FINDING MINIMUM-COST CIRCULATIONS BY SUCCESSIVE APPROXIMATION

Andrew V. Goldberg
Robert E. Tarjan

CS-TR-106-87

July 1987

Finding Minimum-Cost Circulations by Successive Approximation*

Andrew V. Goldberg[†‡]
Laboratory for Computer Science
M.I.T.
Cambridge MA 02139

Robert E. Tarjan[§]
Department of Computer Science
Princeton University
Princeton, New Jersey 08544
and
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

July 1987

*A preliminary version of this paper appeared in the Proceedings of the Nineteenth Annual A.C.M. Symposium on Theory of Computing, May 25-27, 1987.

[†]Current address: Department of Computer Science, Stanford University, Stanford, CA 94305.

[‡]Supported by a Fannie and John Hertz Foundation Fellowship and by the Advanced Research Projects Agency of the Department of Defense, Contract No. N00014-80-C-0622.

[§]Partially supported by the National Science Foundation, Grant No. DCR-8605962, and the Office of Naval Research, Contract No. N00019-87-K-0467.

Abstract

We develop a new approach to solving minimum-cost circulation problems. Our approach combines methods for solving the maximum flow problem with successive approximation techniques based on cost scaling. We measure the accuracy of a solution by the amount that the complementary slackness conditions are violated.

We propose a simple minimum-cost circulation algorithm, one version of which runs in $O(n^3 \log(nC))$ time on an n-vertex network with integer arc costs of absolute value at most C. By incorporating sophisticated data structures into the algorithm, we obtain a time bound of $O(nm \log(n^2/m) \log(nC))$ on a network with m arcs. A slightly different use of our approach shows that a minimum-cost circulation can be computed by solving a sequence of $O(n \log(nC))$ blocking flow problems. A corollary of this result is an $O(n^2(\log n) \log(nC))$-time, n-processor parallel minimum-cost circulation algorithm. Our approach also yields strongly polynomial minimum-cost circulation algorithms.

Our results provide evidence that the minimum-cost circulation problem is not much harder than the maximum flow problem. We believe that a suitable implementation of our method will perform extremely well in practice.

1 Introduction

The *minimum-cost circulation problem* is that of finding a feasible circulation of minimum cost in a network whose arcs have flow capacities and costs per unit of flow. Extensive discussions of this problem and its applications can be found in the books of Ford and Fulkerson [13], Jensen and Barnes [32], Lawler [36], Papadimitriou and Steiglitz [44], and Tarjan [52].

Classical algorithms for the problem, such as the out-of-kilter method [17,41] and the cheapest path augmentation method [8,33], run in exponential time in the worst case. All known polynomial-time algorithms for the problem rely on the concept of *scaling*. Scaling was introduced by Edmonds and Karp [12], who devised the first polynomial-time algorithm for the problem. Scaling has also been applied to various other network optimization problems [18,19]. Scaling algorithms work by solving a sequence of subproblems whose numeric parameters more-and-more closely approximate those of the original problem. A solution to one subproblem helps to solve the next subproblem in the sequence. In the case of the minimum-cost circulation problem, successive subproblems are obtained by adding one bit of precision at a time either to the capacities (*capacity scaling*) or to the costs (*cost scaling*). The algorithm of Edmonds and Karp scales capacities. Röck [45] was the first to propose an algorithm that scales costs.

Table 1 summarizes known polynomial-time algorithms for the minimum-cost circulation problem, including those proposed in this paper. Time bounds are given in terms of the following parameters and functions:

#	Date	Discoverer	Running Time	References
1	1972	Edmonds and Karp	$O(m \log(U) S(n,m))$	[12]
2	1980	Röck	$O(m \log(U) S(n,m))$	[45]
3	1980	Röck	$O(n \log(C) F(n,m,U))$	[45]
4	1984	Tardos	$O(m^4)$	[50]
5	1984	Orlin	$O(m^2 \log(n) S(n,m))$	[43]
6	1985	Fujishige	$O(m^2 \log(n) S(n,m))$	[16]
7	1985	Bland and Jensen	$O(n \log(C) F(n,m,U))$	[7]
8	1986	Galil and Tardos	$O(n^2 \log(n) S(n,m))$	[23]
9	1987	Goldberg and Tarjan	$O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$	
10	1987	Goldberg and Tarjan	$O(nB(n,m) \min\{\log(nC), m \log n\})$	
11	1987	Bertsekas and Eckstein	$O(n^3 \log(nC))$	[5]

Table 1: Polynomial-time algorithms for the minimum-cost circulation problem. Algorithms 9 and 10 are presented in this paper.

n, the number of vertices;

m, the number of arcs;

U, the maximum absolute value of an arc capacity;

C, the maximum absolute value of an arc cost;

$S(n,m)$, the time to solve a single-source shortest path problem with nonnegative arc costs, as a function of n and m;

$F(n,m,U)$, the time to find a maximum flow, as a function of n, m, and U;

$B(n,m)$, the time to find a blocking flow on an acyclic network, as a function of n and m.

