A New Approach to the Maximum Flow Problem

Andrew V. Goldberg[†]

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

Robert E. Tarjan

Computer Science Department
Princeton University
Princeton, NJ 08544
and
AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

All previously known efficient maximum flow algorithms work by finding augmenting paths, either one path at a time (as in the original Ford and Fulkerson algorithm) or all shortest length augmenting paths at once (using the layered network approach of Dinic). We introduce an alternative method based on the preflow concept of Karzanov. 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 algorithm 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 and performing some more complicated analysis, we obtain a version of the algorithm running in $O(nm \log(n^2/m))$ time on an n-vertex, m-edge graph. This is as fast as any known method for any graph density and faster on graphs of moderate density. The algorithm also admits efficient distributed and parallel implementations. We obtain a parallel implementation running in $O(n^2 \log n)$ time and using only $O(m)$ space. This time bound matches that of the Shiloach-Vishkin algorithm, which requires $O(n^2)$ space.

1. Introduction

The problem of finding a maximum flow in a directed graph with edge capacities arises in many settings in operations research and other fields, and efficient algorithms for the problem have received a great deal of attention. Extensive discussion of the problem and its applications can be found in the books of Ford and Fulkerson [7], Lawler [14], Even [5], Papadimitriou and Stieglitz [16], and Tarjan [23]. Table 1 shows a history of algorithms for the problem. Time bounds are stated in terms of n, the number of vertices, m, the number of edges in the problem network, and in one case N, an upper bound on the edge capacities (assumed to be integers for this case).

All algorithms in the table work either by finding augmenting paths one by one (algorithms 1 and 2), or by finding all shortest augmenting paths in one phase, as proposed by Dinic [3] (algorithms 3-10), or by repeated use of Dinic's method (algorithm 11). There is no clear winner among the algorithms in the table; algorithms 4, 6, 10 and 12 are designed to be fast on dense graphs, and algorithms 5, 7, 8, 9, and 11 are designed to be fast on sparse graphs. For dense graphs, the best known bound of $O(n^3)$ was first obtained by Karzanov [13]; Malhotra, Pramodh Kumar, and Maheshwari [15] and Tarjan [24] have given simpler $O(n^3)$-time algorithms. For sparse graphs, Sleator and

[†] Supported by a Fannie and John Hertz Foundation Fellowship.

#	Date	Discoverer	Running Time	References
1	1956	Ford and Fulkerson	-	[6,7]
2	1969	Edmonds and Karp	$O(nm^2)$	[4]
3	1970	Dinic	$O(n^2m)$	[3]
4	1974	Karzanov	$O(n^3)$	[13]
5	1977	Cherkasky	$O(n^2m^{1/2})$	[2]
6	1978	Malhotra, Pramodh Kumar, and Maheshwari	$O(n^3)$	[15]
7	1978	Galil	$O(n^{5/3}m^{2/3})$	[10]
8	1978	Galil and Naamad; Shiloach	$O(nm(\log n)^2)$	[11] [18]
9	1980	Sleator and Tarjan	$O(nm\log n)$	[20,21]
10	1982	Shiloach and Vishkin	$O(n^3)$	[19]
11	1983	Gabow	$O(nm\log N)$	[9]
12	1984	Tarjan	$O(n^3)$	[24]

Table 1. A history of maximum flow algorithms.

Tarjan's bound of $O(nm\log n)$ [20,21] is the best known. For a small range of densities ($m=\Omega(n^2/(\log n)^3)$ and $m=O(n^2)$), Galil's bound of $O(n^{5/3}m^{2/3})$ [10] is best. For sparse graphs with integer edge capacities of moderate size, Gabow's scaling algorithm [9] is best. Among the algorithms in the table, the only parallel algorithm is that of Shiloach and Vishkin [19]. This algorithm has a parallel running time of $O(n^2\log n)$ but requires $O(mn)$ space. Vishkin (private communication) has improved the space bound to $O(n^2)$.

In this paper we present a different approach to the maximum flow problem. Our method uses Karzanov's idea of a *preflow*. During each phase, Karzanov's algorithm maintains a preflow in a layered network. The algorithm pushes flow through the network to find a blocking flow, which determines the layered network for the next phase. Our algorithm abandons the idea of finding a flow in each phase, and indeed abandons the idea of global phases. Instead, our algorithm maintains a preflow in the original network and pushes flow toward the sink along what it estimates to be shortest paths. Only when the algorithm terminates does the preflow become a flow, and then it is a maximum flow.

