selection rule. One way to implement the rule is to maintain an array of sets indexed by vertex label, each set containing all large excess vertices with the corresponding label, and to maintain a pointer to the nonempty set of smallest index. The total time needed to maintain this structure is $O(nm+n^2\log U)$.

Having described the scaling algorithm, we consider the question of whether its running time can be improved by reducing the number of nonsaturating pushes. The proof of Lemma 3.1 bounds the number of nonsaturating pushes by estimating the total increase in the potential Φ. Observe that there is an imbalance in this estimate: $O(n^2\log_k U)$ of the increase is due to phase changes, whereas only $O(n^2)$ is due to relabelings. Our plan is to improve this estimate by decreasing the contribution of the phase changes, at the cost of increasing the contribution of the relabelings. Making this plan work requires changing the algorithm.

We use a nonconstant scale factor k and a slightly more elaborate method of vertex selection. We make use of the *stack-push/relabel* step defined below, which performs a sequence of push and relabel steps using a stack. The stack provides an alternative way of avoiding pushes to large-excess vertices (other than t).

```
Stack-Push/Relabel(r).
    Applicability: Vertex r is active.
    Action: Initialize a stack S to contain r. Repeat the following step until S is empty:
    Stack Step. Let v be the top vertex on S and let (v, w) be the current arc out of
    v. Apply the appropriate one of the following cases:
        Case 1: (v, w) is not eligible.
            Case 1a: (v, w) is not last on A(v). Replace (v, w) as the current arc out
                of v by the next arc on A(v).
            Case 1b: (v, w) is last on A(v). Relabel v and pop it from S. Replace
                (v, w) as the current arc out of v by the first arc on A(v).
        Case 2: (v, w) is eligible.
            Case 2a: $e(w)>\Delta/2$ and $w\neq t$. Push w onto S.
            Case 2b: $e(w)\leqq\Delta/2$ or $w=t$. Perform push(v, w) (modified as at the
                beginning of this section to maintain $e(w)\leqq\Delta$ if $w\neq t$). If $e(v)=0$,
                pop v from S.
```

Some remarks about *stack-push/relabel* are in order. Let us call a nonsaturating push *big* if it moves at least $\Delta/2$ units of flow and *little* otherwise. During an execution of *stack-push/relabel*, every vertex pushed onto S, except possibly the first, has an excess of at least $\Delta/2$ when it is added to S. A vertex v can be popped from S only after it is relabeled or its excess is reduced to zero. Of the pushes from v while v is on S, at most two are nonsaturating, only the last of which can be little.

Our variant of the scaling algorithm, called the *stack scaling algorithm*, consists of phases just as in the scaling algorithm. A phase consists of repeatedly applying the *stack-push/relabel* step to a large-excess active vertex of highest label; a phase ends when there are no large-excess active vertices.

LEMMA 3.2. *The total number of nonsaturating pushes made by the stack scaling algorithm is* $O(kn^2+n^2(\log_k U+1))$.

Proof. To bound the number of nonsaturating pushes, we use an argument like the proof of Lemma 3.1, but with two potentials instead of one. The first potential is that of Lemma 3.1, namely, $\Phi=\sum_{v \text{ active}} e(v)\, d(v)/\Delta$. By the analysis in the proof of Lemma 3.1, every push decreases Φ, the total increase in Φ over all phases is $O(n^2 \log_k U)$, and the difference between the initial and the final values of Φ is $O(n^2)$. Each big push moves at least $\Delta/2$ units of flow and hence decreases Φ by at least $\frac{1}{2}$. Thus the number of big pushes is $O(n^2(\log_k U+1))$.

To count little pushes, we divide them into two kinds, those that result in an empty stack S, called *emptying pushes*, and those that do not, called *nonemptying pushes*. A nonemptying push from a vertex v is such that $e(v)$ was at least $\Delta/2$ when v was added to S, and the push results in $e(v)$ decreasing to zero. We can charge such a push against the cumulative decrease of at least $\frac{1}{2}$ in Φ resulting from moving the original excess on v to vertices of smaller label. Hence there can be only $O(n^2(\log_k U+1))$ nonemptying pushes.

An emptying push from a vertex v can be associated with a decrease of at least $1/k$ in Φ, namely, the drop in Φ caused by the movement of the original excess on v, which is at least Δ/k, to smaller labeled vertices. But using this drop gives a bound on the number of emptying pushes of only $O(kn^2(\log_k U+1))$. We count emptying pushes more carefully by using a second potential, Φ_2. The definition of Φ_2 involves two parameters, an integer l and a set P. The value of l is equal to the minimum of $2n$ and the smallest label of a vertex added to S while S was empty during the current phase. Observe that $l=2n$ at the beginning of a phase and that l is nonincreasing during a phase. The set P contains all vertices that have label greater than l, and also all vertices that have label equal to l and from which an emptying push has been made during the current phase. Observe that P is empty at the beginning of a phase and P never loses a vertex during a phase. The definition of Φ_2 is

$$\Phi_2=\sum_{v\in P:e(v)>0} e(v)(d(v)-l+1)/\Delta. \text{ (If } P=\varnothing, \text{ then } \Phi_2=0.)$$

Observe that $0\leqq\Phi_2\leqq 2n^2$. Any emptying push can be associated either with an addition of a vertex to P or with a decrease in Φ_2 of at least $1/k$. (If the push is from v, either $v\notin P$ when v is added to S but $v\in P$ after the push, or $v\in P$ when v is added to S and Φ_2 drops by at least $1/k$ because of pushes from v while v is on S.) The number of vertices added to P is at most $n\lfloor\log_k U+1\rfloor$ over all phases, and hence so is the number of emptying pushes not associated with decreases in Φ_2.

