IMPROVED ALGORITHMS FOR BIPARTITE NETWORK FLOW

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

Abstract. In this paper, we study network flow algorithms for bipartite networks. A network $G = (V, E)$ is called *bipartite* if its vertex set V can be partitioned into two subsets V_1 and V_2 such that all edges have one endpoint in V_1 and the other in V_2. Let $n = |V|$, $n_1 = |V_1|$, $n_2 = |V_2|$, $m = |E|$ and assume without loss of generality that $n_1 \leq n_2$. We call a bipartite network *unbalanced* if $n_1 \ll n_2$ and *balanced* otherwise. (This notion is necessarily imprecise.) We show that several maximum flow algorithms can be substantially sped up when applied to unbalanced networks. The basic idea in these improvements is a *two-edge push rule* that allows us to "charge" most computation to vertices in V_1, and hence develop algorithms whose running times depend on n_1 rather than n. For example, we show that the two-edge push version of Goldberg and Tarjan's FIFO preflow push algorithm runs in $O(n_1 m + n_1^3)$ time and that the analogous version of Ahuja and Orlin's excess scaling algorithm runs in $O(n_1 m + n_1^2 \log U)$ time, where U is the largest edge capacity. We also extend our ideas to dynamic tree implementations, parametric maximum flows, and minimum-cost flows.

Key words. Network flow, bipartite graphs, maximum flow, minimum-cost flow, parametric maximum flow, parallel algorithms

AMS subject classifications. 90B10 , 68Q25 , 68R10

1. Introduction.

In this paper, we study network flow algorithms for bipartite networks. A network $G = (V, E)$ is called *bipartite* if its vertex set V can be partitioned into two subsets V_1 and V_2 such that all edges have one endpoint in V_1 and the other in V_2. Let $n = |V|$, $n_1 = |V_1|$, $n_2 = |V_2|$, $m = |E|$, and assume without loss of generality that $n_1 \leq n_2$. We call a bipartite network *unbalanced* if $n_1 \ll n_2$ and *balanced* otherwise. We show that several maximum flow algorithms can be substantially sped up when applied to *unbalanced networks.* At first glance, it may appear that unbalanced networks are of limited practical utility. This is not true, however. Gusfield, Martel, and Fernandez-Baca [21] have compiled a list of many practical applications of unbalanced networks. Further applications of unbalanced networks appear in [14].

Specialized bipartite flow algorithms for unbalanced networks were first stud-

* Department of Industrial and Management Engineering, Indian Institute of Technology, Kanpur 208016, India. Research partially supported by 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 Computers, Inc., and Prime Computer.

† Sloan School of Management, M.I.T., Cambridge, MA 02139. Research partially supported by 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 Computers, Inc., and Prime Computer.

‡ Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755 Some of the results in this paper were part of this author's undergraduate thesis at Princeton University [35]. Some of the work was done while this author was a graduate student at the Laboratory for Computer Science, M.I.T., Cambridge, MA 02139. Research partially supported by a graduate fellowship from AT&T. Additional support provided by Air Force Contract AFOSR-86-0078 and by an NSF PYI Grant awarded to David Shmoys, with matching funds from IBM, Sun Microsystems, and UPS.

§ Department of Computer Science, Princeton University, Princeton, NJ 08544, and NEC Research Institute, Princeton, NJ 08540. Research at Princeton University partially supported by the National Science Foundation, Grant DCR-8605952, and the Office of Naval Research, Contract N00014-91-K-1463.

ied by Gusfield, Martel, and Fernandez-Baca [21]. They developed modifications of the algorithms of Karzanov [25] and Malhotra, Pramodh Kumar, and Maheshwari (MPM)[27] for the maximum flow problem that improved their running times from $O(n^3)$ to $O(n_1^2 n_2)$. For the bounded degree case, i.e., when the degree of each vertex in V_2 is bounded by a fixed constant, they developed a further modification of the MPM algorithm that runs in $O(n_1 m + n_1^3)$ time. We suggest several algorithms for the maximum flow problem on unbalanced networks that improve the running times of Gusfield et al. for all classes of unbalanced networks.

Gusfield [20] has shown that on a particular bipartite network in which each vertex in V_2 has constant degree, an algorithm similar to the FIFO preflow push maximum flow algorithm of Goldberg and Tarjan [15],[16] runs in $O(n_1 m + n_1^3)$ time. Further, he observes that this result extends to parametric maximum flow; he solves a series of n_1 maximum flow problems in $O(n_1 m + n_1^3)$ time. We have similar results, which were obtained independently and apply to a more general class of networks.

We begin with the observation of Gusfield et al.[21] that the time bounds for several maximum flow algorithms automatically improve when the algorithms are applied *without modification* to unbalanced networks. A careful analysis of the running times of these algorithms reveals that the worst-case bounds depend on the number of edges in the longest vertex-simple path in the network. We call this the *path length* of the network and denote it by L. For a general network, L may be as large as $n - 1$; but, for a bipartite network, L is at most $2n_1 + 1$. Hence for unbalanced networks the path length is much less than n, and we get an automatic improvement in running times. As an example, consider Dinic's algorithm [10] for the maximum flow problem. This algorithm constructs $O(L)$ layered networks and finds a blocking flow in each one. Each blocking flow computation performs $O(m)$ augmentations and each augmentation takes $O(L)$ time. Consequently, the running time of Dinic's algorithm is $O(L^2 m)$. Thus, when applied to unbalanced networks, the running time of Dinic's algorithm improves from $O(n^2 m)$ to $O(n_1^2 m)$. Column 3 of Table 1.1 summarizes these improvements for several network flow algorithms.

We obtain further running-time improvements by modifying the algorithms. This modification applies only to preflow push algorithms [2, 3, 14, 15, 16, 17]; we call it the *two-edge push rule*. According to this rule, we always push flow from a vertex in V_1 and push flow on two edges at a time, in a step called a *bipush*, so that no excess accumulates at vertices in V_2. This rule allows us to charge all computations to examinations of vertices in V_1, though without this rule they might be charged to vertices in V_2. As an outcome of this rule, we develop algorithms whose running times depend on n_1 rather than n. We incorporate the two-edge push rule in several maximum flow algorithms, dynamic tree implementations, a parametric maximum flow algorithm, and algorithms for the minimum-cost flow problem. Column 4 of Table 1.1 summarizes the improvements obtained using this approach.

In the presentation to follow, we assume some familiarity with preflow push algorithms and we omit many details, since they are straightforward modifications of known results. The reader interested in further details is urged to consult the appropriate paper or papers discussing the corresponding result for general networks or one or both of the survey papers [1],[18].

2. Preliminaries.

2.1. Network Definitions. Let $G = (V, E)$ be a directed bipartite network. We associate with each edge (v, w) in E a finite real-valued *capacity* $u(v, w)$. Let $U = \max\{u(v, w) : (v, w) \in E\}$. Let source s and sink t be the two distinguished

Algorithm	**Running time, general network**	**Running time, bipartite network**	**Running time, modified version**
Maximum Flows			
Dinic[10]	$n^2 m$	$n_1^2 m$	does not apply
Karzanov[25]	n^3	$n_1^2 n$[21]	$n_1 m + n_1^3$
MPM[27]	n^3	$n_1^2 n$[21]	does not apply
FIFO preflow push [15],[16]	n^3	$n_1^2 n$	$n_1 m + n_1^3$
Highest label preflow push[7]	$n^2\sqrt{m}$	$n_1 n\sqrt{m}$	$n_1 m + \min\{n_1^3, n_1^2\sqrt{m}\}$
Excess scaling[2]	$nm + n^2 \log U$	$n_1 m + n_1 n \log U$	$n_1 m + n_1^2 \log U$
Wave scaling [3]	$nm + n^2\sqrt{\log U}$	$n_1 m + n_1 n\sqrt{\log U}$	$n_1 m + n_1^2\sqrt{\log U}$
FIFO w/ dynamic trees [15],[16]	$nm\log(\frac{n^2}{m})$	$n_1 m\log(\frac{n^2}{m})$	$n_1 m\log(\frac{n_1^2}{m}+2)$
Parallel excess scaling[2]	$n^2\log U\log(\frac{m}{n})$, $\lceil m/n\rceil$ processors	$n_1 n\log U\log(\frac{m}{n})$, $\lceil m/n\rceil$ processors	$n_1^2\log U\log(\frac{m}{n_1})$, $\lceil m/n_1\rceil$ processors
Parametric Flows			
GGT[14]	n^3	$n_1 n^2$	$n_1^2 n$
GGT w/ dynamic trees [14]	$nm\log(\frac{n^2}{m})$	$n_1 m\log(\frac{n^2}{m})$	$n_1 m\log(\frac{n_1^2}{m}+2)$
Min-Cost Flows			
Cost scaling [17]	$n^3\log(nC)$	$n_1^2 n\log(n_1 C)$	$n_1 m + n_1^3\log(n_1 C)$
Cost scaling w/ dynamic trees [17]	$nm\log(\frac{n^2}{m})\cdot\log(nC)$	$n_1 m\log(\frac{n^2}{m})\cdot\log(n_1 C)$	$n_1 m\log(\frac{n_1^2}{m}+2)\cdot\log(n_1 C)$

TABLE 1.1

A summary of the results discussed in this paper. Column 2 contains previously known results for general graphs. Column 3 gives bounds on bipartite networks based on the improved bound on L. Column 4 gives our new results based on the two-edge push rule.

vertices in the network. We make the assumption that $s \in V_2$ and $t \in V_1$. We further assume, without loss of generality, that if (v, w) is in E then so is (w, v), and that the network contains no parallel edges. We define the *edge incidence list* $I(v)$ of a vertex $v \in V$ to be the set of edges directed out of vertex v, i.e., $I(v) = \{(v, w) : (v, w) \in E\}$.

2.2. Flow. A *flow* is a function $f : E \to \mathbf{R}$ satisfying

$$f(v,w) \le u(v,w)\ ,\ \ \forall (v,w) \in E \tag{2.1}$$

$$f(v,w) = -f(w,v)\ ,\forall (v,w) \in E \tag{2.2}$$

$$\sum_{v\in V} f(v,w) = 0\ ,\quad \forall w \in V - \{s,t\}. \tag{2.3}$$

The *value* of a flow is the net flow into the sink, i.e.,

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

The *maximum flow problem* is to determine a flow f for which $|f|$ is maximum.

2.3. Preflow. A *preflow* is a function $f : E \to \mathbf{R}$ that satisfies conditions (2.1), (2.2), and the following relaxation of condition (2.3):

$$\sum_{v\in V} f(v,w) \ge 0,\ \ \forall w \in V - \{s\}. \tag{2.4}$$

```
procedure preprocess
begin
        f = 0;
        push u(s,v) units of flow on each edge (s,v) ∈ I(s);
        compute the exact distance label function d by
             backward breadth-first searches from t and from s
             in the residual network;
end
procedure push/relabel(v)
begin
        if there is an eligible edge (v,w)
        then
               begin select an eligible edge (v,w);
                      push δ = min{e(v), u_f(v,w)} units of flow from v to w
               end
        else replace d(v) by min{d(w) + 1 : (v,w) ∈ I(v) and u_f(v,w) > 0}
end
```

FIG. 3.1. *Two procedures for the generic preflow push algorithm*

The maximum flow algorithms described in this paper maintain a preflow during the computation. For a given preflow f, we define, for each vertex $w \in V$, the *excess* $e(w) = \sum_{v \in V} f(v,w)$. A vertex other than t with strictly positive excess is called *active*.

2.4. Residual Capacity. With respect to a preflow f, we define the *residual capacity* $u_f(v,w)$ of an edge (v,w) to be $u_f(v,w) = u(v,w) - f(v,w)$. The *residual network* is the network consisting only of edges that have positive residual capacity.

