A FAST AND SIMPLE ALGORITHM FOR THE MAXIMUM FLOW PROBLEM

R. K. AHUJA

Massachusetts Institute of Technology, Cambridge, Massachusetts and Indian Institute of Technology, Kanpur, India

JAMES B. ORLIN

Massachusetts Institute of Technology, Cambridge, Massachusetts

(Received June 1987; revisions received February, October 1988; accepted June 1988)

We present a simple sequential algorithm for the maximum flow problem on a network with n nodes, m arcs, and integer arc capacities bounded by U. Under the practical assumption that U is polynomially bounded in n, our algorithm runs in time $O(nm + n^2 \log n)$. This result improves the previous best bound of $O(nm \log(n^2/m))$, obtained by Goldberg and Tarjan, by a factor of $\log n$ for networks that are both nonsparse and nondense without using any complex data structures. We also describe a parallel implementation of the algorithm that runs in $O(n^2 \log U \log p)$ time in the PRAM model with EREW and uses only p processors where $p = \lceil m/n \rceil$.

The maximum flow problem is one of the most fundamental problems in network flow theory and has been investigated extensively. This problem was first formulated by Fulkerson and Dantzig (1955) and Dantzig and Fulkerson (1956), and solved by Ford and Fulkerson (1956) using their well known augmenting path algorithm. Since then, a number of algorithms have been developed for this problem; some of them are listed in Table I. In the table, n is the number of nodes, m is the number of arcs, and U is an upper bound on the integral arc capacities. The algorithms whose time bounds involve U assume integral capacities, whereas others run on arbitrary rational or real capacities.

Edmonds and Karp (1972) showed that the Ford and Fulkerson algorithm runs in time $O(nm^2)$ if flows are augmented along shortest paths from source to sink. Independently, Dinic (1970) introduced the concept of shortest path networks, called *layered* networks, and obtained an $O(n^2m)$ algorithm. This bound was improved to $O(n^3)$ by Karzanov (1974), who introduced the concept of *preflows* in a layered network. A *preflow* is similar to a flow except that the amount flowing into a node may exceed the amount flowing out of a node. Since then, researchers have improved the complexity of Dinic's algorithm for sparse networks by devising sophisticated data structures. Among these contributions, Sleator and Tarjan's (1983) dynamic tree data structure is the most attractive from a worst-case point of view.

The algorithms of Goldberg (1985) and of Goldberg and Tarjan (1986) are a novel departure from these approaches in that they do not construct layered networks. Their method maintains a preflow, as per Karzanov, and proceeds by pushing flows to nodes estimated to be closer to the sink. To estimate which nodes are closer to the sink, it maintains a distance label for each node that is a lower bound on the length of a shortest augmenting path to the sink. Distance labels are a better computational device than layered networks because the distance labels are simpler to understand, easier to manipulate, and easier to use in a parallel algorithm. Moreover, by cleverly using the dynamic tree data structure, Goldberg and Tarjan obtain the best computational complexity for sparse as well as dense networks.

For problems with arc capacities polynomially bounded in n, our maximum flow algorithm is an improvement of Goldberg and Tarjan's algorithm and uses concepts of scaling introduced by Edmonds and Karp for the minimum cost flow problem and later extended by Gabow (1985) for other network optimization problems. The bottleneck operation in the straightforward implementation of Goldberg and Tarjan's algorithm is the number of *nonsaturating pushes*, which is $O(n^3)$. However, they reduce the computational time to $O(nm \log(n^2/m))$ by a clever application of the dynamic tree data structure. We show that the number of nonsaturating pushes can be reduced to $O(n^2 \log U)$ by using *excess scaling*. Our

Subject classification: Networks/graphs, flow algorithms: fast and simple algorithm for the maximum flow problem.

0030-364X/89/3705-0748 $01.25

Table I
Running Times of the Maximum Flow Algorithms

No.	Due to	Running Time
1	Ford and Fulkerson (1956)	$O(nm\ U)$
2	Edmonds and Karp (1972)	$O(nm^2)$
3	Dinic (1970)	$O(n^2m)$
4	Karzanov (1974)	$O(n^3)$
5	Cherkasky (1977)	$O(n^2m^{1/2})$
6	Malhotra, Kumar and Maheshwari (1978)	$O(n^3)$
7	Galil (1980)	$O(n^{5/3}m^{2/3})$
8	Galil and Naamad (1980); Shiloach (1978)	$O(nm \log^2 n)$
9	Shiloach and Vishkin (1982)	$O(n^3)$
10	Sleator and Tarjan (1983)	$O(nm \log n)$
11	Tarjan (1984)	$O(n^3)$
12	Gabow (1985)	$O(nm \log U)$
13	Goldberg (1985)	$O(n^3)$
14	Goldberg and Tarjan (1986)	$O(nm \log (n^2/m))$
15	Bertsekas (1986)	$O(n^3)$
16	Cheriyan and Maheshwari (1988)	$O(n^2m^{1/2})$
17	Ahuja and Orlin (this paper)	$O(nm + n^2 \log U)$
18	Ahuja, Orlin and Tarjan (1988)	(a) $O\left(nm + \frac{n^2 \log U}{\log \log U}\right)$ (b) $O(nm + n^2 \sqrt{\log U})$ (c) $O\left(nm \log\left(\frac{n\sqrt{\log U}}{m} + 2\right)\right)$

algorithm modifies the Goldberg-Tarjan algorithm as follows. It performs log U scaling iterations; each scaling iteration requires $O(n^2)$ nonsaturating pushes if we push flows from nodes with *sufficiently large* excesses to nodes with *sufficiently small* excesses while never allowing the excesses to become *too large.* The computational time of our algorithm is $O(nm + n^2 \log U)$.

Under the reasonable assumption that $U = O(n^k)$ for some k, our algorithm runs in time $O(nm + n^2 \log n)$. On networks that are both nondense and nonsparse, i.e., $m = \theta(n^{1+\varepsilon})$ for some ε with $0 < \varepsilon < 1$, our algorithm runs in time $O(nm)$, which improves Goldberg and Tarjan's bound of $O(nm \log(n^2/m))$ on such networks by a factor of log n. Moreover, our algorithm is easier to implement and should be more efficient in practice, because it requires only elementary data structures with little computational overheads.

