A NEW APPROACH TO THE MAXIMUM FLOW PROBLEM

Andrew V. Goldberg

and

Robert E. Tarjan

CS-TR-050-86

April 1986

A New Approach to the Maximum Flow Problem*

*Andrew V. Goldberg***

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Robert E. Tarjan

Department of Computer Science
Princeton University
Princeton, NJ 08544
and

April, 1986

ABSTRACT

All previously known efficient maximum flow algorithms work by finding augmenting paths, either one path at a time (as in the original Ford and Fulkerson algorithm) or all shortest length augmenting paths at once (using the layered network approach of Dinic). We introduce an alternative method based on the *preflow* concept of Karzanov. A preflow is like a flow except that the total amount flowing into a vertex is allowed to exceed the total amount flowing out. The method maintains a preflow in the original network and pushes local flow excess toward the sink along what are estimated to be shortest paths. The algorithm and its analysis are simple and intuitive, yet the algorithm runs as fast as any other known method on dense graphs, achieving an $O(n^3)$ time bound on an n-vertex graph. By incorporating the dynamic tree data structure of Sleator and Tarjan, we obtain a version of the algorithm running in $O(nm \log(n^2/m))$ time on an n-vertex, m-edge graph. This is as fast as any known method for any graph density and faster on graphs of moderate density. The algorithm also admits efficient distributed and parallel implementations. We

obtain a parallel implementation running in $O(n^2 \log n)$ time and using only $O(m)$ space. This time bound matches that of the Shiloach-Vishkin algorithm, which requires $O(n^2)$ space.

June 23, 1986

* A preliminary version of this paper appeared in the proceedings of the 18th Annual ACM Symposium on Theory of Computing, Berkeley, California, May 28-30, 1986.
** Supported by a Fannie and John Hertz Foundation Fellowship. Part of this work was done while this author was at GTE Laboratories, Inc.

A New Approach to the Maximum Flow Problem*

*Andrew V. Goldberg***

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Robert E. Tarjan

Department of Computer Science
Princeton University
Princeton, NJ 08544
and

April, 1986

1. Introduction

The problem of finding a maximum flow in a directed graph with edge capacities arises in many settings in operations research and other fields, and efficient algorithms for the problem have received a great deal of attention. Extensive discussion of the problem and its applications can be found in the books of Ford and Fulkerson [7], Lawler [15], Even [5], Papadimitriou and Stieglitz [17], and Tarjan [24]. Table 1 shows a history of algorithms for the problem. Time bounds are stated in terms of n, the number of vertices, and m, the number of edges in the problem network, and in one case N, an upper bound on the edge capacities (assumed in this case to be integers).

#	Date	Discoverer	Running Time	References
1	1956	Ford and Fulkerson	-	[6,7]
2	1969	Edmonds and Karp	$O(nm^2)$	[4]
3	1970	Dinic	$O(n^2m)$	[3]

* A preliminary version of this paper appeared in the proceedings of the 18th Annual ACM Symposium on Theory of Computing, Berkeley, California, May 28-30, 1986.

** Supported by a Fannie and John Hertz Foundation Fellowship. Part of this work was done while this author was at GTE Laboratories, Inc.

4	1974	Karzanov	$O(n^3)$	[14]
5	1977	Cherkasky	$O(n^2 m^{1/2})$	[2]
6	1978	Malhotra, Pramodh Kumar, and Makeshwari	$O(n^3)$	[16]
7	1978	Galil	$O(n^{5/3} m^{2/3})$	[10]
8	1978	Galil and Naamad; Shiloach	$O(nm(\log n)^2)$	[11] [19]
9	1980	Sleator and Tarjan	$O(nm \log n)$	[21,22]
10	1982	Shiloach and Vishkin	$O(n^3)$	[20]
11	1983	Gabow	$O(nm \log N)$	[9]
12	1984	Tarjan	$O(n^3)$	[25]

Table 1. A history of maximum flow algorithms.

All algorithms in the table work either by finding augmenting paths one by one (algorithms 1 and 2), or by finding all shortest augmenting paths in one phase, as proposed by Dinic [3] (algorithms 3-10 and 12), or by repeated use of Dinic's method (algorithm 11). There is no clear winner among the algorithms in the table; algorithms 4, 6, 10 and 12 are designed to be fast on dense graphs, and algorithms 5, 7, 8, 9, and 11 are designed to be fast on sparse graphs. For dense graphs, the best known bound of $O(n^3)$ was first obtained by Karzanov [14]; Malhotra, Pramodh Kumar, and Maheshwari [16] and Tarjan [25] have given simpler $O(n^3)$-time algorithms. For sparse graphs, Sleator and Tarjan's bound of $O(nm \log n)$ [21,22] is the best known. For a small range of densities (m between $\Omega(n^2/(\log n)^3)$ and $O(n^2)$), Galil's bound of $O(n^{5/3} m^{2/3})$ [10] is best. For sparse graphs with integer edge capacities of moderate size, Gabow's scaling algorithm [9] is best. Among the algorithms in the table, the only parallel algorithm is that of Shiloach and Vishkin [20]. This algorithm has a parallel running time of $O(n^2 \log n)$ but requires $O(nm)$ space. Vishkin (private communication) has improved the space bound to $O(n^2)$.

In this paper we present a different approach to the maximum flow problem. Our method uses Karzanov's idea of a *preflow*. A preflow is like a flow except that the total amount flowing into a vertex can exceed the total amount flowing out. The algorithm maintains a preflow in the network and proceeds by moving local flow excess through the network along what it estimates to be shortest paths to the sink. Eventually the algorithm reaches a state in which no further excess can reach the sink. At this point the algorithm has determined the value of a maximum flow and (implicitly) a minimum cut. The remaining excess is then returned to the source, by backing it up through the network, and the algorithm terminates with a maximum flow.

The algorithm is simple and intuitive. It has natural implementations in sequential and parallel models of computation. We present a simple sequential implementation that runs in $O(n^3)$ time and a more complicated sequential implementation that uses the dynamic tree data structure of Sleator and Tarjan [22,23,24] and runs in $O(nm \log(n^2/m))$ time. The latter bound matches the best known bounds as a function of n and m for both sparse and dense graphs and is better than known bounds on graphs of intermediate density. We present a parallel version of the algorithm running in $O(n^2 \log n)$ time using $O(1)$ words of storage per edge. This matches the time bound of the Shiloach-Vishkin algorithm, but our improved space bound allows implementation on a model of distributed computation in which the amount of space per processor at a vertex is bounded by the vertex degree.