To bound the number of emptying pushes associated with decreases in Φ_2, we bound the total increase in Φ_2. Increases in Φ_2 are due to relabelings and to decreases in l. (A vertex added to P because of an emptying push has zero excess and hence adds nothing to Φ_2.) A relabeling of a vertex v increases Φ_2 by at most the increase in $d(v)$ plus one; the "plus one" accounts for the fact that the relabeling may add v to P. Thus relabelings contribute at most $4n^2$ to the growth of Φ_2.

There are at most $2n$ decreases in l per phase. A decrease in l by one adds at most n/k to Φ_2, since when the decrease occurs every vertex in P has small excess. (Such a decrease occurs because some vertex v of label less than l is added to S while

S is empty. The label of v becomes the new value of l. When this happens, P contains only vertices of label exceeding the new value of l, all of which must have small excess.) Thus the total increase in Φ_2 due to decreases in l is at most $2n^2\lfloor \log_k U+1\rfloor/k$ over all phases.

The total number of emptying pushes associated with decreases in Φ_2 is at most k times the total increase in Φ_2, since Φ_2 is initially zero. Thus the total number of such pushes is at most $4kn^2+2n^2\lfloor \log_k U+1\rfloor$. □

As in the Goldberg-Tarjan and Ahuja-Orlin algorithms, the time to perform saturating pushes and relabeling operations and to examine ineligible arcs is $O(nm)$. The only significant remaining issue is how to choose vertices to add to S when S is empty. For this purpose, we maintain a data structure consisting of a collection of doubly linked lists: $list(j)=\{i\in N: e(i)>\Delta/k$ and $d(i)=j\}$ for each $j\in\{1, 2, \cdots, 2n-1\}$. We also maintain a pointer to indicate the largest index j for which $list(j)$ is nonempty. Maintaining this structure requires $O(1)$ time per push operation plus time to maintain the pointer. Each pointer increase is due to a relabeling (the increase is at most the amount of change in label) or due to a phase change (the increase is at most $2n$). Consequently, the number of times the pointer needs to be incremented or decremented is $O(n^2+n\,(\log_k U+1))$. The overall running time of the algorithm is thus $O(nm+kn^2+n^2(\log_k U+1))$. Choosing $k=\lceil \log U/\log\log U\rceil$ gives the following result:

THEOREM 3.3. *The stack scaling algorithm, with an appropriate choice of k, runs in* $O(nm+n^2\log U/\log\log U)$ *time.*

4. The wave scaling algorithm. Another way to reduce the number of nonsaturating pushes in the scaling algorithm is to keep track of the *total excess*, defined to be the sum of the excesses of all active vertices. The key observation is that if the total excess is sufficiently large, the algorithm can make significant progress by applying *stack-push/relabel* to each active vertex in turn, processing vertices in topological order with respect to the set of eligible arcs. Even though some vertices of very small excess are processed, the overall benefits of this approach yield an even better running time than that of the stack scaling algorithm. The idea of processing vertices in topological order originated in the *wave algorithm* of Tarjan [20], [21]; therefore, we call the new algorithm the *wave scaling algorithm.*

The wave scaling algorithm seems to derive no benefit from using a nonconstant scaling factor; therefore, we fix $k=2$. The algorithm uses another parameter, $l\geqq 1$, whose exact value we shall choose later. A phase of the wave scaling algorithm consists of two parts. First, the *wave step* below is repeated until the total excess is less than $n\Delta/l$. Then *stack-push/relabel* steps are applied to large-excess active vertices in any order until there are no large-excess active vertices.

Wave: Construct a list L of the vertices in $V-\{s, t\}$ in topological order with respect to the set of eligible arcs. (Ordering L in nonincreasing order by vertex label suffices.) For each vertex v on L, if v is active and v has not been relabeled during the current wave, then perform *stack-push/relabel*(v).

Observe that the total excess is a nonincreasing function of time. Constructing L at the beginning of a wave takes $O(n)$ time using a radix sort by vertex label. The time spent during a single wave is $O(n)$ plus time for the relabelings and pushes. Thus the total time required by the wave scaling algorithm is $O(nm)$ plus $O(n)$ per wave plus $O(1)$ per nonsaturating push. We complete the running time analysis with two lemmas.

LEMMA 4.1. *The number of nonsaturating pushes done by the wave scaling algorithm is* $O(n^2+(n^2/l)\log U)$ *plus* $O(n)$ *per wave.*

Proof. We count big pushes (those that move at least $\Delta/2$ units of flow) and little pushes separately. The analysis in the proof of Lemma 3.1 applies to bound the number of big pushes, with the following improvement. At the end of a phase, the total excess is less than $n\Delta/l$. Thus the potential at the beginning of the next phase is at most $4n^2/l$. (The new value of Δ is half of the old; each vertex label is at most $2n$.) Thus the sum of the increases in Φ caused by changes in Δ is $O((n^2/l)\log U)$, which implies that the number of big pushes is $O(n^2+(n^2/l)\log U)$. The same argument gives the same bound on the number of little pushes from vertices that have large excess when added to the stack S, since as in the proof of Lemma 3.2 each such push can be charged against a drop of at least $\frac{1}{2}$ in Φ. The remaining little pushes consist of at most one per vertex to which *stack-push/relabel* is applied during waves, totaling at most n per wave. □

LEMMA 4.2. *The number of waves in the wave scaling algorithm is* $O(\min\{nl+\log U, n^2\})$.

