A New Approach to the Maximum-Flow Problem

ANDREW V. GOLDBERG

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

ROBERT E. TARJAN

Princeton University, Princeton, New Jersey, and AT&T Bell Laboratories, Murray Hill, New Jersey

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). An alternative method based on the *preflow* concept of Karzanov is introduced. 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. A parallel implementation running in $O(n^2 \log n)$ time using n processors and $O(m)$ space is obtained. This time bound matches that of the Shiloach–Vishkin algorithm, which also uses n processors but requires $O(n^2)$ space.

Categories and Subject Descriptors: F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems; G.2.2 **[Discrete Mathematics]**: Graph Theory—*graph algorithms*; *network problems*

General Terms: Algorithms, Design, Theory, Verification

Additional Key Words and Phrases: Dynamic trees, maximum-flow problem

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

A preliminary version of this paper appeared in the *Proceedings of the 18th Annual ACM Symposium on Theory of Computing* (Berkeley, Calif., May 28–30). ACM, New York, 1986, pp. 136–146.

The work of A. V. Goldberg was supported by a Fannie and John Hertz Foundation Fellowship and by the Advanced Research Projects Agency of the Department of Defense under contract N00014-80-C-0622. The work of R. E. Tarjan was partially supported by the National Science Foundation under grant DCR-8605962 and the Office of Naval Research under Contract N00014-87-K-0467.

Authors' present addresses: A. V. Goldberg, Department of Computer Science, Stanford University, Stanford, CA 94305; R. E. Tarjan, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974-2070.

TABLE I. POLYNOMIAL-TIME ALGORITHMS FOR THE MAXIMUM FLOW PROBLEM[a]

Algorithm no.	Date	Discoverer	Running time	References
1	1969	Edmonds and Karp	$O(nm^2)$	[5]
2	1970	Dinic	$O(n^2m)$	[4]
3	1974	Karzanov	$O(n^3)$	[18]
4	1977	Cherkasky	$O(n^2m^{1/2})$	[3]
5	1978	Malhotra, Pramodh Kumar, and Maheshwari	$O(n^3)$	[21]
6	1978	Galil	$O(n^{5/3}m^{2/3})$	[11]
7	1978	Galil and Naamad; Shiloach	$O(nm(\log n)^2)$	[12, 25]
8	1980	Sleator and Tarjan	$O(nm \log n)$	[27, 28]
9	1982	Shiloach and Vishkin	$O(n^3)$	[26]
10	1983	Gabow	$O(nm \log U)$	[10]
11	1984	Tarjan	$O(n^3)$	[31]
12	1985	Goldberg	$O(n^3)$	[14]
13	1986	Goldberg and Tarjan	$O(nm \log(n^2/m))$	[16, 15]
14	1986	Ahuja and Orlin	$O(nm + n^2 \log U)$	[1]

[a] Algorithm 13 is presented in this paper.

discussion of the problem and its applications can be found in the books of Even [6], Ford and Fulkerson [8], Lawler [19], Papadimitriou and Steiglitz [23], and Tarjan [30]. Table I summarizes known polynomial-time algorithms for the problem. Time bounds are stated in terms of the number n of vertices, the number m of edges, and in two cases in terms of an upper bound U on the edge capacities (assumed in these cases to be integers).

The first maximum-flow algorithm, due to Ford and Fulkerson [7], works by finding augmenting paths. Edmonds and Karp [5] observed that augmenting along shortest paths leads to a polynomial-time algorithm (algorithm 1). To improve the efficiency further, Dinic [4] proposed a method to find all shortest augmenting paths in one phase. Algorithms 2–11 use Dinic's method. Algorithms 12–14 are based on the approach described in this paper.

There is no clear winner among the algorithms in the table that are based on Dinic's method. Algorithms 3, 5, 9, and 11 are designed to be fast on dense graphs, and algorithms 4, 6, 7, 8, and 10 are designed to be fast on sparse graphs. For dense graphs, the best known bound of $O(n^3)$ was first obtained by Karzanov [18]; Malhotra et al. [21] and Tarjan [31] have given simpler $O(n^3)$-time algorithms. For sparse graphs, Sleator and Tarjan's bound of $O(nm \log n)$ [27, 28] is the best to date. For a small range of densities (m between $\Omega(n^2/(\log n)^3)$ and $O(n^2)$), Galil's bound of $O(n^{5/3}m^{2/3})$ [11] is best. For sparse graphs with integer edge capacities of moderate size, Gabow's scaling algorithm [10] is best. Among the algorithms based on Dinic's method, the only parallel algorithm is that of Shiloach and Vishkin [26]. This algorithm has a parallel running time of $O(n^2 \log n)$ with n processors but requires $O(nm)$ space. Vishkin (private communication, 1986) has improved the space bound to $O(n^2)$. Our work has been motivated by the Shiloach–Vishkin algorithm.

In this paper we present a different approach to the maximum-flow problem, which is the basis for algorithms 12–14 in Table I. 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. During each phase, Karzanov's algorithm maintains a preflow in an acyclic network. The algorithm pushes flow through the network to find a blocking flow, which determines the acyclic network

for the next phase. Our algorithm abandons the idea of finding a flow in each phase and also abandons the idea of global phases. Instead, our algorithm maintains a preflow in the original network and pushes local flow excess toward the sink along what it estimates to be shortest paths in the residual graph. This pushing of flow changes the residual graph, and paths to the sink may become saturated. Excess that cannot be moved to the sink is returned to the source, also along estimated shortest paths. Only when the algorithm terminates does the preflow become a flow, and then it is 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 $O(nm \log(n^2/m))$-time sequential implementation that uses the dynamic tree data structure of Sleator and Tarjan [28–30] (also used in algorithm 8). 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 "almost dense" graphs (i.e., $m = n^{2-o(1)}$ and $m = n^2/\omega(1)$). We present a parallel version of the algorithm running on $O(n^2 \log n)$ time using n processors and $O(1)$ 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. Recently, Ahuja and Orlin [1] used the approach described in this paper to develop an $O(nm + n^2 \log U)$ algorithm for the problem, improving Gabow's [10] bound of $O(nm \log U)$.