2.5. Distance Labels. A distance function $d : V \to \mathbf{Z}^+ \cup \{\infty\}$ with respect to the residual capacities $u_f(v,w)$ is a function mapping the vertices to the non-negative integers. We say that a distance function is *valid* if $d(s) = 2n_1$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every edge (v,w) in the residual network. We call a residual edge with $d(v) = d(w) + 1$ *eligible*. The eligible edges are exactly the edges on which we push flow.

We refer to $d(v)$ as the *distance label* of vertex v. It can be shown that if the distance labels are valid, then each $d(v)$ is a lower bound on the length of the shortest path from v to t in the residual network. If there is no directed path from v to t, however, then $d(v)$ is a lower bound on $2n_1$ plus the length of the shortest path from v to s. If, for each vertex v, the distance label $d(v)$ equals the minimum of the length of the shortest path from v to t and $2n_1$ plus the length of the shortest path from v to s, then we call the distance labels *exact*.

3. The Generic Preflow Push Algorithm on Bipartite Networks. All maximum flow algorithms described in this paper are *preflow push algorithms*, i.e., algorithms that maintain a preflow at every stage. They work by examining active vertices and pushing excess from these vertices to vertices estimated to be closer to t. If t is not reachable, however, an attempt is made to push the excess back to s. Eventually, there will be no excess on any vertex other than t. At this point the preflow is a flow, and moreover it is a maximum flow[15],[16]. The algorithms use distance labels to measure the closeness of a vertex to the sink or the source.

The generic preflow push algorithm consists of a preprocessing stage followed by repeated application of a procedure called *push/relabel*. These two procedures appear in Figure 3.1.

Increasing the flow on an edge is called a *push* through the edge. We say a push of

algorithm *preflow-push*
begin
    preprocess;
    while the network contains an active vertex **do**
    begin
        select an active vertex v;
        push/relabel(v)
    end
end

FIG. 3.2. *Algorithm* preflow-push

δ units of flow on edge (v, w) is *saturating* if $\delta = u_f(v, w)$ and *nonsaturating* otherwise. A nonsaturating push at vertex v reduces $e(v)$ to zero. We refer to the process of increasing the distance label of a vertex as a *relabel* operation. The purpose of the relabel operation is to create at least one eligible edge on which the algorithm can perform further pushes.

Not specified in Figure 3.1 is an efficient way to choose edges for pushing steps. We assume the same mechanism as that proposed by Goldberg and Tarjan [15],[16]. The algorithm maintains the incidence list $I(v)$ for each vertex v, and a pointer into each such list indicating a *current edge*. Initially the current edge of each incidence list is the first edge on the list. To perform *push/relabel*(v), the current edge pointer for v is moved through the list $I(v)$ until it indicates an eligible edge or it reaches the end of the list. In the former case, a push is done on the current edge. In the latter case, a relabel of v is done and the pointer is reset to indicate the first edge on $I(v)$. Figure 3.2 contains the algorithm *preflow-push*, which combines the two subroutines of Figure 3.1. At the termination of the algorithm, each vertex in $V - \{s, t\}$ has zero excess; thus the final preflow is a flow. It is easy to establish that this flow is maximum. We shall briefly discuss the worst-case time complexity of the algorithm. (We refer the reader to the paper of Goldberg and Tarjan [16] for a complete discussion of the algorithm.)

We begin by stating two lemmas from [15],[16].

LEMMA 3.1. [15],[16] *The generic preflow push algorithm maintains valid distance labels at each step. Moreover, each relabeling of a vertex v strictly increases $d(v)$.*

LEMMA 3.2. [15],[16] *At any time during the preflow push algorithm, for each vertex v with positive excess, there is a directed path from vertex v to vertex s in the residual network.*

Now we can derive the necessary results specific to bipartite networks.

COROLLARY 3.3. *For each active vertex v, $d(v) \leq 4n_1$.*

Proof. When a vertex v is relabeled, it has positive excess, and hence the residual network contains a path P from v to s. Since the vertices on this path are alternately in V_1 and V_2, the maximum possible length of the path is $2n_1$. Since $d(s) = 2n_1$ and, for every edge (w, x) on P, $d(w) \leq d(x) + 1$, it must be the case that $d(v) \leq d(s) + 2n_1 = 4n_1$. □

COROLLARY 3.4. *The number of relabel steps is $O(n_1 n)$. Further, the time spent performing relabels is $O(n_1 m)$. The time spent scanning edges while finding eligible edges on which to push flow is also $O(n_1 m)$.*

Proof. The first statement follows directly from Lemma 3.1 and Corollary 3.3. The second statement follows from the fact that in order to relabel a vertex v, we must look at all of the edges in $I(v)$. Hence, we can bound the total relabeling time by $O((\sum_{v \in V} |I(v)|)(4n_1)) = O(n_1 m)$. The same bound holds for the time spent finding

```
procedure bipush/relabel(v)
begin
    if there is an eligible edge (v, w)
    then
        begin select an eligible edge (v, w);
            if there is an eligible edge (w, x)
            then
                begin select an eligible edge (w, x);
                    push δ = min{e(v), u_f(v, w), u_f(w, x)} units of flow
                        along the path v − w − x
                end
            else replace d(w) by min{d(x) + 1 : (w, x) ∈ I(w) and u_f(w, x) > 0}
        end
    else replace d(v) by min{d(w) + 1 : (v, w) ∈ I(v) and u_f(v, w) > 0}
end
```

FIG. 3.3. *The procedure* bipush/relabel

edges on which to push flow. □

COROLLARY 3.5. *The preflow push algorithm performs $O(n_1 m)$ saturating pushes.*

Proof. Between two consecutive saturating pushes on an edge (v, w), both $d(v)$ and $d(w)$ must increase by 2. By Lemma 3.1 and Corollary 3.3, only $O(n_1)$ saturating pushes can be done on (v, w). Summing over all edges gives the bound. □

LEMMA 3.6. *The preflow push algorithm performs $O(n_1^2 m)$ nonsaturating pushes.*

Proof. Omitted. (Analogous to the proof of Lemma 3.10 in [16].) □

The results in Column 3 of Table 1.1 for preflow push algorithms all follow from the known results by using Corollaries 3.4 and 3.5 to replace certain $O(n)$ bounds in the general case with $O(n_1)$ bounds in the bipartite case. Since all these results are straightforward to obtain and are dominated by those in Column 4, we omit their derivations and move on to the more interesting results in Column 4.

4. The Bipartite Preflow Push Algorithm. The basic idea behind the bipartite preflow push algorithm is to perform bipushes from vertices in V_1. A *bipush* is a push over two consecutive eligible edges; it moves excess from a vertex in V_1 to another vertex in V_1. This approach has all the advantages of the usual approach, and the additional advantage that it leads to improved running times. This approach ensures that no vertex in V_2 ever has any excess. Since all the excess resides at vertices in V_1, it suffices to account for the nonsaturating bipushes emanating from vertices in V_1. Since $|V_1| \leq |V_2|$, the number of nonsaturating bipushes is reduced.

The bipartite preflow push algorithm is a simple generalization of the generic preflow push algorithm. The bipartite algorithm is the same as the generic algorithm given in Section 3 except that the procedure *bipush/relabel* appearing in Figure 3.3 replaces the procedure *push/relabel* in the original algorithm. The algorithm identifies eligible edges emanating from a vertex using the current edge data structure described earlier.

We call a push of δ units on the path $v - w - x$ a *bipush*. The bipush is *saturating* if $\delta = \min\{u_f(v, w), u_f(w, x)\}$ and *nonsaturating* otherwise. Observe that a nonsaturating bipush reduces the excess at vertex v to zero. The following lemma is an easy consequence of the two-edge push rule implemented in *bipush/relabel.*

LEMMA 4.1. *During the execution of the bipartite preflow push algorithm, all excess remains on the vertices in V_1.*

Proof. The first thing the algorithm does is to saturate all edges leaving s. Since

$s \in V_2$, the claim is true immediately after this step. All the other pushes in the algorithm are done using the procedure *bipush/relabel*, which pushes from a vertex in V_1 through a vertex in V_2 to another vertex in V_1, never leaving any excess on a vertex in V_2. No other operations create excess at any vertex. □

As in the original preflow push algorithm, the bipartite preflow push algorithm always pushes flow on eligible edges and relabels a vertex only when there are no eligible edges emanating from it. Hence Lemma 3.1 holds for this algorithm too. Lemma 3.2 also holds. Corollary 3.3 holds for vertices in V_1, but a modified version holds for vertices in V_2: if $v \in V_2$, then either $d(v) \leq 4n_1 + 1$ or $d(v) = \infty$. Corollary 3.4 holds as stated. Corollary 3.5 translates into a bound of $O(n_1 m)$ saturating bipushes. The Lemma 3.6 bound of $O(n_1^2 m)$ on nonsaturating pushes becomes a bound of $O(n_1^2 m)$ on nonsaturating bipushes. Thus we get the following result:

THEOREM 4.2. *The bipartite preflow push algorithm runs in $O(n_1^2 m)$ time.*

We now define the concept of a *vertex examination*. In an iteration, the generic bipartite preflow push algorithm selects an active vertex v and performs a saturating bipush or a nonsaturating bipush or relabels a vertex. In order to develop more efficient algorithms, we incorporate the rule that whenever the algorithm selects an active vertex $v \in V_1$, it keeps pushing flow from that vertex until either its excess becomes zero or it is relabeled. Consequently, there may be several saturating bipushes followed either by a nonsaturating bipush or a relabel operation; there will in general also be relabelings of vertices in V_2. We associate this sequence of operations with a vertex examination. We shall henceforth assume that the bipartite preflow push algorithm follows this rule.

5. Specific Implementations of the Bipartite Preflow Push Algorithm. The bottleneck in the bipartite preflow push algorithm is the time spent doing nonsaturating bipushes. There are two orthogonal approaches to reducing this time. One approach is to reduce the number of nonsaturating bipushes by selecting the vertices for *bipush/relabel* operations cleverly. We shall consider several such selection rules in Sections 5.1–5.4. The second approach is to reduce the time spent per nonsaturating bipush. The idea is to use a sophisticated data structure in order to push flow along a whole path in one step, rather than pushing flow along a single pair of edges. We shall study this approach in Section 5.5. Finally, in Section 5.6 we study a parallel implementation of one version of the bipartite preflow push algorithm.

5.1. The First-In First-Out (FIFO) Algorithm. The FIFO preflow push algorithm examines active vertices in first-in, first-out (FIFO) order. The algorithm maintains a queue Q of active vertices. It selects a vertex v from the front of Q and performs pushes from v while adding newly active vertices to the rear of Q. The algorithm examines v until either it becomes inactive or it is relabeled. In the latter case, v is added to the rear of Q. The algorithm terminates when Q is empty. Goldberg and Tarjan [17] showed that the FIFO algorithm performs $O(n^3)$ nonsaturating pushes. We show, using a similar analysis, that the number of nonsaturating bipushes in the bipartite case is $O(n_1^3)$.

For the purpose of the analysis, we partition the sequence of vertex examinations into several *passes*. The first pass consists of examining the vertices that become active during the *preprocess* step. For $k \geq 2$, the k^{th} pass consists of examining all vertices that were added to the queue during the $k - 1^{st}$ pass.