In bounds containing U or C, the capacities or costs, respectively, are assumed to be integers. The best known bounds for S, F, and B are as follows:

$$\begin{array}{lll} S(n,m) & = O(n \log n + m) & [15] \\ F(n,m,U) & = O(nm \log(n^2/m)) & [29] \\ & = O(n^2 \log U + nm) & [1] \\ B(n,m) & = O(m \log(n^2/m)) & \text{(this paper, Section 8.1).} \end{array}$$

The algorithms in the table split into two classes, those that use capacity scaling (algorithms 1, 2, 5, 6, and 8) and those that use cost scaling (algorithms 3, 4, 7, 9, 10, and 11). All the capacity scaling algorithms require a shortest path subroutine; all the cost scaling

algorithms require a maximum flow or related subroutine. Algorithms 4, 5, 6, 8, 9, and 10 are strongly polynomial: their running times are polynomial in n and m assuming that arithmetic operations take $O(1)$ time, and the numbers they manipulate have a number of bits polynomial in $n, m, \log U$, and $\log C$. The book of Papadimitriou and Steiglitz [44] and the paper of Tardos [50] contain discussions of the notion of a strongly polynomial algorithm.

It is difficult to compare the relative speeds of the various algorithms, because their time bounds depend in different ways upon n, m, U, and C. Among the algorithms not based on our approach (1-8), the dominant algorithms are 1 and 2, 3 and 7, and 8. Under the assumption (see [19]) that $\log U$ and $\log C$ are $\Theta(\log n)$, algorithms 1 and 2 have a bound of $O(m^2 \log n + nm(\log n)^2)$, whereas algorithms 3, 7, and 8 have a bound of $O(n^2 m \log n + n^3(\log n)^2)$, larger by a factor of n^2/m.

In this paper we discuss a general approach to the minimum-cost circulation problem. The approach uses successive approximation based on cost scaling. It combines the ideas Röck [45] used to develop algorithm 3, those Tardos [50] used to develop algorithm 4, those Bland and Jensen [7] used to develop algorithm 7, and those Bertsekas [4] used to develop a pseudopolynomial-time (i.e. polynomial in n, m, and C) algorithm for the problem. The method works by finding an approximate solution and then iteratively refining the current solution, at each iteration halving the error parameter ϵ, until ϵ is small enough so that integrality of arc costs guarantees that the current solution is optimal. To measure the accuracy of a solution, we use the concept of ϵ-optimality, which is based on a relaxation of complementary slackness conditions. This concept is related to the classical technique of perturbing a linear programming problem to avoid degeneracy (see for example the book of Gass [25]). Tardos [50] and Bertsekas [4] both used ϵ-optimality (in different ways) in their algorithms. The algorithm of Tardos fixes the flow on edges with sufficiently high reduced cost. The algorithm of Bertsekas finds a minimum-cost circulation by locally modifying a pseudoflow while preserving ϵ-optimality. Using a lemma of Bertsekas and a generalization of a lemma of Tardos, we show that $O(\min\{log(nC), m \log n\})$ iterations of a refinement subroutine suffice to compute a minimum-cost circulation.

We discuss two general ways to implement the refinement subroutine that is the heart of the algorithm. Both are generalizations of maximum flow algorithms. The first method, which we call the *generic method*, is a slight variation of the algorithm of Bertsekas [4] that generalizes our maximum flow algorithm [29]. The generic method uses a sequence

of transformations to produce an ϵ-optimal circulation. Bertsekas made the important observation that these transformations preserve ϵ-optimality. The analysis of the generic method uses ideas similar to those in the analysis of the maximum flow algorithm [29]. We describe a simple version of the generic method with a running time of $O(n^3)$, giving an $O(n^3 \min\{\log(nC), m \log n\})$ bound for the minimum-cost circulation problem. We call this algorithm the *wave method*, because it resembles the wave algorithm for computing a maximum flow [53]. The wave method was proposed by Charles Leiserson (private communication) and independently (minus certain implementation details and the use of scaling) by Bertsekas [4]. A more complicated algorithm using two sophisticated data structures, dynamic trees [48,49] and finger search trees [39,54], yields an $O(nm \log(n^2/m))$-time bound for refinement and an $O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$-time bound for finding a minimum-cost circulation.

Our second implementation of refinement, the *blocking flow method*, uses $O(n)$ calls of a blocking flow subroutine. It is a generalization of the maximum flow approach of Dinic [11]. The blocking flow approach also yields a simple $O(n^3)$-time refinement algorithm, based on known $O(n^2)$-time blocking flow algorithms [35,38,46,53], and a more complicated $O(nm \log(n^2/m))$-time refinement algorithm, based on a new blocking flow algorithm (presented in Section 8). The blocking flow method also yields an $O(n^2 \log n)$-time, n-processor parallel refinement algorithm, based on the parallel blocking flow algorithm of Shiloach and Vishkin [46]. This gives an $O(n^2 \log n \min\{\log(nC), m \log n\})$-time, n-processor algorithm for the minimum-cost circulation problem.

If $C = O(n^{n^{1-\delta}})$ and $C = O(U^{a(m/(n \log n)+1)}/n)$ for some positive constants δ and a, then our $O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$ sequential time bound for finding a minimum-cost circulation is as small as the bound for any other method. Under the assumption that $\log U$ and $\log C$ are $\Theta(\log n)$, our bound significantly improves over previous bounds.

This paper is a revised and extended version of one chapter of the first author's thesis [26] and of a conference paper [30]. Working from an extended abstract of the conference paper, Bertsekas and Eckstein [5] developed an $O(n^3 \log(nC))$ minimum-cost circulation algorithm using a more traditional cost-scaling approach in combination with the wave method. The bounds we derive in this paper can also be derived using traditional cost scaling; in fact, the first versions of our algorithm used this approach. The successive approximation approach, however, seems preferable for at least two reasons. First, it allows

the use of true costs throughout the algorithm; no rounding is needed. Second, it leads very naturally to strongly polynomial algorithms.