This 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 shows how to use dynamic trees to improve 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 approach presented in this paper has been pioneered by the first author [14]. The version presented here generalizes and improves the original results. Our techniques can be further generalized to solve the minimum-cost flow problem. This generalization is described in [15] and [17].

2. *A Generic Maximum-Flow Algorithm*

Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. We 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, w we define the *distance* $d_G(v, w)$ from v to w in G to be the minimum number of edges on a path from v to w in G; if there is no such path, we define $d_G(v, w) = \infty$. A graph $G = (V, E)$ is a *flow 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:

$$f(v, w) \leq c(v, w) \quad \text{for all} \quad (v, w) \in V \times V \quad \text{(capacity constraint)}, \tag{1}$$

$$f(v, w) = -f(w, v) \quad \text{for all} \quad (v, w) \in V \times V \quad \text{(antisymmetry constraint)}, \tag{2}$$

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

Remark. The antisymmetry constraint (2), used by Sleator [27], 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 conservation constraint (3). To gain an intuition, 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 vertex $v \notin \{s, t\}$ equals the total flow out of v.

The *value* $|f|$ 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* f on the network. A preflow is a real-valued function on vertex pairs satisfying (1) and (2) above, as well as the following weakened form of (3):

$$\sum_{u \in V} f(u, v) \geq 0 \qquad \text{for all} \quad v \in V - \{s\} \qquad \text{(nonnegativity constraint).} \quad (4)$$

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 $\sum_{u \in V} f(u, v)$, the net flow into v.

The preflow algorithm works by examining vertices other than s and t with positive flow excess and pushing excess from them to vertices estimated to be closer to the sink t, with the goal of getting as much excess as possible to t. If the sink is not reachable from a vertex with a positive excess, however, the algorithm pushes this excess to vertices estimated to be closer to the source s. Eventually the algorithm reaches a state in which all vertices other than s and t have zero excess. At this point the preflow f is a flow; in fact, f is a maximum flow.

Before describing the algorithm, we address two issues: how to move flow excess from one vertex to another and how to estimate the distance from a vertex to s or to t.

To deal with the first issue, we define the *residual capacity* $r_f(v, w)$ of a vertex pair (v, w) to be $c(v, w) - f(v, w)$. If vertex v has positive excess and pair (v, w) has positive residual capacity, then an amount of flow excess up to $\delta = \min(e(v), r_f(v, w))$ can be moved from v to w by adding δ to $f(v, w)$ (and subtracting δ from $f(w, v)$). 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 $r_f(v, w) > 0$; the *residual graph* $G_f = (V, E_f)$ for a preflow f is the graph whose vertex set is V and whose edge set E_f is the set of residual edges.

The second issue is how to estimate the distance from a vertex to s or to t. For this purpose we define a *valid labeling* d to be a function from the vertices to the nonnegative integers and infinity,[1] such that $d(s) = n$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every residual edge (v, w). The intent is that, if $d(v) < n$, then $d(v)$ is a lower bound on the actual distance from v to t in the residual graph G_f,

[1] Our definition allows distance labels to be infinite. We shall show, however, that the labels stay finite throughout the execution of the algorithm. Infinite labels are introduced only to simplify the exposition.

Push(v, w).
Applicability: v is active, $r_f(v, w) > 0$ **and** $d(v) = d(w) + 1$.
Action: Send $\delta = \min(e(v), r_f(v, w))$ units of flow from v to w as follows:
$f(v, w) \leftarrow f(v, w) + \delta; f(w, v) \leftarrow f(w, v) - \delta;$
$e(v) \leftarrow e(v) - \delta; e(w) \leftarrow e(w) + \delta.$

Relabel(v).
Applicability: v is active **and** $\forall w \in V, r_f(v, w) > 0 \Rightarrow d(v) \leq d(w)$.
Action: $d(v) \leftarrow \min\{d(w) + 1 \mid (v, w) \in E_f\}$.
(If this minimum is over an empty set, $d(v) \leftarrow \infty$.)

FIG. 1. Push and relabel operations.

```
Procedure Max-Flow (V, E, s, t, c);
  ⟨⟨initialization⟩⟩
    ⟨⟨initialize preflow⟩⟩
      ∀(v, w) ∈ (V − {s}) × (V − [s]) do begin
        f(v, w) ← 0; f(w, v) ← 0;
      end;
      ∀v ∈ V do begin
        f(s, v) ← c(s, v);
        f(v, s) ← −c(s, v);
      end;
    ⟨⟨initialize labels and excesses⟩⟩
      d(s) ← n;
      ∀v ∈ V − {s} do begin
        d(v) ← 0;
        e(v) ← f(s, v);
      end;
  ⟨⟨loop⟩⟩
    while ∃ a basic operation that applies do
      select a basic operation and apply it;
  return(f);
end.
```

FIG. 2. The generic maximum flow algorithm. The running time of the algorithm depends on the order in which basic operations are applied and on details of the implementation.

and if $d(v) \geq n$, then $d(v) - n$ is a lower bound on the actual distance to s in the residual graph. (It can be proved by induction that in the latter case t is not reachable from v in G_f.)

To describe the algorithm, we also need the following definition. We call a vertex v *active* if $v \in V - \{s, t\}$, $d(v) < \infty$, and $e(v) > 0$.

The maximum-flow algorithm begins with the preflow f that is equal to the edge capacity on each edge leaving the source and zero on all other edges, and with some initial sink labeling d. The algorithm then repetitively performs, in any order, the *basic operations*, *push* and *relabel*, described in Figure 1. When there are no active vertices, the algorithm terminates. A summary of the algorithm appears in Figure 2.

The basic operations modify the preflow f and the labeling d. A *push* from v to w increases $f(v, w)$ and $e(w)$ by $\delta = \min(e(v), r_f(v, w))$ and decreases $f(w, v)$ and $e(v)$ by the same amount. The push is *saturating* if $r_f(v, w) = 0$ after the push and *nonsaturating* otherwise. A *relabeling* of v sets the label of v equal to the largest value allowed by the valid labeling constraints.