LEMMA 5.1. *The number of passes over Q is $O(n_1^2)$.*

Proof. Let $\Phi = \max\{d(v) | v \text{ is active}\}$. The initial value of Φ is at most $4n_1$. Consider the effect that a pass over Q can have on Φ. If, during the pass, no vertex in

V_1 is relabeled, then the excess at every vertex is pushed to a vertex with a distance label smaller by at least two, and consequently Φ decreases by at least two. If some vertex in V_1 is relabeled during the pass, however, then Φ can increase or remain the same. In such a case the increase in Φ is bounded by the largest increase in any distance label. Hence, by Corollary 3.3, the total increase in Φ over all passes is at most $4n_1^2$. Consequently, the total number of passes is $O(n_1^2)$. □

Now observe that any pass examines each vertex in V_1 at most once and each vertex examination performs at most one nonsaturating bipush. Consequently, the algorithm performs $O(n_1^3)$ nonsaturating bipushes. We noted in the previous section that all other operations take $O(n_1 m)$ time. Thus we obtain the following result:

THEOREM 5.2. *The bipartite FIFO preflow push algorithm runs in $O(n_1 m + n_1^3)$ time.*

We note that this bound is also achieved by Karzanov's algorithm [25] if it is implemented using the two-edge push rule. A modification of Karzanov's algorithm by Tarjan [36], which he calls the *wave algorithm*, also has the same time bound. The analysis of both of these algorithms is straightforward and hence omitted.

5.2. The Highest-Label Preflow Push Algorithm. The highest-label preflow push algorithm always pushes from an active vertex with highest distance label. This rule can be implemented using a simple bucketing approach so that the overhead for vertex selection is $O(n_1^2)$. The nonsaturating bipushes performed by the algorithm can be divided into passes. A *pass* consists of all bipushes that occur between two consecutive relabel steps of vertices in V_1. Within a pass, vertices in V_2 can possibly be relabeled several times. Notice that in this algorithm, excesses that are most distant from the sink are pushed down two levels at a time. Consequently, if the algorithm does not relabel any vertex during n_1 consecutive vertex examinations, all excess reaches the sink and the algorithm terminates. Since the algorithm performs $O(n_1^2)$ relabel operations on vertices in V_1, we immediately obtain a bound of $O(n_1^3)$ on the number of vertex examinations. As each vertex examination entails at most one nonsaturating bipush, this gives a bound of $O(n_1^3)$ on the number of nonsaturating bipushes and a bound of $O(n_1 m + n_1^3)$ on the running time of the algorithm.

Cheriyan and Maheshwari [7] showed by a clever argument that the highest label preflow push algorithm performs $O(n^2\sqrt{m})$ nonsaturating pushes for general networks. Modifying their argument to fit the bipartite case, we obtain a running time of $O(n_1 m + \min\{n_1^3, n_1^2\sqrt{m}\})$. This improves the above bound of $O(n_1 m + n_1^3)$ if $\sqrt{m} < n_1$. We shall give a potential-based argument that is slightly different from the analysis of Cheriyan and Maheshwari.

We focus on the set of edges that are both current and eligible; we call these edges *live*. Recall that an edge (v, w) is eligible if it has positive residual capacity and $d(v) = d(w) + 1$; (v, w) is current if the current edge pointer for vertex v indicates (v, w). Each vertex has at most one outgoing live edge, and the live edges form no cycles since $d(v) > d(w)$ if (v, w) is a live edge. Thus the set of live edges defines a forest, which we call the *live forest*. We call an active vertex *maximal* if it has no active proper descendant in the live forest. For a vertex v, let $desc(v)$ be the number of descendants of v in the live forest, including v itself, that are in V_1. Let p be a positive integer parameter whose value we shall choose later. For a maximal active vertex v, we define the *uncounted cost* $c(v)$ of v to be $\min\{0, desc(v) - p\}$. For any vertex v that is not maximal active, we define $c(v) = 0$. We use the sum $\sum_{v \in V_1} c(v)$ to help bound the number of nonsaturating bipushes.

We wish to count nonsaturating bipushes. Our strategy is to charge nonsaturating bipushes against changes in current edges, relabelings, increases in the total uncounted cost, and certain other events. We shall obtain an overall bound of $O(n_1mp+n_1^3/p)$ on the number of nonsaturating bipushes. Choosing $p = \max\{1, \lceil n_1/\sqrt{m}\,\rceil\}$ then gives a bound of $O(\min\{n_1m + n_1^3, n_1^2\sqrt{m}\})$ on the number of nonsaturating bipushes.

Define a *pass* of the algorithm to be a maximal interval of time during which all vertices selected for *bipush/relabel* steps have the same distance label. A pass terminates either when a relabeling occurs or when all excess at vertices with maximum distance label is moved to vertices of distance label lower by two.

LEMMA 5.3. *The total number of nonsaturating bipushes is $O(n_1mp + n_1^3/p)$.*

Proof. An argument like that in Lemma 5.1 shows that the total number of passes is $O(n_1^2)$. Consider the nonsaturating bipushes that occur during a pass. Every vertex from which a bipush occurs is maximal active. For a vertex v, call a nonsaturating bipush from v *large* if $c(v) = 0$ before the bipush and *small* otherwise. Two vertices v and w from which nonsaturating bipushes occur during the pass have disjoint sets of descendants in the live forest. If a large bipush occurs from a vertex v, v has at least p V_1-descendants before the bipush. Since the total number of vertices in V_1 is n_1, there can be at most n_1/p large bipushes during the pass.

The following argument shows that every small nonsaturating bipush causes an increase of at least one in the total uncounted cost. Consider such a bipush from a vertex v to a vertex x. The bipush causes vertex v to become inactive and may cause vertex x to become maximal active; no other vertex can become maximal active. If x becomes maximal active, the total uncounted cost increases by at least one, because $\mathrm{desc}(x) > \mathrm{desc}(v)$ and $desc(v) < p$. If x does not become maximal active, then the total uncounted cost still increases by at least one, since the negative term $desc(v)-p$ is removed from the total uncounted cost.

We conclude that there are $O(n_1^3/p)$ nonsaturating bipushes (the large ones) plus those accounted for by increases in the total uncounted cost. It remains to bound the sum of all increases in the total uncounted cost. The total uncounted cost remains between $-pn_1$ and zero. A nonsaturating bipush cannot decrease the total uncounted cost. A saturating bipush or a relabeling or a change in a current edge can reduce the total uncounted cost by at most $O(p)$, since any such operation affects only $O(1)$ maximal active vertices. We conclude that the sum of all decreases in the total uncounted cost is $O(n_1mp)$, and so is the sum of all increases in the total uncounted cost. The lemma follows. □

THEOREM 5.4. *The highest label preflow push algorithm runs in $O(n_1m + \min\{n_1^3, n_1^2\sqrt{m}\})$ time.*

Proof. Immediate from Lemma 5.3 by choosing $p = \max\{1, \lceil n_1/\sqrt{m}\,\rceil\}$. □

5.3. The Excess Scaling Algorithm. The *excess scaling* algorithm, due to Ahuja and Orlin [2], incorporates scaling of the excesses into the generic preflow push algorithm, thereby reducing the number of nonsaturating pushes from $O(n^2m)$ to $O(n^2 \log U)$. The basic idea is to push flow from active vertices with sufficiently large excess to vertices with sufficiently small excess while never letting the excesses become too large. We shall develop an adaptation of the excess scaling algorithm for bipartite networks, which we call the *bipartite excess scaling algorithm.* This algorithm, in contrast to the algorithms in Sections 5.1 and 5.2, requires that the edge capacities be integral.

Figure 5.1 describes the bipartite excess scaling algorithm. The algorithm uses the same *bipush/relabel* step as the generic bipartite preflow push algorithm but with

```
algorithm bipartite excess scaling
begin
        preprocess;
        Δ = 2^⌈log U⌉;
        while Δ ≥ 1 do
        begin
                while the network contains a vertex v ∈ V_1
                        with excess greater than Δ/2 do
                begin
                        among vertices with excess exceeding Δ/2,
                              select a vertex v with smallest distance label;
                        perform bipush/relabel(v)
                              (modified to ensure that no excess exceeds Δ)
                end;
                Δ = Δ/2
        end
end
```

FIG. 5.1. bipartite excess scaling *algorithm*

one slight difference. If $x \neq t$, instead of pushing $\delta = \min\{e(v), u_f(v,w), u_f(w,x)\}$ units of flow, it pushes
$\delta = \min\{e(v), u_f(v,w), u_f(w,x), \Delta - e(x)\}$ units, where Δ is a positive *excess bound* maintained by the algorithm. This change ensures that the algorithm permits no excess on an active vertex to exceed Δ units. Since Δ is integral until the algorithm terminates, all excesses remain integral, which implies that on termination only s and t can have non-zero excess. This implies that the algorithm is correct.

LEMMA 5.5. *The bipartite excess scaling algorithm maintains the following three invariants:*

1. *No vertex in V_2 ever has positive excess.*
2. *Any bipush that does not saturate an edge moves at least $\Delta/2$ units of flow.*
3. *No vertex ever has excess greater than Δ.*

Proof. Invariant 1 is satisfied because the bipartite excess scaling algorithm is a special case of the generic algorithm and the generic algorithm satisfies it. For Invariants 2 and 3, see [2, 3]. ◻

We can use these invariants to establish a bound on the number of nonsaturating bipushes. We define a *scaling phase* to be a maximal period of time during which Δ does not change.

LEMMA 5.6. *The bipartite excess scaling algorithm performs $O(n_1^2 \log U)$ nonsaturating pushes and runs $O(n_1 m + n_1^2 \log U)$ time.*

Proof. As in [3], we consider the potential function $\Phi = \sum_{v \in V} \frac{e(v)d(v)}{\Delta}$, which by Invariant 1 is the same as $\sum_{v \in V_1} \frac{e(v)d(v)}{\Delta}$. By Invariant 3, at the beginning of a scaling phase, $\Phi \leq 4n_1^2$. The actions of the algorithm consist of bipushes and relabels. We consider the two cases separately:
Case 1: A relabel occurs. If a vertex in V_2 was relabeled, Φ remains unchanged. If a vertex in V_1 was relabeled, Φ increases by at least one. By Corollary 3.3, such increases sum to $O(n_1^2)$. (This bound actually applies to the whole algorithm, not just one scaling phase.)
Case 2: A bipush occurs. This must decrease Φ. If the bipush is nonsaturating, then by Invariant 2, it moves at least $\frac{\Delta}{2}$ units of flow to a vertex with distance label two units lower, so Φ decreases by at least 1. As the initial value plus the total increase to

make-tree(v)	Make vertex v into a one-vertex dynamic tree.
find-root(v)	Return the root of v's tree.
find-size(v)	Return the number of vertices in v's tree.
find-value(v)	Return the value of the tree edge from v to its parent. Return ∞ if v is a root.
find-min(v)	Return the ancestor w of v with minimum *find-value*(w). In case of a tie, choose the w closest to the root. Choose v if v is the root.
change-value(v, z)	Add z to the value of every edge from v to *find-root*(v).
link(v, w, x)	Combine the trees containing v and w by making w the parent of v and giving edge (v, w) the value x. Do nothing if v and w are in the same tree or if v is not a root.
cut(v)	Break v's tree into two trees, by deleting the edge joining v and v's parent. Do nothing if v is a root.

FIG. 5.2. **Dynamic Tree Operations**

Φ are $O(n_1^2)$, Φ can decrease $O(n_1^2)$ per scaling phase, which means there are $O(n_1^2)$ nonsaturating pushes per scaling phase.

Observe that originally $\Delta < 2U$, where U is the maximum capacity in the network, and that when Δ decreases below 1, the algorithm terminates. In each scaling phase, Δ decreases by a factor of 2, so there are $O(\log U)$ scaling phases. Thus the total number of nonsaturating pushes is $O(n_1^2 \log U)$.