2 Foundations

In this section we define the *minimum-cost circulation problem* and introduce terminology to be used throughout the paper. We also review conditions for a circulation to be optimal, i.e. minimum-cost, and introduce the notion of ϵ-optimality, which plays a central role in our approach.

The minimum-cost circulation problem is a generalization of the maximum flow problem obtained by adding a cost per unit of flow to each arc. Since this problem is a special case of the linear programming problem, it is usually defined in linear programming terms. Although we use concepts and results that have their roots in linear programming theory, most of the arguments presented in this paper are graph-theoretic. Consequently, we formulate the problem in graph-theoretic terms. Our formulation is equivalent to other formulations of the minimum-cost flow and minimum-cost circulation problems that can be found in standard works on linear programming and network optimization, including those cited in Section 1.

2.1 Definitions

A *circulation network* is a directed graph $G = (V, E)$ with a capacity $u(v, w)$ and a cost $c(v, w)$ associated with each arc[1] (v, w). We require G to be *symmetric*, i.e. $(v, w) \in E$ implies $(w, v) \in E$. We denote the size of V by n and the size of E by m, and we assume (for ease in stating time bounds) that $m \geq n - 1 \geq 1$. We call an unordered pair $\{v, w\}$ such that $(v, w) \in E$ an *edge* of G. We assume that the cost function satisfies the following constraint for each $(v, w) \in E$:

$$c(v, w) = -c(w, v) \quad \text{(cost antisymmetry constraint)}. \tag{1}$$

A *pseudoflow* is a function $f : E \rightarrow R$ satisfying the following constraints for each $(v, w) \in E$:

$$f(v, w) \leq u(v, w) \quad \text{(capacity constraint)}, \tag{2}$$

[1] We refer to ordered pairs of vertices as *arcs*, and to unordered pairs of vertices as *edges*.

$$f(v,w) = -f(w,v) \quad \text{(flow antisymmetry constraint)}. \tag{3}$$

Remark: The flow antisymmetry constraints imply $-u(w,v) \leq f(v,w) \leq u(v,w)$; this means that there are implied lower bounds as well as upper bounds on flow. In the usual formulation of the minimum-cost circulation problem these lower bounds are explicitly given. We have chosen the present formulation because it simplifies the statement of constraints such as (2) and simplifies some proofs. Our results easily extend to allow multiple arcs, infinite capacities, and costs that are not antisymmetric.

For a pseudoflow f and a vertex v, the *excess flow into* v, $e_f(v)$, is defined by $e_f(v) = \sum_{(u,v)\in E} f(u,v)$. A vertex v with $e_f(v) > 0$ is called *active*. Observe that $\sum_{v\in V} e_f(v) = 0$.

A *circulation* is a pseudoflow f such that, for each vertex v,

$$e_f(v) = 0 \quad \text{(flow conservation constraint)}. \tag{4}$$

Observe that a pseudoflow f is a circulation if and only if there are no active vertices. The *cost* of a pseudoflow f is given by the following expression:

$$cost(f) = \frac{1}{2} \sum_{(v,w)\in E} c(v,w) f(v,w). \tag{5}$$

(The factor of $1/2$ appears because the sum counts the cost of the flow through each edge twice.) The *minimum-cost circulation problem* is that of finding a circulation of minimum cost.

We shall need several other concepts. Let G be a directed graph with arc cost function c. The *length* of a path or cycle in G is the number of arcs it contains. The *cost* $c(\Gamma)$ of a path or cycle Γ is the sum of the costs of the arcs on the path or cycle. The *mean cost* of a cycle is the cost of the cycle divided by the length of the cycle.

For a pseudoflow f and an arc (v,w), the *residual capacity* of (v,w) is $u_f(v,w) = u(v,w) - f(v,w)$. An arc (v,w) is *saturated* if $u_f(v,w) = 0$. An arc (v,w) is a *residual arc* if it is not saturated, i.e. $u_f(v,w) > 0$. The *residual graph* $G_f = (V, E_f)$ is the directed graph with vertex set V and arc set $E_f = \{(v,w) | u_f(v,w) > 0\}$.

A *price function* p is a function $p : V \to R$. For a price function p, the *reduced cost* of an arc (v,w) is $c_p(v,w) = c(v,w) - p(v) + p(w)$. The flow conservation constraint (4) implies that the cost of a minimum-cost circulation is the same whether the original costs or

the reduced costs with respect to some price function are used. In the linear programming formulation of the minimum-cost circulation problem, the prices are the dual variables.

2.2 Optimality and Approximate Optimality

There are two classical characterizations of minimum-cost circulations, both of which we shall use.

Theorem 2.1 ([9]) *A circulation f is minimum-cost if and only if the residual graph contains no cycle of negative cost.*

Theorem 2.2 ([13]) *A circulation f is minimum-cost if and only if there is a price function p such that, for each arc (v, w),*

$$c_p(v, w) < 0 \Rightarrow f(v, w) = u(v, w) \quad (\textit{complementary slackness constraint}). \tag{6}$$

For a pseudoflow f, an arc (v, w) is said to be *in kilter* (after the out-of-kilter method [17,41]) if it satisfies (6) and *out of kilter* otherwise. Figure 1a shows a *kilter diagram* [36], which is a pictorial representation of the complementary slackness constraints.