LEMMA 2.1. *If f is a preflow, d is any valid labeling for f, and v is any active vertex, then either a push or a relabel operation is applicable to v.*

PROOF. For any any residual edge (v, w), the definition of a valid labeling implies that $d(v) \leq d(w) + 1$. If a push is not applicable to v, then $d(v) <$

$d(w) + 1$ for every residual edge (v, w). By the integrality of valid labelings, $d(v) \leq d(w)$ for all residual edges (v, w), and a relabeling is applicable to v. □

There is one part of the algorithm we have not yet specified: the choice of an initial labeling d. The simplest choice is $d(s) = n$ and $d(v) = 0$ for $v \in V - \{s\}$. A more accurate choice (indeed, the most accurate possible choice) is $d(v) = \min(d_{G_f}(v, t), d_{G_f}(v, s) + n)$ for $v \in V$, where f is the initial preflow. The latter labeling can be computed in $O(m)$ time using backward breadth-first searches from the sink and from the source in the residual graph. The resource bounds we shall derive for the algorithm are correct for any valid initial labeling. To simplify the proofs, we assume that the algorithm starts with the simple labeling.

3. *Correctness and Termination*

We shall prove that the generic algorithm is correct assuming that it terminates and then prove termination.

LEMMA 3.1. *The algorithm maintains the invariant that d is a valid labeling.*

PROOF. We use induction on the number of pushing and relabeling operations. The simple labeling used initially is valid because labels of all vertices other than s are zero, and all edges leaving s are saturated. Given that d is a valid labeling, a relabeling operation changing $d(v)$ must produce a new valid labeling. Consider a pushing operation that sends flow from v to w. This operation may add (w, v) to G_f and may delete (v, w) from G_f. Since $d(w) = d(v) - 1$, the addition of (w, v) to G_f does not affect the invariant that d is a valid labeling. The deletion of (v, w) removes the corresponding constraint, which also leaves the labeling valid. □

To prove correctness, we use the concept of an augmenting path. An *augmenting path* is a simple path from s to t in the residual graph G_f. Our proof of correctness is based on the following classical theorem of Ford and Fulkerson [7]:

THEOREM 3.2. *A flow f is maximum if and only if there is no augmenting path; that is, t is not reachable from s in G_f.*

LEMMA 3.3. *If f is a preflow and d is any valid labeling for f, then the sink t is not reachable from the source s in the residual graph G_f.*

PROOF. Assume by way of contradiction that there is an augmenting path $s = v_0, v_1, \ldots, v_l = t$. Then $l < n$ and $(v_i, v_{i+1}) \in E_f$ for $0 \leq i < l$. Since d is a valid labeling, we have $d(v_i) \leq d(v_{i+1}) + 1$ for $0 \leq i < l$. Therefore, we have $d(s) \leq d(t) + l < n$, since $d(t) = 0$, which contradicts $d(s) = n$. □

THEOREM 3.4. *Suppose that the algorithm terminates and all distance labels are finite at termination. Then the preflow f is a maximum flow; that is, the algorithm is correct.*

PROOF. If the algorithm terminates and all distance labels are finite, all vertices in $V - \{s, t\}$ must have zero excess, because there are no active vertices. Therefore f must be a flow. This flow is maximum by Lemma 3.3 and Theorem 3.2. □

Now we show that the algorithm terminates and that the distance labels stay finite during the execution of the algorithm. First we prove the following lemma:

LEMMA 3.5. *If f is a preflow and v is a vertex with positive excess, then the source s is reachable from v in the residual graph G_f.*

PROOF. Let S be the set of vertices reachable from v in G_f, and suppose $s \notin S$. Let $\bar{S} = V - S$. The choice of S implies that, for every vertex pair u, w with $w \in S$ and $u \in \bar{S}$, we have $f(u, w) \leq 0$. Thus

$$\begin{aligned}\sum_{w \in S} e(w) &= \sum_{u \in V, w \in S} f(u, w) \\ &= \sum_{u \in \bar{S}, w \in S} f(u, w) + \sum_{u, w \in S} f(u, w) \\ &= \sum_{u \in \bar{S}, w \in S} f(u, w) \\ &\leq 0.\end{aligned}$$

The term $\sum_{u, w \in S} f(u, w)$ in the second line equals zero by antisymmetry. Since f is a preflow, we have $e(w) = 0$ for all $w \in S$, and in particular we have $e(v) = 0$. □

LEMMA 3.6. *For any vertex v, the distance label $d(v)$ never decreases. An application of a relabeling operation to v increases $d(v)$.*

PROOF. Since the labeling d is changed using relabeling operations only, it is enough to prove the second statement of the lemma. Suppose a relabeling operation is applicable to v. Then for all w such that $(v, w) \in E_f$, we have $d(w) \geq d(v)$, which implies that $\min\{d(w) + 1 \mid (v, w) \in E_f\} > d(v)$, so the relabeling must increase $d(v)$. □

We have shown that an application of the relabeling operation to a vertex increases the vertex label. The next lemma shows that the labels cannot increase too much. In particular, the lemma implies that the labels stay finite during an execution of the algorithm.

LEMMA 3.7. *At any time during the execution of the algorithm and for any vertex $v \in V$, $d(v) \leq 2n - 1$.*

PROOF. The lemma is trivial for $v = s$ and $v = t$. Suppose $v \in V - \{s, t\}$. Since the algorithm changes only labels of active vertices, it is enough to prove the lemma for an active vertex v. If v is active, then $e(v) > 0$, so by Lemma 3.5 there is a simple path from v to s in G_f. Let $v = v_0, v_1, \ldots, v_l = s$ be such a path. The length l of the path is at most $n - 1$. Since d is a valid labeling and $(v_i, v_{i+1}) \in E_f$, we have $d(v_i) \leq d(v_{i+1}) + 1$. Therefore, we have $d(v) = d(v_0) \leq d(v_l) + l \leq d(s) + (n - 1) = 2n - 1$. □

Lemma 3.7 allows us to amortize the work done by the algorithm over increases in vertex labels. The next two lemmas bound the number of relabelings and the number of saturating pushes.