Our paper contains six sections in addition to the introduction. Section 2 describes a generic version of the algorithm. Section 3 proves its termination and correctness. Section 4 refines the algorithm to produce an $O(n^3)$-time sequential implementation. Section 5 adds the use of dynamic trees and thereby improves the sequential time bound to $O(nm \log(n^2/m))$. Section 6 discusses efficient distributed and parallel implementations. Section 7 contains some concluding remarks and open problems. The algorithm except for the $O(nm \log(n^2/m))$-time implementation was developed by the first author; an early version appeared in an M.I.T. technical memorandum [12]. A preliminary version of the present paper appeared in the Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing [13].

2. A Generic Maximum Flow Algorithm

Let $G = (V,E)$ be a directed graph with vertex set V and edge set E. We shall denote the size of V by n and the size of E by m. For ease in stating time bounds we assume $m \geq n - 1$. For a pair of vertices v and w we define the *distance* $d_G(v,w)$ from v to w in G to be the minimum number of edges in a path from v to w in G; if there is no such path, $d_G(v,w) = \infty$. A graph $G = (V,E)$ is a *network* if it has two distinguished vertices, a *source* s and a *sink* t, and a positive real-valued *capacity* $c(v,w)$ for each edge $(v,w) \in E$. We extend the capacity function to all vertex pairs by defining $c(v,w) = 0$ if $(v,w) \notin E$. A *flow* f on G is a real-valued function on vertex pairs satisfying the following constraints:

(1) (capacity constraint) $f(v,w) \leq c(v,w)$ for all $(v,w) \in V \times V$.

(2) (antisymmetry constraint) $f(v,w) = -f(w,v)$ for all $(v,w) \in V \times V$.

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

Remark. The antisymmetry constraint (2), which is nonstandard, has two purposes: (i) it eliminates the possibility of having positive flow on both edges of an opposing pair (v,w) and (w,v), a possibility that creates certain technical difficulties, and (ii) it simplifies the formal expression of constraints such as the flow conservation constraint (3). To gain an intuition one should think only of the positive part of the flow function; the appropriate interpretation of the flow conservation constraint is that the total flow into any node $v \notin \{s,t\}$ equals the total flow out of v. □

The *value* of a flow f is the net flow into the sink,

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

A *maximum flow* is a flow of maximum value.

The problem we wish to solve is that of computing a maximum flow in a given network. Our algorithm solves this problem by manipulating a *preflow* g on the network. A preflow is a real-valued function on vertex pairs satisfying (1) and (2) above and the following relaxation of flow conservation:

(4) (positivity constraint) $\sum_{u \in V} g(u,v) \geq 0$ for all $v \in V - \{s\}$.

That is, the total flow into any vertex $v \neq s$ is at least as great as the total flow out of v. We define the *flow excess* $e(v)$ of a vertex v to be infinity if $v = s$ and $\sum_{u \in V} g(u,v)$, the net flow into v, if $v \neq s$.

The preflow algorithm consists of two stages. During the first stage, the algorithm examines vertices with positive flow excess and pushes flow from them to vertices estimated to be closer to t, with the goal of getting as much excess as possible to t. Eventually the algorithm reaches a state in which no more excess can be moved to t. At this point the algorithm begins the second stage, during which it moves flow excesses remaining in the network back to s, without disturbing the excess at t. Once this is done, the preflow g is a flow, and indeed a maximum flow.

The first stage is the dominant part of the computation. In order to describe it, we must first address two issues: how to move flow excess from one vertex to another, and how to estimate the distance from a vertex to t.

To deal with the first issue, we define the *residual capacity* $r(v,w)$ of a vertex pair (v,w) to be $c(v,w) - g(v,w)$. If vertex v has positive excess and pair (v,w) has positive residual capacity, then an amount of flow excess equal to $\delta = \min\{e(v), r(v,w)\}$ can be moved from v to w by adding δ to $g(v,w)$ (and subtracting δ from $g(w,v)$). Observe that there are two ways a pair (v,w) can have positive residual capacity: either (v,w) is an edge with flow less than its capacity (edge (v,w) is said to be *unsaturated*), or (w,v) is an edge with positive flow. In the former case moving excess from v to w increases the flow on edge (v,w); in the latter case it decreases the flow on (w,v). We call a pair (v,w) a *residual edge* if $r(v,w) > 0$; the *residual graph* R for a preflow g is the graph whose vertex set is V and whose edge set is the set of residual edges (pairs (v,w) with $r(v,w) > 0$).

The second issue is how to estimate the distance from a vertex to t. For this purpose we define a *sink labeling* d to be a non-negative integer-valued function on the vertices, such that $d(t) = 0$ and $1 \leq d(v) \leq d(w) + 1$ for every residual edge (v,w). The intent is that $d(v)$ be an estimate of the distance from v to t in the residual graph R. An easy proof by

induction shows that $d(v)$ is always a lower bound on this distance, i.e. $d(v) \le d_R(v,t)$. This implies that $d(v) < n$ if there is any path from v to t in R. We call a vertex v *active* if $e(v) > 0$ and $1 \le d(v) < n$. The active vertices are those having excess that may possibly be movable to the sink.

We are now ready to specify the first stage of the algorithm. The algorithm begins with the zero preflow ($g(v,w) = 0$) and with some initial sink labeling d. The first stage consists of repeatedly performing the following steps, in any order, until there are no active vertices:

Push. Select any active vertex v and any residual edge (v,w) such that $d(v) = d(w) + 1$. Send $\delta = \min\{e(v), r(v,w)\}$ units of flow from v to w. This increases $g(v,w)$ and $e(w)$ by δ and decreases $g(w,v)$ and $e(v)$ by δ. The push is *saturating* if $r(v,w) = 0$ after the push and *nonsaturating* otherwise.

Relabel. Select any vertex v with $1 \le d(v) < n$. Replace $d(v)$ by $\min(\{d(w) + 1 \mid r(v,w) > 0\} \cup \{n\})$.