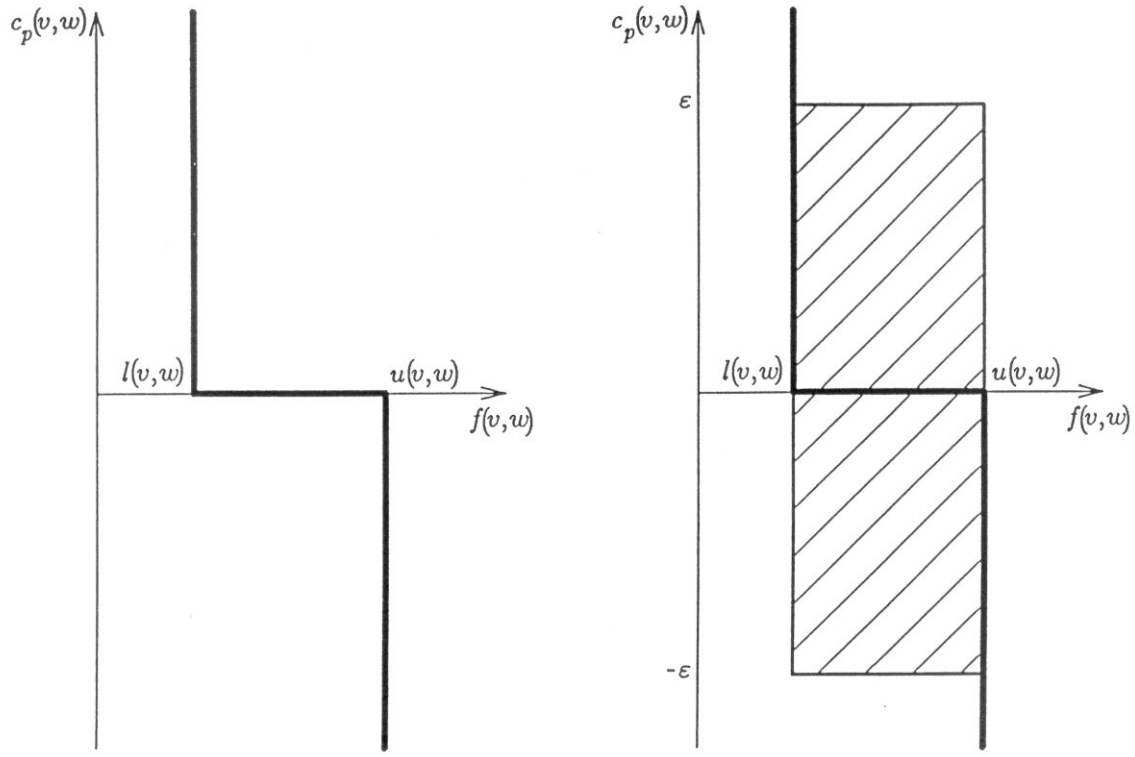
We need a notion of approximate optimality, which we obtain by relaxing the complementary slackness constraints. This relaxation was previously used in the minimum-cost flow algorithms of Tardos [50] and Bertsekas [4,6]. For a constant $\epsilon \geq 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function p* if, for every arc (v, w), we have

$$c_p(v, w) < -\epsilon \Rightarrow f(v, w) = u(v, w) \quad (\epsilon\text{-optimality constraint}). \tag{7}$$

A pseudoflow f is *ϵ-optimal* if f is ϵ-optimal with respect to some price function p. Figure 1b illustrates the concept of ϵ-optimality in terms of kilter diagrams. A circulation f is *ϵ-tight* if f is ϵ-optimal and for any $\epsilon' < \epsilon$, f is not ϵ'-optimal. A circulation f is *ϵ-tight* with respect to a price function p if f is ϵ-tight and f is ϵ-optimal with respect to p.

Observe that a pseudoflow f is ϵ-optimal with respect to a price function p if and only if $c_p(v, w) \geq -\epsilon$ for every residual arc (v, w). It is in this form that we shall use the definition.

A crucial property of ϵ-optimality is that if the arc costs are integers and ϵ is small enough, any ϵ-optimal circulation is minimum-cost. The following theorem is a result of Bertsekas.



(a) Kilter diagram

(b) Approximate optimality

Figure 1:
(a) Kilter diagram. If f is an optimal circulation, the point corresponding to the arc (v, w) is on the heavy curve.
(b) ϵ-optimality. If f is ϵ-optimal, the point corresponding to the arc (v, w) can be in the shaded rectangle as well as on the heavy curve.

Theorem 2.3 ([4]) *If all costs are integers and $\epsilon < 1/n$, then an ϵ-optimal circulation f is minimum-cost.*

Proof: Consider a simple cycle in G_f. The ϵ-optimality of f implies that the reduced cost of the cycle is at least $n\epsilon > -1$. The reduced cost of the cycle equals its original cost, which must be integral and hence nonnegative. Theorem 2.1 implies that f is minimum-cost. ∎

3 Fitting Price Functions and Tight Error Parameters

The definition of ϵ-optimality motivates the following two problems:

1. Given a pseudoflow f and a constant $\epsilon \geq 0$, find a price function p such that f is ϵ-optimal with respect to p, or show that there is no such price function (i.e. f is not ϵ-optimal).

2. Given a pseudoflow f, find the ϵ such that f is ϵ-tight (i.e. the smallest $\epsilon \geq 0$ such that f is ϵ-optimal).