Our algorithm is simple and intuitive. It has natural implementations in sequential and parallel models of computation. We present a sequential implementation that uses the dynamic tree data structure of Sleator and Tarjan [21, 22, 23] and runs in $O(nm\log(n^2/m))$ time. This bound matches the best known bounds as a function of n and m for both sparse and dense graphs and is smaller on graphs of moderate density. We present a parallel version of the algorithm running in $O(n^2\log n)$ time using $O(1)$ words of storage per edge. This matches the time bound of the Shiloach-Vishkin algorithm, but our improved space bound allows implementation under a model of distributed computation in which the amount of space per processor at a vertex is bounded by the vertex degree.

Our paper contains six sections in addition to the introduction. Section 2 contains definitions and notation. Section 3 describes a generic version of the algorithm and proves its termination and correctness. Section 4 refines the algorithm to produce an $O(n^3)$-time sequential implementation. Section 5 introduces dynamic trees and thereby improves the sequential time to $O(nm\log(n^2/m))$. Section 6 discusses efficient distributed and parallel implementation. Section 7 contains some concluding remarks and open problems. The algorithm except for the $O(nm\log(n^2/m))$-time implementation was developed by the first author; an early version appears in an M.I.T. technical memo [12].

2. Definitions and Notation

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. An *undirected edge* of G is an unordered pair $\{v,w\}$ such that $(v,w)\in E$ or $(w,v)\in E$. Vertices v and w are *neighbors* in G. The *undirected version* of G is $G^u=(V,E^u)$ where E^u is the set of undirected edges of G. G is *weakly connected* if G^u is connected. For the purpose of simplifying resource bounds we shall assume $m\geq n-1$, which is true if G is weakly connected.

G is a *network* if it has two distinct distinguished vertices, a *source* s and a *sink* t, and a

positive capacity $c(v, w)$ on each directed edge (v, w). We extend the capacity function to $V \times V$ by defining $c(v, w) = 0$ for $(v, w) \notin E$. A *flow* f on G is a real-valued function on vertex pairs satisfying (1), (2), and (3) below:

(1) $-c(w, v) \leq f(v, w) \leq c(v, w) \quad \forall v, w \in V$

(2) $f(v, w) = -f(w, v) \quad \forall v, w \in V$

(3) $\sum_{w \in V} f(v, w) = 0 \quad \forall v \in V - \{s, t\}$

The *value* of a flow f is the net flow into the sink (or out of the source),

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

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

A *cut* $S, \bar{S}$ is a partition of the vertex set $(S \cup \bar{S} = V, S \cap \bar{S} = \emptyset)$ with $s \in S$ and $t \in \bar{S}$. The *capacity* of the cut is

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

The *flow across the cut* is

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

Any flow has value at most the capacity of any cut. A *minimum cut* is a cut of minimum capacity. The *max-flow, min-cut theorem* of Ford and Fulkerson states that the value of a maximum flow is equal to the capacity of a minimum cut.

A *preflow* g is a real-valued function on vertex pairs satisfying conditions (1), (2), and the following relaxation (4) of condition (3):

(4) $\sum_{u \in V} g(u, v) \geq 0 \quad \forall v \in V - \{s\}$

Given a preflow g, the *flow excess* $e(v)$ is defined for each vertex v by

$$e(v) = \begin{cases} \infty & \text{if } v = s \\ \sum_{u \in V} g(u, v) & \text{if } v \neq s \end{cases}$$

We define the *flow across a cut* for a preflow the same as for a flow. The *residual capacity* r_g for a preflow g is the real-valued function on vertex pairs defined by $r_g(v, w) = c(v, w) - g(v, w)$. The *residual graph* for g is $G_g = (V, E_g)$ where E_g contains all pairs of vertices with positive residual capacity: $E_g = \{(v, w) \mid r_g(v, w) > 0\}$. An edge $(v, w) \in E_g$ is a *residual edge*.

