September 1988 Report No. STAN-CS-88-1227

Finding Minimum-Cost Flows by Double Scaling

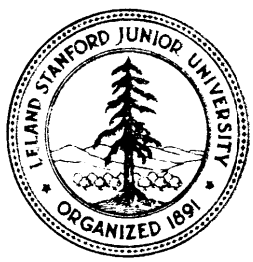
by

R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan

Department of Computer Science

Stanford University

Stanford, California 94305



Finding Minimum-Cost Flows by Double Scaling

Ravindra K. Ahuja [1,2]

Andrew V. Goldberg [3]

James B. Orlin [1]

Robert E. Tarjan [4]

September, 1988

ABSTRACT

Several researchers have recently developed new techniques that give fast algorithms for the minimum-cost flow problem. In this paper we combine several of these techniques to yield an algorithm running in O(nm log log U log(nC)) time on networks with *n vertices, m* edges, maximum arc capacity U, and maximum arc cost magnitude C. The major techniques used are the capacity-scaling approach of Edmonds and Karp, the excess-scaling approach of Ahuja and Orlin, the cost-scaling approach of Goldberg and Tarjan, and the dynamic tree data structure of Sleator and Tarjan. For nonsparse graphs with large maximum arc capacity, we obtain a similar but slightly better bound. We also obtain a slightly better bound for the (noncapacitated) transportation problem. In addition, we discuss a capacity-bounding approach to the minimum-cost flow problem.

[1] Sloan School of Management, M.I.T., Cambridge, MA 02139. Research partially supported by an NSF Presidential Young Investigator Fellowship, Contract 8451517ECS, and grants from Analog Devices, Apple Computer Inc., and Prime Computer.

[2] On leave fmm Indian Institute of Technology, Kanpur, India.

[3] Department of Computer Science, Stanford University, Stanford, CA 94305. Research partially supported by an NSF Presidential Young Investigator Award.

[4] Department of Computer Science, Princeton University, Princeton, NJ 08544 and AT&T Bell Laboratories, Murray Hill, NJ 07974. Research partially supported by National Science Foundation Grant DCR-8605962 and Office of Naval Research Contract N00014-87-K-0467.

Finding Minimum-Cost Flows by Double Scaling

Ravindra K. Ahuja [1,2]

Andrew V. Goldberg [3]

James B. Orlin [1]

Robert E. Tarjan [4]

September, 1988

1. Introduction

The minimum-cost ***circulation*** *problem* calls for finding a circulation of minimum cost in a network whose arcs have flow capacities and costs per unit of flow. Our framework for studying this problem is as follows. See e.g. [9,11,13,17,21]. Let G = (V, E) be a directed graph with vertex set V and arc set E. We require G to be ***symmetric***, i.e. $(v,w) \in E$ *if* and only if $(w,v) \in E$. Graph G is a *network* if each arc (v,w) has a nonnegative real-valued ***capacity*** $u(v,w)$ and a real-*valued cost* $c(v,w)$. We require that the cost function be ***antisymmetric,*** i.e. $c(v,w) = -c(w,v)$ for *all* arcs $(v,w) \in E$. We denote by n, m, U, and C *the* number of vertices, number of arcs, maximum arc capacity, and maximum absolute value of an arc cost, respectively. Time bounds containing U or C are subject to the assumption that all arc capacities, or all arc costs, respectively, are integral. For ease in stating time bounds, we assume (without loss of generality) that C 12 and $U \geq 4$. All logarithms in this paper are base two unless an explicit base is given.

A ***pseudoflow*** fon a network G is a real-valued function on the arcs satisfying the following two constraints:

$$f(v,w) \leq u(v,w) \text{ for all } (v,w) \in E \text{ (capacity constraint)} \qquad (1)$$

[1] Sloan School of Management, M.I.T., Cambridge, MA 02139. Research partially supported by an NSF Presidential Young Investigator Fellowship, Contract 8451517ECS, and grants from Analog Devices, Apple Computer Inc., and Prime Computer.

[2] On leave fran Indian Institute of Technology, Kanpur, India

[3] Department of Computer Science, Stanford University, Stanford, CA 94305. Research partially supported by an NSF Presidential Young Investigator Award.

[4] Department of Computer Science, Princeton University, Princeton, NJ 08544 and AT&T Bell Laboratories, Murray Hill, NJ 07974. Research partially supported by National Science Foundation Grant DCR-8605962 and Office of Naval Research Contract N00014-87-K-0467.

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

For a pseudoflow fand a vertex v, the *balance* $b_f(v)$ at v, is the net flow into v:

$$b_f(v) = \sum_{(u,v) \in E} f(u,v). \quad (3)$$

The *cost* of a **pseudoflow** f is defined as follows:

$$cost(f) = \sum_{f(v,w) > 0} c(v,w)\, f(v,w) \quad (4)$$

A pseudoflow f is a ***circulation*** if the following constraint is satisfied:

$$b_f(v) = 0 \text{ for every vertex v.} \quad (5)$$

The minimum-cost ***circulation*** *problem* is that of finding a circulation of minimum cost in a given network.

The minimum-cost circulation problem has been intensively studied for over **thirty years.** See e.g **[9,11,13,17,21].** Among the known algorithms for this problem, there are three that have the best worst-case time bounds. Each of these algorithms is best for a different range of the parameters n, m, U, and C. **The** algorithms are the $O((m \log U)(m + n \log n))$**-time** method of Edmonds and **Karp [6],** *the* $O(nm \log(n^2/m) \log(nC))$**-time** method of Goldberg and **Tarjan [11],** and the $O((m \log n)(m + n \log n))$**-time** method of Orlin **[16].** The last of these methods is strongly polynomial *.

One important idea is common to all three of these algorithms, that of *scaling* or *successive approximation.* Scaling methods work by solving a sequence of more-and-more accurate approximations to the original problem. The approximations are obtained either by relaxing some of the numerical constraints or by ignoring some of the precision of the numeric parameters. Scaling was introduced by Edmonds and Karp, whose algorithm scales capacities. Orlin's algorithm is a refinement of that of Edmonds and Karp that combines capacity scaling with repeated arc-

* A network algorithm is strongly ***polynomial*** if its running time is polynomial in n and m, assuming arithmetic operations take unit time, and also polynomial in n , m, log$U$, and log$C$, assuming arithmetic operations take time polynomial in the number of bits of the operands. See [20].

shrinking. The algorithm of Goldberg and Tajan scales costs. It relies crucially on the notion of **ε-optimality,** introduced by **Tardos [20]** and independently by Bertsekas **[3,4].**

Consideration of these algorithms suggests the question of whether capacity scaling and cost scaling can be combined to yield an algorithm faster than any algorithm obtainable using either technique alone, at least for a suitable range of n, m, U, and C. A first result along these lines was obtained by Gabow and **Tarjan [8],** who developed an $O(nm \log n \log U \log(nC))$-time algorithm. Although this time bound is never less than that of Goldberg and **Tarjan** [11], the algorithm does not require sophisticated data structures, whereas the **Goldberg-Tarjan** algorithm uses both dynamic trees [**18,19,21**] and finger trees **[14,22].**

Our work is a continuation of efforts in this direction. We obtain an O(nm log $\log U$ log (nC))-time algorithm for the minimum-cost circulation problem. Our result **combines** four known ideas:

(1) Elimination of arc capacities by transforming the minimum-cost circulation problem into a transportation problem.

(2) Cost scaling within the e-optimality framework as proposed by Goldberg and **Tarjan.**

(3) A variant of the **Edmonds-Karp** approach relying on excess scaling, as developed by **Orlin.**

(4) The dynamic tree data structure of Sleator and **Tarjan.**

A simpler version of our algorithm that does not use the dynamic tree data structure runs in O(nm $\log U$ $(1 + \log(nC)/\log\log U)$) time. We obtain slightly better bounds for **nonsparse** graphs with very large arc capacities. We also obtain improved bounds for the **(uncapacitated) transportation** problem.