Proof. Consider any wave except the last in a phase. At the end of the wave, the total excess is at least $n\Delta/l$. The only way for excess to remain on a vertex x at the end of a wave is for x to have been relabeled during the wave; once x is relabeled, $e(x)$ remains constant until the end of the wave. The maximum excess on any single vertex is Δ. It follows that at least n/l relabelings must have occurred during the wave. Since the total number of relabelings is $O(n^2)$, the total number of waves is $O(nl+\log U)$. Furthermore, a wave during which no relabeling occurs causes all vertices to become inactive, and hence terminates the entire algorithm. Thus the number of waves is $O(n^2)$. □

THEOREM 4.3. *The running time of the wave scaling algorithm is* $O(nm+n^2(\log U)^{1/2})$ *if l is chosen equal to* $(\log U)^{1/2}$.

Proof. By Lemmas 4.1 and 4.2, the running time of the wave scaling algorithm is $O(nm+(n^2/l)\log U+\min\{n^2l+n\log U, n^3\})$. Choosing $l=(\log U)^{1/2}$ gives a bound of $O(nm+n^2(\log U)^{1/2}+\min\{n\log U, n^3\})$. But $\min\{n\log U, n^3\}\leqq n^2(\log U)^{1/2}$, since $n\log U\geqq n^2(\log U)^{1/2}$ implies $(\log U)^{1/2}\geqq n$. □

5. Use of dynamic trees. The approach taken in §§ 3 and 4 reduced the total number of pushes. An orthogonal approach is to reduce the total time of the pushes without necessarily reducing their number. This can be done by using the dynamic tree data structure of Sleator and Tarjan [18], [19], [20]. We conjecture that, given a version of the preflow algorithm with a bound of $p\geqq nm$ on the total number of pushes, the running time can be reduced from $O(p)$ to $O(nm\log((p/nm)+1))$ by using dynamic trees. Although we do not know how to prove a general theorem to this effect, we have been able to obtain such a result for each version of the preflow algorithm that we have considered. As an example, the $O(nm\log(n^2/m))$ time bound of Goldberg and Tarjan results from using dynamic trees with the first-in, first-out selection rule; the bound on the number of pushes in this case is $O(n^3)$. In this section we shall show that the same idea applies to the wave scaling algorithm of § 4, resulting in a time bound of $O(nm\log((n/m)(\log U)^{1/2}+2))$. This idea can also be applied to the stack scaling algorithm of § 3, giving a bound of $O(nm\log((n\log U/m\log\log U)+2))$; we omit a description of the latter result since the former bound is better.

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each arc of which has an associated real value. We regard each tree arc as being directed from child to parent, and we regard every vertex as being both an ancestor and a descendant of itself. The data structure supports the following seven operations:

find-root(v):	Find and return the root of the tree containing vertex v.
find-size(v):	Find and return the number of vertices in the tree containing vertex v.
find-value(v):	Find and return the value of the tree arc leaving v. If v is a tree root, the value returned is infinity.
find-min(v):	Find and return the ancestor w of v with minimum *find-value*(w). In case of a tie, choose the vertex w closest to the tree root.
change-value(v, x):	Add real number x to the value of every arc along the path from v to the root of its tree.
link(v, w, x):	Combine the trees containing v and w by making w the parent of v and giving the arc (v, w) the value x. This operation does nothing if v and w are in the same tree or if v is not a tree root.
cut(v):	Break the tree containing v into two trees by deleting the arc joining v to its parent; return the value of the deleted arc. This operation breaks no arc and returns infinity if v is a tree root.

A sequence of h tree operations, starting with an initial collection of singleton trees, takes $O(h \log (z+1))$ time if z is the maximum tree size [10], [18], [19], [20].

In the network flow application, the dynamic tree arcs are a subset of the current arcs out of the vertices. Only eligible arcs are tree arcs. The value of a tree arc is its residual capacity. The dynamic tree data structure allows flow to be moved along an entire path at once, rather than along a single arc at a time. We shall describe a version of the wave scaling algorithm, which we call the *tree scaling algorithm*, that uses this capability. Two parameters govern the behavior of the algorithm, a variable bound Δ on the maximum excess at an active vertex and a fixed bound z, $1 \leqq z \leqq n$, on the maximum size of a dynamic tree. The algorithm is identical to the wave scaling algorithm except that it uses the following step in place of *stack-push/relabel*:

Tree-Push/Relabel(r)

Applicability: Vertex r is active.

Action: Initialize a stack S to contain r. Repeat the following step until S is empty.

Stack Step. Let v be the top vertex on S and let (v, w) be the current arc out of S. Apply the appropriate one of the following cases:

Case 1: *(v, w) is not eligible.*

Case 1a: *(v, w) is not last on $A(v)$.* Replace (v, w) as the current arc out of v by the next arc on $A(v)$.

Case 1b: *(v, w) is last on $A(v)$.* Relabel v and pop it from S. Replace (v, w) as the current arc out of v by the first arc on $A(v)$. For every tree arc (y, v), perform *cut*(y).

Case 2: *(v, w) is eligible.* Let $x =$ *find-root*(w).