We shall prove in Section 3 that at the end of the first stage, if $e(v) > 0$ and $v \ne t$ then t is not reachable from v in the residual graph R. Furthermore, $e(t)$ is the value of a maximum flow.

The second stage of the algorithm returns to s the remaining flow excess on vertices other than t, by pushing it backward along estimated shortest paths from s. The second stage differs from the first in that the flow through an edge is only decreased, never increased. To estimate distances from s, the second stage uses a *source labeling* b, which is a non-negative integer-valued function on the vertices, such that $b(s) = 0$ and $b(u) + 1 \ge b(v) \ge 1$ for every edge (u,v) such that $g(u,v) > 0$. We call an edge (u,v) *positive* if $g(u,v) > 0$. The *positive graph* P for the preflow g is the graph whose vertex set is V and whose edge set is the set of positive edges. A proof by induction shows that $b(v)$ is a lower bound in the distance from s to V in P, i.e. $b(v) \le d_P(s,v)$.

The second stage begins with the preflow g computed by the first stage and with some initial source labeling b. It consists of repeatedly performing the following steps, in

any order, until no vertex $v \notin \{s,,t\}$ has $e(v) > 0$:

Push. Select any vertex $v \notin \{s,t\}$ such that $e(w) > 0$ and any positive edge (u,v) such that $b(u) + 1 = b(v)$. Send $\delta = \min\{e(v), g(u,v)\}$ units of flow from v back to u. This decreases $g(u,v)$ and $e(v)$ by 8 and increases $g(v,u)$ and $e(u)$ by 8. The push is *zeroing* if $g(u,v) = 0$ after the push and *nonzeroing* otherwise.

Relabel. Select any vertex v with $1 \le b(v) < n$. Replace $b(v)$ by $min(\{b(u)+1 \mid g(u,v) > 0\} \cup \{n\})$.

We shall prove in Section 3 that at the end of the second stage, g is a maximum flow.

There is one part of the algorithm we have not yet specified: the choice of an initial sink labeling d for the first stage and an initial source labeling b for the second stage. The simplest choice is $d(t) = 0$, $d(v) = 1$ for $v \neq t$, $b(s) = 0$, $b(v) = 1$ for $v \neq s$. A more accurate choice (indeed, the most accurate possible choice) is $d(v) = d_G(v,t)$, $b(v) = d_P(s,v)$, where P is the positive graph existing at the end the first stage. The latter pair of labelings can be computed in $O(m)$ sequential time or $O(n \log n)$ parallel time, using a backward breadth-first search from the sink at the beginning of the first stage to compute d and a forward breadth-first search from the source at the beginning of the second stage to compute b. The resource bounds we shall derive for the algorithm are valid for either of these pairs of initial labelings.

3. Correctness and Termination

We shall prove that the generic algorithm is correct assuming that it terminates and then prove termination. To prove correctness, we need a classical concept from network flow theory, that of a *cut*. A *cut* S, $\bar{S}$ is a partition of the vertex set V ($S \cup \bar{S} = V$, $S \cap \bar{S} = \emptyset$) such that $s \in S$ and $t \in \bar{S}$. The *capacity* of the cut is

$$c(S,\bar{S}) = \sum_{v \in S,\ w \in \bar{S}} c(v,w)$$

If g is a preflow, the *net flow* across the cut is

$$g(S,\bar{S}) = \sum_{v \in S,\ w \in \bar{S}} g(v,w)$$

If g is a flow, the flow conservation constraint implies that the net flow across any cut equals the flow value. It follows that any flow has value at most the capacity of any cut. The *max-flow, min-cut theorem* of Ford and Fulkerson [6,7] states that the value of a maximum flow equals the capacity of a minimum cut.

LEMMA 1. *The first stage maintains the invariant that d is a sink labeling. For any vertex v, d(v) never decreases.*

Proof. By induction on the number of pushing and relabeling steps. Given that d is a sink labeling, a relabeling step changing $d(v)$ can only increase $d(v)$ and must produce a new sink labeling. Consider a pushing step that sends flow from v to w. This step may add (w,v) to R and may delete (v,w). Since $d(w) = d(v) - 1$, neither the addition of (w,v) to R nor the deletion of (v,w) from R affects the invariant that d is a sink labeling. □

LEMMA 2. *The second stage maintains the invariant that b is a source labeling. For any vertex v, b(v) never decreases.*

Proof. Analogous to the proof of Lemma 1. □

LEMMA 3. *If g is a preflow, any vertex v with positive excess is reachable from s in the positive graph P.*

Proof. Let S be the set of vertices reachable from s in P and suppose $S \neq V$. Let $\bar{S} = V - S$. The choice of S implies that for every vertex pair u,v with $u \in S$, $v \in \bar{S}$, $g(u,v) \leq 0$. Thus

$$\sum_{v \in \bar{S}} e(v) = \sum_{u \in V,\ v \in \bar{S}} g(u,v) = \sum_{u \in S,\ v \in \bar{S}} g(u,v) + \sum_{u,\ v \in \bar{S}} g(u,v) \geq 0$$

since $\sum_{u,\ v \in \bar{S}} g(u,v) = 0$ by antisymmetry. Since g is a preflow, $e(v) = 0$ for all $v \in \bar{S}$. □

For a preflow g, let S_g, $\bar{S}_g$ be the vertex partition such that $\bar{S}_g$ contains all vertices from which t is reachable in the residual graph R.

LEMMA 4. *At the end of the first stage, the cut S_g, $\bar{S}_g$ is such that every vertex pair v,w with $v \in S_g$, $w \in \bar{S}_g$ satisfies $g(v,w) = c(v,w)$.*

Proof. At the end of the first stage, d is a sink labeling by Lemma 2. Every vertex $v \in \bar{S}_g$ has $d(v) < n$ since $d(v)$ is a lower bound on the distance from v to t in R, and this distance is either less than n or infinite. Thus every vertex $v \neq t$ with $e(v) > 0$ is in S_g, since otherwise there would still be an active vertex and the first stage would not have terminated. In particular $s \in S_g$, which means that S_g, $\bar{S}_g$ is a cut. If v,w is a vertex pair such that $v \in S_g$, $w \in \bar{S}_g$, then $r(v,w) = c(v,w) - g(v,w) = 0$ by the definition of $\bar{S}_g$. □