A *valid labeling* d for a preflow g is a non-negative integer-valued function on vertices such that $d(t) = 0$, $d(v) > 0 \ \forall v \neq t$, and $d(w) \geq d(v) - 1$ for every residual edge (v, w). A valid labeling is such that $d(v)$ is a lower bound on the distance from v to t in G_g; i.e. every path from v to t contains at least $d(v)$ edges. (This follows by induction on the distance from v to t.) We denote the distance from v to w in G by $d_G(v, w)$.

3. A Generic Maximum Flow Algorithm

Our maximum flow algorithm maintains a preflow g and a valid labeling d for g. Roughly speaking, the algorithm proceeds by examining vertices with positive flow excess and pushing flow from them to vertices estimated to be closer to t in the residual graph. As a distance estimate, the algorithm uses the labeling d, which it periodically updates to more accurately reflect the current residual graph.

To be more precise, the algorithm consists of two stages. During the first stage, flow is pushed toward the sink. The first stage ends having determined a minimum cut and the value of a maximum flow. During the second stage, remaining flow excess is returned to the source, converting the preflow into a maximum flow. The dominant part of the computation is the first stage.

The first stage begins with the zero flow ($g(v, w) = 0$ for all $v, w \in V$) and an initial valid labeling d. A simple initial valid labeling is $d(t) = 0$, $d(v) = 1$ for $v \neq t$. A more accurate (indeed the most accurate possible) initial valid labeling is $d(v) = d_G(v, t)$. The latter labeling is computable in $O(m)$ sequential time or $O(n \log n)$ parallel time using a backward breadth-first search from t. The resource bounds we shall derive hold for either of these initial labelings.

A vertex v is *active* if $0 < d(v) < n$ and $e(v) > 0$. The first stage consists of repeatedly performing the following steps, in any order, until there are no active vertices:

Push. Select any active vertex v. Select any residual edge (v, w) with $d(w) = d(v) - 1$. Send $\delta = \min\{e(v), r_g(v, w)\}$ units of flow from v to w. The push is *saturating* if $\delta = r_g(v, w)$ and *non-saturating* otherwise. The push has the effect of increasing $f(v, w)$ and $e(w)$ by δ and decreasing $f(w, v)$ and $e(v)$ by δ.

Relabel. Select any vertex v with $0 < d(v) < n$. Replace $d(v)$ by $\min\{d(w) + 1 \mid (v, w)$ is a residual edge$\}$.

Lemma 1. The first stage maintains the invariant that d is a valid labeling for g. For any vertex v, $d(v)$ never decreases.

Proof. By induction on the number of pushing and relabeling steps. The initial labeling is valid by assumption. Given that d is a valid labeling, a relabeling step changing $d(v)$ can only increase $d(v)$ and must produce a new valid labeling. Consider a

pushing step that sends flow from v to w. This step may add (w, v) to E_g and may delete (v, w). Since $d(w) = d(v) - 1$, the addition of (w, v) to E_g does not disturb the validity of d. Delition of an edge also does not disturb the validity of d. •

Lemma 2. The number of relabeling steps that actually change vertex labels is at most $(n-1)^2$.

Proof. The label of t remains zero throughout the algorithm. Every other vertex has an initial label of at least 1 and can have its label increased to at most n. Since no label ever decreases, there are at most $(n-1)^2$ relabeling steps. •

Lemma 3. The number of saturating pushing steps is at most nm.

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

Lemma 4. The number of non-saturating pushing steps is at most $n^2 m$.

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

Theorem 1. The first stage terminates after $O(n^2 m)$ steps, given that every relabeling step changes a vertex label.

Proof. Immediate from Lemmas 2, 3, and 4. •

For a preflow g, let S_g, $\bar{S}_g$ be the vertex partition such that $\bar{S}_g$ contains all vertices from which t is reachable in G_g, and $S_g = V - \bar{S}_g$.

Lemma 5. When the first stage terminates, S_g, $\bar{S}_g$ is a cut such that every pair v, w with $v \in S_g$, $w \in \bar{S}_g$ satisfies $g(v, w) = c(v, w)$.

Proof. Every vertex $v \in \bar{S}_g$ has $d(v) < n$ since $d(v)$ is a lower bound on the distance from v to t in G_g, and this distance is either less than n or infinite. Thus every vertex $v \neq t$ with $e(v) > 0$ is in S_g. This includes s, which means that S_g, $\bar{S}_g$ is a cut. If v, w is a pair such that $v \in S_g$, $w \in \bar{S}_g$, then $r_g(v, w) = c(v, w) - g(v, w) = 0$ by the definition of $\bar{S}_g$. •

We shall eventually show that the cut S_g, $\bar{S}_g$ is a minimum cut. Proving this requires converting g into a flow without disturbing the flow across S_g, $\bar{S}_g$, which is the task of the second stage of the algorithm. The second stage returns excess flow to s along estimated shortest paths. It is like the first stage but simpler, because edge flows need only be reduced and not increased.