This paper also describes a parallel implementation of our maximum flow algorithm. Our algorithm is difficult to make *massively parallel* because the algorithm exploits the fact that it pushes flow from one node at a time. Nevertheless, in the PRAM (Parallel Random Access Machine) model with EREW (Exclusive Read Exclusive Write) our algorithm runs in $O(n^2 \log U \log p)$ time and uses only p processors where $p = \lceil m/n \rceil$. This algorithm easily extends to an $O(((nm/k) + n^2 \log U) \log k)$ algorithm using $2 \leq k \leq p$ processors. The existing parallel algorithms due to Shiloach and Vishkin (1982) and Goldberg and Tarjan (1986) run in $O(n^2 \log n)$ time and use n processors. For $k \leq n$ processors, their algorithms run in $O((n^3/k) \log k)$ time (personal communication with Andrew Goldberg). For $k \leq p/\log U$, our algorithm runs in $O((nm/k) \log k)$ time and provides a speedup of n^2/m over the existing algorithms.

1. NOTATION

Let $G = (N, A)$ be a directed network with a positive integer capacity u_{ij} for every arc$(i, j) \in A$. Let $n = |N|$ and $m = |A|$. The source s and sink t are two distinguished nodes of the network. It is assumed that for every arc$(i, j) \in A$, an arc(j, i) is also contained in A, possibly with zero capacity. We assume without loss of generality that the network does not contain multiple arcs and that capacities of arcs directed into the source or directed out from the sink are zero. We further assume that none of the paths from the source to the sink has infinite capacity as such a path can be detected easily in $O(m)$ time. Observe that if the network contains some infinite capacity arcs but no infinite capacity path, then the capacity of such arcs can be replaced by $\sum_{\{(i,j) \in A: u_{ij} < \infty\}} u_{ij}$. We, therefore, assume that all arcs have finite capacity. Let $U = \max_{(s,j) \in A}\{u_{sj}\}$.

A *flow* is a function $x: A \rightarrow R$ satisfying

$$\sum_{\{j:(j,i) \in A\}} x_{ji} - \sum_{\{j:(i,j) \in A\}} x_{ij} = 0 \quad \text{for all } i \in N - \{s, t\} \tag{1}$$

$$\sum_{\{j:(j,t) \in A\}} x_{jt} = v \tag{2}$$

$$0 \leq x_{ij} \leq u_{ij} \quad \text{for all } (i, j) \in A \tag{3}$$

for some $v \geq 0$. The maximum flow problem is to determine a flow x for which v is maximized.

A *preflow* x is a function $x: A \rightarrow R$ that satisfies (2), (3), and the following relaxation of (1)

$$\sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} \geq 0$$

for all $i \in N - \{s, t\}$. (4)

The algorithms described in this paper maintain a preflow at each intermediate stage.

For a given preflow x, we define for each node $i \in N - \{s, t\}$, the *excess*

$$e_i = \sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij}.$$

A node with positive excess is referred to as an *active node.* We define the excesses of the source and sink nodes to be zero; consequently, these nodes are never active. The *residual capacity* of any arc $(i, j) \in A$, with respect to a given preflow x, is given by $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. The residual capacity of arc(i, j) represents the maximum additional flow that can be sent from node i to node j using the arcs (i, j) and (j, i). The network that consists only of arcs with positive residual capacities is referred to as the *residual network.* Figure 1 illustrates these definitions.

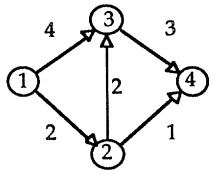
We define the *arc adjacency list* $A(i)$ of a node $i \in N$ as the set of arcs directed out of the node i, i.e., $A(i) := \{(i, k) \in A : k \in N\}$. Note that our adjacency list is a set of arcs rather than the more conventional definition of the list as a set of nodes.

A *distance function* d: $N \rightarrow Z^+$ for a preflow x is a function from the set of nodes to the nonnegative integers. We say that a distance function d is *valid* if it also satisfies the following two conditions:

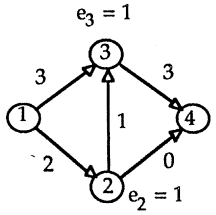
C1. $d(t) = 0$;

C2. $d(i) \leq d(j) + 1$ for every arc $(i, j) \in A$ with $r_{ij} > 0$.

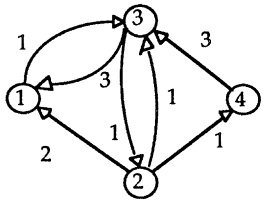
Our algorithm maintains a valid distance function at each iteration. We also refer to $d(i)$ as the *distance label* of node i. It is easy to demonstrate by induction that $d(i)$ is a lower bound on the length of the shortest path from i to t in the residual network. Let $i = i_1 - i_2 - \ldots - i_k - i_{k+1} = t$ be any path of length k in the residual network from node i to the sink. Then from condition **C2** we have, $d(i) = d(i_1) \leq d(i_2) + 1$, $d(i_2) \leq d(i_3) + 1, \ldots, d(i_k) \leq d(i_{k+1}) + 1 = 1$. This yields $d(i) \leq k$ for *any* path of length k in the residual network and, hence, must also hold for the shortest path. If for each i, the distance label $d(i)$ *equals* the minimum length of any path from i to t in the residual network, then we call the distance label *exact.* For example, in Figure 1c, $d = (0, 0, 0, 0)$ is a valid distance label, though $d = (3, 1, 2, 0)$ represents the exact distance label.



a. Network with arc capacities. Node 1 is the source and node 4 is sink. (Arcs with zero capacities are not shown.)



b. Network with a preflow x



c. The residual network with residual arc capacities

Figure 1. Illustrations of a preflow and the residual network.

An arc (i, j) in the residual network is called *admissible* if it satisfies $d(i) = d(j) + 1$. An arc that is not admissible is called an *inadmissible* arc. The algorithms discussed in this paper push flow only on admissible arcs. Lastly, all algorithms in this paper are assumed to be of base 2 unless stated otherwise.

2. PREFLOW-PUSH ALGORITHMS

The *preflow-push algorithms* for the maximum flow problem maintain a preflow at every step and proceed by pushing the node excesses closer to the sink. The first preflow-push algorithm is due to Karzanov. Tarjan (1984) has suggested a simplified version of this algorithm. The recent algorithms of Goldberg (1985) and Goldberg and Tarjan (1986) are based on ideas similar to those presented in Tarjan, but they

use distance labels to direct flows closer to the sink instead of constructing layered networks. We refer to their algorithm as the (*distance-directed*) *preflow-push* algorithm. In this section, we review the basic features of their algorithm, which for the sake of brevity, we simply refer to as the preflow-push algorithm. Here we describe the 1-phase version of the preflow-push algorithm presented by Goldberg (1987). The results in this section are due to Goldberg and Tarjan (1986).