Step **(1),** the elimination of arc capacities, is crucial to the efficiency of our algorithms. An alternative approach is to bound the arc capacities by adding an extra outer capacity-scaling loop, as suggested by Gabow and **Tarjan [8].** Our explorations of this approach lead to algorithms with **time** bounds worse than those mentioned above, but the analytical methods we develop are of independent interest. **Our** results using this approach are described toward the end of the paper. For example, we obtain a polynomial-time algorithm for the problem that uses a classical network simplex algorithm inside a scaling loop.

This paper consists of six sections in addition to the introduction. In Section 2 we define the transportation problem and discuss its relationship with the minimum-cost circulation problem. In Section 3 we develop a generic algorithm for the transportation problem based on cost scaling and **ε-optimality.** In Section 4 we refine the generic algorithm to use excess scaling, and we analyze the resulting method. In Section **5** we add the use of dynamic trees. In Section 6 we

consider the use of capacity bounding as an alternative way of dealing with arc capacities. In Section 7 we summarize our results, comment on the possible practicality of our algorithms, and mention some open problems.

2. The Transportation Problem

The minimum-cost circulation algorithms we develop in Sections 3-5 will be stated in terms of a related problem, the *transportation problem.* In order to discuss this problem, we need some terminology. We *call G bipartite* if V *can be* partitioned into two sets S and T ($S \cup T = V$, $S \cap T = \phi$) such that every arc has exactly one vertex in S and one in T. We call vertices *in S sources* and those in T *sinks; we* denote by n_1 and n_2 *the* sixes of S and T, respectively. We call a bipartite network *uncapacitated* if $u(v,w) = \infty$ for each arc (v,w) with $v \in S$ and $u(v,w) = 0$ for each arc (v,w) with $v \in T$. A *supply-demand vector* d on a bipartite network is a mapping from V to *the real* numbers such that $d(v) \leq 0$ if $v \in S$ ($-d(v)$ is the *supply* at vertex v), $d(v) \geq 0$ if $v \in T$ ($d(v)$ is the *demand* at vertex v), and $\sum_{v \in V} d(v) = 0$ (total supply equals total demand). Given an uncapacitated bipartite network G and a supply-demand vector d, *the transportation problem* is that of finding a minimum-cost pseudoflow f satisfying the following constraint:

$$b_f(v) = d(v) \text{ for all } v \in V \text{ (supply-demand constraint)} \tag{6}$$

We call a pseudoflow *feasible* if it satisfies (6). We call a transportation problem *feasible* if it has some feasible pseudoflow. Checking the feasibility of a transportation problem can be done using any maximum flow algorithm, e.g. [1,2,9,10].

There is a well-known, simple transformation that will convert any minimum-cost circulation problem into an equivalent transportation problem [17,23]. Given a network $G = (V,E)$ *we* construct another network $G' = (V \cup E, A)$, where A contains arcs $((v,w),v)$, $((v,w),w)$ and their reversals for every arc $(v,w) \in E$. The arcs $((v,w),v)$ and $((v,w),w)$ have infinite capacity; their reversals have zero capacity. Arc $((v,w),v)$ has cost zero and arc $((v,w),w)$ has cost $c(v,w)$. We define a supply-demand vector *don* G' by

$$d((v,w)) = -u(v,w) \text{ for all } (v,w) \in E, \tag{7}$$

$$d(v) = \sum_{(v,w) \in E} u(v,w) \text{ for all } v \in V.$$

Any circulation f on G corresponds to a feasible pseudoflow f' on G' such that $cost(f') = cost(f)$, given by $f'((v,w),w) = f(v,w)$, $f'((v,w),v) = u(v,w) - f(v,w)$ for each arc $(v,w) \in E$. This correspondence is invertible. Thus a solution to the transportation problem on G' gives a solution to the minimum-cost circulation problem on G. Observe that if we regard E as being the set of sources of G' and V as being the set of sinks, G' has $n_1 = m$, and $n_2 = n$; arc set A has size $4m$.

We shall derive time bounds for the transportation problem and translate them into time bounds for the minimum-cost circulation problem based on the above transformation.

3. A Generic Algorithm for the Transportation Problem

We obtain a generic algorithm for the transportation problem by translating the minimum-cost circulation algorithm of Goldberg and Tarjan [9,11] into the setting of the transportation problem. We modify the algorithm to be an augmenting path method; the time bounds we derive depend on this modification. We omit proofs of many of the basic results, since they are direct translations of the proofs of Goldberg and Tarjan.

Let $G = (V = S \cup T, A)$ be an uncapacitated bipartite network with source set S of size n_1, sink s&t T of size n_2, arc set A of size m, and supply-demand vector d. We denote the total size of V, i.e., $n_1 + n_2$, by n, and $\min\{n_1, n_2\}$ by no. We denote by U the maximum supply, i.e. $U = \max\{-d(v) \mid v \in S\}$, and by C the maximum absolute value of an arc cost. Note that U and C are defined so that the transformation of Section 2 from a minimum-cost circulation problem to a transportation problem preserves the values of U and C.

For a pseudoflow f on G, we define the excess $e_f(v)$ of a vertex v by

$$e_f(v) = b_f(v) - d(v). \tag{8}$$

Thus a pseudoflow is feasible if every vertex has zero excess. We shall assume that the transportation problem to be solved is feasible, i.e. there is some feasible pseudoflow.

Given a pseudoflow f, the *residual capacity* of an arc (v,w) with respect to f is

$$u_f(v,w) = u(v,w) - f(v,w). \tag{9}$$

Arc (v,w) is *unsaturated* if $u_f(v,w) > 0$ and *saturated otherwise.*

A *price function* p on G is a real-valued function on the vertices. Given a price function p, *the reduced cost* of an arc (v,w) is

$$c_p(v,w) = c(v,w) + p(v) - p(w). \tag{10}$$

Let $\varepsilon > 0$, let f be a pseudoflow, and let p be a price function. Pseudoflow f is &-optimal with respect to price **function** p if

$$c_p(v,w) \geq -\varepsilon \text{ for every unsaturated arc } (v,w) \text{ (}\varepsilon\text{-optimality constraint).} \tag{11}$$

Pseudoflow f is *optimal* with respect top if it is **ε-optimal** for $\varepsilon = 0$. The following theorem is a classical result of network flow theory and follows from the duality theorem of linear programming.

Theorem 3.2 **[11].** A feasible pseudoflow is of minimum cost if and only if it is optimal with respect to some price function *p.*

As Bertsekas **[4]** discovered, a weaker condition **suffices** if **all** arc costs **are** integers:

Theorem 3.2. If all arc costs are integers and ε c $\frac{1}{2n_0}$, then a feasible flow is of minimum cost if and only if it is c-optimal with respect to some price function *p.*

Proof. Analogous to the proof of Theorem 2.3 of **[11],** using the fact that G is bipartite and hence any simple cycle contains at most $2n_0$ *vertices. Cl*

In the remainder of this paper (except in some of the concluding remarks of Section **7),** we **shall** assume that all arc costs **are** integers; thus **Theorem** 3.2 applies.

Our algorithm applies *cost scaling based* on Theorem *3.2.* It *uses* a *cost-scaling factor k 2 2.* It maintains a price function *p* and *an error parameter* ε. *Initially* $\varepsilon = C$ and *p* is identically zero. **The** algorithm consists of repeating the following step until the termination condition is satisfied.

Cost-Scaling Step. Let f be the identically zero pseudoflow. By modifying f and p, find a feasible pseudoflow f' and a price function p' such that f' is e-optimal with respect to p'. If $\varepsilon < \frac{1}{2n_0}$, stop. Otherwise, let *p be* **defined** by $p(v) = p'(v) + \varepsilon$ if v ∈ S, $p(v) = p'(v)$ if v ∈ *T;* replace ε by ε/k.

Note that only the price function p is carried over from iteration to iteration; the pseudoflow is reset to zero after each iteration.