Case 2a: *$e(x) > \Delta/2$ and $x \neq t$.* Push x onto S.

Case 2b: *$e(x) \leqq \Delta/2$ or $x = t$.* Let $\varepsilon = \min\{e(v), u_f(v, w), \textit{find-min}(w)\}$. Let $\delta = \varepsilon$ if $x = t$, $\delta = \min\{\varepsilon, \Delta - e(x)\}$ if $x \neq t$. Send δ units of flow from v to x by increasing $f(v, w)$ by δ and performing *change-value*$(w, -\delta)$. (This is called a *tree push* from v to x. The tree push is *saturating* if $\delta = \min\{u_f(v, w), \textit{find-min}(w)\}$ before the push and *nonsaturating* otherwise.) While it is the case that *find-value*(*find-

min$(w)) = 0$, perform *cut*(*find-min*(w)). If $e(v) = 0$, pop v from S, and if in addition $u_f(v, w) > 0$ and *find-size*(w) + *find-size*$(v) \leqq z$, perform *link*$(v, w, u_f(v, w))$.

The tree scaling algorithm stores flow in two different ways; explicitly for arcs that are not dynamic tree arcs and implicitly (in the dynamic tree data structure) for arcs that are dynamic tree arcs. After each cut, the flow on the arc cut must be restored to its correct current value. In addition, when the algorithm terminates, the correct flow value on each remaining tree arc must be computed. For arcs cut during the computation, the desired flow values are returned by the corresponding *cut* operations. Computing correct flow values on termination can be done using at most $n-1$ *find-value* operations. We have omitted these bookkeeping steps from our description of the algorithm.

The algorithm maintains the following invariants: every active vertex is a tree root, every tree arc is eligible, no excess exceeds Δ, and no tree size exceeds z. Let us call a nonsaturating tree push *big* if it moves at least $\Delta/2$ units of flow, and *little* otherwise. Of the pushes from a given vertex v while v is on S, at most two are nonsaturating and at most one (the last) is small.

The tree scaling algorithm is a variant of the dynamic tree algorithm of Goldberg and Tarjan [9], [10]. Their analysis applies to give the following result:

LEMMA 5.1 [10]. *The total time required by the tree scaling algorithm is* $O(nm \log(z+1))$ *plus* $O(\log(z+1))$ *time per tree push plus the time needed to construct and scan the list L during wave steps. The number of links, cuts, and saturating tree pushes is* $O(nm)$.

Making the tree scaling algorithm efficient requires careful implementation of the list L used in *wave* steps. For the moment we shall ignore the time spent manipulating L. The remaining issue in analyzing the algorithm is bounding the number of nonsaturating pushes. To do this we need two lemmas:

LEMMA 5.2. *The number of waves in the tree scaling algorithm is* $O(\min\{nl + \log U, n^2\})$.

Proof. Identical to the proof of the same result for the wave scaling algorithm (Lemma 4.2). □

LEMMA 5.3. *The number of nonsaturating tree pushes done by the tree scaling algorithm is* $O(nm + (n^2/l) \log U)$ *plus* $O(n/z)$ *per wave.*

Proof. The potential function Φ used in § 3, as applied in the proof of Lemma 3.3, serves to bound the number of big tree pushes by $O(n^2 + (n^2/l) \log U)$. We count little tree pushes as follows. Any little tree push, say from a vertex v, reduces $e(v)$ to zero. We shall count such a push against the most recent previous push that made $e(v)$ nonzero; or, if there is no such previous push, against the preflow initialization. We call the event of $e(v)$ becoming nonzero an *activation* of v. We shall derive a bound of $O(nm + (n^2/l) \log U)$ plus $O(n/z)$ per wave on the number of activations, thereby proving the lemma.

Initializing the preflow f at the beginning of the entire algorithm causes $O(n)$ vertex activations. Every other activation is due to a tree push (Case 2b of *tree-push/relabel*). If such a tree push results in a link or cut, we charge the activation against the corresponding link or cut. Similarly, if the tree push is saturating, we charge the activation against the arc saturation. Such charges account for $O(nm)$ activations.

Any remaining activation, say of a vertex x, is produced by a tree push through an arc (v, w) to $x =$ *find-root*(w), after which *find-size*(v) + *find-size*$(w) > z$, since no link is performed. Let T_v and T_w be the dynamic trees containing v and w, respectively. We call a dynamic tree *large* if it contains more than $z/2$ vertices and *small* otherwise. Just after the push, one of T_v and T_w must be large.

If the tree push is big, we charge the activation of x against the push. By the argument above counting big tree pushes, this accounts for $O(n^2+(n^2/l)\log U)$ activations. Otherwise, the push is little, and thus it reduces $e(v)$ to zero. There can be at most one such push from v per wave. If T_v has changed between the beginning of the most recent wave and the push, we charge the activation of x to the link or cut that most recently changed T_v. There are $O(nm)$ such charges, at most one per link and at most two per cut, since vertex v is the root of T_v when the push occurs. Similarly, if T_w has changed between the beginning of the most recent wave and the push, we charge the activation of x to the link or cut that most recently changed T_w. There are $O(nm)$ such charges, since a vertex x can be activated at most once per wave, and x is the root of T_w when the push occurs. If neither T_v nor T_w has changed, we charge the activation of x to whichever of T_v or T_w is large. Each large tree existing at the beginning of a wave can be charged at most twice (once as a T_v, once as a T_w). Since there are fewer than $2n/z$ large trees at the beginning of any wave, the total number of such charges is $O(n/z)$ per wave. Combining all our estimates gives the claimed bound on vertex activations. □

THEOREM 5.4. *The number of tree pushes done by the tree scaling algorithm is* $O(nm)$ *if* z *is chosen equal to* $\min\{n, \max\{1, (n^2/m^2)\log U\}\}$, *and* l *is chosen equal to* m/n *if* $z=1$, $(n/m)\log U$ *if* $1<z\leqq n$.