All operations of the preflow-push algorithm are performed using only local information. At each iteration of the algorithm (except at the initialization and at the termination) the network contains at least one active node, i.e., a nonsource and nonsink node with positive excess. The goal of each iterative step is to choose some active node and to send its excess *closer* to the sink, with closer being judged with respect to the current distance labels. If excess at this node cannot be sent to nodes with smaller distance labels, then the distance label of the node is increased. The algorithm terminates when the network contains no active nodes. The preflow-push algorithm uses the following subroutines:

PREPROCESS. On each arc $(s, j) \in A(s)$, send u_{sj} units of flow. Let $d(s) = n$ and $d(t) = 0$. Let $d(i) = 1$ for each $i \neq s$ or t. (Alternatively, any valid labeling can be used, e.g., the distance label for each node $i \neq s, t$ can be determined by a backward breadth first search on the residual network starting at node t.)

PUSH(i). Select an admissible arc (i, j) in $A(i)$. Send $\delta = \min\{e_i, r_{ij}\}$ units of flow from node i to j.

We say that a push of flow on arc (i, j) is *saturating* if $\delta = r_{ij}$, and *nonsaturating* otherwise.

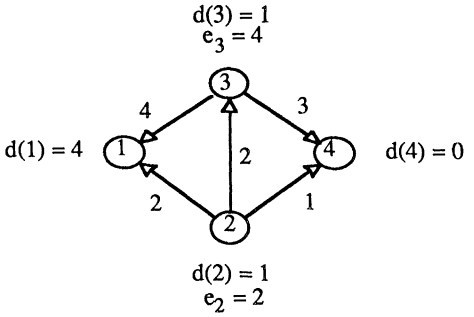
RELABEL(i). Replace $d(i)$ by $\min\{d(j) + 1 : (i, j) \in A(i)$ and $r_{ij} > 0\}$.

This is called a *relabel* step. The result of the relabel step is to create at least one admissible arc on which further pushes can be performed.

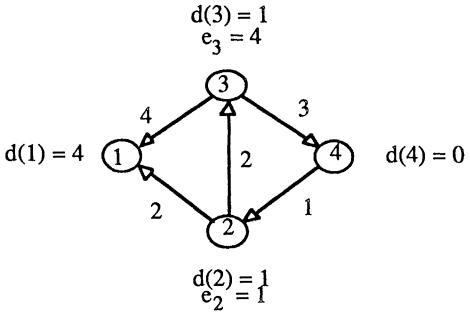
The generic version of the preflow-push algorithm is given below.

```
algorithm PREFLOW-PUSH;
begin
  PREPROCESS;
  while there is an active node do
  begin
    select an active node i;
    if there is an admissible arc in A(i) then PUSH(i)
    else RELABEL(i);
  end;
end.
```

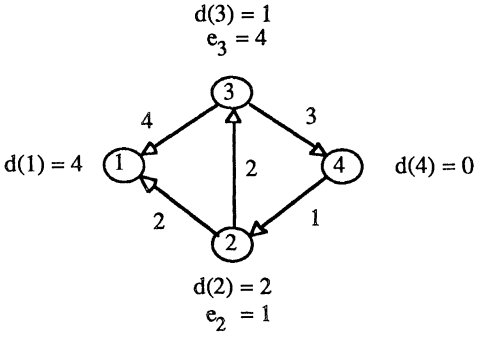
Figure 2 illustrates the steps PUSH(i) and RELABEL(i) as applied to the network in Figure 1a. The number beside each arc represents its residual capacity. Figure 2a specifies the residual network after the PREPROCESS step. Suppose the algorithm selects node 2 for examination. Since arc (2, 4) has a residual capacity $r_{24} = 1$ and $d(2) = d(4) + 1$, the algorithm performs a saturating push of value $\delta = \min\{2, 1\}$ units. The push reduces the excess of node 2 to 1. Arc (2, 4) is deleted from the residual network and



(a) The residual network after the pre-processing step.



(b) After the execution of step PUSH(2).



(c) After the execution of step RELABEL(2).

Figure 2. Illustrations of push and relabel steps.

arc (4, 2) is added to the residual network. Since node 2 is still an active node, it can be selected again for further pushes. Arcs (2, 3) and (2, 1) have positive residual capacities, but they do not satisfy the distance condition. Hence, the algorithm performs RELABEL(2), and gives node 2 a new distance $d'(2) = \min\{d(3) + 1, d(1) + 1\} = \min\{2, 5\} = 2$.

The preprocess step accomplishes several important tasks. First, it causes the nodes adjacent to s to have positive excess, so that subsequently we can select nodes for push or relabel steps. Second, by saturating arcs incident to s, the feasibility of setting $d(s) = n$ is immediate. Third, since the distance label $d(s) = n$ is a lower bound on the length of the minimum path from s to t, there is no path from s to t after the PREPROCESS step. Furthermore, since distance labels are nondecreasing (see Lemma 1), we also are guaranteed that in subsequent iterations the residual network will never contain a directed path from s to t, and so there can never be any need to push flow from s again.

In our improvement of the preflow-push algorithm, we need a few of the results given in Goldberg and Tarjan (1986). We include some of their proofs in order to make this presentation more self-contained. The interested reader is encouraged to read the original paper by Goldberg and Tarjan which discusses other versions of preflow-push algorithms.

Lemma 1. *The generic preflow-push algorithm maintains valid distance labels at each step. Moreover, at each relabel step the distance label of some node strictly increases.*

Proof. First note that the preprocess step constructs valid distance labels. Assume inductively that the distance function is valid prior to an operation, i.e., it satisfies the validity conditions **C1** and **C2**. A push operation on the arc (i, j) may create an additional arc (j, i) with $r_{ji} > 0$, and an additional condition $d(j) \leq d(i) + 1$ needs to be satisfied. This validity condition remains satisfied since $d(i) = d(j) + 1$ by the property of the push operation. A push operation on arc (i, j) might delete this arc from the residual network, but this does not affect the validity of the distance function. During a relabel step, the new distance label of node i is $d'(i) = \min\{d(j) + 1: (i, j) \in A(i)$ and $r_{ij} > 0\}$, which again is consistent with the validity conditions. The relabel step is performed when there is no arc $(i, j) \in A(i)$ with $d(i) = d(j) + 1$ and $r_{ij} > 0$. Hence, $d(i) < \min\{d(j) + 1: (i, j) \in A(i)$ and $r_{ij} > 0\} = d'(i)$, thereby proving the second part of the lemma.