LEMMA 3.8. *The number of relabeling operations is at most $2n - 1$ per vertex and at most $(2n - 1)(n - 2) < 2n^2$ overall.*

PROOF. Relabeling operations apply only to vertices $v \in V - \{s, t\}$. A relabeling of v increases $d(v)$. The label $d(v)$ is zero initially, and the label can grow to at most $2n - 1$. Therefore, there are at most $2n - 1$ relabelings of each vertex in $V - \{s, t\}$, and the total number of relabelings is at most $(2n - 1)(n - 2)$. □

LEMMA 3.9. *The number of saturating push operations is at most $2nm$.*

PROOF. For any pair of vertices v and w, consider the saturating pushes from v to w and from w to v. If there are any such pushes, then either $(v, w) \in E$ or $(w, v) \in E$. 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 2. Similarly, $d(v)$ must increase by at least 2 between saturating pushes from w to v. Since $d(v) + d(w) \geq 1$ when the first push between v and w occurs and $d(v) + d(w) \leq 4n - 3$ when the last such push occurs (by Lemma 3.7), the total number of saturating pushes between v and w is at most $2n - 1$. Thus the total number of saturating pushes is at most $2n - 1$ per edge, for a total over all edges of at most $(2n - 1)m < 2nm$. □

Next we bound the number of nonsaturating pushes.

LEMMA 3.10. *The number of nonsaturating pushing operations is at most* $4n^2m$.

PROOF. Let $\Phi = \sum_{\{v \mid v \text{ is active}\}} d(v)$. Each nonsaturating push from a vertex v to another vertex w causes Φ to decrease by at least 1, since the push makes v inactive and $d(w) = d(v) - 1$. A saturating pushing operation causes Φ to increase by at most $2n - 1$. The total increase in Φ due to saturating pushes is at most $(2n - 1) \times 2nm$ by Lemma 3.9. The total increase in Φ over the entire algorithm due to relabeling operations is at most $(2n - 1)(n - 2)$ by Lemma 3.8. Before the first push or relabel operation Φ is zero, and at the end of the algorithm Φ is also zero. Thus the total decrease in Φ, and hence the total number of nonsaturating pushing operations, is equal to the total increase in Φ, which is at most $(2n - 1)2nm + (2n - 1)(n - 2) \leq 4n^2m$ (recall the assumption $m \geq n - 1$). □

THEOREM 3.11. *The generic algorithm terminates after* $O(n^2m)$ *basic operations.*

PROOF. Immediate from Lemmas 3.8, 3.9, and 3.10. □

The running time of the generic algorithm depends on the order in which basic operations are applied and the details of implementation, but it is clear that any reasonable sequential implementation of the algorithm will run in polynomial time. In the next section we discuss one possible implementation with an $O(n^2m)$ time bound. We also show that a particular ordering of the basic operations yields an $O(n^3)$ time bound.

We conclude this section with a discussion of a variant of the generic maximum-flow algorithm. Let us recall a classical concept from network flow theory, that of a cut. A *cut* $S, \bar{S}$ is a partition of the vertex set V (that is, $S \cup \bar{S} = V$ and $S \cap \bar{S} = \emptyset$) such that $s \in S$ and $t \in \bar{S}$. The *capacity* of the cut is

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

A cut is *minimum* if it has minimum possible capacity. The *max-flow, min-cut* theorem of Ford and Fulkerson [7, 8] states that the value of a maximum flow equals the capacity of a minimum cut.

In many applications in which the maximum-flow problem occurs, only the maximum-flow value or a minimum cut is needed, not an actual maximum flow [24]. For such applications our maximum-flow algorithm can be modified to compute a minimum cut and the maximum-flow value without actually computing a maximum flow. The only change necessary is to redefine an active vertex to be a vertex $v \in V - \{s, t\}$ such that $e(v) > 0$ and $d(v) < n$. When the modified algorithm terminates, the excess $e(t)$ at the sink is the value of a maximum flow, and the cut $S, \bar{S}$ such that $\bar{S}$ contains exactly those vertices from which t is reachable in G_f is a minimum cut [16]. For this variant of the algorithm, the bounds in Lemmas 3.8–3.10 can be improved by roughly a factor of 2. The results we derive

in subsequent sections for the maximum-flow algorithm apply to this minimum-cut algorithm as well.

4. *Sequential Implementation*

Any reasonable implementation of the generic maximum-flow algorithm runs in polynomial time. Some implementations, however, are more efficient than others. We start with a simple implementation that runs in $O(n^2m)$ time and then refine it to improve its efficiency.

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 the three values $c(v, w)$, $c(w, v)$, and $f(v, w)(= -f(w, v))$ with each undirected edge $\{v, w\}$. 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 operation from v. Initially, the current edge of v is the first edge on the edge list of v. The main loop of the implementation consists of repeating the *push/relabel* operation described in Figure 3 until there are no active vertices. (We discuss the maintenance of active vertices later.) The *push/relabel* operation is applicable to an active vertex v. This operation pushes excess through the current edge $\{v, w\}$ of v if a pushing operation is applicable to this edge. If not, the operation replaces $\{v, w\}$ as the current edge of v by the next edge on the edge list of v; or if $\{v, w\}$ is the last edge on this list, it makes the first edge on the list the current one and relabels v.

We need to show that *push/relabel* uses the relabeling operation correctly.

LEMMA 4.1. *The push/relabel operation does a relabeling only when the relabeling operation is applicable.*

PROOF. A *push/relabel* operation relabels a vertex v only when v is active. Just before the relabeling, for each edge (v, w), either $d(v) \leq d(w)$ or $r_f(v, w) = 0$. This is because the distance label $d(v)$ has not changed since (v, w) was the current edge, $d(w)$ never decreases, and $r_f(v, w)$ cannot increase unless $d(w) > d(v)$. The lemma follows from the definition of a relabeling operation. □

The implementation needs one additional data structure, a set Q containing all active vertices. Initially $Q = \{v \in V - \{s, t\} \mid c(s, v) > 0\}$. Maintaining Q takes only $O(1)$ time per *push/relabel* operation. (Such an operation applied to an edge $\{v, w\}$ may require adding w to Q and/or deleting v.)