Proof. By Lemmas 5.2 and 5.3, the number of tree pushes done by the algorithm is $O(nm+(n^2/l)\log U+(n/z)\min\{nl+\log U, n^2\})$. We consider three cases:

Case 1. $(n^2/m^2)\log U\leqq 1$. Then $z=1$ and $l=m/n\geqq 1$. The number of tree pushes is $O(nm+(n^3/m)\log U+n\min\{m+\log U, n^2\})$. Since $\log U\leqq m^2/n^2$, the bound on tree pushes is $O(nm+\min\{m^2/n, n^3\})=O(nm)$, because $m^2/n\leqq n^3$ implies $m\leqq n^2$, which means $m^2/n\leqq nm$, and $m^2/n\geqq n^3$ implies $m\geqq n^2$, which means $n^3\leqq nm$. (This analysis allows for the possiblity of multiple arcs in G, i.e., the possibility that $m>n^2$.)

Case 2. $1<(n^2/m^2)\log U<n$. Then $z=(n^2/m^2)\log U$ and $l=(n/m)\log U\geqq 1$. The number of tree pushes is $O(nm+(m^2/n\log U)\min\{(n^2/m)\log U+\log U, n^2\})=O(nm+\min\{m^2/n, m^2n/\log U\})=O(nm+\min\{m^2/n, n^3\})$, since $\log U>m^2/n^2$. Since $\min\{m^2/n, n^3\}\leqq nm$ (see Case 1), the number of tree pushes is $O(nm)$. (As in Case 1, this analysis allows for the possiblity that $m>n^2$.)

Case 3: $(n^2/m^2)\log U\geqq n$. Then $z=n$ and $l=(n/m)\log U\geqq 1$. The number of tree pushes is $O(nm+\min\{(n^2/m)\log U+\log U, n^2\})=O(nm)$. □

Observe that if z is chosen as in Theorem 5.4, i.e., equal to $\min\{n, \max\{1, (n^2/m^2)\log U\}\}$, the time per dynamic tree operation is $O(\log(z+1))=O(\log((n/m)(\log U)^{1/2}+2))$. Thus the running time of the tree scaling algorithm, with k and z chosen as in Theorem 5.4, is $O(nm\log((n/m)(\log U)^{1/2}+2))$ (ignoring time spent manipulating L).

All that remains is to show that the manipulations of the list L in *wave* steps can be performed in $O(nm\log((n/m)(\log U)^{1/2}+2))$ time. It is not sufficient to represent L simply as a linked list, for then the time spent scanning L will be $O(n)$ per wave, and this scanning time will dominate the time for the rest of the computation. Instead, we represent L as follows. For each integer i in the range $1\leqq i\leqq 2n-1$, we maintain the set $K(i)$ of active vertices with label i that have not been relabeled during the current wave. For each integer j in the range $1\leqq j\leqq\lceil(2n-1)/z\rceil$, we maintain a heap (priority queue) $H(j)$ of the integers i with $K(i)\neq\varnothing$ and $\lceil i/z\rceil=j$. We also maintain a current index j^*, initially equal to $\lceil(2n-1)/z\rceil$.

During a wave, we find the next vertex v to which to apply a *tree-push/relabel* step by decrementing j^* until $H(j^*) \neq \emptyset$, then finding the maximum integer $i \in H(j^*)$, and finally selecting v to be any vertex in $K(i)$. During each *tree push/relabel* step, the sets $K(i)$ and heaps $H(j)$ are updated as vertices become active or inactive and labels change. Each insertion or deletion in a set $K(i)$ takes $O(1)$ time. The heaps $H(j)$ are implemented using any standard heap implementation (see [20]), so that insertion or deletion in a heap $H(j)$ takes $O(\log(z+1))$ time, as does finding the maximum integer in a given heap $H(j)$. (The maximum number of elements in any heap $H(j)$ is z.) A set $K(i)$ or a heap $H(j)$ can change only as the result of a tree push or a relabeling, and the time required for the associated updating of sets and heaps is $O(\log(z+1))$. Thus we obtain the following result:

LEMMA 5.5. *The total time spent maintaining the sets $K(i)$, the heaps $H(j)$, and the index j^* during the tree scaling algorithm is $O(nm \log(z+1))$, if z and l are chosen as in Theorem* 5.4.