Lemma 2. *At any stage of the preflow-push algorithm, for each node i with positive excess, there is a directed path from node i to node s in the residual network.*

Proof. By the flow decomposition theory of Ford and Fulkerson, any preflow x can be decomposed with respect to the original network G into the sum of nonnegative flows along: i) paths from s to t, ii) paths from s to active nodes, and iii) flows around directed cycles. Let i be an active node relative to the preflow x in G. Then, there must be a path P from s to i in the flow decomposition of x because paths from s to t and flows around cycles do not contribute to the excess at node i. Then the reversal of P (P with the orientation of each arc reversed) is in the residual network, and hence, there is a path from i to s in the residual network.

Corollary 1. *For each node $i \in N$, $d(i) < 2n$.*

Proof. The last time node i was relabeled, it had a positive excess, and hence, the residual network contained a path of length at most $n - 1$ from i to s. The fact that $d(s) = n$ and condition **C2** imply that $d(i) \leq d(s) + n - 1 < 2n$.

Lemma 2 also implies that a relabel step never minimizes over an empty set.

Corollary 2. *The number of relabel steps is less than $2n^2$.*

Proof. Each relabel step increases the distance label of a node by at least one, and by Corollary 1 no node can be relabeled more than $2n$ times.

Corollary 3. *The number of saturating pushes is no more than nm.*

Proof. Suppose that arc (i, j) becomes saturated at some iteration (at which time $d(i) = d(j) + 1$). Then no more flow can be sent on (i, j) until flow is sent back from j to i, at which time $d'(j) = d'(i) + 1 \geq d(i) + 1 = d(j) + 2$; this flow change cannot occur until $d(j)$ increases by at least 2. Thus by Corollary 1, arc (i, j) can become saturated at most n times, and the total number of arc saturations is no more than nm. (Recall that we assume that (i, j) and (j, i) are both in A, so the number of arcs in the residual network is no more than m.)

Lemma 3. *The number of nonsaturating pushes is at most $2n^2m$.*

Proof. See Goldberg and Tarjan (1986).

Lemma 4. *The algorithm terminates with a maximum flow.*

Proof. When the algorithm terminates, each node in $N - \{s, t\}$ has zero excess; so the final preflow is a feasible flow. Furthermore, since the distance labels satisfy conditions **C1** and **C2** and $d(s) = n$, it follows that upon termination, the residual network contains no directed path from s to t. This condition is the classical termination criterion for the maximum flow algorithm of Ford and Fulkerson.

The bottleneck operation in many preflow-based algorithms, such as the algorithms due to Karzanov; Tarjan; and Goldberg and Tarjan (1986), is the number of nonsaturating pushes. A partial explanation of why the number of nonsaturating pushes dominates the number of saturating pushes is as follows: The saturating pushes cause structural changes—they delete saturated arcs from the residual network. This observation leads to a bound of $O(nm)$ on the number of saturating pushes—no matter in which order they are performed. The nonsaturating pushes do not change the structure of the residual network and seem more difficult to bound. Goldberg (1985) showed that the number of nonsaturating pushes is $O(n^3)$ when nodes are examined in a first-in first-out order. Goldberg and Tarjan (1986) reduced the running time of this algorithm by using dynamic trees to reduce the average time per nonsaturating push. Cheriyan and Maheshwari (1988) showed that the number of nonsaturating pushes can be decreased to $O(n^2m^{1/2})$ if flow is always pushed from a node with a highest distance label, and they showed that this bound is tight. They also showed that the bounds $O(n^2m)$ and $O(n^3)$ are, respectively, tight for the generic preflow-push and first-in first-out preflow-push algorithms. In the next section, we show that by using scaling, we can dramatically reduce the number of nonsaturating pushes to $O(n^2 \log U)$. We recently discovered new scaling algorithms that further reduce the number of nonsaturating pushes to $O(n^2 \log U/\log\log U)$ or to $O(n^2\sqrt{\log U})$. These results are presented in Ahuja, Orlin and Tarjan (1988).

3. THE EXCESS SCALING ALGORITHM

Our maximum flow algorithm improves the generic preflow-push algorithm of Section 2 by using *excess scaling* to reduce the number of nonsaturating pushes from $O(n^2m)$ to $O(n^2 \log U)$. The basic idea is to push flow from active nodes with *sufficiently large* excesses to nodes with *sufficiently small* excesses while never letting the excesses become *too large.* We refer to our algorithm as the *excess scaling algorithm.*

The algorithm performs $K = \lceil \log U \rceil + 1$ scaling iterations. For a scaling iteration, the *excess-dominator* is defined to be the least integer Δ that is a power of 2 and satisfies $e_i \leq \Delta$ for all $i \in N$. Furthermore, a new scaling iteration is considered to have begun whenever Δ decreases by a factor of 2. In a scaling iteration, we guarantee that each nonsaturating push sends at least $\Delta/2$ units of flow and that the excess-dominator does not increase. To ensure that each nonsaturating push has a value of at least $\Delta/2$, we consider only nodes with an excess more than $\Delta/2$; and among these nodes with large excess, we select a node with a minimum distance label. This choice ensures that the flow will be sent to a node with a small excess. We show that after at most $8n^2$ nonsaturating pushes, the excess-dominator decreases by a factor of at least 2, and a new scaling iteration begins. After at most K scaling iterations, all node excesses drop to zero and we obtain a maximum flow.

In order to select an active node with excess more than $\Delta/2$ and with a minimum distance label among such nodes, we maintain the lists $\text{LIST}(r) = \{i \in N: e_i > \Delta/2 \text{ and } d(i) = r\}$ for each $r = 1, \ldots, 2n - 1$. These lists can be maintained in the form of either linked stacks or queues (see, for example, Aho, Hopcroft and Ullman 1974), which enables insertion and deletion of elements in $O(1)$ time. The variable *level* represents a lower bound on the smallest index r for which $\text{LIST}(r)$ is nonempty.

As per Goldberg and Tarjan, we use the following data structure to efficiently select the admissible arc for pushing flow out of a node. We maintain with each node i the list, $A(i)$, of arcs directed out of it. Arcs in each list can be arranged arbitrarily, but once the order is decided, it remains unchanged throughout the algorithm. A special arc named *null* is appended to the end of each list. Each node i has a *current arc* (i, j), which is the current candidate for pushing flow out of i. Initially, the current arc of node i is the first arc in its arc list. This list is examined sequentially, and whenever the current arc is found to be inadmissible for pushing flow, the next arc in the arc list is made the current arc. When the arc list has been examined completely, the null arc is reached. At this time, the node is relabeled and the current arc is again set to the first arc in the arc list.