Lemma 3.3. At the beginning of a cost-scaling step, *f (the zero* pseudoflow) is ~-optimum with respect to *p.*

Proof. Any arc (v,w) that is unsaturated with respect to f has $v \in S$ and $w \in T$. Suppose that the current cost-scaling step is the first. Then $c_p(v,w) = c(v,w) \geq -C = -\varepsilon$. Suppose on the other hand that the cost scaling step is not the first. Let f' be the pseudoflow and p' *the* price function computed in the previous step. Then (v,w) is unsaturated with respect to f', since $u(v,w) = \infty$, but any pseudoflow has all arc flows finite. Thus $c_p(v,w) = c_{p'}(v,w) + k\varepsilon \geq -k\varepsilon + k\varepsilon \geq 0$, since f' is $k\varepsilon$ optimal with respect top', and $p(v) = p'(v) + k\varepsilon$, $p(w) = p'(w)$. □

Theorem 3.4. The transportation algorithm is correct and terminates after $0(1 + \log_k(n_0 C))$ iterations.

Proof., Correctness follows from Theorem 3.2. The bound on the number of iterations is obvious. □

The heart of the algorithm is the conversion of an e-optimal pseudoflow into an ~-optimal feasible pseudoflow and the corresponding modification of the price function. We call this the. *refinement computation.* Our generic refinement algorithm consists a sequence of two kinds of local transformations, one of which modifies the pseudoflow and the other of which modifies the price function. To define the transformations, we use the following terminology. A vertex v is *active* if $e_f(v) > 0$. *An unsaturated arc* (v,w) is *eligible* if $c_p(v,w) < 0$. The refinement algorithm consists of repeating the following steps, in any order, until no vertex is active, and then defining $f = f$, $p' = p$:

push (v,w):

Applicability: Vertex v is active, $u_f(v,w) > 0$, and $c_f(v,w) \leq \varepsilon$.

Action: Push up to $\delta = \min\{ e_f(v), u_f(v,w) \}$ units of flow from v to w by increasing $f(v,w)$ by an amount up to 6.

relabel (v).

Applicability: Vertex v is reachable from some active vertex by a path of eligible arcs, and there is no eligible arc (v,w).

Action: Replace $p(v)$ by $\max\{p(w) - c(v,w) - \varepsilon\}$.

Lemma 3.5. Any pushing or relabeling step preserves e-optimality. A relabeling of a vertex v decreases $p(v)$ by at least ε.

Proof Analogous to the proofs of Lemmas 5.2 and 5.3 of [11]. □

Lemma 3.6. The price of a vertex v decreases by $0(kn_0\varepsilon)$ during refinement. Hence v is relabeled $O(kn_0)$ times.

Proof. Analogous to the proof of Lemmas 5.7, 5.8, and 5.9 of [11]. The bound on price changes in Lemma 5.7 of [11] is $0(n\varepsilon)$, where the cost scaling is by a factor of two. Revising the argument to include a cost scaling factor of k yields an $0(kn\varepsilon)$ bound. Observing that G is bipartite, and hence that any simple path in G contains at most $2n_0$ *arcs, reduces the* bound to $0(kn_0\varepsilon)$. □

Now we describe a version of the refinement algorithm that is based on the idea of finding augmenting paths. The algorithm uses a fixed incidence list $I(v)$ for each vertex v. This list contains each arc (v,w). One such arc is designated the *current arc* out of v. Initially the current arc out of v is the first arc on I(v). The algorithm repeatedly attempts to find a path of eligible arcs from an active vertex to a vertex of negative excess. When such a path is found, flow is pushed along it. To find such paths, the algorithm uses depth-first search, implemented using a stack S. During a search, vertices are relabeled as necessary to extend the path. The algorithm consists of initializing S to be empty and repeating the following steps until termination occurs in Step 1.

Step I (start new path). If there are no active vertices, stop. Otherwise, select some active vertex v and push it onto S. Go to Step 2.

Step 2 (extend path). Let v be the top vertex on S. While the current arc of v is not eligible, replace the current arc by the next arc on I(v). If this eventually produces an eligible current arc (v,w), push w onto S and go to Step 4. If the end of $I(v)$ is reached without finding an eligible arc, go to Step 3.

Step 3 (relabel). Relabel v, the top vertex on S. Reset the current arc of v to be the first arc on I(v). If v is not the only vertex on S, pop it from S. Go to Step 2.

Step 4 (augment). Let w be the top vertex on S. If ew) ≥ 0, go to Step 2. Otherwise, let δ be any positive quantity not more than the minimum of $e_f(v)$ and $\min\{u_f(x,y) \mid x \text{ is on S}, x \neq w, \text{ and}$

(x,y) is the current arc out of x}. For each current arc (x,y) such that x is on S and $x \neq w$, increase $f(x,y)$ by 6. Empty S and go to Step 1.

We call *this* method *the augmenting path version* if the refinement algorithm, or the *augmenting path algorithm* for short. We call an execution of Step 4 that actually moves flow an *augmentation.*

Lemma 3.7. The augmenting path algorithm maintains the invariant that there is no cycle of eligible arcs.

Proof. Analogous to the proof of Corollary 5.6 of [11]. Cl

Remark. The proof of Lemma 3.7 uses the fact that pushes take place only along eligible arcs. Cl

Lemma 3.8. The maximum size of S is at most $2n_0$.

Proof., The vertices on S always define a path of eligible arcs. By Lemma 3.7 such a path is simple. The fact that G is bipartite gives the claimed bound on the size of S. Cl

Theorem 3.9. The augmenting path algorithm is correct and runs in $O(kn_0m)$ time plus $O(n_0)$ time per augmentation.

Proof. Correctness follows from **Lemma 3.5.** We bound the running time as follows. The number of additions to S equals the number of pops from S. The number of pops from S is $O(n_0)$ per augmentation by **Lemma** 3.8 plus at most one per relabeling. The time to relabel a vertex v is 0(I $I(v)$ I), which is also the time spent in Step 2 changing current arcs of v between relabelings. By Lemma 3.6, the relabeling time and time spent changing current arcs, summed over all vertices, is $O(kn_0m)$. *An* execution of Step 2 that does not change the current arc of v causes an addition to S. The time to do an augmentation is $O(n_0)$. *The* claimed time bound follows. Cl

4. Bounds for the Augmenting Path Algorithm

In this section we derive time bounds for various versions of the augmenting path algorithm. Observe that there are two kinds of freedom in this algorithm, in the choice of starting vertices for augmenting paths in Step 1, and in the amount by which the flow is augmented in Step 4.

Let us first analyze the simple method in which each augmentation is by an amount that is as large as possible; that is, in Step 4, δ is selected as follows: $\delta = \min \{ e_f(v), \min \{u_f(x,y) \text{ I } x \text{ is}$

on $S, x \neq$ w, and (x,y) is a current am}). With this method, each augmentation either reduces the number of active vertices by one, reduces the number of vertices of negative excess by one, or saturates an arc. Lemma 3.6 implies that the total number of arc saturations is $O(kn_0m)$ (see Lemma 5.10 of [11]); hence the total running time of the augmenting path algorithm is $O(kn_0^2m)$, or $O(n_0^2m)$ if k is chosen equal to two. This bound is analogous to Dinic's bound of 0 (n^2m) for the maximum flow problem [5]; indeed, the augmenting path algorithm itself can be viewed as an analogue of Dinic's algorithm.

We obtain a better bound (if all arc capacities are integral and not too large) by using excess scaling. This method is based on the capacity-scaling algorithm of Edmonds and Karp [6] for the minimum-cost circulation problem and is also analogous to the maximum flow algorithm of Gabow [7]. Henceforth (except in Section 7) we shall assume that all arc capacities are integral.

The excess-scaling algorithm maintains an estimate A of the maximum excess. Initially A is the largest power of two not exceeding U. The algorithm maintains two invariants:

(i) The sum of all positive excesses is at most $2nA$;

(ii) The residual capacity of any arc is either infinity or an integer (possibly zero) multiple of A.