Solutions to these problems are needed in our strongly polynomial minimum-cost circulation method and can be used to give heuristic improvements in our weakly polynomial method. The problem of finding an optimal price function given an optimal circulation is the special case of problem 1 with $\epsilon = 0$. In this section we shall show that both problems can be solved in $O(nm)$ time, the first by doing a Bellman-Ford shortest path computation (see e.g. [52]), the second by applying an algorithm of Karp [34] for computing the minimum mean cost of a cycle in a directed graph with arc costs. We also show that an important special case of problem 1 arising in our weakly polynomial method can be solved in $O(m)$ time, by using the Dial-Wagner implementation [10,55] of Dijkstra's shortest path algorithm (see e.g. [52]).

To address these problems, we need some results about shortest paths and shortest path trees (see e.g. [52]). Let G be a directed graph with a distinguished *source vertex* s and a cost $c(v, w)$ on every arc (v, w). For a spanning tree T rooted at s, the *tree cost function* $d : V \rightarrow R$ is defined recursively as follows: $d(s) = 0$, $d(v) = d(parent(v)) + c(parent(v), v)$ for $v \in V - \{s\}$, where $parent(v)$ is the parent of v in T. A spanning tree T rooted at s is a *shortest path tree* if and only if, for every vertex v, the path from s to v in T is a minimum-cost path from s to v in G, i.e. $d(v)$ is the cost of a minimum-cost path from s to v.

Lemma 3.1 (see e.g. [52]) *Graph G contains a shortest path tree if and only if G contains no negative-cost cycle.*

Lemma 3.2 (see e.g. [52]) *A spanning tree T rooted at s is a shortest path tree if and only if $c(v, w) + d(v) \geq d(w)$ for every arc (v, w) in G.*

3.1 Finding a Fitting Price Function

Consider problem 1: given a pseudoflow f and a nonnegative ϵ, find a price function p with respect to which f is ϵ-optimal, or show that f is not ϵ-optimal. Define a new cost

function $c^{(\epsilon)} : E \to R$ by $c^{(\epsilon)}(v,w) = c(v,w) + \epsilon$. Extend the residual graph G_f by adding a single vertex s and arcs from it to all other vertices to form an *auxiliary graph* $G_{aux} = (V_{aux}, E_{aux}) = (V \cup \{s\}, E_f \cup (\{s\} \times V))$. Extend $c^{(\epsilon)}$ to G_{aux} by defining $c^{(\epsilon)}(s,v) = 0$ for every arc (s,v), where $v \in V$. Note that every vertex is reachable from s in G_{aux}.

Theorem 3.3 *Pseudoflow f is ϵ-optimal if and only if G_{aux} (or equivalently G_f) contains no cycle of negative $c^{(\epsilon)}$-cost. If T is any shortest path tree of G_{aux} (rooted at s) with respect to the arc cost function $c^{(\epsilon)}$, and d is the associated tree cost function, then f is ϵ-optimal with respect to the price function p defined by $p(v) = -d(v)$ for all $v \in V$.*

Proof: Suppose f is ϵ-optimal. Any cycle in G_{aux} is a cycle in G_f, since vertex s has no incoming arcs. Let Γ be a cycle of length l in G_{aux}. Then $c(\Gamma) \geq -l\epsilon$, which implies $c^{(\epsilon)}(\Gamma) = c(\Gamma) + l\epsilon \geq 0$. Therefore G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost.

Suppose G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost. Then by Lemma 3.1 G_{aux} has some shortest path tree rooted at s. Let T be any such tree and let d be the tree cost function. By Lemma 3.2, $c^{(\epsilon)}(v,w) + d(v) \geq d(w)$ for all $(v,w) \in E_f$, which is equivalent to $c^{(\epsilon)}(v,w) + d(v) - d(w) \geq -\epsilon$ for all $(v,w) \in E_f$. But these are the ϵ-optimality constraints for the price function $p = -d$. Thus f is ϵ-optimal with respect to p. ∎

Remark: Theorem 3.3 holds for *any* assignment of costs to the arcs (s,v). We have assigned zero costs to these edges merely for simplicity.

Using Theorem 3.3, we can solve problem 1 by constructing the auxiliary graph G_{aux} and finding either a shortest path tree or a negative-cost cycle. Constructing G_{aux} takes $O(m)$ time. Finding a shortest path tree or a negative-cost cycle takes $O(nm)$ time using the Bellman-Ford shortest path algorithm (see e.g. [52]).

An important special case of problem 1 arises in connection with Theorem 2.3. Suppose the arc cost function c is integral, f is a pseudoflow ϵ-optimal with respect to a price function p_ϵ, and $\epsilon < 1/n$. By Theorem 2.3, f is optimal. The question arises of how to compute a price function p with respect to which f is optimal. Knowing p_ϵ helps to find such a price function; we shall show how this computation can be done using Dijkstra's shortest path algorithm.

We use a variant of the previous construction. Define G_{aux} as above:

$$G_{aux} = (V_{aux}, E_{aux}) = (V \cup \{s\}, E_f \cup (\{s\} \times V)).$$

Extend p_ϵ and c to G_{aux} by defining $p_\epsilon(s) = 0$, $c(s,v) = -\lfloor p_\epsilon(v) \rfloor$ for all $v \in V$. Define a new cost function c' on E_{aux} as follows:

$$c'(v,w) = c_{p_\epsilon}(v,w) + \epsilon = c(v,w) - p_\epsilon(v) + p_\epsilon(w) + \epsilon.$$

Observe that ϵ-optimality implies $c'(v,w) \geq 0$ for all $(v,w) \in E_f$, and the definition of c' implies that $0 \leq c'(s,v) < 1 + \epsilon$ for all $v \in V$.