THEOREM 4.2. *The push/relabel implementation of the maximum-flow 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 be a vertex in $V - \{s, t\}$, 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 3.8, the total number of passes through the edge list of v is at most $4n - 1$, one for each of the at most $(2n - 1)$ relabelings of v, one before each relabeling as the current edge runs through the list, and one after the last relabeling. Every *push/relabel* operation selecting v either causes a push, changes the current edge of v, or increases $d(v)$. The total time spent in *push/relabel* operations selecting v is thus $O(n\Delta_v)$ plus $O(1)$ per push out of v. Summing over all vertices and applying Lemmas 3.9 and 3.10, we obtain the theorem. □

Push/Relabel(v).
Applicability: v is active.
Action: Let $\{v, w\}$ be the current edge of v.
If *push*(v, w) is applicable **then** *push*(v, w)
else
if $\{v, w\}$ is not the last edge on the edge list of v **then**
replace $\{v, w\}$ as the current edge of v
by the next edge on the edge list of v
else begin
make the first edge on the edge list of v the current edge;
relabel(v);
end.

FIG. 3. The push/relabel operation.

Discharge.
Applicability: $Q \neq \emptyset$.
Action: Remove the vertex v on the front of Q.
(Vertex v must be active.)
Repeat
push/relabel(v);
if w becomes active during this *push/relabel* operation **then**
add w to the rear of Q;
until $e(v) = 0$ **or** $d(v)$ increases.
If v is still active **then** add v to the rear of Q.

FIG. 4. The discharge operation.

To obtain a better running time, we need to reduce the number of nonsaturating pushes. We do this in a way similar to that used by Shiloach and Vishkin [26]. Namely, we exploit the freedom we have in selecting vertices for *push/relabel* operations by using a first-in, first-out selection strategy; that is, we maintain Q as a queue. The *first-in, first-out algorithm* consists of applying the *discharge* operation until Q is empty. The discharge operation consists of removing the vertex on the front of Q, applying *push/relabel* operations to this vertex at least until the excess becomes zero or the label of the vertex increases, and adding any newly active vertices to the rear of Q (including v if it is still active).

There is still some flexibility in this algorithm, namely, in how long we keep applying *push/relabel* operations to a vertex v. Figure 4 describes one extreme case, in which we 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. In the sequential case, our analysis is valid for both extremes and all intermediate variants. In the parallel case, our analysis applies only to the version described in Figure 4, but it can be easily modified to handle other cases as well.

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 operations applied to the vertices added to the queue during the initialization. Given that pass i is defined, pass $i + 1$ consists of the discharging operations applied to vertices on the queue that were added during pass i.

LEMMA 4.3. *The number of passes over the queue is at most* $4n^2$.

PROOF. Let $\Phi = \max\{d(v) \mid v \text{ is active}\}$. Consider the effect on Φ of a single pass over the queue. If no distance label changes during the pass, each vertex has its excess moved to lower-labeled vertices, so Φ decreases during the pass. If Φ is not

changed by the pass, some vertex label must increase by at least 1. If Φ increases, some vertex label must increase by at least as much as Φ increases. The total number of passes in which Φ stays the same or increases is thus at most $2n^2$ by Lemma 3.7. Since $\Phi = 0$ initially and at the end of the algorithm, the total number of passes in which Φ decreases is also at most $2n^2$. Hence the total number of passes is at most $4n^2$. □

COROLLARY 4.4. *The number of nonsaturating pushes during the first-in, first-out algorithm is at most* $4n^3$.

PROOF. There is at most one nonsaturating push per vertex in $V - \{s, t\}$ per pass. □

THEOREM 4.5. *The first-in, first-out algorithm runs in* $O(n^3)$ *time.*

PROOF. Immediate from Theorem 4.2 and Corollary 4.4 □

An alternative strategy for vertex selection, which we call the *maximum-distance method*, is always to select a vertex v in Q with $d(v)$ maximum. This strategy also gives an $O(n^3)$ running time, as a proof similar to that of Theorem 4.5 shows. A third method giving an $O(n^3)$ bound is the *wave method*, described in [15] and [17].

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 operation below $O(1)$. We do this by using the dynamic tree data structure of Sleator and Tarjan [28–30]. This data structure allows us to maintain a set of vertex-disjoint rooted trees in which each vertex v has an associated real value $g(v)$, possibly ∞ or $-\infty$. We regard a tree edge as directed toward the root, that is, 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 described in Figure 5.

The total time for a sequence of l tree 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 [29] and [30] 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 form a subset of the current edges of the vertices. The current edge $\{v, w\}$ of a vertex $v \in V - \{s, t\}$ is *eligible* to be a dynamic tree edge (with $p(v) = w$) if $d(v) = d(w) + 1$ and $r_f(v, w) > 0$. Not all eligible edges are tree edges, however. The value $g(v)$ of a vertex v in its dynamic tree is $r_f(v, p(v))$ if v has a parent and ∞ 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 later.

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. By 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 to within a constant factor at $O(nm \log(n^2/m))$ for the choice $k = n^2/m$.

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 $g(v)$.

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

change-value(v, x): Add real number x to $g(w)$ for each ancestor 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.

FIG. 5. Dynamic tree operations.

Send(v).
Applicability: v is active.
Action:
While *find-root*(v) $\neq v$ **and** $e(v) > 0$ **do begin**
  send $\delta \leftarrow \min(e(v), \textit{find-value}(\textit{find-min}(v)))$ units of flow
    along the tree path from v by performing *change-value*(v, $-\delta$);
  while *find-value*(*find-min*(v)) $= 0$ **do begin**
    $u \leftarrow$ *find-min*(v);
    perform *cut*(u) followed by *change-value*(u, ∞);
  end;
end.

FIG. 6. The Send operation.

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 in Figure 6, which pushes excess from a nonroot vertex v to the root of its tree, cuts edges saturated by the push, and repeats these steps until $e(v) = 0$ or v is a tree root.