In Step 1, the algorithm always chooses a starting vertex v with $e_f(v) \geq A$; if no such vertex exists, the algorithm replaces A by A/2 and tries again. In Step 4, the algorithm always pushes A units of flow along the augmenting path. The choice of starting vertices guarantees that invariant (i) is maintained; immediately after A changes, the sum of all positive excesses is at most $2n\Delta$. Augmenting by A preserves invariant (ii), which in turn guarantees that A units of flow can actually be pushed each time an augmentation occurs. When A first becomes less than one, all excesses are zero, and the algorithm terminates. We call a maximal period of time during which A stays constant a *phase* of the algorithm.

Lemma 4.1. The total number of augmentations done by the excess-scaling algorithm is O(n log U).

Proof. Each augmentation either reduces the number of vertices with negative excess by one or reduces the sum of positive excesses by A. By (i), the latter case can occur only $O(n)$ *times* during a phase. The number of phases is $O(\log U)$. The bound follows. □

Theorem 4.2. The excess-scaling version of the augmenting path algorithm runs in

$O(n_0(km + n\log U))$ time. Using this method in the transportation algorithm gives a bound for the transportation problem of $O(n_0(km + n \log U) \log_k (nC))$ *time* for any k such that $2 \le k \le nC$. Choosing $k = \min\{2 + \frac{n}{m} \log U, nC\}$ yields the following time bounds for the transportation problem:

$$O(n_0 m \log (nC)) \text{ if } \log U < 2m/n;$$

$$O(n_0 n \log U (1 + \log (nC)/\log (\frac{n}{m} \log U))) \text{ if } \log U \ge 2m/n.$$

Proof. Immediate from Lemma 4.1 and Theorem 3.4. □

Corollary 4.3. The excess-scaling version of the transportation algorithm combined with the transformation of Section 2 will solve a minimum-cost circulation problem in $O(nm \log U (1 + \log (nC)/\log \log U))$ time.

By changing the excess-scaling algorithm slightly, we can obtain a bound of $O(n_0 \log U + n \log \min\{n, U\})$ on the number of augmentations. This is an improvement on the bound of Lemma 4.1 only if $\log U = \omega((n/n_0) \log n)$, which only holds if U grows nonpolynomially with n. Nevertheless, we shall present the result, since it suggests the possibility of obtaining an $O(n \log n)$ bound on the number of augmentations for some suitable modification of the algorithm. We shall assume that $n_1 \ge n_2$, i.e. $n_0 = n_2$, which is without loss of generality: if $n_1 < n_2$, exchange the source set and the sink set and negate the supply-demand vector.

We modify the excess-scaling algorithm by changing Step 1 to the following:

Step 1′. If there are no active vertices, stop. Otherwise, if some active vertex $v \in S$ has an outgoing arc (v,w) such that $f(v,w) > 7n\Delta$, increase the flow on (v, w) by $e_f(v)$ and repeat Step 1′. *(We call this* a *special push.)* Otherwise, if some active vertex v has $e_f(v) \ge \Delta$, push vertex v onto S and go to Step 2. Otherwise, replace Δ by $\Delta/2$ and repeat Step 1′.

Before analyzing the modified algorithm in detail, we make several observations. A special push is actually a push, since iff $(v,w) > 0$ then $u_f(w,v) > 0$, which implies by ε-optimality that $c_p(w,v) \ge -\varepsilon$ and by cost antisymmetry that $c_p(v,w) \le \varepsilon$. A special push maintains the invariant that there is no cycle of eligible arcs. Once a vertex $v \in S$ has zero excess, its excess remains zero until the end of the algorithm. The excess on any vertex $v \in S$ never exceeds 2Δ. The total

flow moved by special pushes is thus at most $2nA$. The total flow moved during augmentations that decrease the number of vertices with negative excess is at most n A. The total flow moved by other augmentations during a single phase is at most the sum of the positive excesses, which is at most $2nA$. *Thus the* total flow moved from a given time until the end of the algorithm is at most $3nA + \sum_{i=0}^{\infty} 2n\Delta/2^i = 7n\Delta$. It follows that once an arc (v,w) has flow exceeding $7n\Delta$, its flow remains positive until the end of the algorithm, and (w,v) can never be saturated. We call an arc (v,w) that can never be saturated *open*. The modified algorithm maintains invariant (i) (the sum of all positive excesses is at most 2nd) and, in place of invariant (ii), the following:

(ii)′ Every arc (v, w) is either open or has a residual capacity that is an integer (possibly zero) multiple of A.

We can verify invariant (ii)′ by induction on the number of steps taken by the algorithm, simultaneously showing that every augmentation in Step 4 can actually move A units of flow.

Lemma 4.4. The total number of augmentations made by the modified excess-scaling algorithm is $0(n \log U + n \log \min\{n, U\})$.

Proof. Consider a vertex $v \in S$. There are at most two augmentations starting from v per phase. Suppose that the first augmentation from v is during phase i. This augmentation moves A units of flow. Henceforth until the end of the algorithm there is always an arc (v,w) with $f(v,w) \geq A/n$. After $2 \log n + 3$ more phases, the current value of the excess estimate is $A' = \Delta/8n^2$, and there is some arc (v,w) with $f(v,w) \geq 8n\,A'$. When such an arc exists, if not before, the excess at v is reduced to zero by a special push. Hence v can have positive excess only during $0(\log \min\{n, U\})$ phases, and there are $O(n \log \min\{n, U\})$ augmentations starting from vertices in S.

Now consider a vertex $v \in T$. If v does not receive additional flow from special pushes during a phase, there can be at most two augmentations starting from v during the phase. A special push can move up to 2A units of flow to v, which can account for at most two augmentations starting from v during the phase. We charge such augmentations to the corresponding special pushes. Since there are only n special pushes, the number of augmentations starting from vertices in T, summed over all phases, is $0(n + n_0 \log U)$. This gives the desired bound. Cl

In presenting time bounds for the modified method, we assume that $\log U = \Omega((n/n_0) \log a)$, since otherwise the bounds are the same as those in Theorem 4.2 and Corollary 4.3.

Theorem 4.5. Assume that $\log U = \Omega((n/n_0)\log n)$. Then the modified excess-scaling version of the augmenting path algorithm runs in $O(n_0(km + n_0 \log U))$ time. Using this method in the transportation algorithm gives a time bound for the transportation problem of $O(n_0(km + n_0 \log U) \log_k (n_0 C))$ for any k such that $2 \le k \le n_0 C$. *The* choice of $k = \min\{2 + \frac{n_0}{m} \log U, n_0 C\}$ yields the following time bounds for the transportation problem:

$$O(n_0 m \log (n_0 C)) \text{ if } \log U < 2m/n_0;$$

$$O(n_0^2 \log U (1 + \log (n_0 C)/\log (\frac{n_0}{m} \log U))) \text{ if } \log U \ge 2m/n_0.$$

Corollary 4.6. If $\log U = \Omega(\frac{m}{n} \log n)$, *the* modified excess-scaling version of the transportation algorithm combined with the transformation of Section 2 will solve a minimum-cost circulation problem in $O(n^2 \log U(1 + \log (nC)/\log \log U))$ time.

Remark. Every bound derived in this Section remains valid if each occurrence of the parameter U is replaced by another smaller parameter U^*. For the transportation problem, $U^* = 4 + \sum_{v \in S} (-d(v))/n$. For the minimum-cost circulation problem, $U^* = 4 + \sum_{(v,w) \in E} u(v,w)$. The bound on the number of augmentations in Lemma 4.1 can be reduced to $O(n \log U^*)$ by observing that the sum of positive excesses is initially at most nU^*, which implies that the number of augmentations during phases in which $A > U^*$ is $O(n)$. Similarly, the bound in Lemma 4.4 can be reduced to $O(n \log U^* + n \log \min\{n, U^*\})$. Corresponding improvements in the bounds of Theorem 4.2, Corollary 4.3, Theorem 4.5, and Corollary 4.6 follow. These improved bounds are analogous to the bound Edmonds and Karp obtained for their transportation algorithm [6].