The running time of the algorithm is $O(n_1 m + n_1^2 \log U)$ plus the time required to select smallest-distance vertices for *push/relabel* steps. The bucket-based data structure described in [3] makes the total time for vertex selection $O(n_1 m + n_1^2 \log U)$. ☐

5.4. Variants of Excess Scaling. Ahuja, Orlin, and Tarjan [3] have developed two variants of the excess scaling algorithm that achieve improved time bounds. The faster of these, called the *wave scaling algorithm*, runs in $O(nm + n^2\sqrt{\log U})$ time. The idea of bipushes can easily be incorporated into both of their algorithms, thereby improving the running times for bipartite networks. The following theorem states the running time of the bipartite wave scaling algorithm.

THEOREM 5.7. *The bipartite wave scaling algorithm runs in* $O(n_1 m + n_1^2\sqrt{\log U})$ *time.*

The derivation of this time bound is similar to that of the excess scaling algorithm. The analysis of the original algorithm uses arguments based on potential functions defined over the vertex set V. For bipartite networks, we define the potential functions over the set V_1 and are able to replace n by n_1 in the running time. The detailed proof of this theorem is quite lengthy but contains no new ideas; therefore we omit it. A similar improvement can be obtained in Ahuja, Orlin, and Tarjan's less efficient algorithm, called the *stack scaling algorithm*.

5.5. Dynamic Trees. In the previous four sections, we reduced the time needed to compute a maximum flow by reducing the number of nonsaturating pushes. In this section, we consider a different approach: we reduce the time spent per nonsaturating push. The idea is to use a sophisticated data structure in order to push flow along a whole path in one step, rather than pushing flow along a single edge. The *dynamic tree* data structure of Sleator and Tarjan [34, 33, 37] is ideally suited for this purpose.

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each edge of which has an associated real value. We adopt the convention that tree edges are directed towards the root. We denote the parent of

```
procedure tree-push/relabel(v)
begin
        if there is an eligible edge (v, w)
        then
               begin link(v, w, u_f(v, w))
                      p(v) ← w
                      δ ← min{e(v), find-value(find-min(v))}
                      change-value(v, −δ)
(*)                   while v ≠ find-root(v) and find-value(find-min(v)) = 0 do
                             begin z ← find-min(v)
(**)                                cut(z)
                             end
               end
        else   begin replace d(v) by min{d(w) + 1 : (v, w) ∈ I(v) and u_f(v, w) > 0}
(†)                   for all children y of v do
(‡)                          cut(y)
               end
end
```

FIG. 5.3. **The** *tree-push/relabel* **operation**

v by $p(v)$ and regard each vertex as an ancestor and descendent of itself. We call a dynamic tree *trivial* if it contains only one V_2-vertex and *non-trivial* otherwise. The data structure supports the operations in Figure 5.2. It is shown in [34] that if the maximum number of vertices in any tree is k, we can perform an arbitrary sequence of l tree operations in $O(l \log k)$ time.

In maximum flow algorithms, the dynamic tree edges are a subset of the *current edges*. The value of a tree edge is its residual capacity. We maintain the invariant that every active vertex is a dynamic tree root. For this section, we relax the invariant that all excess is on vertices in V_1 and allow excess to accumulate on vertices in V_2.

The key to the dynamic tree implementation is the *tree-push/relabel* operation in Figure 5.3. The operation is applied to an active vertex v. If there is an eligible edge (v, w) then the operation adds (v, w) to the forest of dynamic trees, pushes as much flow as possible from v to the root of the tree containing w, and then deletes from the forest all edges which are saturated by this push. Otherwise, v is relabeled and its children are cut off. We refer to the operation of pushing flow from a node of a dynamic tree to the root as a *tree-push*.

The first dynamic tree algorithm we consider is just the generic *preflow-push* algorithm with the *push/relabel* operation replaced by the *tree-push/relabel* operation of Figure 5.3. We modify the initialization so that each vertex is in its own one-vertex dynamic tree and we add a post-processing step which extracts the correct flow on each edge that remains in a dynamic tree. We call this algorithm the *generic bipartite dynamic tree* algorithm.

The correctness of this algorithm is straightforward to verify (see [15], [16]). We show that this implementation yields an efficient algorithm.

LEMMA 5.8. *The number of* tree-push/relabel *operations done by the generic bipartite dynamic tree algorithm is $O(n_1 m)$.*

Proof. Each *tree-push/relabel* operation either relabels a vertex or pushes flow along a tree path. If it pushes flow then it must either saturate an edge or decrease the number of tree roots by one. By Corollaries 3.4 and 3.5 a relabeling or an edge saturation can occur at most $O(n_1 m)$ times. Furthermore the total increase in the number of tree roots caused by such operations is $O(n_1 m)$. Thus a push which decreases the number of tree roots by one can occur at most $O(n_1 m + n)$ times, which

```
procedure bi-cut(v)
begin
        if v ∈ V1
        then cut(p(v))
        cut(v)
end
```

FIG. 5.4. **The *bi-cut* operation**

is the sum of the number of times the number of tree roots can increase by one plus the number of initial tree roots. ∎

Recalling the assumption about vertex examinations that bounds the time spent deciding which vertex and edge to process, we get the following theorem:

THEOREM 5.9. *The generic bipartite dynamic tree algorithm runs in $O(n_1 m \log n)$ time.*

Proof. Each call to *tree push/relabel* does $O(1)$ dynamic tree operations and then executes the *while* loop in line (*) or the *for* loop in line (†) a number of times. Each execution of the *while* loop takes $O(1)$ dynamic tree operations, and the *while* loop is executed at most $O(n_1 m)$ times over the course of the whole algorithm, since each *cut* in line (**) corresponds to a saturating push. Similarly the *cut*s in line (‡) correspond to edges looked at while relabeling and by Corollary 3.4 there are only $O(n_1 m)$ of these. Thus the algorithm performs $O(n_1 m)$ dynamic tree operations. Since the maximum tree size is n, the algorithm takes $O(n_1 m \log n)$ time. ∎

Note that we have used the fact that the number of links, the number of cuts, the number of saturating pushes, and the relabeling time are all $O(n_1 m)$.

5.5.1. Further Improvements. While for many values of n, n_1, m, and U, the bound given by Theorem 5.9 is an improvement over those of the algorithms in the previous four sections, it is possible to use dynamic trees in a more sophisticated manner to achieve a running time of $O(n_1 m \log(\frac{n_1^2}{m} + 2))$. In order to realize this bound, we must overcome a few obstacles. First, as in [3] and [15, 16], we need to limit the tree size. Moreover, we need to make the tree size bound solely a function of n_1 rather than n. Finally, we must deal with the fact that a *cut* can make a V_2-vertex a tree root. This leaves open the possibility that a V_2-vertex will become active, thus violating one of the invariants we have previously maintained. We see no way to avoid this – instead we control how this happens and use a fairly complicated analysis to show that we can achieve the desired time bounds.

To ensure that the tree size is a function of n_1 and not n, we use the following:

LEMMA 5.10. *If all the leaves in a non-trivial dynamic tree are V_1-vertices, then the number of vertices in the tree is at most twice the total number of V_1-vertices in the tree.*

Proof. Since no V_2-vertex is a leaf, all V_2-vertices have at least one child. The graph is bipartite, which means that all these children must be V_1-vertices. Therefore, the total number of V_1 vertices in the tree must be at least as large as the total number of V_2-vertices. ∎

We will use two rules to enforce this invariant. First, if a *link* operation could make a V_2-vertex a leaf, we do not perform that *link*. This rule will be respected in all the procedures that follow. Second, if a *cut* causes a V_2-vertex to become a leaf, we immediately cut that vertex from the tree. This idea is implemented in procedure *bi-cut*, which appears in Figure 5.4. Procedure *bi-cut* will be used in place of *cut*. Observe that procedure *bi-cut* performs at most two dynamic tree operations.

```
procedure bi-send(v)
begin
        f ← find-root(v)
        if r ∈ V1
        then
                δ ← min{e(v), find-value(find-min(v))}
        else
(*)             δ ← min{e(v), find-value(find-min(v)), out-cap(r) − e(r)}
        change-value(v, −δ)
        while v ≠find-root(v) and find-value(find-min(v)) = 0 do
                begin z ← find-min(v)
                        bi-cut(z)
                end
end
```

FIG. 5.5. **The *bi-send* operation**

We also want to maintain the invariant that no tree have more than k vertices (k will be chosen later). As in [15, 16] we achieve this by preceding each *link* operation by a calculation of whether or not the result of the link will be a tree of greater than k vertices. If so, we do not perform the link. Since trees only grow as the result of *link* operations, it is clear that this maintains the desired invariant.

The main problem left to address is the complexity added by allowing excess to remain on V_2-vertices. In general, this yields slower running times. We maintain the following invariant, however:

INVARIANT 5.11. *Whenever a V_2-vertex is relabeled, it does not have any excess on it.*

As we shall see, this will allow us to get a good bound on the number of tree operations.

To maintain this invariant we need to ensure that we always have the flexibility to send all the excess from a V_2-vertex out over the current edge. The following lemma gives a condition sufficient to guarantee this flexibility:

LEMMA 5.12. *Let* out-cap(v) *be the residual capacity of the current edge of v. If for all V_2-vertices v that are dynamic tree roots, we maintain that*

$$e(v) \leq \text{out-cap}(v) \tag{5.1}$$

and that the current edge of v is eligible, then Invariant 5.11 can be satisfied with $O(1)$ additional work per tree-push or relabeling operation.

Proof. The left side of (5.1) can change when we do a push that involves v, and the right side can change when the current edge of v changes. We deal with these two cases separately. When doing a tree-push that terminates at a root r that is a V_2-vertex we must ensure that the new excess does not exceed *out-cap*(r). To do this we simply push less flow. This idea is captured in a new procedure called *bi-send*, which appears in Figure 5.5. This procedure will be used whenever we want to push flow along a path from a tree vertex to the root.

Next we have to deal with the case when *out-cap*(v) changes. Let (v, w) be the current edge of v. The value of *out-cap*(v) may change in two different ways. One way is that (v, w) may become saturated. When this happens, invariant (5.1) implies that the push saturating (v, w) rids v of all its excess. After the push, we advance the current edge pointer of v to the next eligible edge, doing a relabeling if necessary. The second case is that w may be relabeled, thus making (v, w) ineligible. The current edge pointer of v is advanced to the next eligible edge; for this new edge, (5.1) may be

procedure *bi-relabel*(w)
begin
    if $w \in V_1$
    then for all v s.t. the current edge of v is (v, w) **do**
            push $e(v)$ units of flow over edge (v, w)
    replace $d(v)$ by $\min\{d(w) + 1 : (v, w) \in I(v) \text{ and } u_f(v, w) > 0\}$
    for all children y of v **do**
        bi-cut(y)
end

FIG. 5.6. **The** *bi-relabel* **operation**

violated, however. To handle this case, we always push flow over edge (v, w) before relabeling w. This change is summarized in procedure *bi-relabel*(w), which appears in Figure 5.6. Observe that since all edges incident to w must be inspected in order to relabel w, procedure *bi-relabel* runs in the same asymptotic time as procedure *relabel*.

What we have shown is that whenever the current edge pointer of $w \in V_2$ advances, there is no excess at w. Since this pointer advances to the end of the list before a relabel, it must be true that at the time of a relabel there is no excess on w. Further, the only algorithmic changes are the change in line (*) of *bi-send*, which adds $O(1)$ work per tree push, the change in *bi-relabel*, which adds $O(1)$ work per relabel, and a change in the current edge advancement procedure, to make sure that current edges from V_2-vertices are always eligible. □

Given these building blocks we can give the procedure *bi-tree push/relabel*, which incorporates all of these ideas. The procedure appears in Figure 5.7. The basic idea is similar to that used in [3, 15, 16], in that we do a tree-push, but only perform a *link* if the size of the resulting tree is not too large. We also have the additional constraint of not performing a *link* that will cause a V_2-vertex to become a leaf. This leads to lines (T1) through (T2) of *bi-tree push/relabel* which handle the case when we are pushing from a trivial dynamic tree. In this case we first push flow over v's eligible edge (v, w). Then we do a *bi-send*(w) and proceed as if we had started at the root of w's dynamic tree. We also make one technical change and use a procedure called *bi-send** instead of *bi-send* in line (TB). Procedure *bi-send** differs from *bi-send* in that it defers doing its *cut*s until line (**) of procedure *bi-tree push/relabel*. This is done in order to avoid the case that the *link* performed in line (†) is linking a trivial dynamic tree, as this would make a V_2-vertex a leaf. (This is done purely for ease of presentation and is not necessary.)