Theorem 3.4 *Let T be a shortest path tree in G_{aux} (rooted at s) with respect to the cost function c'. Let d_ϵ be the tree cost function on T with respect to the arc cost function c_{p_ϵ}. Then f is optimal with respect to the price function p defined by $p(v) = p_\epsilon(v) - d_\epsilon(v)$.*

Proof: Let $(v,w) \in E_f$. We must prove that $c(v,w) - p_\epsilon(v) + d_\epsilon(v) + p_\epsilon(w) - d_\epsilon(w) \geq 0$, i.e. $c_{p_\epsilon}(v,w) + d_\epsilon(v) \geq d_\epsilon(w)$. Let d' be the tree cost function with respect to the arc cost function c'. The definition of c' implies that $c'(v,w) = c_{p_\epsilon}$, $d'(v) \leq d_\epsilon(v) + n\epsilon$, and $d'(w) \geq d(w) + \epsilon$. Combining these with the inequality $c'(v,w) + d'(v) \geq d'(w)$ given by Lemma 3.2, we find that

$$c_{p_\epsilon}(v,w) + d_\epsilon(v) \geq d_\epsilon(w) - n\epsilon > d_\epsilon(w) - 1.$$

But the integrality of c and the definitions of c_{p_ϵ} and d_ϵ imply that $c_p(v,w) + d_\epsilon(v) = i + p_\epsilon(w)$ and $d_\epsilon(w) = j + p_\epsilon(w)$ for some integers i and j. Since $i + p_\epsilon(w) > j + p_\epsilon(w) - 1$ implies that $i \geq j$, we must have $c_{p_\epsilon}(v,w) + d_\epsilon(v) \geq d_\epsilon(w)$. ∎

By Theorem 3.4, we can compute a price function p with respect to which f is optimal by solving a single source shortest path problem on a graph with nonnegative arc costs and doing $O(m)$ additional work (to construct G_{aux} and to compute p given a shortest path tree). We can solve the shortest path problem in $O(m + n \log n)$ time with Dijkstra's algorithm implemented using Fibonacci heaps [15].

If $1/\epsilon$ is an integer of size $O(n)$ and, for every v, $p_\epsilon(v)$ is an integer multiple of ϵ, then the time to solve the shortest path problem can be reduced to $O(m)$. In this case, $c'(v,w)$ is an integer multiple of ϵ for every (v,w). Since in this case $c'(s,v) \leq 1$ for every $v \in V$, it follows that $d'(v) \leq 1$. Thus if all edge costs are multiplied by $1/\epsilon$, the result is a shortest path problem with nonnegative integer arc costs such that all minimum-cost paths from s have cost $O(n)$. The algorithm of Dial [10] and Wagner [55] will solve such a problem in $O(m)$ time.

3.2 Finding a Tight Error Parameter

Let us turn to problem 2: given a pseudoflow f, find the ϵ such that f is ϵ-tight. We need a definition. For a directed graph G with arc cost function c, the *minimum cycle mean* of G, denoted by $\mu(G,c)$, is the minimum mean cost of a cycle in G. Karp [34] has given an $O(nm)$ algorithm to compute $\mu(G,c)$. This problem is a special case of the so-called "tramp steamer" problem; see [36] for additional references. The connection between minimum cycle means and tight error parameters is given by the following theorem:

Theorem 3.5 *For a pseudoflow f, the ϵ for which f is ϵ-tight is $\epsilon = \max\{0, -\mu(G_f,c)\}$.*

Proof: Consider any cycle Γ in G_f. Let the length of Γ be l. For any ϵ, let $c^{(\epsilon)}$ be the cost function defined in Section 3.1: $c^{(\epsilon)}(v,w) = c(v,w)+\epsilon$ for $(v,w) \in E_f$. Let ϵ be such that f is ϵ-tight, and let $\mu = \mu(G_f,c)$. By Theorem 3.3, $0 \leq c^{(\epsilon)}(\Gamma) = c(\Gamma) + l\epsilon$, i.e. $c(\Gamma)/l \geq -\epsilon$. Since this is true for any cycle Γ, $\mu \geq -\epsilon$, i.e. $\epsilon \geq -\mu$. Conversely, for any cycle Γ, $c(\Gamma)/l \geq \mu$, which implies $c^{(-\mu)}(\Gamma) \geq 0$. By Theorem 3.3, this implies $\max\{0,-\mu\} \geq \epsilon$. ∎

By Theorem 3.5 a tight error parameter can be computed in $O(nm)$ time by applying Karp's minimum cycle mean algorithm.

We conclude this section with an observation regarding the structure of ϵ-tight pseudoflows. Suppose f is an ϵ-tight pseudoflow and $\epsilon > 0$. Let p be a price function such that f is ϵ-optimal with respect to p, and let Γ be a cycle in G_f with mean cost $-\epsilon$. Since $-\epsilon$ is a lower bound on the reduced cost of an arc in G_f, every arc of Γ must have reduced cost exactly ϵ.