5. Use of Dynamic Trees

The algorithms discussed in Section 4 are quite simple and do not require the use of any complicated data structures. By adding the use of dynamic trees [18,19,21], we can improve the bounds derived in Section 4 by almost a logarithmic factor. Our use of dynamic trees is analogous to their use in other network flow algorithms [2,9,10,11,12,18,21].

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each arc of which has an associated value. Each tree is an in-tree; that is, if vertex v is a child of vertex w, there is a tree arc from v to w. Each vertex in a tree is regarded as being both an ancestor and a descendant of itself. The data structure supports the following seven operations:

find-root(*v*): Find and return the root of the tree containing vertex v.

find-size(v): Find and return the number of vertices in the tree containing vertex v.

jind-value(v): Find and return the value of the tree arc leaving *v*. If v is a tree root, the value returned is infinity.

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

change-value(*v*,*x*): Add real number *x* to the value of every arc along the path from v to find-root(v).

link(*v*,*w*,*x*): Combine the trees containing v and w by making w the patent of v and giving the arc (*v*,*w*) the value x. This operation does nothing if v and w are in the same tree or if *v* is not a tree root.

cut(v): Break the tree containing v into two trees by deleting the edge joining v to its parent; return the value of the deleted edge. This operation breaks no edge and returns infinity if *v* is a tree root.

A sequence of *l* tree operations, starting with an initial collection of singleton trees, takes $O(l \log (z + 1))$ time if *z* is the maximum tree size [10,17,19,21].

We use this data structure to represent a subset of the eligible current arcs. The value of an arc is its residual capacity. The data structure allows flow to be moved along an entire path at once, rather than along one arc at a time.

In applying this data structure to the transportation problem, we can improve the resulting time bounds if we take advantage of the special structure of the problem, specifically the fact that G is bipartite, and hence so is every dynamic tree. Let us assume that $n_1 \geq n_2$, i.e. $|S| \geq |T|$. We redefine the *size* of a dynamic tree to be the number of vertices of *T* it contains. This changes

the semantics of the find *size* operation, but does not affect its implementation significantly. We also modify the data structure so that any dynamic tree contains at most twice as many vertices as its size. To do this we introduce an extra layer of abstraction. We represent each of the actual dynamic trees (the ones manipulated by the operations) by a *virtual dynamic tree,* which consists of the actual tree with all leaves in S deleted. Each of the deleted leaves has a pointer to its parent in the actual tree, and has stored with it the value of the outgoing **tree** arc. Every virtual **tree** contains a number of vertices at most twice its size, since every virtual **tree** vertex in S has a virtual tree child in T. Every operation on actual trees translates into O(1) operations on virtual trees. It follows that a sequence of l operations on actual dynamic **trees** takes $O(l \log (z + 1))$ time, where z is the maximum tree size according to the new definition of size.

The following version of the excess-scaling algorithm uses these modified dynamic trees. In addition to an excess estimate A, the algorithm uses a fixed bound z, $1 \le z \le n_0$, on the maximum size of a dynamic tree. The algorithm maintains a stack S that defines a path of eligible current arcs as follows: if vertex v appears just below vertex w on S, then the tree path from v to *find-root*(v) followed by the arc (*find-root*(v),w) is a path of eligible current arcs. Initially S is empty and each vertex forms a one-vertex dynamic **tree.** The algorithm consists of repeating the following steps until termination occurs in Step 1.

Step I (start new path). If no vertex has positive excess, stop. Otherwise, if no vertex has excess at least A, replace A by A/2 and **repeat** Step 1. Otherwise, let v be a vertex of excess at least A.. Push v onto S and go to Step 2.

Step 2 (extend path). Let v be the top vertex on S. Compute w = *find root (v). If* $e_f(w) < 0$, *go* to Step 4. Otherwise, while the current arc of w is not eligible, replace the current arc of w by the next arc on I (w). If the end of I (w) is reached without finding an eligible arc, go to Step 3. If an eligible arc (w,x) is found, test whether *fin&size(v) + find-size(x)* $\le z$. If so, perform *link*(w,x, $u(v,w) - f(v,w)$). If not, push x onto S. Repeat Step 2.

Step 3 (relabel). Relabel w. For each tree arc (y,w), perform *cut*(y). If v = w and v is not the only vertex on S, pop v from S. Go to Step 2.

Step 4 (augment). Add A to $e_f(w)$.

Step 4a. Perform *change-value (v,* -A). *While find-value*(*find-min*(v)) = *0,* **perform** *cut* (*find-min*(v)). *Go* to Step 4b.

Step 4b. Pop v from S. If S is empty, subtract A from e_f (v) and go to Step 1. Otherwise, let x = v and replace v by the new top vertex on S. Let w = *find-root*(v). Add A to $f(w,x)$ and go to Step 4a.

This algorithm stores flow explicitly for arcs that are not in dynamic trees and implicitly for tree arcs. Whenever a cut is performed, the arc cut must have its flow restored to its correct current value. When the algorithm terminates, every arc still in a dynamic tree must have its correct flow computed. These computations have been omitted from the description above.

The analysis of this algorithm is similar to the analysis of other network flow algorithms that use dynamic trees, e.g. [2,9,10,11,12,18,21]. Since with this method the time bound for the transportation problem is not improved by using a non-constant cost-scaling factor, we shah choose $k = 2$. *The total* number of links and cuts performed in the dynamic tree version of the excess-scaling algorithm is $O(n_0 m)$ *(see e.g.* Lemma 7.2 of [11]), taking time $O(n_0 m \log(z+1))$. The proof of Lemma 4.1 is valid for this version of the excess-scaling algorithm, which means that there are $O(n \log U)$ augmentations. The definition of the algorithm guarantees that if v and w are consecutive vertices on S, and w is not the top vertex on S , then *find-size(v) + @d-size(w)* $> z$, i.e. either the tree containing v or the tree containing w has size exceeding $z/2$. Since every vertex on S is in a different dynamic tree, the maximum height of S is $O(n_0/z)$, and the time per augmentation is $O((n_0/z) \log(z+1))$. Thus we obtain the following result.

Theorem 5.1. The dynamic tree version of the excess-scaling algorithm runs in $O(n_0 (m + \frac{n}{z} \log U) \log(z+1))$ time, for any z satisfying $1 \le z \le n_0$. With this method, the transportation algorithm runs in $O(n_0 (m + \frac{n}{z} \log U) \log(z+1) \log(n_0 C))$ *time.* Choosing $z = \min\{1 + \frac{n}{m} \log U, n_0\}$ gives the following time bounds for the transportation problem:

$$O(n_0 m \log(2 + \frac{n}{m} \log U) \log(n_0 C)) \text{ if } \log U < n_0 m/n;$$

$$O(n \log U \log n_0 \log(n_0 C)) \text{ if } \log U \ge nom\ ln.$$

Remark. The bound *in* Theorem 5.1 for *the case* $\log U \ge n_0 m/n$ is not an interesting one, since a better bound of $O(n_0 m \log n_0 \log(n_0 C))$ can be obtained by implementing the generic augmenting path algorithm (without excess scaling) using dynamic trees. □

Corollary 5.2. The dynamic tree implementation of excess scaling can be used to solve the minimum-cost circulation problem in O(nm log $\log U$ log (n_0C)) time if log $U \le n$. If log $U > n$, a bound of O(nm *logn* log (n_0C)) is obtainable with a dynamic tree implementation of the augmenting path algorithm without excess scaling.

. We can reduce the bound on the number of augmentations to $O(n_0 \log U + n \log \min\{n, U\})$ by modifying the excess-scaling method as in Section 4. This leads to the following results.