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 operations until Q is empty. However, we replace the *push/relabel* operation with the *tree-push/relabel* operation described in Figure 7.

A *tree-push/relabel* operation applies to an active vertex v that is the root of a dynamic tree. There are two main cases. The first case occurs if the current edge $\{v, w\}$ of v is eligible for a pushing operation. If the trees containing v and w together have at most k vertices, the *tree-push/relabel* operation links these trees by making w the parent of v and then does a send operation from v. If these trees together contain more than k vertices, *tree-push/relabel* does an ordinary pushing operation from v to w followed by a send from w. The second case occurs if the edge $\{v, w\}$ is not eligible for a pushing operation. In this case *tree-push/relabel* updates the current edge of v and relabels v if necessary. If v is relabeled, *tree-push/relabel* cuts all tree edges entering v, thereby maintaining the invariant that all dynamic tree edges are eligible for pushing operations.

It is important to realize that this algorithm stores values of the preflow f in two different ways. If $\{v, w\}$ is an edge that is not a dynamic tree edge, $f(v, w)$ is stored

Tree-Push/Relabel(v).
Applicability: v is an active tree root.
Action: Let $\{v, w\}$ be the current edge of v.
(1) If $d(v) = d(w) + 1$ **and** $r_f(v, w) > 0$ **then begin**
(1a) If *find-size*(v) + *find-size*(w) $\leq k$ **then begin**
make w the parent of v by performing
change-value($v, -\infty$), *change-value*($v, r_f(v, w)$), and *link*(v, w);
push excess from v to w by performing *send*(v);
end
(1b) else $\langle\langle$*find-size*(v) + *find-size*(w) $> k\rangle\rangle$ **begin**
apply a pushing operation to move excess from v to w;
perform *send*(w);
end;
end
(2) else $\langle\langle d(v) \leq d(w)$ **or** $r_f(v, w) = 0\rangle\rangle$
(2a) if $\{v, w\}$ is not the last edge on the edge list of v **then**
replace $\{v, w\}$ as the current edge by the next edge on the list
(2b) else $\langle\langle\{v, w\}$ is the last edge on the edge list of $v\rangle\rangle$ **begin**
make the first edge on the list the current one;
perform *cut*(u) and *change-value*(u, ∞) for every child u of v;
apply a relabeling operation to v;
end.

FIG. 7. The tree-push/relabel operation.

explicitly, with $\{v, w\}$. If $\{v, w\}$ is a dynamic tree edge, with w the parent of v, then $g(v) = c(v, w) - f(v, w)$ is stored implicitly in the dynamic tree data structure. Whenever a tree edge (v, w) is cut, $g(v)$ must be computed and $f(v, w)$ updated to its current value. In addition, when the algorithm terminates, flow values must be computed for all edges remaining in dynamic trees.

Two observations imply that the dynamic tree algorithm is correct. First, any edge $\{v, w\}$ that is in a dynamic tree has $d(v) = d(w) + 1$. Therefore in case (1a) of *tree-push/relabel*, vertices v and w are in different trees, and the algorithm never attempts to link a dynamic tree to itself. Second, a vertex v that is not a tree root can have positive excess only in the middle of case (1) of a *tree-push/relabel* operation. To see this, note that only in this case does the algorithm create an active vertex that is not a tree root, and this event is followed by a *send* operation that moves the nonroot excess to one or more roots.

LEMMA 5.1. *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* operation 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 $2nm$ by a proof like that of Lemma 3.9. (Another way of getting a bound on the number of cut and link operations is to observe that a cut operation corresponds to a saturating push or to an edge scan during relabeling, and the number of link operations exceeds the number of cut operations by at most $n - 1$.) The total number of *tree-push/relabel* operations 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 4.3 remains valid, which means that the number of passes is at most $4n^2$.

The next lemma is the crucial part of the analysis.

LEMMA 5.2. *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 $2n^2$ times, or as a result of $e(v)$ increasing from zero, which can happen only in subcases (1a) and (1b) of *tree-push/relabel*. In either subcase the number of vertices added to Q is at most one more than the number of cuts performed during the invocation of *send* in the subcase. Thus the number of additions to Q in subcases (1a) and (1b) is at most $2nm$ (the maximum number of cuts) plus the number of occurrences of the subcases. There are at most $2nm$ occurrences of (1a) (the maximum number of links). There are at most $2nm$ occurrences of (1b) in which the invocation of *send*(w) causes a cut and at most $2nm$ occurrences of (1b) 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 nonsaturating 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 $6nm$. (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 $4n^3/k$ over all passes.

Suppose, on the other hand, that T_w is large. The occurrence of (1b) adds the root of T_w, say r, to Q (otherwise this occurrence of (1b) is not counted). 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 $6nm$. If T_w has not changed, we charge the occurrence to T_w. The number of such charges over all passes is at most $4n^3/k$.

Summing our estimates, we find that there are at most $2n^2 + 20nm + 8n^3/k$ additions to Q altogether, giving the lemma. □

THEOREM 5.3. *The dynamic tree algorithm runs in $O(nm \log(n^2/m))$ time if k is chosen equal to n^2/m.*

PROOF. The proof is immediate from Lemmas 5.1 and 5.2. □

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

```
Procedure Pulse.
For all active vertices v in parallel do begin
  ⟨⟨stage 1⟩⟩
    push flow from v until e(v) = 0 or ∀w such that d(w) = d(v) − 1, r_f(v, w) = 0;
    ⟨⟨when pushing flow from v to w, reduce e(v) but do not increase e(w)⟩⟩
  ⟨⟨stage 2⟩⟩
    if e(v) > 0 then begin
      d'(v) ← min{d(w) + 1 | r_f(v, w) > 0};
  ⟨⟨stage 3⟩⟩
      if d(v) ≠ d'(v) then begin
        d(v) ← d'(v);
        broadcast d(v) to all neighbors of v;
      end;
    end;
  ⟨⟨stage 4⟩⟩
    add flow pushed to v in stage 1 to e(v).
end.
```

FIG. 8. The Pulse operation.