Proof. The number of relabelings is $O(nm)$, as is the number of tree pushes by Theorem 5.4. There is $O(n/z)$ time per wave spent decrementing j^*, but this amount of time is included in the $O(nm)$ bound of Theorem 5.4. □

THEOREM 5.6. *The tree scaling algorithm runs in $O(nm \log((n/m)(\log U)^{1/2}+2))$ time if z is chosen equal to* $\min\{n, \max\{1, (n^2/m^2)\log U\}$ *and l is chosen equal to m/n if $z=1$ or to $(n/m)\log U$ if $z>1$.*

Proof. Immediate from Theorem 5.4 and Lemma 5.5, since $\log(z+1) = O(\log((n/m)(\log U)^{1/2}+2))$. □

6. Bounds in a semilogarithmic computation model. The time bounds derived in §§ 3-5 are all based on the assumption that addition and comparison of integers of magnitude at most U takes $O(1)$ time. If U is huge, this assumption may be unjustified. In this section we explore the consequences to our results of using a semilogarithmic computation model in which each computer word is allowed to hold $O(\log n)$ bits, and any operation involving $O(1)$ words takes $O(1)$ time. In this model, all the elementary graph and list manipulation operations needed by our algorithms take $O(1)$ time each, but adding or comparing two capacity or flow values can take $O(\log_n U)$ time if an exact answer is required. Thus a straightforward translation of our results into the semilogarithmic model increases each of the time bounds by a factor of $O(\log_n U)$.

By suitably modifying the algorithms, however, it is possible to reduce the extra time each of our algorithms requires in the semilogarithmic model to an additive term of $O(m \log_n U)$. Thus, for example, the running time of the tree scaling algorithm becomes $O(nm \log((n/m)(\log U)^{1/2}+2) + m \log_n U)$. Note that the time needed to read all the arc capacities is $\Theta(m \log_n U)$ in this model. Thus, as U grows exponentially large, the total running time becomes linear in the size of the input.

The general approach is to approximately solve a sequence of $O(\log_n U)$ closer-and-closer approximations to the original problem. Each approximation uses as a starting point the approximate solution to the previous problem. Solving each problem requires manipulation of integers of only $O(\log n)$ bits.

Making this approach work involves a number of messy technical details. Since the result is mainly of theoretical interest, we shall merely sketch the ideas involved in modifying the algorithms of §§ 4 and 5. We assume (without loss of generality) that n is a power of two and that U is a power of n. For $k = 1, 2, \cdots, \log_n U$, we define *problem k* to be the maximum flow problem on the graph G with arc capacities $u_k(v, w)$ defined by $u_k(v, w) = \lceil u(v, w)/\delta_k \rceil \delta_k$, where $\delta_k = U/n^k$. Observe that for $k = \log_n U$,

$\delta_k = 1$; thus, problem $\log_n U$ is the original problem. Also, for a general value of k, all capacities in problem k are divisible by δ_k; and, for every arc (v, w), $0 \leqq u_{k-1}(v, w) - u_k(v, w) \leqq \delta_{k-1}$.

We solve problem k for $k = 1, 2, \cdots, \log_n U - 1$ approximately and then solve problem $\log_n U$ exactly. Throughout the computation we maintain a preflow f and a valid labeling d for the current problem. Preflow f satisfies the following constraint:

$$(5) \qquad e(v) \geqq \min \left\{ \sum_{f(u,v)>0} f(u, v), n(\delta_k - 1) \right\} \forall v \neq s \text{ (flow holdback constraint).}$$

(In inequality (5), the sum over $f(u, v)$ is taken to be zero if $f(u, v) \leqq 0$ for all arcs (u, v).)

For a vertex v, the *available excess* at v, which is the amount actually allowed to be pushed from v, is $a(v) = \max \{0, e(v) - n(\delta_k - 1)\}$. Throughout the computation, the available excesses are used in place of the actual excesses to determine when vertices are active and how much flow can be pushed.

The flow holdbacks are needed to maintain the preflow property when converting a solution for one problem into a good initital preflow for the next problem. For the first problem, the preflow f and valid labeling d are initialized exactly as in § 2 (using the arc capacities $u_1(v, w)$). Once a solution to problem $k-1$ is computed, it is converted into an initial preflow for problem k by replacing the current preflow f by the preflow f' defined by $f'(v, w) = \min \{f(v, w), u_k(v, w)\}$. Since $0 \leqq u_{k-1}(v, w) - u_k(v, w) \leqq \delta_{k-1}$, f' is obtained from f by decreasing the flow on each arc (v, w) by an amount between zero and $\min \{f(v, w), \delta_{k-1}\}$ (inclusive). Since $\delta_{k-1} = n\delta_k$, the validity of the flow holdback constraint for f in problem $k-1$ implies the validity of the flow holdback constraint for f' in problem k. The flow holdback constraint implies that f' is a preflow, since f' satisfies the capacity constraint by construction. Since every arc saturated by f in problem $k-1$ remains saturated by f' in problem k, d is a valid labeling for f' in problem k.

For $1 \leqq k < \log_n U$, a preflow f and valid labeling d constitute a solution to problem k if $a(v) \leqq 2n(\delta_k - 1)$ for every active vertex v. Thus a solution to problem $\log_n U$ gives a maximum flow in the original network. After the initialization of the preflow for problem k, every active vertex has an excess of at most $4n^2\delta_k$, of which $3n^2\delta_k = 3n\delta_{k-1}$ comes from the excess on v in problem $k-1$ (since $a(v) \leqq 2n(\delta_{k-1} - 1)$), and $n^2\delta_k = n\delta_{k-1}$ comes from the changes made to f in initializing the preflow for problem k. (These are overestimates.) The initial value of Δ (the bound on maximum available excess) for problem k is $4n^2\delta_k$. Problem k is solved by performing a number of phases, continuing until the maximum available excess is at most $2n$ $(\delta_k - 1)$, which happens by the time Δ is reduced to $n\delta_k$ (or to $\frac{1}{2}$ if $k = \log_n U$). If $k < \log_n U$, the number of phases needed is at most $\log n + 2$; if $k = \log_n U$, the number of phases needed is at most $2 \log n + 3$. Thus each problem requires $O(\log n)$ phases, and the total number of phases over all subproblems is $O(\log n \log_n U) = O(\log U)$.