Theorem 5.3. Assume that log $U = \Omega((n/n_0)\, logn)$. Then the dynamic tree implementation of the modified excess scaling algorithm runs in $O(n_0 (m + \frac{n_0}{z} \log U) \log (z + 1))$ time, for any z satisfying $1 \le z \le n_0$. With this method, the transportation algorithm runs in $O(n_0 (m + \frac{n_0}{z} \log U) \log (z + 1) \log (n_0C))$ time. Choosing z = min $(1 + \frac{n_0}{m} \log U, n_0)$ *gives* the following time bounds for the transportation problem:

$$O(n_0 m \log (2 + \frac{n_0}{m} \log U) \log (n_0C)) \text{ if } \log U < m;$$

$$O(n_0 \log U \log n_0 \log (n_0C)) \text{ if } \log U \ge m.$$

Corollary 5.4. If log $U = \Omega(\frac{m}{n}\, logn)$, *the* minimum-cost circulation problem can be solved in $O(nm \log (2 + \frac{n}{m} \log U) \log (nC))$ *time.*

Remark. Every bound derived in this Section remains valid if each occurrence of U is replaced by U^*, where U^* is as defined in the remark at the end of Section 4. This follows from the corresponding improvements in the bounds of Lemmas 4.1 and 4.4 discussed in that remark. Cl

6. The Capacity Bounding Technique

The results derived in Sections 3-5 depend crucially on the elimination of arc capacities via the transformation to a transportation problem discussed in Section 2. One may ask whether there is some more direct, or at least alternative, way to deal with arc capacities. A question that turns out to be related is whether the Ahuja-Orlin excess-scaling algorithm for the maximum flow problem [1,2] generalizes in a natural way to the minimum-cost flow problem via cost scaling, in

analogy with the generalization of other maximum flow algorithms to this problem [9,10].

In this section we show that the answer to both these questions is a qualified "yes." We consider the minimum-cost circulation problem as defined in Section 1, with integer capacities and costs. We propose a way of solving this problem using an outer capacity-scaling loop whose effect is to convert the original problem into a sequence of $O(\log U)$ problems in each of which the arc capacities are integers bounded by m. This idea was used by Gabow and Tarjan [8]. The method requires a standard "vertex-splitting" transformation. To solve the resulting capacity-bounded problems, we propose a modification of the Ahuja-Orlin excess-scaling maximum flow *algorithm* nested inside *an e-scaling* loop. The resulting *triple scaling algorithm runs* in $O((n^2 \log m + nm) \log U \log(nC))$ time using no fancy data structures. Although this algorithm has an inferior complexity bound as compared to the bounds obtained in previous sections, the method and its analysis has independent interest.

Now we give details. The outer capacity-scaling loop constructs two minimum-cost circulation problems at each iteration, a *target problem* and a *restricted (capacity-bounded) problem.* These are obtained from the current network as follows. First, the next bit of precision is added to the arc capacities by doubling the current capacity and adding one to the capacity of each arc (v,w) such that the current bit of $u(v,w)$ is 1. We denote the resulting capacity function by u'. The target problem is (G,u',c), where G is the original graph and c is the original cost function.

The restricted problem is obtained from the target problem by bounding the flow through every vertex of the target problem by m. More formally, the restricted problem is obtained by splitting each vertex v of G = (V,E) *into two to obtain the* graph G' = (V',E'), where V' contains vertices v_1 and v_2 for each $v \in V$, and E' *contains an arc* (v_1,v_2) of capacity $u(v_1,v_2) = m$ and cost $c(v_1,v_2) = 0$ for each $v \in V$ and an arc (v_2,w_1) of capacity $u''(v_1,w_1) = u'(v,w)$ and cost $c'(v_1,w_1) = c(v,w)$ for each arc $(v,w) \in E$. (Network G' also contains opposite-directed arcs of capacity zero.) The restricted problem is (G',u'',c'). We call a vertex v_1 in the restricted problem *inner* and a *vertex* v_2 *outer, we call an arc* (v_1,v_2) a *split arc.*

Observe that every circulation f' in the restricted network corresponds to a circulation f in the target network given by $f(v,w) = f'(v_2,w_1)$. By construction, the costs off and f' are the same. Conversely, for every circulation fin the target network, there is a corresponding arc function f' given by $f'(v_2,w_1) = f(v,w)$, $f'(v_1,v_2) = \sum_{(x,v) \in E} f(x,v)$. Function f is a circulation if and only if $f'(v_1,v_2) \le m$ for every split arc (v_1,v_2).

We assume (without loss of generality) that the original problem has no negative capacities, i.e. the identically zero arc function is a circulation. At a high level, the algorithm consists of initializing fand u" to be zero on all arcs and repeating the following steps for each bit of precision in the capacities, proceeding left-to-right through the bits:

Step 1. (Construct the new target problem by introducing the next capacity bit.) For each $(v,w) \in E$, replace $u'(v,w)$ by $2u'(v,w)$ if the current bit of $u(v,w)$ is zero, by $2u'(v,w) + 1$ if the current bit of $u(v,w)$ is one.

Step 2. Construct the restricted problem (G', u'', c').

Step 3. Find an optimal solution f' to the problem (G', u'', c').

Step 4. Construct the circulation fin (G, u', c') corresponding to f'.

Step 5. (Modify the target problem so that the zero circulation is optimal.) For each $(v,w) \in E$, replace $u'(v, w)$ by $u'(v, w) - f(v, w)$.

The algorithm maintains the invariant that on entry to Step 1, the zero circulation is optimal for the old target problem (as modified in Step 5). The following result is similar to a lemma of Gabow and Tarjan [8].

Lemma 6.1. The circulation f computed in Step 4 is an optimal solution to the target problem (G, u', c').

Proof. By induction on the number of iterations. On the most recent entry to Step 1, the zero circulation is optimal for the old target problem. This is true for the initial entry because of the initialization, and true for each subsequent entry by the induction hypothesis. Thus there is some price function p for which the reduced costs of all positive-capacity arcs are non-negative in the old target problem. By the construction in Step 1, all negative-reduced-cost arcs in the new target problem have capacity one or zero. Let f^* be an optimal solution to the new target problem such that f^* contains no zero-cost cycle of positive flow (any such cycle can be eliminated by reducing its flow). Circulation f^* can be decomposed into at most m simple cycles, each of flow value one. (There is at most one such cycle for each negative reduced-cost arc). The arc function f' on the restricted network that corresponds to f^* thus has $f'(v_1, v_2) \leq m$ for each split arc (v_1, v_2), i.e. f' is a circulation. Since f^* is optimal for the target problem, f' is optimal for the restricted problem. It follows that any optimal solution to the restricted problem corresponds to an optimal solution to the target problem, and the lemma is true. Cl

We shall describe an implementation of Step 4 that uses the c-scaling approach of Goldberg and Tarjan [9,11] (already discussed in Section 3 in the context of the transportation problem) with an inner loop that is a modification of the Ahuja-Orlin maximum flow algorithm [1,2]. The

e-scaling loop starts with a zero circulation and a zero price function; the zero circulation is ε-optimal with respect to the zero price function for $\varepsilon = C$. Then the method iteratively applies a refinement subroutine that halves ε and produces a circulation f' and a price function p such that f' is E-optimal with respect to p. When $\varepsilon < \frac{1}{2n}$, the method terminates with an optimal solution. (Recall that the restricted network has *2n vertices.) The* formal definition of the e-scaling loop is as follows:

Step4.1. **Let** $f' = 0$, $\varepsilon = C$, and $p = 0$.

Step 4.2. For each outervertex v_2, **let** $p(v_2) = \max \{c'(v_2,w_1) \mid (v_2,w_1) \in E'\}$.

Step 4.3. While $\varepsilon \geq \frac{1}{2n}$, perform (e, f, p) $\leftarrow$ *refine* (ε, f, p).

The special structure of the restricted network allows the maintenance of the following invariant in Step 4.3: for all residual arcs (v,w) with negative reduced costs, v is an inner vertex. Step **4.2 guarantees** that this invariant holds on entry to Step 4.3.