LEMMA 5. *The second stage maintains the following invariants: (i) if u,v is a vertex pair such that $g(u,v) > 0$ and $u \in S_g$, then $v \in S_g$; (ii) if $v \neq t$ and $e(v) > 0$, then $v \in S_g$; and (iii) the partition S_g, $\bar{S}_g$ remains fixed.*

Proof. Straightforward by induction on the number of pushing and relabaeling steps. □

THEOREM 1. *The cut S_g, $\bar{S}_g$ fixed throughout the second stage is a minimum cut. The preflow g at the end of the second stage is a maximum flow.*

Proof. Lemmas 2 and 3 imply that during the second stage, any vertex v with positive excess has $b(v) < n$. It follows that the second stage can only terminate when $e(v) = 0$ for all $v \notin \{s,t\}$, i.e. when g is a flow. Part (i) of Lemma 5 implies that the second stage moves flow excess only within S_g, which means that Lemma 4 remains true throughout the second stage. But if g is a flow such that $g(v,w) = c(v,w)$ for $v \in S_g$, $w \in \bar{S}_g$, then g is a maximum flow and S_g, $\bar{S}_g$ is a minimum cut. □

We prove termination of the algorithm by means of a sequence of lemmas. The lemmas come in pairs, each pair consisting of one result for the first stage and an analogous result for the second.

LEMMA 6. *During the first stage, the number of relabeling steps that actually change vertex labels is at most $(n-1)^2$.*

Proof. Throughout the algorithm, $d(t) = 0$. For any vertex $v \neq t$, $d(v)$ has at most n possible values (1 through n). Since by Lemma 1 $d(v)$ never decreases, it changes at most $n-1$ times. □

LEMMA 7. *During the second stage. the number of relabeling steps that actually change vertex labels is at most* $(n-1)^2$.

Proof. Analogous to the proof of Lemma 6. □

LEMMA 8. *The number of saturating pushing steps in the first stage is at most nm.*

Proof. For any pair of vertices v,w, consider the saturating pushes from v to w and from w to v. If there are any such pushes, it must be the case that $(v,w) \in E$ or $(w,v) \in E$. If $w = t$, there is at most one saturating push from v to w and none from w to v, since $d(w)$ is always zero. Suppose $v \neq t$, $w \neq t$. Consider a saturating push from v to w. In order to push flow from v to w again, the algorithm must first push flow from w to v, which cannot happen until $d(w)$ increases by at least two. Similarly, $d(v)$ must increase by at least two between saturating pushes from w to v. Since $d(v) + d(w) \geq 3$ when the first push between v and w occurs and $d(v) + d(w) \leq 2n - 3$ when the last such push occurs, the total number of saturating pushes between v and w is at most $n-2$. Thus the total number of saturating pushes is at most $\max\{1, n-2\} \leq n$ per edge, for a total over all edges of at most nm. □

LEMMA 9. *The number of zeroing pushing steps in the second stage is at most m.*

Proof. A given edge of G can have its flow made zero only once in the second stage. □

LEMMA 10. *The number of nonsaturating pushing steps in the first stage is at most* n^2m.

Proof. Let $\Phi = \Sigma\{d(v) \mid v \text{ is active}\}$. Each nonsaturating pushing step causes Φ to decrease by at least one. A saturating pushing step causes Φ to increase by at most $n-2$. The total increase in Φ over the entire first stage due to relabeling steps is at most $(n-1)^2$. Initially Φ is at most $n-1$, and Φ is always nonnegative. Thus the total decrease in Φ over the first stage, and hence the total number of nonsaturating pushing steps, is at most

$(n-1) + (n-2)\,nm + (n-1)^2 \le n^2 m$. □

LEMMA 11. *The number of nonzeroing pushing steps in the second stage is at most* $2n^2 + nm$.

Proof. Let $\Phi = \Sigma\{b(v) \mid v \notin \{s,t\} \text{ and } e(v) > 0\}$. Each nonzeroing pushing step causes Φ to decrease by at least one. A zeroing pushing step causes Φ to increase by at most $n-2$. The total increase in Φ over the entire second stage due to relabeling steps is at most $(n-1)^2$. Initially Φ is at most $(n-1)(n-2)$, and Φ is always non-negative. Thus the total decrease in Φ over the second stage, and hence the number of nonzeroing pushing steps, is at most $(n-1)(n-2) + (n-2)m + (n-1)^2 \le 2n^2 + nm$. □

THEOREM 2. *The first stage terminates after* $O(n^2 m)$ *steps, if every relabeling step changes a vertex label.*

Proof. Immediate from Lemmas 6, 8, and 10. □

THEOREM 3. *The second stage terminates after* $O(nm)$ *steps, if every relabeling step changes a vertex label.*

Proof. Immediate from Lemmas 7, 9, and 11. □

We conclude this section with two observations about the second stage. All previously known maximum flow algorithms first find a maximum flow and then a minimum cut; furthermore they must find a minimum cut to guarantee that a maximum flow has been found. In contrast, our algorithm first finds the maximum flow value and a minimum cut (in the first stage) and then a maximum flow (in the second stage). In applications requiring only the maximum flow value or a minimum cut (of which there are many; see [18]), the second stage need not be performed at all.

Although we have described the second stage as a simplified version of the first stage, there is an alternative way to convert a preflow g to a flow that is more efficient, at least theoretically. We first eliminate circulations (cycles of flow) by applying the algorithm of Sleator and Tarjan [23], which works for preflows even though originally stated for flows. This converts the positive graph P into an acyclic graph, without affecting the

cut $S_g, \bar{S}_g$. The sequential time required is $O(m \log n)$. Next, we eliminate flow excesses by processing the vertices of P in reverse topological order. To process a vertex $v \notin \{s,t\}$ with $e(v) > 0$, we reduce the flow on incoming edges by a total of $e(v)$. This increases the excess on predecessors of v. After all vertices have been processed, g is a flow, and $S_g, \bar{S}_g$ has not been changed. Thus g is a maximum flow and $S_g, \bar{S}_g$ is a minimum cut. Processing the vertices of P in topological order takes $O(m)$ sequential time. Thus the total time to convert g to a flow is $O(m \log n)$.

Either method of converting g to a flow takes less time than the first stage. Thus in the remainder of the paper we shall discuss only the first stage.