The algorithm can be formally described as follows.

```
Algorithm MAX-FLOW;
begin
  PREPROCESS;
  K := 1 + ⌈log U⌉;
  for k = 1 to K do
  begin
    Δ = 2^(K−k)
    for each i ∈ N do if e_i > Δ/2 then add i to
        LIST(d(i));
    level := 1;
    while level < 2n do
      if LIST(level) = ∅ then level := level + 1
      else
      begin
        select a node i from LIST(level);
        PUSH/RELABEL(i);
      end;
  end;
end;

Procedure PUSH/RELABEL(i);
begin
  found := false;
  let (i, j) be the current arc of node i;
  while found = false and (i, j) ≠ null do
    if d(i) = d(j) + 1 and r_ij > 0 then found :=
        true
    else replace the current arc of node i by the next
        arc (i, j);
  if found = true then   {found an admissible arc}
  begin
    push min{e_i, r_ij, Δ − e_j} units of flow on arc
    (i, j);
    update the residual capacity r_ij and the excesses e_i
        and e_j;
    if (the updated excess) e_i ≤ Δ/2, then delete node
        i from LIST(d(i));
    if j ≠ s or t and (the updated excess) e_j > Δ/2,
      then add node j to LIST(d(j)) and set level :=
          level − 1;
  end
  else   {finished arc list of node i}
  begin
    delete node i from LIST(d(i));
    update d(i) := min{d(j) + 1; (i, j) ∈ A(i) and
        r_ij > 0};
    add node i to LIST(d(i)) and set the current arc
        of node i to the first arc of A(i);
  end;
end.
```

4. COMPLEXITY OF THE ALGORITHM

In this section, we show that the distance directed preflow-push algorithm with excess scaling correctly computes a maximum flow in $O(nm + n^2 \log U)$ time.

Lemma 5. *The excess scaling algorithm satisfies the following two conditions:*

C3. *Each nonsaturating push from a node i to a node j sends at least $\Delta/2$ units of flow.*

C4. *No excess increases above Δ (i.e., the excess-dominator does not increase subsequent to a push).*

Proof. For every push on arc (i, j) we have $e_i > \Delta/2$ and $e_j \leq \Delta/2$ because node i is a node with the smallest distance label among nodes whose excess is more than $\Delta/2$, and $d(j) = d(i) - 1 < d(i)$ by the property of the push operation. Hence, by sending $\min\{e_i, r_{ij}, \Delta - e_j\} \geq \min\{\Delta/2, r_{ij}\}$ units of flow, we ensure that in a nonsaturating push the algorithm sends at least $\Delta/2$ units of flow. Furthermore, the push operation increases the excess at node j only. Let e_j' be the excess at node j after the push. Then $e_j' = e_j + \min\{e_i, r_{ij}, \Delta - e_j\} \leq e_j + \Delta - e_j \leq \Delta$. All node excesses thus remain less than or equal to Δ.

While there are other ways of ensuring that the algorithm always satisfies the properties stated in **C3** and **C4**, pushing flow from a node with excess greater than $\Delta/2$ and with minimum distance among such nodes is a simple and efficient way of enforcing these conditions.

With properties **C3** and **C4**, the push operation may be viewed as a kind of *restrained greedy approach.* Property **C3** ensures that the push from i to j is sufficiently large to be effective. Property **C4** ensures that the maximum excess never exceeds Δ during an iteration. In particular, rather than greedily getting rid of all its excess, node i shows some restraint to prevent e_j from exceeding Δ. Keeping the maximum excess lower may be useful in practice as well as in theory. Its major impact is to encourage flow excesses to be distributed fairly equally in the network. This distribution of flows should make it easier for nodes to send flow towards the sink. This also may be important because of the following consideration: suppose several nodes send flow to a single node j creating a large excess. It is likely that node j is not able to send the accumulated flow closer to the sink, in which case, its distance label increases and much of its excess has to be returned. This phenomenon is prevented by maintaining condition **C4**.

Lemma 6. *If each push satisfies conditions* ***C3*** *and* ***C4****, then the number of nonsaturating pushes per scaling iteration is at most* $8n^2$.

Proof. Consider the potential function $F = \sum_{i \in N} e_i\, d(i)/\Delta$. The initial value of F at the beginning of the Δ-scaling iteration is bounded by $2n^2$ because e_i is bounded by Δ and $d(i)$ is bounded by $2n$. When the algorithm examines node i, one of the following two cases must apply.

Case 1. The algorithm is unable to find an arc along which flow can be pushed. This case occurs when the current arc of node i reaches the end of $A(i)$. Observe that if an arc (i, j) is found to be inadmissible earlier, then it remains inadmissible until $d(i)$ increases because $d(j)$ is nondecreasing. Hence, there exists no admissible arc emanating from node i and the relabel operation increases $d(i)$ by $\varepsilon \geq 1$ units. This increases F by at most ε units. Since the total increase in $d(i)$ throughout the running of the algorithm for each i is bounded by $2n$, the total increase in F due to relabelings of nodes is bounded by $2n^2$ in the scaling iteration. (Actually, the increase in F due to node relabelings is at most $2n^2$ over all scaling iterations.)

Case 2. The algorithm is able to identify an arc on which flow can be pushed and so it performs either a saturating or a nonsaturating push. In either case, F decreases. A nonsaturating push on arc (i, j) sends at least $\Delta/2$ units of flow from node i to node j and since $d(j) = d(i) - 1$, this decreases F by at least $1/2$ units. As the initial value of F for a scaling iteration plus the increases in F sum to at most $4n^2$, this case cannot occur more than $8n^2$ times.

Theorem 1. *The scaling algorithm performs* $O(n^2 \log U)$ *nonsaturating pushes.*

Proof. The initial value of the excess-dominator Δ is $2^{\lceil \log U \rceil} \leq 2U$. By Lemma 6, the value of the excess-dominator decreases by a factor of 2 within $8n^2$ nonsaturating pushes and a new scaling iteration begins. After $1 + \lceil \log U \rceil$ such scaling iterations, $\Delta < 1$; and by the integrality of the flow $e_i = 0$ for all $i \in N - \{s, t\}$. The algorithm thus obtains a feasible flow, which by Lemma 4 must be a maximum flow.

Theorem 2. *The complexity of the excess scaling algorithm is* $O(nm + n^2 \log U)$.

Proof. The complexity of the algorithm depends upon the number of executions of the **while** loop in the main program. In each such execution, either a PUSH/RELABEL(i) step is performed or the value of the variable *level* increases. Each execution of the procedure PUSH/RELABEL(i) results in one of the following outcomes.

Case 1. A push is performed. Since the number of saturating pushes is $O(nm)$ and the number of nonsaturating pushes is $O(n^2 \log U)$, this case occurs $O(nm + n^2 \log U)$ times.