The correctness of the &-scaling loop and the fact that it terminates after $O(\log(nC))$ iterations of *refine* follow from the **results** of Goldberg and **Tarjan [9,11].**

The following implementation of *refine* is based on the generic implementation of *refine* described by Goldberg and **Tarjan [9,11],** specialized to use excess scaling as in the **Ahuja-Orlin** maximum flow algorithm **[1,2].**

refine (ε, f', p).

Step ***R.1.*** (Saturate negative-cost arcs.) For each split arc (v_1, v_2) such that $c'_p(v_1,v_2) < 0$, *let* $f'(v_1,v_2) = u''(v_1,v_2)$.

Step R.2. (Initialize Δ, ε.) Let A be the smallest power of two not less than m. Replace ε by $\varepsilon/2$.

Step R.3. (Inner loop.) While $A \geq 1$ **repeat** the following steps:

Step ***R.3.1.*** While there is a *push* or ***relabel*** operation that applies, perform such an operation.

Step R.3.2. Replace A by $\Delta/2$.

Step R.4. For each inner vertex v_1, replace $p(v_1)$ by $p(v_1)$ -e/2.

Step R.5. Return (ε, f', p).

The push and ***relabel*** operations are defined as follows:

push(v,w).

Applicability: $e_{f'}(v) > \Delta/2$, $e_{f'}(w) \leq A/2$, $u''_{f'}(v,w) > 0$, and $c'_p(v,w) < -\varepsilon/4$.

Action: Send min $\{\Delta/2, u''_{f'}(v,w)\}$ units of flow from v to w.

relabel(v).

Applicability: $e_{f'}(v) > 0$ and $c'_p(v,w) \geq -\varepsilon/4$ for each residual arc (v, w).

Action: Replace $p(v)$ by max $\{p(w) - c'(v,w) - \varepsilon/2\}$.

Some remarks are in order here. Step R.l saturates all negative reduced-cost arcs, thereby making f' into an optimal pseudoflow but introducing excesses and deficits at vertices. Step R.3 moves the excess flow amounts to the vertices with deficits while maintaining $\varepsilon/2$-optimality (for the new value of ε). Step R.4 is assures that for any residual arc (v,w) which has a negative reduced cost, v is an inner vertex. After this step, f' is no longer $\varepsilon/2$-optimal but only ~-optimal.

Step 4.3 maintains a value A that is an upper bound on the largest excess. When no excess exceeds A/2, A is halved. All excesses are integers; by the time $A < 1$, all deficits have been canceled.

Each pushing step moves excess from a vertex with excess exceeding A/2 to a vertex with excess not exceeding A/2, and through an edge of cost between $-\varepsilon/2$ and $-\varepsilon/4$. Thus each push is either saturating or it moves at least A/2 units of flow; in the latter case it reduces the cost off by at least $\Delta\varepsilon/8$. We shall use this cost reduction to bound the number of nonsaturating pushes.

We have omitted a description of how to determine ***push*** and ***relabel*** *operations* that can be applied. These details can be found in [1,2]. The total time spent in such overhead in a single execution of *refine* is $O(nm)$.

The proofs of the following lemmas are easv modifications of proofs of analogous lemmas in [9,10]:

Lemma 6.2. The push and relabel operations preserve ε-optimality.

Lemma *6.3.* (i) Each operation ***relabel*** *(v)* decreases *p(v)* by at least e/4. (ii) During an execution of *r&e, the* maximum amount by which *p(v)* can decrease is $6\varepsilon n$, for every vertex v. (iii) The total number of relabel operations during an execution of ***refine*** is $O(n^2)$, *taking* $O(nm)$ time. (iv) The number of saturating pushes during an execution of *refine* is $O(nm)$.

Lemma 6.3 implies that the running time of *refine* is $O(nm)$ plus O(1) per nonsaturating push. We shall establish a bound of $O(n^2 \log m + nm)$ on the number of nonsaturating pushes, thereby obtaining an $O((n^2 \log m + nm) \log U \log(nC))$ bound on the triple scaling algorithm.

To bound the number of nonsaturating pushes, we define the *cost* of a pseudoflow f' with respecttoapricefunctionptobe

$$cost_p(f') = \sum_{f'(v,w) > 0} c'_p(v,w) f'(v,w) = \sum_{f(v,w) > 0} c'(v,w) f'(v,w) + \sum_{v \in V} e_{f'}(v) p(v).$$

Observe that if f' is a circulation its cost does not depend on p, i.e. $cost_p(f') = cost(f')$ where $cost(f) = \sum_{f(v,w)} c'(v,w) f(v,w)$ *(as* defined in Section 1).

As noted above, a nonsaturating push decreases $cost_p(f')$ by at least $\Delta\varepsilon/8$. A relabeling of a vertex v that decreases $p(v)$ by an amount x increases $cost_p(f')$ by $x e_{f'}(v)$.

We want to study how much $cost_p(f')$ can vary during an execution of ***refine***. We do this by relating $cost_p(f')$ to the cost of an optimal circulation. This requires the following lemma, which states a general result about circulations and pseudoflows.

Lemma 6.4. Let fbe a circulation, f' a pseudoflow, and p a price function on a network f with capacity and cost functions u and c, respectively. Then

$$cost_p(f') - cost(f) \le \sum_{(v,w):\, c_p(v,w) < 0} -c_p(v,w) u_{f'}(v,w).$$

Proof. Let f'' be the pseudoflow obtained from f by saturating all negative reduced-cost arcs. Then

$$cost,(f) = cost_p(f'') + \sum_{(v,w):\, c_p(v,w) < 0} -c_p(v,w)\, u_{f'}(v,w).$$

But f can be obtained from f" by increasing flow along a collection of paths and cycles of arcs in $G_{f''}$, each of which has nonnegative reduced cost. Thus $cost(f) \geq cost_p(f'')$, and the lemma follows. cl

Consider a time during the execution of *refine*. For the current pseudoflow f' and price function p, *we* define a ***potential*** Φ by $\Phi = (cost_p(f') - cost(f^*))/(\Delta\varepsilon)$, where f^* is any optimal circulation. The following lemma bounding Φ is the heart of the analysis of nonsaturating pushes.

Lemma 6.5. $-48n^2 \leq \Phi \leq 4nm/\Delta$.

Proof. We shall prove that $-48\Delta\varepsilon n^2 \leq cost_p(f') - cost(f^*) \leq 4\varepsilon nm$. To obtain the upper bound, we note first that f' is $\varepsilon/2$-optimal with respect to p. By decreasing prices on inner vertices by $\varepsilon/2$, we obtain a price function p' with respect to which f' is e-optimal and for any residual arc (v,w) that has a negative reduced cost with respect to p', v is an inner vertex. Since no vertex has excess exceeding $A \leq 2m$, $cost_{p'}(f') \leq cost_p(f') + \varepsilon nm$. By Lemma 6.4,

$$cost_{p'}(f') - cost(f^*) \leq \sum_{(v,w):\, c'_{p'}(v,w) < 0} (v,w) \leq 3enm.$$

To justify the last inequality, we show that the sum of residual capacities of all arcs residual arcs with negative reduced costs is bounded by $3nm$. To see this, recall that if (v,w) is such an arc, that v is an inner vertex. For an inner vertex v, the total residual capacity of arcs going out of v is $m - e_{f'}(v)$. Since the total excess is bounded by $n\Delta \leq 2nm$, the total residual capacity of arcs going out of inner vertices is bounded by 3nm; therefore the total residual capacity of arcs with negative reduced costs is at most $3nm$.

Combining inequalities gives $cost_p(f') - cost(f^*) \leq 4\varepsilon nm$.