4 The Minimum-Cost Flow Framework

In this section we give a high-level description of our minimum-cost circulation algorithms. First we describe a successive approximation framework. The running times of the algorithms developed within this framework depend on C, the biggest absolute value of the costs. Then we show that a natural modification of this framework leads to strongly polynomial algorithms.

```
procedure min-cost(V, E, u, c);
    [initialization]
    ε ← C;
    ∀v,  p(v) ← 0;
    if ∃ a circulation then f ← a circulation else return(null);
    [loop]
    while ε ≥ 1/n do
        (ε, f, p) ← refine(ε, f, p);
    return(f);
end.
```

Figure 2: The successive approximation algorithm. Versions of *refine* described in this paper decrease ϵ by a factor of two.

4.1 The Successive Approximation Framework

In this section we give a high-level description of our successive approximation algorithm (see Figure 2). We assume here that the arc cost function is integral. The algorithm maintains a circulation f_ϵ and a price function p_ϵ, such that f_ϵ is ϵ-optimal with respect to p_ϵ. The algorithm starts with $\epsilon = C$ (or alternatively $\epsilon = 2^{\lceil \log_2 C \rceil}$), with $p(v) = 0$ for all $v \epsilon V$, and with any initial circulation. An initial circulation can be found using one invocation of any maximum flow algorithm. (See e.g. [44], problem 11(e), p. 215.) Any initial circulation is C-optimal. The algorithm maintains the invariant that the circulation f is ϵ-optimal with respect to the price function p. The main loop of the algorithm repeatedly reduces the error parameter ϵ. When $\epsilon < 1/n$, the current circulation is minimum-cost, and the algorithm terminates.

Reducing the error in the optimality of the current circulation is the task of the subroutine *refine*. The input to *refine* is an error parameter ϵ, a circulation f, and a price function p such that f is ϵ-optimal with respect to p. The output from *refine* is a reduced error parameter ϵ, a new circulation f, and a new price function p such that f is ϵ-optimal with respect to p. The implementations of *refine* described in this paper reduce the error parameter ϵ by a constant factor (two unless mentioned otherwise).

The correctness of the algorithm is immediate from Theorem 2.3, assuming that *refine* is correct. The number of iterations of *refine* is $O(\log(nC))$. The initialization is dominated by the time to compute a circulation, which is $O(nm \log(n^2/m))$ [29]. The bounds we shall obtain for various versions of *refine* are all $\Omega(mn \log(n^2/m))$. Thus the iterations of *refine* dominate the running time of the algorithm, giving us the following theorem:

Theorem 4.1 *A minimum-cost circulation can be computed in the time required for $O(\log(nC))$ iterations of refine, if refine reduces ϵ by a factor of two.*

Remark: The minimum-cost circulation algorithm need not begin with a circulation; it can begin with any pseudoflow. If the problem is infeasible, this fact will be discovered during the first invocation of *refine*. Beginning with a circulation simplifies the presentation below, because we do not have to worry about feasibility.

Remark: Some applications require an optimal price function in addition to an optimal circulation. Once an optimal circulation f is found, a price function p with respect to which f is optimal can be computed in $O(m)$ time as described in Section 3.1, assuming that the initial value of ϵ was chosen to be a power of two.

We conclude this section by comparing our approximation approach with the traditional cost-scaling algorithms of Röck [45] and of Bland and Jensen [7]. These algorithms use a maximum flow routine in the inner loop. Our algorithms are similar to these cost-scaling algorithms in that *refine*, as we implement it, can be viewed as a generalization of a maximum flow algorithm. The previous algorithms, however, require $O(n)$ maximum-flow computations to halve the error parameter, whereas our algorithm requires only one invocation of *refine*. This accounts for our improved time bounds.

The traditional cost-scaling approach, where precision of the costs is increased bit-by-bit, can be used instead of our successive approximation approach. We used traditional scaling in the first versions of our algorithm; later, traditional scaling was used by Bertsekas and Eckstein [5]. Suppose that the current precision of costs is ϵ and we have an ϵ-optimal circulation. If we add the next bit of costs, the circulation is $(3/2)\epsilon$-optimal with respect to the resulting cost function. Applying a version of the *refine* subroutine that reduces the error parameter by a factor of three, we obtain an $\epsilon/2$-optimal circulation. Repeating this step $O(\log(nC))$ times, we obtain an optimal circulation. Asymptotic bounds for this approach are the same as for the successive approximation approach, but the constant factors are somewhat worse because of the additional errors introduced when new bits of costs are added. A potential advantage of this approach is that a smaller number of bits must be considered when manipulating the costs. For most practical applications, however, the costs are small and word operations can be used on them.

```
procedure min-cost(V, E, u, c);
    [initialization]
    ε ← C;
    ∀v,  p(v) ← 0;
    if ∃ a circulation f then f ← a circulation else return(null);
    [loop]
    while ε > 0 do begin
(*)     find λ and p_λ such that f is λ-tight with respect to p_λ;
        if λ > 0 then (ε, f, p) ← refine(λ, f, p_λ)
        else return(f);
end.
```

Figure 3: The strongly polynomial algorithm. Versions of *refine* described in this paper decrease ϵ by a factor of two.

4.2 The Strongly Polynomial Framework

The minimum-cost circulation algorithm of Section 4.1 has the disadvantage that the number of iterations of *refine* depends on the magnitudes of the costs. If the costs are huge integers, the method need not run in time polynomial in n and m; if the costs are irrational, the method need not even terminate. In this section we show that a natural modification of our successive approximation approach produces strongly polynomial algorithms, i.e. algorithms running in time polynomial in n and m, assuming arbitrary real numbers as capacities and costs and infinite-precision arithmetic. The running time bounds we shall derive for algorithms based on the approach of Section 4.1 are valid for the modified approach presented in this section. Unlike the previous strongly polynomial algorithms, our approach uses no rounding: true capacities and costs are used throughout.