We now use procedure *bi-tree-push/relabel* in a FIFO algorithm. We call this the *FIFO bipartite dynamic tree algorithm*.

Since, by Invariant 5.11, whenever a V_2-vertex is relabeled it has no excess, we can derive a bound of $O(n_1^2)$ passes over the queue, by a proof similar to that of 5.1. Define a *vertex activation* to be the event that either a vertex with zero excess receives positive excess, or a vertex with positive excess is relabeled. This corresponds to a vertex being placed on the queue. We will need to bound the number of times this occurs.

First, we give a lemma, the proof of which is simliar to that of Lemma 5.8 and Theorem 5.9, with the additional observations that the time spent in an iteration of *bi-tree-push/relabel* is within a constant factor of the amount of work done by *tree-push/relabel*.

LEMMA 5.13. *The FIFO bipartite dynamic tree algorithm runs in* $O(n_1 m \log k)$ *time plus* $O(\log k)$ *time per vertex activation.*

```
procedure bi-tree-push/relabel(v)
begin
        if there is an eligible edge (v, w)
(T1)    then begin if v is a trivial V2 tree
                   then begin push flow on edge (v, w)
                              r ← find-root(w)
(TB)                          bi-send*(w)
                              if there is an eligible edge (r, q)
                              then begin v ← r
                                         w ← q
                                   end
                              else bi-relabel(r)
(T2)                      end

                   if find-size(v) + find-size(w) ≤ k
(†)                then begin link(v, w, u_f(v, w))
                              p(v) ← w
                        end
(*)                else  begin push flow on edge (v, w)
                              bi-send(w)
(**)                          Perform the cuts from line (TB) (there may be none)
                        end
             end
        else
             bi-relabel(v)
end
```

FIG. 5.7. **The** *bi-tree-push/relabel* **procedure**

All that remains is to bound the number of vertex activations. First we introduce some terminology. We denote the tree containing vertex v by T_v. We call a tree *large* if the number of nodes in the tree is at least $k/2$. As a consequence of Lemma 5.10, there are only $2n_1$ vertices in all the non-trivial dynamic trees, hence there are no more than $4n_1/k$ large trees at any time. In particular we will use the fact that there are $O(n_1/k)$ large trees at the beginning of a pass over the queue.

LEMMA 5.14. *The number of vertex activations is $O(n_1 m + n_1^3/k)$.*

Proof. By invariant 5.11, all V_2-vertices have zero excess when relabeled, thus the only vertex activations due to relabelings are from V_1-vertices. There are at most $O(n_1^2)$ of these. There can be only $O(n_1 m)$ vertex activations for which the corresponding *bi-tree-push/relabel* executions perform a *cut* or *link* or a saturating push in line (*).

It remains to count the vertex activations for which the corresponding invocation of *bi-tree-push/relabel* does neither a cut nor a link nor a saturating push. If this occurs then it must be that *find-size*(v) + *find-size*$(w) \geq k$, i.e. either T_v or T_w is large. We consider the two cases separately.

Suppose T_v is large. Vertex v is the root of T_v. Since the push is non-saturating, it must rid v of all its excess. If T_v has changed since the beginning of the current pass, we charge the activation to the *link* or *cut* that most recently changed T_v. This occurs at most once per *cut* and twice per *link* for a total of $O(n_1 m)$ time overall. If T_v has not changed since the beginning of the pass, we charge the activation to T_v. There are at most $O(n_1/k)$ large trees at the start of a pass, hence this case counts for $O(n_1^3/k)$ charges overall.

Suppose T_w is large. In this case the root r of T_w may be added to the queue. As before, if T_w changed during the pass we charge the activation to the *link* or *cut* which caused it, otherwise we charge it to the large tree.

We have ignored so far the possible activations in lines (T1) through (T2). It is easy to verify that these only add a constant factor to the bounds mentioned above. The reason for adding this case is to ensure that in every iteration either a *link*, *cut*, or saturation is performed, or a large tree is involved. This additional case allows us to ensure this with no asymptotic loss in the running time of the procedure.

Combining all these cases we get $O(n_1 m + n_1^3/k)$ vertex activations. □

THEOREM 5.15. *The FIFO bipartite dynamic trees algorithm runs in* $O(n_1 m \log(\frac{n_1^2}{m} + 2))$ *time.*

Proof. Apply Lemmas 5.13 and 5.14 and choose $k = \frac{n_1^2}{m} + 2$. □

5.6. A Parallel Implementation. In this section, we give a parallel implementation of the bipartite excess scaling algorithm. Our model of computation is an exclusive-read exclusive-write parallel random access machine (EREW PRAM) [13]. Our algorithm runs in $O((\frac{n_1 m}{d} + n_1^2 \log U)\log d)$ time using $d = \lceil \frac{m}{n_1} \rceil$ processors, thus achieving near-optimal speedup for the given number of processors. We assume familiarity with parallel prefix operations [22] and refer the reader to [2, 16, 26, 32] for examples of the use of parallel prefix operations in network flow algorithms. Specifically, we use the fact that using d processors and $O(\log d)$ time, we can execute the following parallel prefix operation:

> Parallel Prefix Operation: Given $l \leq d$ numbers $f(v_1), \ldots, f(v_l)$, compute the partial sums $f(v_1), f(v_1) + f(v_2), \ldots, f(v_1) + \ldots + f(v_l)$.

Our algorithm will be the same as the excess-scaling algorithm of Section 5.3 with a parallel implementation of *bipush/relabel* and a few additional data structures. The same approach was taken by Ahuja and Orlin [2] in developing a parallel version of their original excess scaling algorithm.

The first step in our algorithm is to transform the input graph so that each vertex has out-degree no greater than d. This transformation yields a graph with $O(n_1)$ V_1-vertices, $O(n_2)$ V_2-vertices and $O(m)$ edges. We achieve this by repeating the following step until it is no longer applicable:

> splitting step: Pick a vertex v with out-degree $k > d$. Create two new vertices v' and v'' and replace edges $(v, v_{k-d+1}) \ldots (v, v_k)$ with edges $(v, v'), (v', v'')$, and $(v'', v_{k-d+1}) \ldots (v'', v_k)$. Edges (v, v') and (v', v'') have infinite capacity, while each edge (v'', v_k) has its capacity set equal to $u(v, v_k)$.

The splitting step creates one new V_1-vertex, one new V_2-vertex, and 2 more edges. Let $\Phi = \sum_v \max\{0, \lceil \frac{\text{out-degree}(v) - d}{d-1} \rceil\}$. Each splitting step reduces Φ by one. Initially $\Phi = O(n_1)$ and $\Phi \geq 0$ when the algorithm terminates. Thus, we only need to perform the splitting step $O(n_1)$ times overall, adding $O(n_1)$ vertices and $O(n_1)$ edges. Similarly, we can repeat the same step to reduce the in-degree of each vertex.

Further, we can perform this step in $O(n_1 \log m)$ time on d processors. We explain how to reduce the in-degree; the out-degree can be reduced in a similar manner. First, we lexicographically sort the list of edges by their tails. This can be done on d processors in $O(n_1 \log m)$ time using Cole's sorting algorithm [8] and Brent's Theorem [6]. Next, we assign one processor to each of the last d edges on the list. In $O(\log d)$ time, we can determine if all these edges have the same tail. If so, we perform the splitting step, which can be done in $O(1)$ time on d processors. We then delete these edges from the list and continue on the remainder of the list. If they do not all have the same tail, then the last vertex on the list must have degree $\leq d$. In this case we delete all edges which have the same tail as the last edge and continue on the remainder of the list. In each iteration we either delete all the edges incident to a vertex or we process d edges. Hence there are $O(n_1 + \frac{m}{d}) = O(n_1)$ iterations, each of which can be performed in $O(\log m)$ time on d processors.

For the rest of this section, we will assume, w.l.o.g., that every vertex in our graph has both in-degree and out-degree $\leq d$.

We first address the problem of implementing a bipush in parallel. In the bipush operation for the maximum flow problem, it is necessary to scan the edge list for vertex v starting with the current edge for vertex v until either an eligible edge is determined or until the edge list is exhausted. In the parallel algorithm, we will scan these edges in parallel.

We begin by introducing some terminology. Let $I(v)$ denote the set of vertices w such that (v, w) is an edge, and let $\hat{I}(v)$ denote the set of vertices w such that (v, w) is an eligible edge. Let us assume that the vertices in $I(v)$ are denoted $v_1, v_2, \ldots, v_k$, where $k = |I(v)|$. Thus the j-th edge emanating from vertex v is edge (v, v_j).

For each vertex $v \in V_2$, we let $\hat{r}(v) = \sum_{w \in \hat{I}(v)} r(v, w)$, and refer to $\hat{r}(v)$ as the effective residual capacity of vertex v. Note that we can always push all of the excess out of a vertex v in V_2 prior to a relabeling of v so long as the excess does not exceed the effective residual capacity.

We define the *effective residual capacity* $\hat{r}(v, w)$ of edge (v, w) as

$$\hat{r}(v, w) = \begin{cases} 0 & \text{if } (v, w) \text{ is not eligible} \\ r(v, w) & \text{if } (v, w) \text{ is eligible and } v \in V_2, w \in V_1 \\ \min\{r(v, w), \hat{r}(w)\} & \text{if } (v, w) \text{ is eligible and } v \in V_1, w \in V_2. \end{cases}$$

In the algorithm, we will be performing pushes from one vertex in V_1 at a time, and we will subsequently push from several vertices in V_2 in parallel. By defining the effective residual capacity for edges (v, w) as we do, we will ensure that we never push more flow into any vertex $v \in V_2$ than the effective residual capacity of v. Subsequently, all of the flow can be pushed out prior to a relabel of v.

In order to achieve the speedup desired, we cannot assign one processor to each edge of $I(v)$ in a push from vertex v. Thus, we will have to more efficiently allocate processors to edges on which we wish to push flow. In order to do so, we introduce the following four procedures. In all these procedures v is a vertex from which we wish to push δ units of flow.

1. $\underline{NextCurrent(v, \delta)}$: if pushing δ units of flow would saturate all of v's admissible edges, then output $|I(v)| + 1$. Otherwise, output the index of the edge that will be current after pushing δ units of flow from v.

2. $\underline{NewRelabel(v, \delta)}$: output true if NextCurrent $(v, \delta) = |I(v)| + 1$ and false otherwise.

3. $\underline{NextIncrement(v, \delta)}$: output the amount of flow to be sent in edge NextCurrent(v, δ) when pushing flow from v.

4. $\underline{Requirement(v, \delta)}$: output the number of edges scanned in order to send δ units of flow from v without a relabel. It is equal to $NextCurrent(v, \delta) - Current(v) + 1$.