4. Sequential Implementation

As a first step toward obtaining an efficient sequential implementation, we shall describe a simple refinement of the generic algorithm that runs in $O(n^2 m)$ time, matching Dinic's bound. We need some data structures to represent the network and the preflow. We call an unordered pair $\{v,w\}$ such that $(v,w) \in E$ or $(w,v) \in E$ an *undirected edge* of G. We associate three values with each undirected edge $\{v,w\}$: $c(v,w)$, $c(w,v)$, and $g(v,w)$ $(=-g(w,v))$. Each vertex v has a list of the incident undirected edges $\{v,w\}$, in fixed but arbitrary order. Thus each edge $\{v,w\}$ appears in exactly two lists, the one for v and the one for w. Each vertex v has a *current edge* $\{v,w\}$, which is the current candidate for a pushing step out of v. Initially the current edge of v is the first edge on the edge list of v. The refined algorithm consists of repeating the following step until there are no active vertices:

Push/Relabel. Select any active vertex v (i.e. one with $e(v) > 0$ and $1 \le d(v) < n$). Let $\{v,w\}$ be the current edge of v. If $d(w) = d(v) - 1$ and $r(v,w) > 0$, apply a pushing step to send flow from v to w. Otherwise, replace $\{v,w\}$ as the current edge of v by the next edge on the edge list of v; if v has no next edge, make the first edge on the edge list of v the current one and apply a relabeling step to v.

LEMMA 12. *Each relabeling of a vertex v in a push/relabel step causes $d(v)$ to increase by at least one.*

Proof. Just before the relabeling, for each edge (v,w), either $d(v) \leq d(w)$ or $r(v,w) = 0$, because $d(v)$ has not changed since (v,w) was the current edge, $r(v,w)$ cannot increase unless $d(w) > d(v)$, and $d(w)$ never decreases. The lemma follows from the definition of a relabeling step. □

The refined algorithm needs one additional data structure, a set Q containing all active vertices. Initially $Q = \{s\}$. Maintaining Q takes only $O(1)$ time per push/relabeling step. (Such a step applied to an edge $\{v,w\}$ may require adding w to Q and/or deleting v.)

THEOREM 4. *The refined algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating pushing step, for a total of $O(n^2m)$ time.*

Proof. Let $v \neq t$ and let Δ_v be the number of edges on the edge list of v. Relabeling v requires a single scan of the edge list of v. By Lemma 12, the total number of passes through the edge list of v is at most $2n - 2$, one for each of the at most $n - 1$ relabelings of v and one before each relabeling as the current edge runs through the list. Every push/relabel step selecting v either causes a push, changes the current edge of v, or increases $d(v)$. It follows that the total time spent in push/relabel steps selecting v is $O(n\,\Delta_v)$ plus $O(1)$ time per push out of v. Summing over all vertices and applying Lemmas 8 and 10 gives the theorem. □

Remark. The analogous implementation of the second stage runs in $O(nm)$ time. □

To obtain a better running time we need to reduce the number of nonsaturating pushes. We can do this by exploiting the freedom we have in selecting vertices for push/relabel steps. We use a first-in, first-out selection strategy, i.e. we maintain Q as a queue. The *first-in, first-out algorithm* consists of applying the following step until Q is empty:

Discharge. Remove the vertex v on the front of Q. Apply push/relabel steps to v at least until $e(v)$ becomes zero or $d(v)$ increases. If a push from v to another vertex w causes $e(w)$ to become positive, add w to the rear of Q. After the push/relabel steps on v are completed, add v to the rear of Q if it is still active.

Note that there is still some flexibility in this algorithm, namely in how long we keep applying push/relabel steps to a vertex v. At one extreme, we can stop as soon as $e(v) = 0$ or v is relabeled. At the other extreme we can continue until v becomes inactive, which may involve several relabelings of v. Our analysis is valid for both extremes and all intermediate variants.

To analyze the first-in, first-out algorithm, we need to introduce the concept of *passes* over the queue. Pass one consists of the discharging step applied to the initial vertex on the queue, s. Given that pass i is defined, pass $i+1$ consists of the discharging steps applied to vertices on the queue that were added during pass i.

LEMMA 13. *The number of passes over the queue is at most* $2n(n-1)$.

Proof. Let $\Phi = \max\{d(v) \mid v \text{ is active}\}$. Consider the effect on Φ of a single pass over the queue. If Φ stays the same, some vertex label must increase by at least one. If Φ increases, some vertex label must increase by at least the same amount. The total number of passes in which Φ stays the same or increases is thus at most $(n-1)^2$. Since $\Phi < n$ initially and $\Phi > 0$ before the last pass, the total number of passes in which Φ decreases is at most $(n-1)^2 + (n-1)$. Hence the total number of passes is at most $2(n-1)^2 + n - 1 = 2n(n-1)$. □

COROLLARY 1. *The number of nonsaturating pushes during the first-in, first-out algorithm is at most* $2n(n-1)^2$.

Proof. There is at most one nonsaturating push per vertex other than t per pass. □

THEOREM 5. *The first-in, first-out algorithm runs in* $O(n^3)$ *time.*

Proof. Immediate from Theorem 4 and Corollary 1. □

An alternative strategy for vertex selection, which we call the *maximum distance method*, is to always select a vertex v in Q with $d(v)$ maximum. This strategy also gives an $O(n^3)$ running time, as a proof like that of Theorem 5 shows.

5. Use of Dynamic Trees

We have now matched the $O(n^3)$ time bound of Karzanov's algorithm. To obtain a better bound, we must reduce the time per nonsaturating pushing step below $O(1)$. We do this by using the dynamic tree data structure of Sleator and Tarjan [22, 23, 24]. This data structure allows the maintenance of a set of vertex-disjoint rooted trees in which each vertex v has an associated real value $h(v)$, possibly ∞ or $-\infty$. We shall regard a tree edge as directed toward the root, i.e. from child to parent. We denote the parent of a vertex v by $p(v)$. We adopt the convention that every vertex is both an ancestor and a descendant of itself.

The tree operations we shall need are the following:

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): Compute and return $h(v)$.

find min (v): Find and return the ancestor w of v of minimum value $h(w)$. In case of a tie, choose the vertex w closest to the root.

change value (v,x): Add x to $h(w)$ for all ancestors w of v. (We adopt the

convention that $\infty + (-\infty) = 0$.)

link (v,w): Combine the trees containing vertices v and w by making w the parent of v. 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 edge from v to its parent. This operation does nothing if v is a tree root.