Case 2. The distance label of node i goes up. By Corollary 1, this case can occur $O(n)$ times for each node i and $O(n^2)$ in total.

Thus the algorithm calls the procedure PUSH/RELABEL $O(nm + n^2 \log U)$ times. The effort needed to find an arc to perform the push operation is $O(1)$ plus the number of times the current arc of node i is replaced by the next arc in $A(i)$. After $|A(i)|$ such replacements for node i, Case 2 occurs and the distance label of node i goes up. Thus, the total effort needed is $\sum_{i \in N} 2n\,|A(i)| = O(nm)$ plus the number of PUSH/RELABEL operations. This is clearly $O(nm + n^2 \log U)$.

Next consider the time needed for relabeling operations. Computing the new distance label of node i requires examining arcs in $A(i)$. This yields a total of $\sum_{i \in N} 2n\,|A(i)| = O(nm)$ time for all relabeling operations. The lists LIST(r) are stored as linked stacks and queues, hence, addition and deletion of any element takes $O(1)$ time. Consequently, updating these lists is not a bottleneck operation.

Finally, we need to bound the number of increases of the variable *level*. In each scaling iteration, *level* is bounded above by $2n - 1$ and bounded below by 1. Hence, its number of increases per scaling iteration is bounded by the number of decreases plus $2n$. Furthermore, *level* can decrease only when a push is performed and, in such a case, it decreases by 1. Hence, its increases over all scaling iterations are bounded by the number of pushes plus $2n(1 + \lceil \log U \rceil)$, which is again $O(nm + n^2 \log U)$.

5. REFINEMENTS

As a practical matter, several modifications of the algorithm might improve its actual execution time without affecting its worst-case complexity. We suggest three modifications:

1. Modify the scale factor.
2. Allow some nonsaturating pushes of a small amount.
3. Try to locate nodes disconnected from the sink.

The first suggestion is to consider the scale factor. The algorithm in the present form uses a scale factor of 2, i.e., it reduces the excess-dominator by a factor of 2 in each scaling iteration. In practice, however, some other fixed integer scaling factor $\beta \geq 2$ might yield better results. The excess-dominator then will be the least power of β that is no less than the excess at any node, and property **C3** becomes the following.

C3′. Each nonsaturating push from a node i to a node j sends at least Δ/β units of flow.

The scaling algorithm presented earlier easily can be altered to incorporate the β scale factor by letting LIST(r) = $\{i \in N: e_i > \Delta/\beta$ and $d(i) = r\}$. The algorithm can be shown to run in $O(nm + \beta n^2 \log_\beta U)$ time. From the worst-case point of view any fixed value of β is optimum; the best choice for the value of β in practice should be determined empirically.

The second suggestion focuses on the nonsaturating pushes. Our algorithm as stated selects a node with $e_i > \Delta/2$ and performs a saturating or a nonsaturating push. We could, however, keep pushing the flow out of this node until either we perform a nonsaturating push of a value at least $\Delta/2$ or reduce its excess to zero. This variation might produce many saturating pushes from the node and even allow pushes after its excess decreases below $\Delta/2$. Also, the algorithm as stated earlier sends at least $\Delta/2$ units of flow during every nonsaturating push. The same complexity of the algorithm is obtained if for some fixed $r \geq 1$, one out of every $r \geq 1$ nonsaturating pushes sends at least $\Delta/2$ units of flow.

The third suggestion recognizes that in practice one potential bottleneck is the number of relabels. In particular, the algorithm *recognizes* that the residual network contains no path from node i to node t only when $d(i)$ exceeds $n - 2$. Goldberg (1987) suggested that it may be desirable to occasionally perform a breadth first search to make the distance labels exact. He discovered that a judicious use of breadth first search dramatically speeds up the algorithm.

An alternative approach is to keep track of the number n_k of nodes whose distance is k. If n_k decreases to 0 after any relabel for some k, then each node with a distance greater than k is disconnected from the sink in the residual network. (Once node j is disconnected from the sink, it stays disconnected since the shortest path from j to t is nondecreasing in length.) We avoid selecting such nodes until all nodes with positive excess become disconnected from the sink. At this time, the excesses of the nodes are sent back to the source. This approach essentially yields the two phase approach to solve the maximum flow problem as outlined in Goldberg and Tarjan (1986). The first phase constructs a maximum preflow that is converted to a maximum flow in the second phase.

6. PARALLEL IMPLEMENTATION

In this section, we describe a parallel implementation of the excess scaling algorithm. We analyze this implementation in the PRAM (Parallel Random Access Machine) Model with EREW (Exclusive Read Exclusive Write). Our algorithm runs in $O(n^2 \log U \log p)$ time and uses p processors, where $p = \lceil m/n \rceil$. More generally, our algorithm runs in $O(((nm/k) + n^2 \log U) \log k)$ time for any $k \leq p$. We describe the algorithm for $k = p$ processors, but the extension to $k \leq p$ is immediate.

Our algorithm performs the following *parallel prefix* operations.

Operation 1. Given $l \leq p$ numbers $d(j_1), d(j_2), \ldots, d(j_l)$, determine the minimum of these numbers.

Operation 2. Given $l \leq p$ numbers $f(j_1), f(j_2), \ldots, f(j_l)$, compute the partial sums $f(j_1), f(j_1) + f(j_2), f(j_1) + f(j_2) + f(j_3), \ldots, f(j_1) + f(j_2) + \ldots + f(j_l)$.

Operation 3. Given $l \leq p$ numbers $F(j_1), F(j_2), \ldots, F(j_l)$, such that $F(j_1) \leq F(j_2) \leq \ldots \leq F(j_l) \geq \Delta/2$, determine the minimum index w such that $F(j_w) \geq \Delta/2$.

Operation 4. Given $l \leq p$ numbers $e(j_1), e(j_2), \ldots, e(j_l)$, form the doubly linked list of numbers whose value is more than $\Delta/2$.

Using a parallel prefix, each of these operations can be performed in $O(\log p)$ time using $p/(\log p)$ processors (see Kindervater and Lenstra 1988 for Operations 1 and 2, and Dekel and Sahni 1983 for Operation 3.) Operation 4 can also be performed in $O(\log p)$ time using *recursive doubling* (personal communication with J. K. Lenstra).