To obtain the upper bound, we note first that we can assume that there is a price function p^* with respect to which f^* is optimal and such that $I\, p(v) - p^*(v)\, I \leq 12\varepsilon n$ for every vertex v. This is since the repeated executions of ***refine*** in Step 4.3 of the E-scaling method will produce an optimal circulation f^* and a final price function p^* such that f^* is optimal with respect to p^*; the

total price change of any vertex over all the iterations of *refine* is at most $\sum_{i=0}^{\infty} 6n\varepsilon/2^i = 12\varepsilon n$. We have $cost(f^*) - cost_{p'}(f') \leq 0$, since f^* can be converted into f' by increasing flow on residual arcs of $G_{f'}$, all of which have nonnegative reduced cost. But

$$cost_{p'}(f') - cost_p(f') \leq \sum_{V\&V} |p^*(v) - p(v)|\,|e_{f'}(v)| \leq 48\Delta\varepsilon n^2$$

since the sum of the positive excesses is at most $2n\Delta$, as is the negative of the sum of the negative excesses. Combining inequalines gives $cost(f) - cost_p(f') \leq 48\Delta\varepsilon n^2)$. □

Lemma 6.7. The number of nonsaturating pushes during an execution of ***refine*** is $O(n^2 \log m + nm)$.

Proof. Each nonsaturating push decreases Φ by at least $1/8$. Any saturating push also decreases Φ. A relabeling that decreases the price of a vertex by x increases Φ by at most x.

There are at most log m + 1 iterations of Step R.3 in refine. Consider the i^{th} iteration. Suppose that during this iteration the total decrease in vertex prices is x_i. If p_i is the number of nonsaturating pushes during this iteration, then the iteration causes a net decrease in Φ of at least $p_i/8 - x_i$. Changing A between two iterations can increase Φ by at most n^2. Summing over all phases and applying Lemma 6.5, we have

$$\sum(p_i/8 - x_i) \leq 2nm + 48n^2 + n^2(\log m + 1).$$

Since $\sum x_i = O(n^2)$, it follows that $\sum_i p_i = O(n^2 \log m + nm)$. □

Theorem 6.8. The triple scaling algorithm for finding a minimum-cost circulation runs in $O((n^2 \log m + nm) \log U \log(nC))$ time.

Using dynamic trees in the inner loop of the triple scaling algorithm, as in [2], reduces the time bound to O(nm log $(1 + \frac{n}{m}$ logm) $\log U$ log (nC)). Further minor improvements might be possible using additional ideas in [2]. We shall not pursue this possibility further, however, since in any case the approach of Sections 3-5 produces better bounds.

We conclude this section by noting that there is an alternative way to solve the sequence of restricted problems generated in the outer capacity-scaling loop. Namely, we can use a simple version of the network simplex rule, specifically Dantzig's minimum reduced-cost pivot rule with lexicography to avoid cycling. This rule was studied by Orlin [15], who obtained a bound of $O(nmU \log (mUC))$ on the number of pivot steps. In a restricted problem, $U = m$, and the capacity-scaling method with Step 3 implemented using the network simplex method runs in $O(nm^3 \log U \log (mC))$ time, if the time to do one pivot step is $O(m)$. *In this algorithm,* it is not necessary to transform the graph by splitting vertices; it suffices to impose a capacity bound of m on every arc in G. This time bound is not noteworthy; more interesting is the mere fact that combining one scaling loop with a standard version of the network simplex algorithm gives a polynomial-time algorithm. Without the scaling loop, the same version of the network simplex algorithm can take exponentially many pivot steps [24].

7. Remarks

We have shown that the minimum-cost circulation problem can be solved in $O(nm \log \log U \log (nC))$ *time,* and even in $O(nm \log (2 + \frac{n}{m} \log U) \log (nC))$ time if $\log U = \Omega(\frac{m}{n} \log n)$. We have derived analogous bounds for the transportation problem. Our algorithms use scaling of both costs and capacities, combined with an augmenting path method and an implementation based on dynamic trees. If dynamic trees are not used, the time bound for the minimum-cost circulation problem is $O(nm \log U (1 + \log (nC)/ \log \log U))$, or $O(n^2 \log U (1 + \log (nC)/ \log \log U))$ if $\log U = \Omega(\frac{m}{n} \log n)$. Under the similarity assumption [7], namely $\log U = O(\log n)$ and $\log C = O(\log n)$, *our time* bound with dynamic trees is $O(nm \log n \log \log n)$, which beats the best previous bound of $O(nm \log n \log(\frac{n^2}{m}))$ [11] except for very dense graphs $(m = \Omega(\frac{n^2}{\log n}))$.

We expect that some version of our algorithm, probably without dynamic trees, will in practice be competitive with or superior to previously existing algorithms. We have not yet done the required experiments to confirm or refute this hypothesis. Our experiments with similar algorithms (see e.g. [9]) suggest that periodic scans to tighten prices may increase the practical, though not the theoretical, speed of the algorithm.

We have discussed a capacity-bounding technique, which allows us to use a modification of the Ahuja-Orlin maximum flow algorithm in the inner loop of the Goldberg-Tarjan minimum-cost flow method. The analysis of this technique uses the cost of the current pseudoflow as a measure of its quality. This makes the analysis very intuitive.

A tantalizing open question is whether there is an 0 (nm log logn log (*nC*))-time *algorithm* for the minimum-cost circulation problem. We believe that the answer is yes and that a modification of our methods will lead to such a bound. Such a result would probably give a time bound of 0 (nm log *logn)* for the maximum flow problem, which would also be an improvement over known results. (See [2,10].)

8. References

[1] R. K. Ahuja and J. B. Orlin, "A fast and simple algorithm for the maximum flow problem," Technical Report 1905-87, Sloan School of Management, M.I.T., Cambridge, MA, 1987.

[2] R.K. Ahuja, J.B. Orlin, and R.E. Tarjan, "Improved time bounds for the maximum flow problem," to appear.

[3] D.P. Bertsekas, "A distributed algorithm for the assignment problem," unpublished working paper, Laboratory for Information and Decision Sciences, M.I.T., 1979.

[4] D.P. Bertsekas, "Distributed asynchronous relaxation methods for linear network flow problems," Technical Report LIDS-P-1606, Laboratory for Information and Decision Sciences, M.I.T., 1986.

[5] E.A. Dinic, 'Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.* 11 (1970), 1277-1280.

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

[7] H.N. Gabow, "Scaling algorithms for network problems," *J. Comput. System Sci.* 31 (1985), 148-168.

[8] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for network problems," *SIAM J. Comput.*, submitted.

[9] A.V. Goldberg, "Efficient graph algorithms for sequential and parallel computers," Ph.D. Thesis, M.I.T., 1987.

[10] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," *J. Assoc. Comput. Mach.*, to appear.

[11] A.V. Goldberg and R.E. Tarjan, "Fiiding minimum-cost circulations by successive approximation," *Math. of Oper. Res.*, to appear.

[12] A.V. Goldberg and R.E. Tarjan, "Finding minimum-cost circulations by canceling negative cycles," *J. Assoc. Comput. Mach.*, submitted; *also Proc. Twentieth Annual ACM Sww. on Theory of Computing* (1988), *388-397.*

[13] EL. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Reinhart, and Winston, New York, NY, 1976.

[14] R. Mehlhorn, *Data Structures and Algorithms, Volume I: Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[15] J. B. Orlin, "On the simplex algorithm for networks and generalized networks," *Math. Programming 24* (1985), *166-178.*

[16] J. Orlin, "A faster strongly polynomial minimum cost *flow algorithm," Proc. Twentieth Annual ACM Symp. on Theory of Computing* (1988), 377-387.

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

[18] D.D. Sleator and R.E. Tarjan, "A data *structure* for dynamic trees," *J. Comput. System Sci. 26* (1983), 362-391.

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

[20] E. Tardos, "A strongly polynomial minimum cost circulation algorithm," *Combinatorica, 5* (1985), 247-255.

[21] R.E. Tarjan, *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA 1983.

[22] R.E. Tarjan and C.J. van Wyk, "An 0 *(n* log logn)-time algorithm for triangulating a simple polygon, " *SIAM J. Comput., 17* (1988), *143-178.*

[23] H.M. Wagner, "On a class of capacitated transportation problems," *Management Science* 5 (1959). 304-318.

[24] N. Zadeh, "A bad network flow problem for the simplex method and other minimum cost flow algorithms,'* *Math. Programming 5* (1973), 255-266.