During the solution of problem k, the preflow f is maintained so that $f(v, w)$ is a multiple of δ_k for each arc (v, w). (This happens automatically, since Δ, the initial flow values, and all capacities are multiples of δ_k.) Thus the flow values and capacities can be represented in units of δ_k. Furthermore, the flow values and capacities are not represented explicitly, but rather as differences from the initial flow values. To be more precise: let f_{k-1} be the final preflow for problem $k-1$; let $f_0 = 0$. Then the current preflow f is represented by the value $f(v, w) - f_{k-1}(v, w)$ for each arc (v, w). The capacity function u_k is represented by the value $\min \{u_k(v, w) - f_{k-1}(v, w), 9n^5\delta_k\}$ for each arc (v, w). We claim that the amount by which the flow on an arc can change during the solution of problem k and subsequent problems is at most $8n^5\delta_k$. This claim implies that if $u_k(v, w)-f_k(v, w)>9n^5\delta_k$, the extra residual capacity (beyond $9n^5\delta_k$) on arc (v, w) will never be used and can be ignored. To prove the claim, we note that between relabelings at most $n\Delta \leqq 4n^3\delta_k$ units of flow can be moved through any given arc, and there are at most $2n^2$ relabelings over the entire algorithm. This accounts for $8n^5\delta_k$ units of change. The change on an arc due to modifying f between subproblems is at most $\sum_{j=k-1}^{\log_n U} \delta_j \leqq 2\delta_{k-1}=2n\delta_k$. Thus the claim is true.

Storing f and u_k in such a difference form, in units of δ_k, allows the algorithm to manipulate numbers consisting of only $O(\log n)$ bits, and all necessary additions and comparisons of flows and capacities take $O(1)$ time. Constructing the final preflow is merely a matter of adding together the successive flow differences $f_1-f_0, f_2-f_1, \cdots$ computed when solving successive problems. By adding these differences from right to left, the time per addition can be made $O(m)$, and the total time is $O(m \log_n U)$. This is also the total time required for initializing the preflow for each problem ($O(m)$ time per problem).

If the wave scaling algorithm is used, the bounds in § 4 remain valid almost without modification and apply to the calculations involved in solving all $\log_n U$ problems. Specifically, the number of relabelings is $O(n^2)$, the number of saturating pushes is $O(nm)$, the number of nonsaturating pushes is $O(n^2+(n^2/l)\log U)$ plus $O(n)$ per wave, and the number of waves is $O(\min\{nl+\log U, n^2+\log_n U\})$. The bound on waves is the only one that changes, in that n^2 becomes $n^2+\log_n U$; this occurs because a wave without relabelings terminates only the solution of the current problem and not the entire algorithm. Choosing $l=(\log U)^{1/2}$ gives an overall bound of $O(nm+n^2(\log U)^{1/2}+m \log_n U)$ time.

Use of the tree scaling algorithm of § 5 results in a time bound of $O(nm \log((n/m)(\log U)^{1/2}+2)+m \log_n U)$. To obtain this bound, we must verify that the extra time required for initializing the sets $K(i)$, the heaps $H(j)$, and the dynamic trees for each problem, and the extra time for tree pushes associated with vertices that become active at the beginning of a new problem, is $O(m)$ per problem.

We need two extra facts about the dynamic tree data structure. By traversing the entire data structure, the flow values for all the dynamic tree arcs can be computed in $O(n)$ time. Furthermore, a new set of flow values for all these arcs can be installed in the data structure in $O(n)$ time. These facts can easily be verified by checking the description of the data structure [19], [20].

When the flow f is modified at the beginning of a new problem, new vertices with available excess are created, possibly as many as $n-2$. Handling these newly active vertices is the hardest part of the initialization of the new problem. To begin a new problem, the current flow on all dynamic tree arcs is computed explicitly. (This takes $O(n)$ time as noted above.) Then the flow and capacities are modified to their initial values for the new problem. The bound on maximum available excess is set equal to $4n^3\delta_k$. (This is n times the value proposed earlier in this section and is greater than the total excess on all active nodes.)

Next, the vertices of G are scanned in topological order with respect to the set of eligible arcs. A vertex v is scanned by applying to it a *push/relabel* step of the kind defined in § 2. Such a step either relabels v or reduces its available excess to zero. After all vertices have been scanned, all the available excess is on vertices that have been relabeled during the scanning. The current flow values for dynamic tree arcs are reinstated in the dynamic tree data structure, and every dynamic tree arc that has been saturated during the scanning or that has had one of its end vertices relabeled is cut. Finally, the sets $K(i)$ and heaps $H(j)$ are reinitialized, which takes $O(n)$ time.

(A heap can be initialized in linear time [20].) Then the tree scaling algorithm is begun.