A straightforward parallel implementation of the excess scaling algorithm requires n processors. Our reduction of the number of processors required by the algorithm is based on the following method. While performing a push at node i or relabeling this node, the algorithm does not consider all arcs in $A(i)$ simultaneously, but it partitions arcs in $A(i)$ into groups, each containing at most p arcs, and considers only one group at a time. For example, while relabeling node i, the algorithm sequentially examines $\lceil |A(i)|/p \rceil$ groupings and using Operation 1 computes the minimum of a group in $O(\log p)$ time. Since any

node can be relabeled at most $2n$ times, the total number of groups examined for all nodes is

$$\sum_{i=1}^{n} \left\lceil \frac{|A(i)|}{p} \right\rceil (2n) \leq \sum_{i=1}^{n} \left(\frac{|A(i)|}{p} + 1 \right)(2n) = \left(\frac{m}{p} + n \right)(2n) \leq 4n^2.$$

Thus, partitioning arcs in $A(i)$ into groups does not increase the computational time in the worst case.

Our parallel algorithm performs the PUSH/RELABEL(i) step on a group of arcs in $A(i)$ in parallel. The algorithm maintains for each node i an index *current group*, which plays the same role as the current arc does for the sequential algorithm. If the current group of the node i is null (i.e., all of its groups have been examined), then the node is relabeled. Otherwise it assigns a parallel processor to each arc in the group which easily can be done in $O(\log p)$ time. Let the arcs in this group be $(i, j_1), (i, j_2), \ldots, (i, j_l)$ for some $l \leq p$. Each processor checks the arc (i, j) it is assigned to and defines a number $f(j) = r_{ij}$ if arc (i, j) is admissible, and $f(j) = 0$ otherwise. Then the algorithm performs Operation 2 to compute the partial sums of the $f(\cdot)$ values. This operation results in the following two cases.

Case 1. $f(j_1) + f(j_2) + \ldots + f(j_l) \leq \Delta/2$. The algorithm saturates all admissible arcs in the group, updates the residual capacities of arcs and excesses at the nodes, and replaces the current group of node i by the next group.

Case 2. $f(j_1) + f(j_2) + \ldots + f(j_l) > \Delta/2$. The algorithm performs Operation 3 with $F(\cdot)$ equal to the partial sums at the nodes. The algorithm saturates all arcs $(i, j_1), (i, j_2), \ldots, (i, j_{w-1})$ and performs a nonsaturating push of value $\min\{e(i) - F(j_{w-1}), r_{ij_w}, \Delta - e(j_w)\}$ on arc (i, j_w). The algorithm updates the residual capacities of arcs and excesses at the nodes.

Let $e(j)$ denote the updated excesses of nodes in both cases. The algorithm finally performs Operation 4 on numbers $e(j_1), e(j_2), \ldots, e(j_l)$ to form the doubly linked list LIST($d(j_1)$). If this list is nonempty, then the variable *level* is also updated.

We count the number of parallel prefix operations performed by the algorithm. Each iteration of the algorithm results in one of the following outcomes: a) the algorithm saturates all arcs in a group; b) the algorithm performs a nonsaturating push; and c) the algorithm relabels a node. Since an arc is saturated at most n times, the number of occurrences of case a is bounded by $\sum_{i=1}^{n} \lceil |A(i)|/p \rceil n \leq 2n^2$. Each occurrence of case b pushes at least $\Delta/2$ units of flow; hence, this case occurs at most $O(n^2 \log U)$ times. The algorithm performs $O(1)$ parallel prefix operations for each occurrence of case a and case b. Furthermore, it has been shown that the algorithm performs at most $4n^2$ parallel prefix operations during all relabel steps. Consequently, the algorithm performs $O(n^2 \log U)$ parallel prefix operations and runs in $O(n^2 \log U \log p)$ time.

The only modification to the algorithm for $k \leq p$ parallel processors is that the arcs in each adjacency list $A(i)$ are partitioned into groups of at most k arcs. In this case, the number of parallel prefix operations for group saturations and node relabeling is $O(nm/k)$ and the number of nonsaturating pushes remains unchanged. We have, thus, established the following result.

Theorem 3. *In the PRAM model with EREW, the parallel excess scaling algorithm runs in* $O(((nm/k) + n^2 \log U) \log k)$ *time using* $k \leq p$ *processors.*

7. RELATED RESEARCH

Our improvement of the distance directed preflow-push algorithm has several advantages over other algorithms for the maximum flow problem. Our algorithm is superior to all previous algorithms for the maximum flow problem under the reasonable assumption that U is polynomially bounded in n. Also, the algorithm utilizes very simple data structures that make it attractive from an implementation viewpoint.

Our algorithm is a novel approach to combinatorial scaling algorithms. In the previous scaling algorithms developed by Edmonds and Karp, Rock 1980, and Gabow, scaling involved a sequential approximation of either the cost coefficients or the capacities and right-hand sides. For example, we first solve the problem with the costs C approximated by $C/2^T$ for some integer T; we then reoptimize to solve the problem with costs approximated by $C/2^{T-1}$, and then reoptimize for the problem with costs approximated by $C/2^{T-2}$, and so forth. Our excess scaling algorithm does not fit into this standard framework. Rather, our algorithm works with true data, relaxes the flow conservation constraints and scales the maximum amount of relaxation.

The preflow-push algorithms for the maximum flow problem have been extended for the minimum cost flow problem. A generalization of the Goldberg-Tarjan maximum flow algorithm to the minimum cost flow problem was developed by Bertsekas (1986). This algorithm, however, was only pseudopolynomial. Goldberg and Tarjan (1987) incorporated cost scaling

in this approach and obtained a polynomial time algorithm. The algorithm of Goldberg and Tarjan (1987) is similar to our algorithm—it also works with true data, relaxes the complementary slackness conditions and gradually decreases the relaxation to zero.

The scaling algorithm for the maximum flow problem can be improved further by using more clever rules to push flow or by using dynamic trees. We describe such improvements in Ahuja, Orlin and Tarjan. We show that by using a larger scale factor and by pushing flow from a node with the highest distance label among nodes having *sufficiently large* excess, the algorithm runs in $O(nm + n^2 \log U/\log\log U)$ time. (Assume that $U \geq 4$.) We describe another variation of the excess scaling algorithm that runs in $O(nm + n^2\sqrt{\log U})$ time. Use of the dynamic tree data structure further improves the complexity of this algorithm to $O(nm \log(n\sqrt{\log U}/m + 2))$.

We also have undertaken an extensive empirical study to assess the computational merits of the preflow-push algorithms versus the previous best algorithms, those of Dinic and Karzanov. Our experiments so far suggest that preflow-push algorithms are substantially faster than Dinic's and Karzanov's algorithms.