For a preflow g, let $E_g^* = \{(v, w) \mid g(v, w) > 0\}$, and let $G_g^* = (V, E_g^*)$. The second stage begins with the preflow g from the first stage and a vertex labeling d' such that $d'(s) = 0$ and $d'(w) \leq d'(v) + 1$ for $(v, w) \in E_g^*$. A simple initial labeling is $d'(s) = 0$, $d(v) = 1$ for $v \neq s$. The most accurate initial labeling is $d(v) = d_{G_g^*}(s, v)$. The latter is computable in $O(m)$ sequential time or $O(n \log n)$ parallel time by breadth-first search. As with the first stage, all our results hold for either initial labeling. The second stage consists of repeating the following steps in any order until there is no vertex $w \neq t$ with $0 < d'(w) < n$ and $e(w) > 0$:

Reduce. Select any vertex $w \neq t$ with $0 < d'(w) < n$ and $e(w) > 0$. Select any edge $(v, w) \in E_g^*$ with $d'(v) = d'(w) - 1$. Send $\delta = \min\{e(w), g(v, w)\}$ units of flow from w to v. The reducing step is *zeroing* if $\delta = g(v, w)$, *non-zeroing* otherwise.

Relabel. Select any vertex w with $0 < d'(w) < n$. Replace $d'(w)$ by $\min\{d'(v) + 1 \mid (v, w) \in E_g^*\}$.

Remark. The *reduce* step can only decrease flow through an edge in the original network.

The results for the second stage are analogous to those for the first stage, except for the Lemma 8 below, which follows easily from the above remark. We state these results without proof.

Lemma 6. The second stage maintains the invariant that $d'(s) = 0$ and $d'(w) \leq d'(v) + 1$ for $(v, w) \in E_g^*$. For any vertex v, $d'(v)$ never

decreases.

Lemma 7. The number of relabeling steps in the second stage is at most $(n-1)^2$.

Lemma 8. The number of zeroing reducing steps is at most m.

Lemma 9. The number of non-zeroing reducing steps is at most $n(n+m)$.

Theorem 2. The second stage terminates after $O(nm)$ steps, given that every relabeling step changes a vertex label.

Theorem 3. The vertex partition S_g, $\bar{S}_g$ remains fixed during the second stage. When the second stage terminates, S_g, $\bar{S}_g$ is a minimum cut and g is a maximum flow.

Proof. An induction on the number of steps shows that the second stage maintains the following invariants:

(i) if v, w is a pair such that $g(v,w) > 0$ and $w \in S_g$, then $v \in S_g$

(ii) if $v \neq t$ and $e(v) > 0$, then $v \in S_g$

(iii) the partition S_g, $\bar{S}_g$ remains fixed

Another induction on the number of steps shows that the first and second stages maintain the following invariant:

(iv) if $e(v) > 0$, $d_{G_g^*}(s,v) < n$.

Invariant (iv) implies that when the second stage terminates, g is a flow. Invariant (iii) and Lemma 5 imply that when the second stage terminates, S_g, $\bar{S}_g$ is a minimum cut and g is a maximum flow. •

All previously known maximum flow algorithms first find a maximum flow and then a minimum cut; furthermore they must find a minimum cut to guarantee that a maximum flow has been found. In contrast, our algorithm first finds a minimum cut (in the first stage) and then a maximum flow (in the second stage). In applications requiring only a minimum cut (of which there are many; see [17]), the second stage need not be performed at all.