6. *Parallel and Distributed Implementation*

The synchronous parallel version of our algorithm is a modification of the first-in, first-out algorithm of Section 4. The algorithm proceeds in pulses, each of which consists of a number of operations applied in parallel. Each pulse is divided into four stages. Pushing of flow is done during the first stage, relabeling of vertices is done in the second stage, broadcasting of new labels is done in the third stage, and flow pushed to a vertex in the first stage is added to its excess in the fourth stage. 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 v during a pulse is not added to $e(v)$ until the fourth stage. To be more precise, the parallel version consists of repeating the *pulse* operation described in Figure 8 until there are no active vertices.

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, these computations 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 4.3 gives the following analogous result for the parallel algorithm:

LEMMA 6.1. *The number of pulses made by the parallel algorithm is at most* $4n^2$.

COROLLARY 6.2. *The number of nonsaturating pushes made by the parallel algorithm is at most* $4n^3$.

For the distributed implementation of this algorithm, our computing model is as follows [2, 13]. 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 interprocessor 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.

A synchronous distributed implementation of the parallel algorithm works as follows: Each vertex processed during a pulse sends updated flow values to the appropriate neighbors. New vertex labels are also transmitted to neighbors, but after flow pushing. Since flow always travels in the direction from larger to smaller labels, this delaying of the label broadcasting 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.

For parallel implementation, our computing model is an exclusive-read, exclusive-write parallel random-access machine (PRAM) [9] or a distributed random-access machine (DRAM) [20]. 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 [26] apply to our algorithm to show that $O(n)$ processors suffice to obtain the $O(n^2 \log n)$ time bound. See [26] for details. A different parallel implementation of the algorithm that uses parallel prefix operations as primitives is described in [15].

Now we discuss two implementations of the algorithm in the asynchronous distributed model of parallel computation [2, 13]. Awerbuch (private communication, 1985) has observed that in the asynchronous case, the synchronization protocol of [2] can be used to implement our algorithm in $O(n^2 \log n)$ rounds and $O(n^3)$ messages. The same bounds can be obtained for the Shiloach–Vishkin algorithm [26] but only by allowing more memory per processor: The processor at a vertex v needs $O(n\Delta_v)$ storage. Vishkin (private communication, 1986) 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 in which memory is at a premium.

Our algorithm can be modified to work in the asynchronous model without the use of the synchronization protocol, achieving a better running time but using more messages. This asynchronous version of the algorithm synchronizes locally using *acknowledgments*. When a vertex v pushes flow to a vertex w such that, according to the local information at v, $d(v) = d(w) + 1$, it sends a message $(v, \delta, d(v))$ and updates $e(v)$. The vertex v will not push flow to w again or change $d(v)$ until v receives an acknowledgment from w. When a vertex w receives a message $(v, \delta, d(v))$, it first checks if $d(v) = d(w) + 1$ (because the value of $d(w)$ in the processor v at the time it sent the message may be out of date). If $d(v) = d(w) + 1$, then w sends to v a message of the form (accept, w, δ, $d(w)$). Otherwise, it sends to v a message of the form (reject, w, δ, $d(w)$), where $d(w)$ is the correct value of the distance label of w. The accepting or rejecting messages serve as acknowledgments. In addition, a rejecting message causes v to update its excess, its local value of $d(w)$, and $d(v)$ if necessary. When a distance label of a vertex increases, it informs its neighbors about the new value of the label.

THEOREM 6.3. *The asynchronous distributed implementation of the algorithm that uses acknowledgments runs in $O(n^2)$ time using $O(n^2 m)$ messages and $O(\Delta_v)$ memory per processor.*

PROOF. To analyze the message complexity of the algorithm, note that the total number of messages is the number of messages generated by the distance label increases plus twice the number of (accepting or rejecting) acknowledge messages.

The number of messages generated by the distance label increases is at most $2nm$. There is at most one rejecting message per edge per distance label increase, for a total of $2nm$ (by Lemma 3.7). The same arguments as in the proofs of Lemmas 3.9 and 3.10 give $2nm$ and $4n^2m$ bounds on the number of accepting messages corresponding to saturating and nonsaturating pushes. The total message complexity of the algorithm is thus $O(n^2m)$.

To bound the running time of the algorithm, we need to introduce a unit of time. Given an execution of an algorithm, we define a time unit to be the longest interval from the time when a message is originated by a sender to the time when the message is processed by the receiver. For example, a push–acknowledgment pair of operations takes two units of time. Note that if during a time interval $(t, t + 4]$ no vertex label increases, then the Φ function defined as in the proof of Lemma 4.3 must decrease during this time interval. To see this, observe that during the time interval $[t + 1, t + 4)$ each vertex has the correct information about the distance labels of its neighbors, so all pushes initiated during the time interval $[t + 1, t + 2)$ are accepted by time $t + 4$. A proof similar to that of Lemma 4.3 yields an $O(n^2)$ bound on the running time of the algorithm. □

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* operations 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. Recently, Ahuja and Orlin [1] devised a scaling algorithm based on the approach described in this paper. Their algorithm runs in $O(nm + n^2 \log U)$ time, assuming that the edge capacities are integers not exceeding U. This improves Gabow's bound of $O(nm \log U)$ mentioned in the introduction. 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 the use of a parallel version of the dynamic tree data structure.

It is possible to modify our algorithm so that when a pushing operation is executed, each distance label corresponds exactly to the distance to the sink or to the source in the residual graph (i.e., if $d(v) < n$, then $d(v) = d_{G_f}(v, t)$; if $d(v) \geq n$, then $d(v) = d_{G_f}(v, s) + n$). The modification involves a stronger interpretation of the current 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 in total during the algorithm.

Whether maintaining exact distance labels improves the practical performance of the algorithm is not clear, because the work of maintaining the exact labels may exceed the extra work due to nonexact labels. The above observation, however, 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 for free.