LEMMA 5.16. *There exists a data structure that allows us to implement each of these operations in $O(\log d)$ time on one processor.*

We defer the proof until later. Assume for now that such an implementation exists.

procedure *Parallel-push*(v, δ, S)
begin
  $c = Current(v)$. $k = NextCurrent(v, \delta)$. $s = |S|$
(*) **For each** i from c to $\min(k-1, c+s-1)$ **do in parallel**
    send $\hat{r}(v, v_i)$ units of flow in edge (v, v_i), and update $\hat{r}$.
  if $s \geq k - c + 1$ and $k \leq |I(v)|$
    then send $NextIncrement(v, \delta)$ units of flow in edge (v, v_k).
  $Current(v) = NextCurrent(v, \delta)$.
  end

FIG. 5.8. *The procedure parallel push*

Using these procedures, we can implement the main operation, which we call *parallel-push*(v, δ, S). This operation tries to push up to δ units of flow from vertex v using the set S of parallel processors, and so that no relabel occurs. The implementation is straightforward, and appears in Figure 5.8.

LEMMA 5.17. *Parallel-push can be implemented in $O(\log d)$ time on d processors.*

Proof. Step (*) can be implemented by a parallel prefix operation on d processors. By Lemma 5.16 all the other steps can be implemented on 1 processor in $O(\log d)$ time. □

Part of the input to parallel-push is a set of processors. We use a procedure Allocate(v, D) to implement this.

Allocate(v, D)

input: vertex v, and D, a d-dimensional vector of demands for processors from the vertices in $I(v)$. $D(j)$ is the number of processors requested by vertex v_j.

output: The vector Processors(), where Processors(j) is the set of processors allocated to vertex v_j.

It is straightforward to implement Allocate with a parallel prefix operation.

Now, we are ready to put all the pieces together to get an implementation of *parallel bipush/relabel.* This simply consists of a parallel push from v, followed by a set of parallel pushes from vertices $w \in V_2$ with excess, each of which is preceded by processor allocation. The procedure concludes by relabeling the necessary vertices. The details appear in Figure 5.9. One detail deserves explanation. We always try to push exactly $\Delta/2$ units of flow from a vertex in V_1. This is necessary to maintain the invariant that no vertex ever accumulates more than Δ units of excess.

To begin the analysis, we bound the number of iterations of this procedure.

LEMMA 5.18. *There are $O(n_1^2 \log U)$ calls to* parallel bipush/relabel *over the course of the whole algorithm.*

Proof. Each parallel bipush/relabel in the first line either moves $\Delta/2$ units of flow or results in a relabeling. By a proof similar to that of Lemma 5.6, there are at most $O(n_1^2 \log U)$ such pushes over the whole algorithm. □

LEMMA 5.19. *Each call to* parallel bipush/relabel *takes*
$O(\#$ of iterations of the while loop $\times \log d +$ time spent relabeling$)$ time on d processors.

Proof. By Lemma 5.16 and the fact that Allocate takes $O(\log d)$ time, each step except for the parallel push in line (*) takes $O(\log d)$ time. We know from lemma 5.17 that a push takes $O(\log d)$ time. It is easy to see that a set of pushes which use a total of d edges can also be completed in $O(\log d)$ time; thus each iteration of the while loop takes $O(\log d)$ time. The lemma follows. □

It remains to bound the number of iterations of the while loop.

```
procedure parallel bipush/relabel(v)
begin
        Parallel push(v, Δ/2, d)
        while e(v_j) ≠ 0 for some v_j ∈ I(v) do
        begin
                for each j = 1 to d do in parallel
                    D(v_j) = Requirement(v_j, e(v_j)).
                Allocate(v, D, d).
                for i = 1 to d do in parallel
                begin
(*)                 push(v_i, e(v_i), processors(i)).
                    update data structures.
                end
        end
        create a list L of indices j s.t. j ∈ V_2 and NewRelabel(v_j) = true.
        for each i ∈ L do Relabel(v_i).
        if NewRelabel(v) = true then relabel(v).
end
```

FIG. 5.9. *Procedure parallel bipush/relabel*

LEMMA 5.20. *The while loop is executed $O(\frac{n_1 m}{d} + n_1^2 \log U)$ times over the whole algorithm.*

Proof. First we observe that each vertex in $I(v)$ may have at most one non-saturating push from it per execution of the while loop. Lemma 5.18 implies that the number of non-saturating pushes is at most $O(n_1^2 d \log U)$ overall. Let nsp be the number of non-saturating pushes that have occurred since the beginning of the algorithm. Consider the potential function $F = \sum_v current(v) + nsp$. Initially $F = 0$ and at termination $F = (\#\text{ of non-saturating pushes}) = O(n_1^2 d \log U)$. The only way for F to decrease is by a relabel. Each relabel decreases F by at most $|I(v)|$; the total decrease is $O(n_1 m)$. So, the total increase in F over the algorithm is $O((n_1^2 d \log U + n_1 m)$. A parallel push with k processors increases F by k or results in a relabeling. Each iteration in a while loop except for the last one allocates d processors; hence it increases F by d or results in a relabeling. Ignoring the last iteration of the while loop in each call to parallel bipush/relabel, we find that there are at most $O((n_1^2 d \log U + n_1 m)/d)$ iterations of the while loop. To count the last iterations, we observe that there is one last iteration per call for a total of $O(n_1^2 \log U)$. Thus, overall there are $O(\frac{n_1 m}{d} + n_1^2 \log U)$ iterations. □

LEMMA 5.21. *The total time spent relabeling is $O((\frac{n_1 m}{d} + n_1^2 \log U) \log d)$.*

Proof. We spend a total of $O(n_1 m)$ work relabeling. However, at each relabeling step we look at d edges at a time, except for the last relabel step in a call to parallel bipush/relabel. Hence the total time is $O(\frac{n_1 m}{d} + n_1^2 \log U)$ □

Now we turn to the proof of Lemma 5.16.

Proof. (of Lemma 5.16) Assume for now that $k = |I(v)|$ is a power of 2 for each vertex. We create a complete binary tree whose leaves are the indices of the vertices in I(v). The key of each leaf j in the binary tree is $\hat{r}(v, v_j)$. The key of each internal vertex of the binary tree is the sum of the keys of its descendent leaves.

Whenever a vertex v is relabeled, each vertex v_j of $I(v)$ is assigned a processor, and its binary tree is updated. The assignment of processors takes $O(\log d)$ steps per relabel. Moreover, each processor updates its binary tree in $O(\log d)$ steps.

When a push from vertex v is performed, the binary tree for vertex v must be updated. If k processors are assigned then Current is increased by $\leq k$, and the updating can be accomplished with k processors in O(log d) time.

In order to compute $NextCurrent(v, \delta)$, we start at the root of the binary tree for v, and we select the right child or the left child depending on whether δ is less than or greater than the key of the right child. We then recurse on the selected child. We also can compute $NextIncrement$ in this manner. $\square$

Combining all the above results, we have the following theorem:

THEOREM 5.22. *Algorithm* Bipartite Excess Scaling *with* bipush/relabel *replaced by* parallel bipush/relabel *runs in* $O((\frac{n_1 m}{d} + n_1^2 \log U)\log d)$ *time on* d *processors on an EREW PRAM.*

Plugging in $d = \lceil \frac{m}{n_1} \rceil$, we can restate the theorem as the following corollary:

COROLLARY 5.23. *Algorithm* Bipartite Excess Scaling *with* bipush/relabel *replaced by* parallel bipush/relabel *runs in* $O(n_1^2 \log U \log \frac{m}{n_1})$ *time on* $\lceil \frac{m}{n_1} \rceil$ *processors on an EREW PRAM.*

The work done by this algorithm is within a logarithmic factor of the running time of the sequential algorithm *Bipartite Excess Scaling.*

6. Parametric Maximum Flow. A natural generalization of the maximum flow problem is obtained by making the edge capacities functions of a single parameter λ. This problem is known as the *parametric maximum flow problem.* We consider parametric maximum flow problems in which the capacities of the edges out of the sink are non-decreasing functions of λ, the capacities of the edges into the sink are non-increasing functions of λ, and the capacities of the remaining edges are constant. Although this type of parameterization appears to be quite specialized, Gallo, Grigoriadis, and Tarjan [14] have pointed out that this parametric problem has many applications, in computing subgraph density and network vulnerability and in solving other problems, some of which are mentioned at the end of this section.

Let $u_\lambda(v, w)$ denote the capacity of edge (v, w) as a function of λ and suppose that we wish to solve the maximum flow problem for parameter values $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_l$. Clearly, for l different values of λ, a solution can be found using l invocations of a maximum flow algorithm. This approach takes no advantage of the similarity of the successive problems to be solved, however. Gallo, Grigoriadis, and Tarjan [14] gave an algorithm for finding the maximum flow for $O(n)$ increasing values of λ in the same asymptotic time that it takes to run the Goldberg-Tarjan maximum flow algorithm *once.* If the capacities are linear functions of λ, it is easy to show that the value of the maximum flow, when viewed as a function of λ, is a piecewise linear function with no more than $n-2$ breakpoints. In this case, they give an algorithm for finding all of the breakpoints of this function in the same asymptotic time as it takes to run the Goldberg-Tarjan maximum flow algorithm once.

In this section we give an algorithm which for l increasing values of λ finds all l maximum flows in $O(ln + ln_1^2 + n_1^3 + n_1 m)$ time. Using the dynamic tree data structure, this algorithm runs in $O(ln + n_1 m \log(\frac{ln_1 + n_1^2}{m} + 2))$ time.

We begin by giving one iteration of the algorithm, i.e., determining the maximum flow for parameter value λ_i, if the maximum flow for parameter value λ_{i-1} is given. The algorithm appears in Figure 6.1. First, we update the capacities. The capacity of an edge leaving the source may have increased. If so, we saturate the edge, by setting its flow equal to its new capacity. The capacity of an edge leaving the sink may have decreased. If it has decreased below the flow on the edge, we decrease the flow so that it is equal to the capacity. Since $t \in V_1$ by assumption, this may create excess on vertices in V_2. Therefore, we immediately push any such excess to vertices in V_1, thus re-establishing the invariant that no excess is on vertices in V_2. The second step consists of running the bipartite FIFO algorithm in the network beginning with the

Step 1 (Update preflow)
Let $i = i + 1$
$\forall (s,v) \in E$ with $d(v) < 2n_1$, let $f(s,v) = \max\{u_{\lambda_i}(s,v), f(s,v)\}$.
$\forall (v,t) \in E$, let $f(v,t) = \min\{u_{\lambda_i}(v,t), f(v,t)\}$.
$\forall v \in V_2$ **while** $e(v) > 0$, **do** *push/relabel*(v).
Step 2 (Find maximum flow) Run the bipartite FIFO algorithm on the network with capacities u_{λ_i}, beginning with flow f and distance labels d.

FIG. 6.1. *Algorithm* parametric bipartite flow

current f and d. This gives us a maximum flow for the parameter value λ_i.

Remark: In applications of the parametric maximum flow problem, it may happen that $s \in V_1$ or $t \in V_2$, contrary to our assumption. Such a possibility can be handled by making minor changes to the algorithm, without affecting its running time.

Now we must prove that the algorithm is correct and efficient. We do this by means of the following lemmas:

LEMMA 6.1. *At the end of each step in the algorithm, there is no excess on any vertex in V_2.*

Proof. It suffices to restrict our attention to Step 1, since Step 2 always maintains this condition. Since by assumption $s \in V_2$, increasing the flow on edges out of s can increase the excess only on vertices in V_1. Since $t \in V_2$, decreasing the flow on edges into t may create excesses on vertices in V_2. This excess is immediately removed from vertices in V_2 by the procedure *push/relabel*, however. □

LEMMA 6.2. *Throughout all iterations of the* parametric bipartite flow algorithm, *distance labels are non-decreasing.*

Proof. We first show that updating the residual capacities and the preflow between iterations maintains the validity of the distance labels. Increasing the flow on an edge (s,v) may create a new residual edge (v,s), but since $d(v) < 2n_1$, the labeling is still valid. Decreasing the flow on edges into t does not create any new residual edges, so the distance labels are still valid. We noted earlier that procedures *push/relabel* and *bipush/relabel* maintain a valid labeling. The lemma follows. □

A consequence of Lemma 6.2 is that, over all iterations of the algorithm, each vertex is relabeled $O(n_1)$ times, and the total relabeling time is $O(n_1 m)$. Furthermore, the total number of saturating pushes over the whole algorithm is $O(n_1 m)$. We bound the number of nonsaturating bipushes in the next lemma.