Our algorithms and those due to Goldberg and Tarjan suggest the superiority of distance label based approaches over the layered network based approaches. The improvements we obtain do not seem to be possible for the algorithms utilizing layered networks. The distance labels implicitly store dynamically changing layered networks and hence are more powerful. We show the use of distance labels in augmenting path algorithms, capacity scaling algorithms and for unit capacity networks in Orlin and Ahuja (1987).

The maximum flow problem on bipartite networks is an important class of the maximum flow problem (see Gusfield, Martel and Fernandez-Baca 1985). The bipartite network is a network $G = (N, A)$, such that $N = N_1 \cup N_2$ and $A \subseteq N_1 \times N_2$. Let $n_1 = |N_1|$ and $n_2 = |N_2|$. For cases where $n_1 \ll n_2$, our maximum flow algorithm can be modified to run in $O(n_1 m + n_1^2 \log U)$ time, thus resulting in significant speedup over the original algorithm. Our results on bipartite network flows will appear in a future paper jointly with C. Stein and R. Tarjan.

ACKNOWLEDGMENT

We wish to thank John Bartholdi, Tom Magnanti, and Hershel Safer for their suggestions which led to improvements in the presentation. We also are grateful to the referees for their perceptive comments. This research was supported in part by the Presidential Young Investigator Grant 8451517-ECS of the National Science Foundation, by grant AFOSR-88-0088 from the Air Force Office of Scientific Research, and by grants from Analog Devices, Apple Computer, Inc., and Prime Computer.

REFERENCES

AHO, A. V., J. E. HOPCROFT AND J. D. ULLMAN. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

AHUJA, R. K., J. B. ORLIN AND R. E. TARJAN. 1988. Improved Time Bounds for the Maximum Flow Problem. Research Report, Sloan School of Management, MIT, Cambridge, Mass. (to appear in *SIAM J. Comp.*).

BERTSEKAS, D. 1986. Distributed Relaxation Methods for Linear Network Flow Problems. *Proc. of 25th IEEE Conference on Decision and Control*, Athens, Greece.

CHERIYAN, J., AND S. N. MAHESHWARI. 1988. Analysis of Preflow Push Algorithms for Maximum Network Flow. Technical Report, Dept. of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India. (Revised).

CHERKASKY, R. V. 1977. Algorithm for Construction of Maximal Flow in Networks With Complexity of $O(V^2\sqrt{E})$ Operation (in Russian). *Math. Methods Solution Econ. Prob.* **7,** 112–125.

DANTZIG, G. B., AND D. R. FULKERSON. 1956. On the Max-Flow Min-Cut Theorem of Networks. In *Linear Inequalities and Related Systems*, pp. 215–221, H. W. Kuhn and A. W. Tucker (eds.) (Annals of Mathematics Study 38). Princeton University Press, Princeton, N.J.

DEKEL, E., AND S. SAHANI. 1983. Binary Trees and Parallel Scheduling Algorithms. *IEEE Trans. Comput.* **10,** 657–675.

DINIC, E. A. 1970. Algorithm for Solution of a Problem of Maximum Flow in Networks With Power Estimation. *Soviet Math. Dokl.* **11,** 1277–1280.

EDMONDS, J., AND R. M. KARP. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. Assoc. Comput. Mach.* **19,** 248–264.

FORD, L. R., AND D. R. FULKERSON. 1956. Maximal Flow Through a Network. *Can. J. Math.* **8,** 399–404.

FULKERSON, D. R., AND G. B. DANTZIG. 1955. Computations of Maximum Flow in Networks. *Naval Res. Log. Quart.* **2,** 277–283.

GABOW, H. N. 1985. Scaling Algorithms for Network Problems. *J. Comput. Syst. Sci.* **31,** 148–168.

GALIL, Z. 1980. An $O(V^{5/3}E^{2/3})$ Algorithm for the Maximal Flow Problem. *Acta Inform.* **14,** 221–242.

GALIL, Z. AND A. NAAMAD. 1980. An $O(VE \log^2 V)$

Algorithm for the Maximum Flow Problem. *J. Comput. Syst. Sci.* **21,** 203–217.

GOLDBERG, A. V. 1985. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, Mass.

GOLDBERG, A. V. 1987. Efficient Graph Algorithms for Sequential and Parallel Computers. Ph.D. Dissertation, Laboratory for Computer Science, MIT, Cambridge, Mass. Available as Technical Report MIT/LCS/TR-374.

GOLDBERG, A. V., AND R. E. TARJAN. 1986. A New Approach to the Maximum Flow Problem. *Proc. of the 18th Annual ACM Symposium on the Theory of Computing*, 136–146. (also in *J. Assoc. Comput. Mach.* **35** (1988), 921–940).

GOLDBERG, A. V., AND R. E. TARJAN. 1987. Solving Minimum Cost Flow Problem by Successive Approximation. *Proc. of the 19th Annual ACM Symposium on the Theory of Computing*, 7–18.

GUSFIELD, D., C. MARTEL AND D. FERNANDEZ-BACA. 1985. Fast Algorithms for Bipartite Network Flow. Technical Report YALEV/DCS/TR-356, Department of Computer Science, Yale University, New Haven, Conn.

KARZANOV, A. V. 1974. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dokl.* **15,** 434–437.

KINDERVATER, G. A. P., AND J. K. LENSTRA. 1988. Parallel Computing in Combinatorial Optimization. *Ann. Opns. Res.* **14,** 245–289.

MALHOTRA, V. M., M. P. KUMAR, AND S. N. MAHESHWARI. 1978. An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks. *Inform. Process. Lett.* **7,** 277–278.

ORLIN, J. B., AND R. K. AHUJA. 1987. New Distance-Directed Algorithms for Maximum Flow and Parametric Maximum Flow Problems. Working Paper No. 1908-87, Sloan School of Management, MIT, Cambridge, Mass.

ROCK, H. 1980. Scaling Techniques for Minimal Cost Network Flows. In *Discrete Structures and Algorithms*, pp. 181–191, V. Page and Carl Hanser (eds.) München, West Germany.

SHILOACH, Y. 1978. An $O(nI \log^2(I))$ Maximum Flow Algorithm. Technical Report STAN-CS-78-702, Computer Science Department, Stanford University, Stanford, Calif.

SHILOACH, Y., AND U. VISHKIN. 1982. An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. *J. Algorithms* **3,** 128–146.

SLEATOR, D. D., AND R. E. TARJAN. 1983. A Data Structure for Dynamic Trees, *J. Comput. Syst. Sci.* **24,** 362–391.

TARJAN, R. E. 1984. A Simple Version of Karzanov's Blocking Flow Algorithm. *Opns. Res. Lett.* **2,** 265–268.