Our algorithm is practical [15, 22]. We offer a heuristic that may make the algorithm even faster. The heuristic periodically updates the distance labels by performing breadth-first searches backward from the sink and source in the residual graph. These searches compute, for each vertex v, the distances $d_{G_f}(v, s)$ and $d_{G_f}(v, t)$. The new distance label of v is set to $\min(d_{G_f}(v, s) + n, d_{G_f}(v, t))$. There are several possible strategies for deciding when to recompute labels. One is to do so after every n relabeling operations. Another is to do so every time an edge into the sink is saturated or an edge out of the source has its flow reduced to zero. Neither of these strategies affects the asymptotic time bound of the algorithm, but they may improve its practical performance.

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 selection, maximum-distance selection, and the wave method [15, 17], other strategies, such as last-in, first-out selection and maximum-excess selection, deserve consideration. The Ahuja–Orlin algorithm [1], for example, uses a variant of the idea of always selecting a vertex of largest excess.

Appendix: Finding Sizes of Dynamic Trees

The dynamic tree implementation of [29] and [30] does not support *find-size* operations but can be easily modified to do so as follows: We assume some familiarity with [29] or [30]. 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 vertex the size of its virtual subtree. 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.

Notes Added in Proof. The conference publication of the early version of this paper [16] prompted much related work, and several improvements in the results mentioned in the paper have been obtained. Cheriyan and Maheshwari [2a] proved that the maximum-distance method defined at the end of Section 4 has a running time of $O(n^2m^{1/2})$. Ahuja et al. [1a] improved the Ahuja–Orlin algorithm and incorporated the dynamic tree data structure, thereby obtaining a running time of $O(nm \log((n/m) \log U + 2))$ Ahuja and Orlin (private communication, 1987) observed that the running time of the parallel algorithm presented in Section 6 can be reduced to $O(n^2\log((m/n) + 2))$ by transforming the problem graph so that the maximum vertex degree is $O(m/n)$; this increases the number of vertices by only a constant factor.

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

REFERENCES

1. AHUJA, R. K. AND ORLIN, J. B. A fast and simple algorithm for the maximum flow problem. Tech. Rep. 1905–87, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Mass., 1987.

1a. AHUJA, R. K., ORLIN, J. B., AND TARJAN, R. E. Improved time bounds for the maximum flow problem. Unpublished mansucript.

2. AWERBUCH, B. Complexity of network synchronization. *J. ACM 32*, 4 (Oct. 1985), 804–823.

2a. CHERIYAN, J., AND MAHESHWARI, S. N. Analysis of preflow push algorithms for maximum network flow. Department of Computer Science and Engineering, Indian Institute of Tech., New Delhi, India, 1987.

3. CHERKASKY, R. V. An algorithm for constructing maximal flows in networks with complexity of $O(V^2\sqrt{E})$ operations. *Math. Methods Solution Econ. Probl. 7* (1977), 112–125. (In Russian.)

4. DINIC, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Sov. Math. Dokl. 11* (1970), 1277–1280.

5. EDMONDS, J., AND KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19*, 2 (Apr. 1972), 248–264.

6. EVEN, S. *Graph Algorithms*. Computer Science Press, Potomac, Md., 1979.

7. FORD, L. R., JR., AND FULKERSON, D. R. Maximal flow through a network. *Can. J. Math. 8* (1956), 399–404.

8. FORD, L. R., JR., AND FULKERSON, D. R. *Flows in Networks*. Princeton University Press, Princeton, N.J., 1962.

9. FORTUNE, S., AND WYLLIE, J. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing* (1978), pp. 114–118.

10. GABOW, H. N. Scaling algorithms for network problems. *J. Comput. Syst. Sci. 31* (1985), 148–168.

11. GALIL, Z. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Inf. 14* (1980), 221–242.

12. GALIL, Z., AND NAAMAD, A. An $O(EV \log^2 V)$ algorithm for the maximal flow problem. *J. Comput. Syst. Sci.* 21 (1980), 203–217.

13. GALLAGER, R. G., HUMBLET, P. A., AND SPIRA, P. M. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 66–77.

14. GOLDBERG, A. V. A new max-flow algorithm. Tech. Rep. MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1985.

15. GOLDBERG, A. V. Efficient graph algorithms for sequential and parallel computers. PhD dissertation, Massachusetts Institute of Technology, Cambridge, Mass., Jan. 1987. Also available as Tech. Rep. TR-374, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1987.

16. GOLDBERG, A. V., AND TARJAN, R. E. A new approach to the maximum flow problem. In *Proceedings of the 18th ACM Symposium on Theory of Computing*. ACM, New York, 1986, pp. 136–146.

17. GOLDBERG, A. V., AND TARJAN, R. E. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, to appear.

18. KARZANOV, A. V. Determining the maximal flow in a network by the method of preflows. *Sov. Math. Dokl. 15* (1974), 434–437.

19. LAWLER, E. L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.

20. LEISERSON, C., AND MAGGS, B. Communication-efficient parallel graph algorithms. In *Proceedings of the International Conference on Parallel Processing*. IEEE Computer Society Press, Silver Spring, Md., 1986, pp. 861–868.

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

22. OGIELSKI, A. T. Integer optimization and zero-temperature fixed point in Ising random-field systems. *Phys. Rev. Lett. 57* (1986), 1251–1254.

23. PAPADIMITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N.J., 1982.

24. PICARD, J. C., AND RATLIFF, H. D. Minimum cuts and related problems. *Networks 5* (1975), 357–370.

25. SHILOACH, Y. An $O(nI \log^2 I)$ maximum-flow algorithm. Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, Calif., 1978.

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

27. SLEATOR, D. D. An $O(nm \log n)$ algorithm for maximum network flow. Tech. Rep. STAN-CS-80-831, Computer Science Dept., Stanford Univ., Stanford, Calif., 1980.
28. SLEATOR, D. D., AND TARJAN, R. E. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26* (1983), 362–391.
29. SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.
30. TARJAN, R. E. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.
31. TARJAN, R. E. A simple version of Karzanov's blocking flow algorithm. *Oper. Res. Lett. 2* (1984), 265–268.

RECEIVED JUNE 1986; REVISED MARCH AND AUGUST 1987; ACCEPTED AUGUST 1987