Although we have described the second stage as a simplified version of the first stage, there is an alternative way to convert g to a flow that is more efficient, at least theoretically. We first eliminate circulations (cycles of flow) by applying the algorithm of Sleator and Tarjan [22], which though stated for flows, works for preflows as well. This converts G_g^* to an acyclic graph, without affecting the cut S_g, $\bar{S}_g$. This computation takes $O(m \log n)$ sequential time. Next, we process the vertices of G_g^* in reverse topological order to eliminate flow excesses. To process a vertex v with $e(v) > 0$, we reduce the flow on incoming edges by a total of $e(v)$. This increases the excess on predecessors of v. After all vertices are processed, g is a flow, and S_g, $\bar{S}_g$ has not changed. Thus g is a maximum flow and S_g, $\bar{S}_g$ is a minimum cut. Processing the vertices of G in reverse topological order in this way takes $O(m)$ sequential time. The total time to convert g to a flow is thus $O(m \log n)$.

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

4. Sequential Implementation

As a first step toward obtaining an efficient sequential implementation, we shall describe a simple refinement of our algorithm that runs in $O(n^2 m)$ time, matching Dinic's bound. We need certain data structures to represent the network and the preflow. Recall that a pair $\{v, w\}$ such that $(v, w) \in E$ or $(w, v) \in E$ is an undirected edge of G. We associate three values with each undirected edge: $c(v,w)$, $c(w,v)$, and $g(v,w)$ $(=-g(w,v))$. Each vertex v has a list of the incident undirected edges $\{v, w\}$, in fixed but arbitrary order. (Thus each edge $\{v, w\}$ occurs in exactly two lists, the one for v and the one for w.) Each vertex v also has a *current edge* $\{v, w\}$, which is the current candidate for a pushing step out of v. Initially the current edge of v is the first edge on the edge list of v. The refined algorithm consists of repeating the following step until no vertex v has $e(v) > 0$ and $0 < d(v) < n$:

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

Lemma 10. Each relabeling of a vertex v in a push/relabel step causes $d(v)$ to increase by at least one.

Proof. Just before the relabeling, for each neighbor w of v either $d(w) \geq d(v)$ or $r_g(v,w) = 0$, because $d(v)$ have not changed since (v, w) was the current edge, w cannot push to v unless $d(w) > d(v)$, and by Lemma 1 $d(w)$ does not decrease. The lemma follows from the definition of the relabeling step. •

The refined algorithm needs one additional data structure, a set Q containing all active vertices

(which are candidate vertices for push/relabeling steps). Initially $Q = \{s\}$. Maintaining Q takes only $O(1)$ time per push/relabeling step. (Such a step applied to an edge $\{v, w\}$ may require adding w to Q and/or deleting v.)

Theorem 4. The refined algorithm runs in $O(nm)$ time plus $O(1)$ time per non-saturating pushing step, for a total of $O(n^2 m)$ time.

Proof. Let $v \neq t$ and let Δ_v be the size of the edge list of v. Relabeling v requires a single scan of the edge list of v. By Lemma 10, the total number of passes through the edge list of v is at most $2n - 2$, one for each of the at most $n - 1$ relabelings and one before each relabeling as the current edge runs through the list. Every push/relabel step selecting v either causes a push, changes the current edge of v, or increases $d(v)$. It follows that the total time spent in push/relabel steps selecting v is $O(n\,\Delta_v)$ plus $O(1)$ time per push out of v. Summing over all vertices and applying Lemmas 3 and 4 gives the theorem. •

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

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

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

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

Lemma 11. The number of passes over the queue is at most $2n(n-1)$.

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

Corollary 1. The number of non-saturating pushes during the last-in, first-out algorithm is at most $2n(n-1)^2$.

Proof. There is at most one non-saturating push per vertex other than t per pass. •

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

Proof. Immediate from Theorem 4 and Corollary 1. •

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 non-saturating pushing step below $O(1)$. We do this by using the dynamic tree data structure of Sleator and Tarjan [21, 22, 23]. This structure allows the maintenance of a set of vertex-disjoint rooted trees, each of whose vertices v has an associated real value $h(v)$, possibly ∞ or $-\infty$. We shall regard a tree edge as directed toward the root, i.e. from child to parent. We denote the parent of a vertex v by $p(v)$. We adopt the convention that every vertex is both an ancestor and a descendant of itself. The tree operations we shall need are described in Figure 1.

The total time for a sequence of l operations starting with a collection of single-vertex trees is $O(l \log k)$, where k is an upper bound on the maximum number of vertices in a tree. (The implementation of dynamic trees presented in [22, 23] does not support *find size* operations, but it is easily modified to do so. See the appendix.)