The total time for a sequence of l operations starting with a collection of single-vertex trees is $O(l \log k)$, where k is an upper bound on the maximum number of vertices in a tree. (The implementation of dynamic trees presented in [23,24] does not support *find size* operations, but it is easily modified to do so. See the appendix.)

In our application the edges of the dynamic trees are a subset of the current edges of the vertices. The current edge $\{v,w\}$ of a vertex v is eligible to be a dynamic tree edge (with $p(v) = w$) if $d(v) < n$, $d(w) = d(v) - 1$, and $r(v,w) > 0$, but not all eligible edges are tree edges. The value $h(v)$ of a vertex v in its dynamic tree is $r(v,p(v))$ if v has a parent, ∞ if v is a tree root. Initially each vertex is in a one-vertex dynamic tree and has value ∞. We limit the maximum tree size to k, where k is a parameter to be chosen below.

By using appropriate tree operations we can push flow along an entire path in a tree, either causing a saturating push or moving flow excess from some vertex in the tree all the way to the tree root. Combining this idea with a careful analysis, we are able to show that the number of times a vertex is added to Q is $O(nm + n^3/k)$. At a cost of $O(\log k)$ for each tree operation, the total running time of the algorithm is $O((nm + n^3/k)\log k)$, which is minimized at $O(nm \log(n^2/m))$ for the choice $k = n^2/m$.

The details of the improved algorithm, which we call the *dynamic tree algorithm*, are as follows. The heart of the algorithm is the procedure *send* (v) defined below, which pushes excess from a non-root vertex v to the root of its tree, and cuts edges saturated by the push.

Send (v). Send $\delta = \min\{e(v)$, *find value* $(v)\}$ units of flow along the tree path from v by performing *change value* $(v, -\delta)$. Repeat the following step until *find root* $(v) = v$ or *find value* (*find min* $(v)) > 0$:

(*) Let $u =$ *find* min (v). Perform *cut* (u) followed by *change value* (u, ∞).

At the top level, the dynamic tree algorithm is exactly the same as the first-in, first-out algorithm of Section 4: we maintain a queue Q of active vertices and repeatedly perform discharging steps until Q is empty. However, we replace push/relabel steps with steps of the following kind:

Tree Push/Relabel. Let v be an active vertex and let $\{v,w\}$ be its current edge. Apply the apropriate one of the following cases:

(1) If $d(w) = d(v) - 1$ and $r(v,w) > 0$, apply the appropriate one of the following subcases:

(1a) If *find size* (v) + *find size* $(w) \le k$, make w the parent of v by performing *change value* $(v, -\infty)$, *change value* $(v, r(v,w))$, and *link* (v,w). Then push excess from v by performing *send* (v).

(1b) If *find size* (v) + *find size* $(w) > k$, apply a pushing step to move excess from v to w. Then perform *send* (w).

(2) If $d(w) > d(v) - 1$ or $r(v,w) = 0$, apply the appropriate one of the following subcases:

(2a) If $\{v,w\}$ is not the last edge on the edge list of v, replace $\{v,w\}$ as the current edge of v by the next edge on the edge list.

(2b) If $\{v,w\}$ is the last edge on the edge list of v, make the first edge on the edge list the current one, perform *cut* (u) and *change value* (u, ∞) for each vertex u whose parent is v, and apply a relabeling step to v.

It is important to realize that this algorithm stores values of the preflow g in two different ways. If $\{v,w\}$ is an edge that is not a dynamic tree edge, $g(v,w)$ is stored explicitly, with $\{v,w\}$. If $\{v,w\}$ is a dynamic tree edge, with w the parent of v, then $h(v) = c(v,w) - g(v,w)$ is stored implicitly in the dynamic tree data structure. Whenever a

tree edge (v,w) is cut, $h(v)$ must be computed and $g(v,w)$ restored to its current value. In addition, when the algorithm terminates, preflow values must be computed for all edges remaining in dynamic trees.

Before analyzing the running time of this algorithm, let us make a useful observation. An application of *send* (v) moves excess from v to the root of the tree containing it. If v still has excess, *send* cuts the tree edge from v to its parent. A proof by induction shows that the dynamic tree algorithm maintains the invariant that except in the middle of case (1) of a tree push/relabel step, all excess is on tree roots.

LEMMA 14. *The dynamic tree algorithm runs in* $O(nm \log k)$ *time plus* $O(\log k)$ *time per addition of a vertex to* Q.

Proof. The condition in subcase (1a) of tree push/relabel guarantees that the maximum size of any dynamic tree is k. Thus the time per dynamic tree operation is $O(\log k)$. Each tree push/relabel step takes $O(1)$ time plus $O(1)$ tree operations plus $O(1)$ tree operations per *cut* operation (in invocations of *send* and in subcase (2b)) plus time for relabeling (in subcase (2b)). The total relabeling time is $O(nm)$. The total number of *cut* operations is at most the number of *link* operations, which is at most nm (by a proof like that of Lemma 8). The total number of tree push/label steps is $O(nm)$ plus one per addition of a vertex to Q. Combining these observations gives the lemma. □

We define passes over the queue Q exactly as in Section 4. the proof of Lemma 13 remains valid, which means that the number of passes is at most $2n(n-1)$.

The next lemma is the crucial part of the analysis.

LEMMA 15. *The number of times a vertex is added to* Q *is* $O(nm + n^3/k)$.

Proof. A vertex v can be added to Q only after $d(v)$ increases, which happens at most $(n-1)^2$ times, or as a result of $e(v)$ increasing from zero, which can happen only in subcases (1a) and (1b) of tree push/relabel. An occurrence of (1a) can add one vertex to Q but also performs a link. Hence occurrences of (1a) cause at most nm additions to Q in total.

An occurrence of (1b) can add at most two vertices to Q, namely w and the root of the tree containing w before *send* (w) is invoked. If w is not the root of its tree, it can be added to Q only if *send* (w) causes a cut. Since there are at most nm cuts, subcase (1b) accounts for at most nm additions to Q plus one per occurrence of the subcase. There are at most nm occurrences of (1b) in which the invocation of *send* (w) causes a cut, and at most nm occurrences in which the push from v to w is saturating. Let us call an occurrence of (1b) *nonsaturating* if it adds a vertex to Q but causes neither a cut nor a saturating push. It remains for us to count the number of nonsaturating occurrences.