Because the starting value of Δ for each problem is increased, the number of phases increases, but only by $\log n$ per subproblem, or $\log U$ overall. The time to carry out all the scanning in the initialization is $O(m)$. The time to initialize the dynamic tree data structure is $O(m)$ plus $O(\log(z+1))$ per cut; the time for a cut can be charged against the corresponding arc saturation or vertex relabeling. The time to initialize the sets $K(i)$ and heaps $K(j)$ is $O(n)$. At the beginning of a subproblem, the only active vertices are those that have been relabeled during the initialization, and the timing analysis for the tree scaling algorithm is virtually the same as the analysis in § 5. Thus we obtain a total time bound of $O(nm \log((n/m)(\log U)^{1/2}+2)+m \log_n U)$.

The techniques discussed above can also be applied to the maximum flow algorithm of Goldberg and Tarjan [10], resulting in a time bound in the semilogarithmic model of $O(nm \log(n^2/m)+m \log_n U)$. The details are straightforward.

7. Remarks.

The algorithms of §§ 3-4 not only have good theoretical time bounds, but they may be very efficient in practice. We hope to conduct experiments to determine whether either of these algorithms is competitive with previously known methods. The tree scaling algorithm of § 5 is perhaps mainly of theoretical interest, although for huge networks there is some chance that the use of dynamic trees may be practical.

The two obvious open theoretical questions are (1) whether further improvement in the time bound for the maximum flow problem is possible and (2) whether our ideas extend to the minimum cost flow problem. It is not unreasonable to hope for a bound of $O(nm)$ for the maximum flow problem; note that the bounds of Theorems 4.3 and 5.6 are $O(nm)$ for graphs that are not too sparse and whose arc capacities are not too large, i.e., $(\log U)^{1/2}=O(m/n)$. Obtaining a bound better than $O(nm)$ would seem to require major new ideas.

As a partial answer to the second question, we have been able to obtain a time bound of $O(nm \log\log U \log(nC))$ for the minimum cost flow problem, where C is the maximum arc cost, assuming all arc costs are integral [2]. This result uses some ideas in the present paper and some additional ones [11].

As a final remark, we note that in certain cases the bounds of our algorithms can be improved by substituting a smaller value for U. To be precise, suppose that f is a flow of value U_1 obtained by some fast heuristic and that U_2 is an upper bound on the maximum flow value, also obtained by some fast heuristic. Then all of our bounds can be improved by substituting $U^*=\max\{4, \min\{U, (U_2-U_1)/n\}\}$ for U.

To see this, we first construct the residual network for the flow f. This network has the same graph as the original network, with the capacity of each arc (v, w) equal to $u_f(v, w)$. A flow of value x in the residual network corresponds to a flow of value U_1+x in the original network and vice-versa. It thus suffices to find a maximum flow in the residual network. For this network, U_2-U_1 is an upper bound on the maximum flow value.

We now add to the network a new source s_0, new vertices $v_1, \cdots, v_n$, and arcs (s_0, v_i), (v_i, s), (v_i, s_0), and (s, v_i) for $1 \leqq i \leqq n$. We define $u(s_0, v_i)=u(v_i, s)=\lceil U^* \rceil$, $u(v_i, s_0)=u(s, v_i)=0$ for $1 \leqq i \leqq n$. This augmented network has the same maximum flow value as the residual network. Then we run one of our excess-scaling algorithms on the augmented network, choosing as the original excess bound Δ the smallest possible allowed value exceeding U^*.

As an application of this result, we can take $U_1=0$ and $U_2=\sum_{v \in V} u(s, v)$, giving a value for U^* of $\max\{4, \sum_{v \in V} u(s, v)/n\}$.

Acknowledgments. We thank Andrew Goldberg for inspiring ideas and stimulating conversations.

REFERENCES

[1] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Oper. Res., to appear.

[2] R. K. AHUJA, A. V. GOLDBERG, J. B. ORLIN, AND R. E. TARJAN, *Finding minimum-cost flows by double scaling*, Math. Prog., submitted.

[3] J. CHERIYAN AND S. N. MAHESHWARI, *Analysis of preflow push algorithms for maximum network flow*, Dept. of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, 1987.

[4] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[5] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[6] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[7] H. N. GABOW, *Scaling algorithms for network problems*, J. Comput. System Sci., 31 (1985), pp. 148–168.

[8] A. V. GOLDBERG, *A New Max-Flow Algorithm*, Tech. Report MIT/LCS/TM-291, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, MA, 1985.

[9] ———, *Efficient Graph Algorithms for Sequential and Parallel Computers*, Ph.D. thesis, Mass. Inst. of Technology, Cambridge, MA, 1987.

[10] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, in Proc. 18th ACM Symp. on Theory of Computing, 1986, pp. 136–146; J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[11] ———, *Finding minimum-cost circulations by successive approximation*, Math. Oper. Res., to appear.

[12] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[13] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, NY, 1976.

[14] V. M. MALHOTRA, M. PRAMODH KUMAR, AND S. N. MAHESHWARI, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Inform. Process. Lett., 7 (1978), pp. 277–278.

[15] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[16] Y. SHILOACH AND U. VISHKIN, *An $O(n^2 \log n)$ parallel max-flow algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[17] D. D. SLEATOR, *An $O(nm \log n)$ algorithm for maximum network flow*, Tech. Report STAN-CS-80-831, Computer Science Dept., Stanford University, Stanford, CA, 1980.

[18] D. D. SLEATOR AND R. E. TARJAN, A data structure for dynamic trees, J. Comput. System Sci., 26 (1983), pp. 362–391.

[19] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[20] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[21] ———, *A simple version of Karzanov's blocking flow algorithm*, Oper. Res. Lett., 2 (1984), pp. 265–268.