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

find root (v):	Find and return the root of the tree containing vertex v.
find size (v):	Find and return the number of vertices in the tree containing vertex v.
find value (v):	Compute and return $h(v)$
find min (v):	Find and return the ancestor w of v of minimum $h(w)$. In case of a tie, choose the vertex w closest to the root.
change value (v, x):	Add x to $h(w)$ for all ancestors w of v. (We adopt the convention that $\infty + (-\infty) = 0$.)
link (r, v):	Combine the trees containing vertices r and v by making v the parent of r. This operation does nothing if r and v are in the same tree or if r 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.

Figure 1. Dynamic tree operations.

to k, where k is a parameter to be chosen below.

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

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

Tree Push/Relabel. Let v be an active vertex. Let $\{v, w\}$ be the current edge of v. If $d(w) = d(v) - 1$ and $r_g(v, w) > 0$, apply the appropriate one of the following cases:

(a) If *find root*$(v) = v$ and *find size*(v) + *find size*$(w) \leq k$, then make w the parent of v by performing *change value*$(v, -\infty)$, *change value*$(v, r_g(v, w))$, and *link*(v, w).

(b) If *find root*$(v) = v$ and *find size*(v) + *find size*$(w) > k$, then apply a pushing step to send flow from v to w.

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

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

If on the other hand $d(w) > d(v) - 1$ or $r_g(v, w) = 0$, replace $\{v, w\}$ as the current edge of v by the next edge on the edge list of v; if v has no next edge, make the first edge on the edge list of v the current edge, perform *cut*(u) and *change value*(u, ∞) for each vertex u whose parent is v, and apply a relabeling step to v.

It is important to realize that this algorithm stores values of the preflow g in two different ways. If $\{v, w\}$ is an edge that is not a dynamic tree edge, $g(v, w)$ is stored explicitly, with $\{v, w\}$. If $\{v, w\}$ is a dynamic tree edge, with w the parent of v, $h(v) = c(v, w) - g(v, w)$ is stored implicitly in the dynamic tree data structure. Whenever a tree edge $\{v, w\}$ is cut, $h(v)$ must be computed and $g(v, w)$ restored to its current value. In addition, when the algorithm terminates, preflow values must be computed for all edges remaining in dynamic trees.

Lemma 12. The algorithm with dynamic trees runs in $O(nm \log k)$ time plus $O(\log k)$ time per addition of a vertex to Q.

Proof. The condition "*find size*(v) + *find size*$(w) \leq k$" in case (a) of tree push/relabel guarantees that the maximum size of any dynamic tree is k. Thus the time per dynamic tree operation is $O(\log k)$. Each tree push/relabel step takes $O(1)$ time plus $O(1)$ tree operations plus $O(1)$ tree operations per *cut* operation plus time for relabeling. The total relabeling time is $O(nm)$. By a proof like that of Lemma 3, the total number of *cut* operations is at most nm. The total number of *link* operations is at most n plus the number of *cut* operations, i.e. at most $n(m+1)$. The total number of tree push/relabel steps is $O(nm)$ plus one per addition of a vertex to Q. Combining these obser-

vations gives the lemma. •

We define passes over the queue Q exactly as in Section 4. The proof of Lemma 11 remains valid, which means that the number of passes is at most $2n(n-1)$.

The next lemma is the heart of the analysis.

Lemma 13. The number of times a tree root has its excess made positive is $O(nm + n^3/k)$.

Proof. For any vertex v, we denote the tree containing v by T_v and its size by $|T_v|$. If r is a tree root, $e(r)$ can be made positive in either case (b) or case (c) of tree push/relabel. We shall analyze the number of times this happens in case (c); the analysis of case (b) is similar but simpler.

Suppose a tree root r has its excess made positive because of a push from a vertex w in case (c). Let i be the pass in which this happens. Vertex w was added to Q during pass $i-1$. Let I be the interval of time from the beginning of pass $i-1$ up to the increase in $e(r)$. If T_w changes during I (because of a link or cut), charge the increase in $e(r)$ to the last link or cut during I that changes T_w. Over the entire algorithm there are at most $3nm$ such charges, one per link and two per cut.