We need a few definitions. for any vertex u, we denote the dynamic tree containing u by T_u and the number of vertices it contains by $|T_u|$. Tree T_u is *small* if $|T_u| \leq k/2$ and *large* otherwise. At any time, and in particular at the beginning of any pass, there are at most $2n/k$ large trees.

Consider a nonsaturating occurrence of (1b) during a given pass, say pass i. The condition in (1b) guarantees that either T_v or T_w is large, giving us two cases to consider.

Suppose T_v is large. Vertex v is the root of T_v. The occurrence of (1b) removes all the excess from v, which means that a nonsaturating occurrence can apply to a given vertex v only once during a given pass. If T_v has changed since the beginning of pass i, we charge the occurrence of (1b) to the link or cut that changed T_v most recently before the occurrence. The number of such occurrences over all passes is at most one per link and two per cut, for a total of at most $3nm$. (A link forms one new tree; a cut, two.) If T_v has not changed since the beginning of pass i, we charge the occurrence of (1b) to T_v. Since T_v is large and there are at most $2n/k$ large trees at the beginning of pass i, there are at most $2n/k$ such charges per pass, for a total of at most $2n^2(n-1)/k$ over all passses.

Suppose on the other hand that T_w is large. The occurrence of (1b) adds the root of T_w, say r, to Q. A given vertex r can be added to Q at most once during a given pass. If T_w has changed since the beginning of pass i, we charge the occurrence of (1b) to the link or cut that changed T_w most recently before the occurrence. The number of such occurrences over all passes is at most $3nm$. If T_w has not changed, we charge the occurrence to T_w. The number of such charges over all passes is at most $2n^2(n-1)/k$.

Summing our estimates, we find that there are at most $(n-1)^2 + 10nm + 4n^2(n-1)/k$ additions to Q altogether, giving the lemma. □

THEOREM 6. *The dynamic tree algorithm runs in $O(nm\log(n^2/m))$ time if k is chosen equal to n^2/m.*

Proof. Immediate from Lemmas 14 and 15. □

As in Section 4, we can replace first-in, first-out selection of vertices for discharging steps by maximum distance selection, and still obtain the same running time bound.

6. Distributed and Parallel Implementation

The parallel version of our algorithm is a modification of the first-in, first-out algorithm of Section 4. We make three changes in the algorithm. first, we restrict the algorithm so that it stops processing a vertex v as soon as $e(v) = 0$ or v is relabeled. Second, instead of using a queue for selection of vertices to be processed, we process all active vertices in parallel. Third, the flow pushed to a vertex w during a parallel step is not added to $e(w)$ until the end of the step. To be more precise, the parallel version consists of repeating the following step until there are no active vertices:

Pulse. Perform the following computation in parallel for every active vertex v: apply push/relabel steps to v until $e(v)$ becomes zero or $d(v)$ increases. When relabeling v, compute its new label from the values of its neighbors' labels at the beginning of the pulse. Upon completion of the computation, recompute $e(v)$ for each affected vertex using the new flow.

The parallel algorithm is almost a special case of the first-in, first-out algorithm, the only difference being in the values used in relabeling and flow excess computations: in the first-in, first-out algorithm, relabeling steps in pass i use the most recent label and excess values, some of which may have been computed earlier in pass i. Nevertheless, a proof just like that of Lemma 13 gives the following analogous result for the parallel algorithm:

LEMMA 16. *The number of pulses made by the parallel algorithm is at most* $2n(n-1)$.

COROLLARY 2. *The number of non-saturating pushes made by the parallel algorithm is at most* $2n(n-1)^2$.

For distributed implementation of this algorithm, our computing model is as follows. We allow each vertex v of the graph to have a processor with an amount of memory proportional to Δ_v, the number of neighbors of v. This processor can communicate directly with the processors at all neighboring vertices. We assume that local computation is much faster than inter-processor communication. Thus as a measure of computation time we use the number of rounds of message-passing. We are also interested in the total number of messages sent. We shall consider both the synchronous and the asynchronous cases.

Each vertex processed during a pulse sends updated flow values to the appropriate neighbors. New vertex labels are also transmitted to neighbors, but only at the end of the pulse. Since flow always travels in the direction from larger to smaller labels, this delaying of the label broadcasting until the end of a pulse guarantees that flow only travels through an edge in one direction during a pulse. An easy analysis shows that in the synchronous case the distributed algorithm takes $O(n^2)$ rounds of message-passing and a total of $O(n^3)$ messages. Awerbuch (private communication, 1985) has observed that in the asynchronous case the synchronization protocol of [1] can be used to implement the algorithm in $O(n^2 \log n)$ rounds and $O(n^3)$ messages. The same bounds can be obtained for the Shiloach-Vishkin algorithm [20], but only by allowing more memory per processor: the processor at a vertex v needs $O(n\,\Delta_v)$ storage. Vishkin (private communication) has reduced the space required by this algorithm to a total of $O(n^2)$ (from $O(nm)$). Nevertheless, our algorithm has an advantage in situations where memory is at a premium.

For parallel implementation, our computing model is a PRAM [8] without concurrent writing. The implementation in this model is very similar to that of the distributed implementation, except that computations on binary trees must be performed to

allow each vertex to access its incident edges fast. Because of these binary trees, each pulse takes $O(\log n)$ time, and the parallel time of the algorithm is $O(n^2 \log n)$. The ideas of Shiloach and Vishkin [20] apply to our algorithm to show that $O(n)$ processors suffice to obtain the $O(n^2 \log n)$ time bound.

An alternative way to obtain a fast distributed or parallel algorithm is to use a parallel version of maximum distance selection: during each pulse, apply push/relabel steps to every active vertex v for which $d(v)$ is maximum. This requires a preprocessing step at the beginning of each pulse to compute the maximum $d(v)$, but it simplifies other calculations, since during a given pulse a vertex cannot both send and receive flow, which allows the computations of flow excess to proceed concurrently with the push/relabel steps.