LEMMA 6.3. *The algorithm performs a total of $O(ln_1^2 + n_1^3)$ nonsaturating bipushes over all l iterations.*

Proof. As in the bipartite FIFO algorithm, consider the potential function $\Phi = \max\{d(v) | v \text{ is active}\}$. The potential function increases due to relabelings, and this increase has already been shown to be at most $4n_1^2$. The potential function may also increase in Step 1 when the preflow is updated. But this increase is at most $O(n_1)$ per iteration, and $O(n_1 l)$ over all iterations. Thus the total number of passes over Q is $O(ln_1 + n_1^2)$ and the total number of nonsaturating bipushes is $O(ln_1^2 + n_1^3)$. □

THEOREM 6.4. *A total of l iterations of the* parametric bipartite flow algorithm *take $O(ln + n_1 m + ln_1^2 + n_1^3)$ time.*

Proof. Each execution of Step 1 takes $O(n)$ time to update the residual capacities and flows. Getting rid of the excesses at vertices in V_2 by performing *push/relabel* steps takes $O(n)$ time per iteration plus the time to perform saturating pushes, which is $O(n_1 m)$ time overall. Hence l executions of Step 1 take $O(ln + n_1 m)$ time. The l

executions of Step 2 take a total of $O(n_1 m + l n_1^2 + n_1^3)$ time, as was shown previously. The theorem follows. □

The dynamic tree data structure can be incorporated into the parametric maximum flow algorithm to improve its computational complexity. Using the ideas described in Section 5.5, it can be shown that the dynamic tree implementation of the parametric maximum flow problem runs in $O\left(ln + n_1 m \log\left(\frac{ln_1+n_1^2}{m}+2\right)\right)$ time.

Often applications of the parametric maximum flow problem require that the minimum cut be determined for each of the parameter values $\lambda_1, \lambda_2, \cdots, \lambda_l$. Obviously each such minimum cut can be determined by a breadth-first search of the network, requiring $O(m)$ effort per cut. Overall this time would be $O(ml)$ and for larger values of l would be a bottleneck. In order to achieve a faster time bound we maintain exact distance labels of vertices as explained in [16]. Maintaining exact distance labels requires some additional effort but no more than $O(n_1 m)$ time over all iterations. While using this method, the *minimum cut* $(X_i, \overline{X_i})$, at the end of iteration i is defined as $X_i = \{v \in V : d(v) \geq 2n_1\}$ and $\overline{X_i} = \{v \in V : d(v) < 2n_1\}$. It may also be pointed out the minimum cuts in the parametric maximum flow problem are nested, i.e., for $\lambda_1 \leq \lambda_2 \leq \lambda_3$, with corresponding cuts $(X_1, \overline{X_1}), (X_2, \overline{X_2}), (X_3, \overline{X_3})$, we have that $X_1 \subseteq X_2 \subseteq X_3$ [12]. This property allows us to store all l cuts in $O(n+l)$ space, and recreate any one cut in $O(n)$ time.

While we have only given an algorithm for the case where the λ's are given in increasing order, actually we can solve a more general problem. Let $\kappa(\lambda)$, the *min-cut capacity function*, be the capacity of the minimum cut as a function of λ. If the edge capacities are linear functions of λ, then $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n-2$ breakpoints. We can actually compute all of these breakpoints in $O(n^2 + n_1 m \log(\frac{nn_1}{m}+2))$ time, and can do even better if we know *a priori* that $l = o(n)$. This result directly follows from the results of [14] and the details appear in [35].

We conclude by noting that the bipartite parametric flow problem has many applications including multiprocessor scheduling with release times and deadlines [21, 24], 0-1 integer programming problems [29, 30], maximum subgraph density [21], finding a maximum-size set of edge-disjoint spanning trees in an undirected graph [28, 29, 30], network vulnerability [9, 19], partitioning a data base between fast and slow memory [11], and the sportswriter's end-of-season problem [23, 31]. For all these problems we improve on or match the best known bounds.

7. Minimum-Cost Circulation. In this section we examine the *minimum-cost flow problem* on bipartite networks. We consider the recent cost-scaling minimum-cost flow algorithm of Goldberg and Tarjan [17], and describe the improvement in its running time that can be obtained when it is adapted for bipartite networks. We shall be very sketchy in our description, since all the results are analogous to the results in Section 5.

The minimum cost flow problem is a generalization of the maximum flow problem. In this problem, each edge (v, w) has a cost $c(v, w)$. We formulate the problem as a circulation problem, since it is equivalent to other formulations. (See [1],[18].) We assume that the costs are *antisymmetric*, i.e., $c(v, w) = -c(w, v)$ for each edge (v, w). Let $C = \max\{c(v, w) : (v, w) \in E\}$. The minimum-cost circulation problem can be formulated as follows:

$$\text{Minimize} \sum_{(v,w)\in E} c(v,w)f(v,w)$$

subject to

$$f(v,w) \le u(v,w), \quad \forall (v,w) \in E \tag{7.1}$$

$$f(v,w) = -f(w,v), \ \forall (v,w) \in E \tag{7.2}$$

$$\sum_{w\in V} f(v,w) = 0, \quad \forall v \in V \tag{7.3}$$

A *circulation* is a function of satisfying constraints (7), (8), and (9). A *pseudoflow* is a function f satisfying only constraints (7.1) and (7.2). For any pseudoflow f, we define the *excess* of vertex w to be

$$e(w) = \sum_{v:(v,w)\in E} f(v,w) \tag{7.4}$$

The excess at a vertex may be positive or negative. A vertex v is called *active* if $e(v) > 0$. The residual network is defined as for the maximum flow problem. We associate with each vertex v a real-valued *price* $p(v)$. The prices correspond to linear programming dual variables. In the analysis, the prices play a role similar to that played by the distance labels in the maximum flow algorithm. The *reduced cost* of an edge (v,w) with respect to the price function p is denoted by $c_p(v,w)$ and is defined by $c_p(v,w) = c(v,w) + p(v) - p(w)$.

7.1. The Cost-Scaling Algorithm. The cost-scaling algorithm of Goldberg and Tarjan[17], relies on the concept of *approximate optimality*. A circulation f is said to be *ϵ-optimal* for some $\epsilon > 0$ if f together with some price function p satisfies the following condition:

$$u_f(v,w) > 0 \Rightarrow c_p(v,w) \ge -\epsilon \Longrightarrow (\epsilon\text{-optimality}). \tag{7.5}$$

We refer to this condition as the *ϵ-optimality condition*. Let l be the number of edges on the longest simple cycle in the network. It can be shown that any feasible flow is ϵ-optimal for $\epsilon \ge C$ and any ϵ-optimal feasible flow for $\epsilon < 1/l$ is an optimum flow [4]. Since in a bipartite network every other vertex on a cycle must be a vertex in V_1, any ϵ-optimal feasible flow for $\epsilon < \frac{1}{2n_1}$ is an optimum flow.

The cost-scaling algorithm treats ϵ as a parameter and iteratively obtains ϵ-optimal flows for successively smaller values of ϵ. Initially, $\epsilon = C$; on termination, $\epsilon < \frac{1}{2n_1}$. The algorithm performs repeated cost-scaling phases, each of which consists of applying an *improve-approximation* procedure that transforms a 2ϵ-optimal circulation into an ϵ-optimal circulation. After $1 + \lceil \log(2n_1C) \rceil$ cost scaling phases, $\epsilon < \frac{1}{2n_1}$, and the algorithm terminates with an optimal circulation. To get the algorithm started, an initial circulation can be found by using any maximum flow algorithm, such as one of those discussed in Section 5. A more formal description of this algorithm appears in Figure 7.1.

Recall that in the maximum flow algorithm, we maintained the invariant that all excess was on V_1-vertices. This will be our goal in the minimum cost circulation algorithm also. The procedure *improve-approximation* given in Figure 7.2 first converts the 2ϵ-optimal circulation it receives as input into a 0-optimal pseudoflow (lines

```
algorithm cost scaling
begin
        p = 0; ε = C;
        let f be any initial circulation;
        while ε ≥ 1/(2n_1) do
                begin ε = ε/2
                        improve-approximation(f, p, ε);
                end
end
```

FIG. 7.1. *Algorithm* Cost Scaling

```
procedure improve-approximation(f, p, ε)
begin
(*)     if c_p(v, w) < 0
        then begin f(v, w) = u(v, w);
                   f(w, v) = −f(v, w)
             end;
(**)    compute vertex imbalances;
(†)     while the network contains an active V_2-vertex v do
                push/update(v)
(‡)     while the network contains an active vertex v do
                bipush/update(v)
end
```

FIG. 7.2. *The procedure* improve-approximation

(*) through (**)). This may leave positive excess on V_2-vertices. So we execute the while loop at line (†), which applies *push/update* operations to these vertices until they are rid of all their excess. Now we have established the invariant that the only vertices with positive excess are V_1-vertices. We will maintain this invariant for the rest of procedure *improve-approximation*. The remainder of the procedure moves flow from vertices with positive excess to vertices with negative excess. As vertices in V_2 may have negative excess, this will sometimes involve a one-edge push and sometimes involve a two-edge push.

We call an edge (v, w) in the residual network *admissible* if $c_p(v, w) < 0$. We define the subnetwork of G consisting solely of admissible edges to be the *admissible network*. The basic operations in the procedure are selecting active vertices, pushing flows on admissible edges, and updating vertex prices. The details of *improve-approximation*, adapted to the bipartite case, appear in Figures 7.2 and 7.3.

To identify admissible edges emanating from a vertex, the algorithm uses the same *current edge* data structure used in the preflow push algorithm for the maximum flow problem.

A movement of flow along a path $v - w - x$ in *bipush/update* is called a *bipush*. The bipush is *saturating* if $\delta = \min\{u_f(v, w), u_f(w, x)\}$ and *nonsaturating* otherwise. The correctness and efficiency of the algorithm rest on the following results:

LEMMA 7.1.

1. *The* improve-approximation *procedure always maintains* ϵ*-optimality of the pseudoflow, and at termination yields an* ϵ*-optimal circulation.*

2. *Each vertex price never increases, and it decreases* $O(n_1)$ *times during an execution of the procedure.*

3. *There are* $O(n_1 m)$ *saturating pushes and bipushes during an execution of the procedure.*

4. *Immediately before, during, and immediately after the while loop in line* (‡)

```
procedure bipush/update(v)
begin
        if there exists an admissible edge (v, w)
        then if e(w) < 0
                then push min{e(w), e(v), u_f(v, w)} units of flow on (v, w)
                else  if there exists an admissible edge (w, x)
                        then push δ = min{e(v), u_f(v, w), u_f(w, x)} units of flow
                                    along the path v − w − x
                        else  replace p(w) by max_{(w,x)∈E_f} {p(x) − c(w, x) − ε}
        else  replace p(v) by max_{(v,w)∈E_f} {p(w) − c(v, w) − ε}
end
procedure push/update(v)
begin
        if there exists an admissible edge (v, w)
        then push min{e(w), e(v), u_f(v, w)} flow on (v, w)
        else replace p(v) by max_{(v,w)∈E_f} {p(w) − c(v, w) − ε}
end
```

FIG. 7.3. *The procedures* push/update *and* bi-push/update

of improve-approximation*, all excess is on V_1-vertices.*

Proof. These results follow directly from the proofs of Goldberg and Tarjan [17] adapted for bipartite networks. □

As in the preflow push algorithm, it can easily be shown that the time spent updating prices in an execution of *improve-approximation* is $O(n_1 m)$. The bottleneck in the procedure is the number of nonsaturating pushes and bipushes. Observe that there are three different types of pushes and bipushes to bound:

1. pushes in the while loop at line (†),
2. bipushes in the while loop at line (‡),
3. pushes in the while loop at line (‡).