On the other hand, suppose T_w does not change during I. Consider the addition of w to Q in pass $i-1$. This addition occurs either after a relabeling of w or because of a pushing step in case (b) of tree push/relabel. In the former case, we charge the increase in $e(r)$ to the relabeling of w. Over the entire algorithm, there are at most $(n-1)^2$ such charges, one per relabeling. In the latter case, let the pushing step in (b) push flow from v to w. If the pushing step is saturating, we charge the increase in $e(r)$ to the saturating push. Over the entire algorithm, there are at most nm such charges, one per saturating push.

There is still the possibility that the pushing step from v to w is not saturating. In this case, if the current edge of v is not $\{v, w\}$ at the beginning of pass $i-1$, we charge the increase in $e(r)$ to the event of $\{v, w\}$ becoming the current edge of v. Over the entire algorithm there are at most nm such charges. If the current edge of v is $\{v, w\}$ at the beginning of pass $i-1$ but the tree T_v changes between the beginning of pass $i-1$ and the push from v to w, we charge the increase in $e(r)$ to the last link or cut changing T_v that occurs in pass $i-1$ before the push from v to w. Over the entire algorithm there are at most $3nm$ such charges, one per link and two per cut.

We have narrowed the situation down to one final case: T_w is unchanged from the beginning of pass $i-1$ until the push from w to r in pass i, $\{v, w\}$ is the current edge of v from the beginning of pass $i-1$ until the push from v to w in pass $i-1$, and T_v is unchanged from the beginning of pass $i-1$ until the push from v to w. In this case we change the increase in $e(r)$ to the pair of trees T_v, T_w existing at the beginning of pass $i-1$.

Let us count the number of such pairs of trees for a fixed pass $i-1$. A tree T existing at the beginning of pass $i-1$ can participate as T_w in only one such pair T_v, T_w, since only one push is responsible for adding w to Q in pass $i-1$, and w is uniquely determined by r. Tree T can also participate as T_v in only one pair T_v, T_w, since v is the root of T_v and $\{v, w\}$ is the current edge of v at the beginning of pass $i-1$, which means that the choice of T_v uniquely determines T_w. It follows that the sum of $|T_v| + |T_w|$ over all pairs T_v, T_w is at most $2n$. But the condition "$find\ size(v) + find\ size(w) > k$ in case (b) implies that $|T_v| + |T_w| > k$ for any pair T_v, T_w, which means that the number of such pairs is at most $2n/k$.

Summing the number of tree pairs over all passes, we find that there are at most $4n^2(n-1)/k$. Combining this with our estimates for the other cases, we obtain a bound of at most $7nm + (n-1)^2 + 4n^2(n-1)/k$ excess increases caused by case (c). We can derive a similar bound for case (b) by the same method. •

Lemma 14. The number of times a non-root has its excess made positive is $O(nm + n^3/k)$.

Proof. Let w be a non-root that has its excess made positive. This can happen only in case (b) of tree push/relabel, as a result of a push from a tree root v to w. Only nm such pushes can be saturating. Suppose the push from v to w is non-saturating. There can only be one such push per vertex v removed from Q. Suppose that v was a non-root when it was added to Q. Then we charge the push from v to w to the cut that made v a root after its addition to Q. This leaves only the case that v was a root when it was added to Q. A root can be added to Q only after $d(v)$ increases, which happens at most $(n-1)^2$ times, or as a result of $e(v)$ increasing from zero, which by Lemma 13 happens $O(nm + n^3/k)$ times. Combining all cases gives the lemma. •

Lemma 15. The number of additions of a vertex to Q is $O(nm + n^3/k)$.

Proof. A vertex v can be added to Q only after $d(v)$ increases, which happens at most $(n-1)^2$ times, or as a result of $e(v)$ increasing from zero, which happens $O(nm + n^3/k)$ times by Lemmas 13 and 14. •

Theorem 6. The algorithm with dynamic trees runs

in $O(nm\log(n^2/m))$ time if k is chosen equal to n^2/m.