7. Remarks

Our concluding remarks concern three issues: (i) better bounds, (ii) exact distance labeling, and (iii) efficient practical implementation. Regarding the possibility of obtaining better bounds for the maximum flow problem, it is interesting to note that the bottleneck in the sequential version of our algorithm is the nonsaturating pushes, whereas the bottleneck in the parallel version is the saturating pushes. We wonder whether an $O(nm)$ sequential time bound can be obtained through more careful handling of the nonsaturating pushes, possibly avoiding the use of the dynamic tree data structure. Perhaps also an $O(m(\log n)^k)$ parallel time bound can be obtained through use of a parallel version of the dynamic tree data structure.

On graphs with integer edge capacities bounded by N, Gabow's scaling algorithm [9] runs in $O(nm \log N)$ sequential time. It consists of $\log N$ applications of Dinic's algorithm, each taking $O(nm)$ time. We would like to know whether some version of our algorithm runs in $O(nm \log N)$ time on this class of networks. A variant that suggests itself is to always push flow from the vertex with largest excess.

It is possible to modify the first-in first-out algorithm so that when a pushing step is executed, the distance labels are exact distances to the sink in the residual graph. The

modification involves a stronger interpretation of the next edge of a vertex, which should be unsaturated and lead to a vertex with a smaller label. If a pushing step saturates the current edge of v, a new current edge is found by scanning the edge list of v and relabeling if the end of the list is reached, as in a push/relabel step. If a relabeling step changes the label of v, the current edge must be updated for each vertex u such that (u,v) is the current edge of u. One can show that these computations take $O(nm)$ time total during the algorithm.

It is not clear whether maintaining exact distance labels will actually improve the performance of the algorithm in practice, because the work of maintaining the exact labels may exceed the extra work due to nonexact labels. However, the above observation suggests that as long as we are interested in an $\Omega(nm)$ upper bound on an implementation of the generic algorithm, we can assume that the exact labeling is given to us for free.

Our algorithm is simple enough to be potentially practical. In a practical implementation it is important to make the algorithm as fast as possible. We offer two heuristics that may speed up the algorithm by more quickly showing that vertices with positive excess have no paths to the sink, while avoiding the extra work entailed in maintaining exact distance labels. The first idea is to periodically bring the distance labels up to date by performing a breadth-first search backward from the sink. Doing this each time an edge into the sink is saturated, and/or after every n relabelings, will not affect the worst-case running time of the algorithm. The second idea is to maintain a set D of *dead* vertices, those with no paths to the sink. After the edges out of the source are saturated, the source becomes dead. In general, any vertex v with $d(v) \geq n - |D|$ can be declared dead. Once all the excess is on dead vertices, the first stage terminates.

Another important issue in a practical implementation is what strategy to use for selecting vertices for discharging steps. Although the best theoretical bounds we have obtained are for first-in, first-out and maximum distance selection, other strategies, such as last-in, first-out and maximum excess, deserve consideration. It may be that a hyprid strategy will be best in practice.

8. Acknowledgements

We thank Baruch Awerbuch, Charles Leiserson, and David Schmoys for many suggestions and stimulating discussions. The first author is very grateful to the Fannie and John Hertz Foundation for his support.

Appendix. Finding Sizes of Dynamic Trees

The dynamic tree implementation of [23,24] does not support *find size* operations, but can be easily modified to do so, as follows. We assume some familiarity with [23] or [24]. The implementation represents each dynamic tree by a *virtual tree* having the same vertex set but different structure. To allow find size operations to be performed efficiently, we store with each virtual tree root its tree size, and with every other vertex the difference between its virtual subtree size and that of its virtual parent. This information is easy to update after each dynamic tree operation; the updating increases the time per operation by only a constant factor. A *find size* operation is performed just like a *find root* operation: *find root(v)* has the effect of locating the root of the virtual tree containing v, which contains the tree size.

References

[1] B. Awerbuch, "An efficient network synchronization protocol," *Proc. 16th ACM Symp. on Theory of Computing* (1984), 522-525.

[2] R. V. Cherkasky, "Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations," *Mathematical Methods of Solution of Economical Problems* 7 (1977), 112-125 (in Russian).

[3] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.* 11 (1980), 1277-1280.

[4] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. Assoc. Comput. Mach. 19 (1972), 248-264.*

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

[6] L. R. Ford, Jr. and D. R. Fulkerson, "Maximal flow through a network," *Can. J. Math.* 8 (1956), 399-404.

[7] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[8] S. Fortune and J. Wylle, "Parallelism in random access machines," *Proc. 10th ACM Symp. on Theory of Computing* (1978), 114-118.

[9] H. N. Gabow, "Scaling algorithms for network problems," *Proc. 24th IEEE Symp. on Found. of Comput. Science* (1983), 248-258.

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

[11] Z. Galil and A. Naamad, "An $O(EV\log^2 V)$ algorithm for the maximal flow problem," *J. Comput. System Sci.* 21 (1980), 203-217.

[12] A. V. Goldberg, "A new max-flow algorithm," Tech. Mem. MIT/LCS/TM-291, Laboratory for Computer Science, Mass. Inst. of Tech., Cambridge, MA, 1985.

[13] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," *Proc. 18th Annual ACM Symp. on Theory of Computing* (1986), to appear.

[14] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Math. Dokl.* 15 (1974), 434-437.

[15] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Reinhart, and Winston, New York, NY, 1976.

[16] 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), 277-278.

[17] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ. 1982.

[18] J. C. Picard and H. D. Ratliff, "Minimum cuts and related problems," *Networks* 5 (1975), 357-370.

[19] Y. Shiloach, "An $O(n \cdot I \log^2 I)$ maximum-flow algorithm," Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford University, Stanford, CA, 1978.

[20] Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ parallel max-flow algorithm," *J. Algorithms* 3 (1982), 128-146.

[21] D. D. Sleator, "An $O(n\,m \log n)$ algorithm for maximum network flow," Tech. Rep. STAN-CS-80-831, Computer Science Dept., Stanford University, Stanford, CA, 1980.

[22] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," J. Comput. System Sci. 24 (1983), 362-391.

[23] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees, *J. Assoc. Comput. Mach.* 32 (1985), 652-686.

[24] R. E. Tarjan, Data Structures and Network Algorithms, Society for Industrial and Applied Math., Philadelphia, PA, 1983.

[25] R. E. Tarjan, "A simple version of Karzanov's blocking flow algorithm," *Operations Research Lett.* 2 (1984), 265-268.