We bound the first type by observing the all the pushes are saturating except for at most one per V_2-vertex. Therefore, there are at most n nonsaturating pushes.

To bound the second type of bipushes, we need the following lemma from [17]:

LEMMA 7.2 ([17]). *The admissible network remains acyclic throughout the execution of the* improve-approximation *procedure.*

The number of nonsaturating bipushes performed by the procedure depends upon the order in which active vertices are examined. Goldberg and Tarjan [17] show that the generic version of the procedure, in which active vertices are examined in an arbitrary order, performs $O(n^2 m)$ pushes for general networks. They show that a specific implementation of the generic implementation, called the *first-active method* algorithm, performs $O(n^3)$ nonsaturating pushes, as does a related method, the *wave method.* (The wave method was developed independently by Bertsekas and Eckstein[5].) We shall show that an adaptation of the first-active method for bipartite networks performs $O(n_1^3)$ nonsaturating bipushes.

The first-active method uses the acyclicity of the admissible network. As is well known, the vertices of an acyclic network can be ordered so that for each edge (v, w), v precedes w in the ordering. Such an ordering of vertices is called a *topological ordering* and can be found in $O(m)$ time. The first-active method maintains a list L of all vertices in V_1 in topological order. The algorithm examines each vertex $v \in L$ in order. If v is active, it performs *bipush/update* operations on vertex v until either it becomes inactive or $p(v)$ is updated. In the former case, the algorithm examines the next vertex on L. In the latter case, the algorithm moves v from its current position on L to the front of L, and restarts the scan of L at the front. Moving v to the front of L preserves the invariant that L is a topological order of the vertices, because immediately after v is assigned a new prices, it has no incoming admissible edges. The algorithm terminates when L is scanned in its entirety. Note that updating the price of a vertex in V_2 does not affect the topological order of vertices in V_1. On termination, no vertex can be active.

Observe that if within n_1 consecutive vertex examinations the algorithm performs no price updates then all active vertices have discharged their excesses and the algorithm obtains a flow. This follows from the fact that when vertices are examined in the topological order, active vertices push flow to vertices after them in the topological order. As there are $O(n_1^2)$ price updates of vertices in V_1, we immediately obtain an $O(n_1^3)$ bound on the number of vertex examinations. Each vertex examination entails at most one nonsaturating bipush. Consequently, the wave algorithm performs $O(n_1^3)$ nonsaturating bipushes per execution of *improve-approximation*.

Now we bound the third type of push. A push in this case is performed over an edge (v, w) such that $e(w) < 0$. There are three cases. Either the value of the push is $u_f(v, w)$ (saturating), $e(v)$ (non-filling), or $e(w)$ (filling). For the first case we have already bounded the number of saturating pushes. In the second case, we can bound the number of non-filling pushes by $O(n_1^3)$ by arguments similar to those for non-saturating pushes above. For filling pushes, observe that each vertex is filled at most once per iteration of *improve-approximation*; thus there are a total of n such pushes overall.

Combining the three cases, we find that the number of nonsaturating pushes and bipushes is $O(n_1^3 + n)$. As all other steps take $O(n_1 m)$ time per execution of the *improve-approximation* procedure and the procedure is called $O(\log(n_1 C))$ times, we get the following result:

THEOREM 7.3. *The wave algorithm solves the minimum cost flow problem on a bipartite network is $O((n_1 m + n_1^3)\log(n_1 C))$ time.*

8. Summary and Conclusions. We have considered a number of maximum flow algorithms and algorithms for other network flow problems for bipartite networks in which one side is much smaller than the other. Our work is motivated by and improves upon the work of Gusfield, Martel, and Fernandez-Baca [21]. In that paper, the authors demonstrated the importance of bipartite maximum flow problems in which one side is much smaller than the other. In addition, they showed that existing algorithms run much faster on these "unbalanced" networks.

We have extended the results of Gusfield et al. in several ways. First of all, we showed that their analysis applies to other maximum flow algorithms. In addition, we developed the concept of the *bipush* for preflow-push algorithms and showed that bipushes lead to further improvements in several algorithms for the maximum flow problem. We further generalized the results to algorithms for the parametric maximum flow problem, as well as the minimum cost flow problem. We also showed that the results apply as well to dynamic tree implementations if the dynamic tree algorithms are modified appropriately.

Although the theory in this paper has been concerned with bipartite networks, it would be just as valid for networks in which we allow edges joining two vertices in V_1. More generally, it is valid for networks in which have a small *vertex cover*. A vertex cover of a network $G = (V, E)$ is a set S of vertices such that each edge in E is incident to at least one vertex in S. A minimum vertex cover is one with the smallest number of vertices. Although it is NP-hard to determine a minimum vertex cover of a graph, it is possible to find a vertex cover in $O(n+m)$ time whose cardinality is within a factor of 2 of the cardinality of a minimum vertex cover. (Just find any maximal matching and include each of the matched vertices).

If the size of the minimum vertex cover of a graph is n_1, then all of the time bounds presented in the previous sections apply. It is easy to show that the length of the longest path in such a network is at most $2n_1$. As for bipushes they would have to be replaced as follows. Suppose G is a network, not necessarily bipartite, in which V_1 is a vertex cover. As before we maintain the invariant that each active vertex is in V_1. Suppose that v is active, and that (v, w) is eligible. If w is in V_1 then we perform a normal push. If w is not in V_1, then each edge incident to w is in V_1 and we perform a bipush. All of the results in this paper are thus easily generalized to networks with small vertex covers, and the time bounds stated in Table 1.1 apply to such networks.

It is likely that improvements could be obtained in the running times of other algorithms for network flow problems on unbalanced bipartite networks, or on networks in which the cardinality of a minimum vertex cover is small. For example, one can obtain improved running times for dynamic programming algorithms for the shortest path problem, and one can improve the running time for all pairs shortest path algorithms. We conjecture that one can also obtain improved time bounds for the b-matching problem on networks with small vertex covers. We also conjecture that one can obtain improved results for polymatroidal network flows.

Acknowledgments. We are grateful to David Shmoys for many helpful conversations and suggestions, and for a careful reading of an earlier draft of this paper.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows*, in Handbook in operations research and management science, Volume 1: Optimization, G. Nemhauser, A. H. G. R. Kan, and M. J. Todd, eds., North-Holland, Amsterdam, 1990, pp. 211–360.
[2] R. K. Ahuja and J. B. Orlin, *A fast and simple algorithm for the maximum flow problem*, Operations Research, 37 (1989), pp. 748–759.
[3] R. K. Ahuja, J. B. Orlin, and R. Tarjan, *Improved time bounds for the maximum flow problem*, SIAM J. Comput., 18 (1989), pp. 939–954.
[4] D. Bertsekas, *Distributed asynchronous relaxation methods for linear network flow problems*, Tech. Report LIDS-P-1606, MIT, Laboratory for Information and Decision Sciences, Cambridge, MA, 1986.
[5] D. Bertsekas and J. Eckstein, *Dual coordinate step methods for linear network flow problems*, Mathematical Programming, 42 (1988), pp. 202–243.
[6] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, Journal of the ACM, 21 (1974), pp. 201–208.
[7] J. Cheriyan and S. N. Maheshwari, *Analysis of preflow push algorithms for maximum network flow*, SIAM Journal on Computing, 18 (1989), pp. 1057–1086.
[8] R. Cole, *Parallel merge sort*, SIAM Journal of Computing, 17 (1988), pp. 770–785.
[9] W. Cunningham, *Optimal attack and reinforcement of a network*, Journal of the ACM, 32 (1985), pp. 549–561.
[10] E. Dinic, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.
[11] M. J. Eisner and D. G. Severance, *Mathematical techniques for efficient record segmentation in large shared databases*, Journal of the ACM, 23 (1976), pp. 619–635.
[12] L. R. Ford and D. R. Fulkerson, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.
[13] S. Fortune and J. Wyllie, *Parallelism in random access machines*, in Proceedings of the 10th Annual ACM Symposium on Theory of Computing, 1978, pp. 114–118.
[14] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan, *A fast parametric maximum flow algorithm and applications*, SIAM Journal on Computing, 18 (1989), pp. 30–55.

[15] A. V. Goldberg, *Efficient graph algorithms for sequential and parallel computers*, PhD thesis, MIT, Cambridge, MA, Jan. 1987.
[16] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum flow problem*, Journal of the ACM, 35 (1988), pp. 921–940.
[17] ———, *Solving minimum-cost flow problems by successive approximation*, Mathematics of Operations Research, 15 (1990), pp. 430–466.
[18] A. V. Goldberg, R. E. Tarjan, and E. Tardos, *Network flow algorithms*, in Paths, Flows, and VLSI-Layout, B. Korte, L. Lovász, H. Prömel, and A. Shriver, eds., Springer-Verlag, Berlin, 1990, pp. 101–164.
[19] D. Gusfield, *Connectivity and edge disjoint spanning trees*, Information Processing Letters, 16 (1983), pp. 87–89.
[20] ———, *Computing the strength of a graph*. submitted for publication, July 1989.
[21] D. Gusfield, C. Martel, and D. Fernandez-Baca, *Fast algorithms for bipartite network flow*, SIAM J. Comput., 16 (1987).
[22] W. Hillis and J. G.L. Steele, *Data parallel algorithms*, Communications of the ACM, 29 (1986), pp. 1170–1183.
[23] A. Hoffman and T. Rivlin, *When is a team "mathematically" eliminated?*, in Symposium on Mathematical Programming, Princeton University Press, Princeton, NJ, 1970, pp. 391–396.
[24] W. Horn, *Some simple scheduling algorithms*, Naval Research Logistics Quarterly, 21 (1974), pp. 177–185.
[25] A. V. Karzanov, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.
[26] T. Leighton, C. Leiserson, B. Maggs, S. Plotkin, and J. Wein, *Advanced parallel and VLSI computation*, Research Seminar Series MIT/LCS/RSS 2, MIT, 1988.
[27] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Information Processing Letters, 7 (1978), pp. 277–278.
[28] C. S. J. A. Nash-Williams, *Edge disjoint spanning trees of finite graphs*, Journal of the London Mathematics Society, 36 (1961), pp. 445–450.
[29] J. Picard and M. Querayne, *A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory*, Networks, 12 (1982), pp. 141–159.
[30] ———, *Selected applications of minimum cuts in networks*, INFOR., 20 (1982), pp. 394–422.
[31] B. Schwartz, *Possible winners in partially completed tournaments*, SIAM Review, 8 (1966), pp. 302–388.
[32] Y. Shiloach and U. Vishkin, *An $O(\log n)$ parallel connectivity algorithm*, Journal of Algorithms, 3 (1982), pp. 57–67.
[33] D. Sleator and R. Tarjan, *Self-adjusting binary search trees*, Journal of the ACM, 32 (1985), pp. 652–686.
[34] D. D. Sleator and R. Tarjan, *A data structure for dynamic trees*, Journal of Computer and System Sciences, 26 (1983), pp. 362–391.
[35] C. Stein, *Efficient algorithms for bipartite network flow*. Unpublished Manuscript, 1987.
[36] R. E. Tarjan, *A simple version of Karzanov's blocking flow algorithm*, Operations Research Letters, 2 (1981), pp. 265–268.
[37] ———, *Data structures and network algorithms*, SIAM, Philadelphia, PA, 1983.