Proof. Immediate from Lemmas 12 and 15. •

6. Distributed and Parallel Implementation

The parallel version of our algorithm is a modification of the algorithm of Section 4. We make three changes in the algorithm. First, we restrict the algorithm so that it stops processing a vertex 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 into a vertex w during a parallel step is not added to $e(w)$ until the end of the step. To be more precise, the parallel version consists of repeating the following step until every vertex $v \neq t$ has $e(v)=0$ or $d(v) \geq n$:

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

The parallel algorithm is almost a special case of the algorithm in Section 4, the difference being in the values used in relabeling and flow excess computations: in the algorithm of Section 4, relabeling steps in pass i use the most recent label and excess values, some of which may have been computed earlier in pass i. Nevertheless, a proof just like that of Lemma 11 gives the following analogous result for the parallel algorithm:

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

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

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

Each vertex processed during a pulse sends updated flow values to the appropriate neighbors. If a vertex label has changed during a pulse, it is also transmitted to neighbors, but only at the end of the pulse. Since flow always travels from larger to smaller labels, this delaying of the label broadcasting until the end of a pulse guarantees that flow only travels through an edge in one direction during a pulse. An easy analysis shows that in the synchronous case the distributed algorithm takes $O(n^2)$ rounds of message-passing and a total of $O(n^3)$ messages. Awerbuch (private communication, 1985) has observed that in the asynchronous case the synchronization protocol of [1] can be used to implement the algorithm in $O(n^2\log n)$ rounds and $O(n^3)$ messages. The same bounds can be obtained for the Shiloach-Vishkin algorithm [19], but only by allowing more memory per processor: the processor at a vertex v needs $O(n\,\Delta_v)$ storage. Vishkin (private communication) has reduced the space required by this algorithm to a total of $O(n^2)$ (from $O(nm)$). Nevertheless, our algorithm has an advantage in situations where memory is at a premium.

For parallel implementation, our computing model is a PRAM [8] without concurrent writing. The implementation in this model is very similar to that of the distributed implementation, except that computations on binary trees must be performed to allow each vertex to access its incident edges fast. Leafs of the binary trees correspond to edges incident to a vertex. The information about flow through an edge can be stored at the corresponding leaf processors. This implementation requires $O(m)$ processors with $O(1)$ memory per processor. 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 [19] apply to our algorithm to show that $O(n)$ processors suffice to obtain the $O(n^2\log n)$ time bound.

7. Remarks

Our concluding remarks concern three issues: (i) better bounds, (ii) exact 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 non-saturating pushes, whereas the bottleneck in the parallel version is the saturating pushes. We wonder whether an $O(nm)$ sequential time bound can be obtained through more careful handling of the non-saturating pushes, possibly avoiding the use of the dynamic tree data structure. Perhaps also an $O(m(\log n)^k)$ parallel time bound can be obtained through use of a parallel version of the dynamic tree data structure.

On graphs with integer edge capacities bounded by N, Gabow's scaling algorithm [9] runs in $O(nm\log N)$ sequential time. It consists of $\log N$ applications of Dinic's algorithm, each taking

$O(nm)$ time. We would like to know whether some version of our algorithm runs in $O(nm \log N)$ time on this class of networks. A variant that suggests itself is to always push flow from the vertex with largest excess.

It is possible to modify the first-in first-out algorithm so that when a push step is executed, the distance labels are exact distances to the sink in the residual graph. The modification involves a stronger interpretation of the next edge of a vertex, which requires the next edge to be unsaturated and point to a vertex with a smaller label. If a push step saturates the next edge of v, a new next edge must be found by scanning the edge list of v and relabeling if the end of the list is reached, like in the push/relabel step. If a relabeling step changes the label of v, next edges must be updated for all vertices u such that (u, v) is the next edge of u. It is easy to see that the above computations take $O(nm)$ time during the first stage of the algorithms.

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

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

Acknowledgements

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 his support.

Appendix. Finding Sizes of Dynamic Trees

The dynamic tree implementation of [22,23] does not support *find size* operations, but can be easily modified to do so, as follows. We assume some familiarity with [22] or [23]. The implementation represents each dynamic tree by a *virtual tree* having the same vertex set but different structure. To allow find size operations to be performed efficiently, we store with each virtual tree root its tree size, and with every other vertex the difference between its virtual subtree size and that of its virtual parent. This information is easy to update after each dynamic tree operation; the updating increases the time per operation by only a constant factor. A *find size* operation is performed just like a *find root* operation: *find root*(v) has the side effect of locating the root of the virtual tree containing v, which contains the tree size.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] V. M. Malhotra, M. Pramodh Kumar, and S. N. Mahcshwart, "An $O(|V|^3)$ algorithm for finding maximum flows in networks," *Inform. Process. Lett.* 7 (1978), 277-278.

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

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

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

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

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

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

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

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

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