The changes needed to make our approach strongly polynomial are to add an extra computation to the main loop of the algorithm and to change the termination condition. Before calling *refine* to reduce the error parameter ϵ, the new method computes the value λ and a price function p_λ such that the current circulation f is λ-tight with respect to p_λ. The strongly polynomial method is described on Figure 3. The value of λ and the price function p_λ in line $(*)$ are computed as described in Section 3. The algorithm terminates when the circulation f is optimal, i.e. $\lambda = 0$.

The time to perform line $(*)$ is $O(nm)$ by the results of Section 3. Since all the implementations of *refine* that we shall consider have a time bound greater than $O(nm)$, the extra time per iteration in the new version of the algorithm exceeds the time per iteration in the original version by less than a constant factor. Since each iteration at least halves ϵ,

the bound of $O(\log(nC))$ on the number of iterations derived in Section 4.1 remains valid, assuming that the costs are integral.

For arbitrary real-valued costs, we shall derive a bound of $O(m \log n)$ on the number of iterations, assuming that the *refine* subroutine used decreases the error parameter by a constant factor. We start by proving the following theorem, which is a slight generalization of a lemma of Tardos [50]. (To obtain the lemma of Tardos, take $\epsilon' = 0$.)

Theorem 4.2 *Let $\epsilon > 0$, $\epsilon' \geq 0$ be constants. Suppose that a circulation f is ϵ-optimal with respect to a price function p, and that there is an arc $(v, w) \in E$ such that $|c_p(v, w)| \geq n(\epsilon + \epsilon')$. Then for any ϵ'-optimal circulation f', we have $f(v, w) = f'(v, w)$.*

Proof: By antisymmetry, it is enough to prove the theorem for the case $c_p(v, w) \geq n(\epsilon + \epsilon')$. Let f' be a circulation such that $f'(v, w) \neq f(v, w)$. Since $c_p(v, w) > \epsilon$, the flow through the arc (v, w) must be as small as the capacity constraints allow, namely $-u(w, v)$, and therefore $f'(v, w) \neq f(v, w)$ implies $f'(v, w) > f(v, w)$. We show that f' is not ϵ'-optimal, and the theorem follows.

Consider $G_> = \{(x, y) \in E | f'(x, y) > f(x, y)\}$. Note that $G_>$ is a subgraph of G_f, and (v, w) is an arc of $G_>$. Since f and f' are circulations, $G_>$ must contain a simple cycle Γ that passes through (v, w). Let l be the length of Γ. Since all arcs of Γ are residual arcs, the cost of Γ is at least

$$c_p(v, w) - (l - 1)\epsilon \geq n(\epsilon + \epsilon') - (n - 1)\epsilon > n\epsilon'.$$

Now consider a cycle $\overline{\Gamma}$ obtained by reversing the arcs on Γ. Note that $\overline{\Gamma}$ is a cycle in $G_< = \{(x, y) \in E | f'(x, y) < f(x, y)\}$ and therefore a cycle in $G_{f'}$. By antisymmetry, the cost of $\overline{\Gamma}$ is less than $-n\epsilon'$ and thus the mean cost of $\overline{\Gamma}$ is less than $-\epsilon'$. Theorem 3.5 implies that f' is not ϵ'-optimal. ∎

To state an important corollary of Theorem 4.2, we need the following definition. We say that an arc $(v, w) \in E$ is *ϵ-fixed* if the flow through this arc is the same for all ϵ-optimal circulations.

Corollary 4.3 *Let $\epsilon > 0$, suppose f is ϵ-optimal with respect to a price function p, and suppose that for some arc (v, w), $|c_p(v, w)| \geq 2n\epsilon$. Then (v, w) is ϵ-fixed.*

Define F_ϵ to be the set of ϵ-fixed arcs.

Lemma 4.4 *Assume $\epsilon' \leq \frac{\epsilon}{2n}$. Suppose that there exists an ϵ-tight circulation f. Then $F_{\epsilon'}$ properly contains F_ϵ.*

Proof: Since every ϵ'-optimal circulation is ϵ-optimal, we have $F_\epsilon \subseteq F_{\epsilon'}$. To show that the containment is proper, we have to show that there is an ϵ'-fixed arc that is not ϵ-fixed.

Since the circulation f is ϵ-tight, there exists a price function p such that f is ϵ-optimal with respect to p, and there exists a simple cycle Γ in G_f with a mean cost of $-\epsilon$ (see Section 3.2). Since increasing f along Γ preserves ϵ-optimality, arcs of Γ are not ϵ-fixed.

We show that at least one arc of Γ is ϵ'-fixed. Let f' be a circulation that is ϵ'-optimal with respect to some price function p'. Since the mean cost of Γ is $-\epsilon$, there is an arc (v, w) of Γ with $c_{p'}(v, w) \leq -\epsilon \leq -2n\epsilon'$; by antisymmetry $c_{p'}(w, v) \geq 2n\epsilon'$. By Corollary 4.3, the arc (v, w) is ϵ'-fixed. But (v, w) is not ϵ-fixed since (v, w) is an arc of Γ. ∎

As the strongly polynomial algorithm runs, the value of the error parameter ϵ decreases; each time this value decreases by a factor of $2n$, at least one arc becomes fixed. If all arcs are fixed with respect to the current value of ϵ, the current circulation is optimal and the algorithm terminates during the next execution of line $(*)$. (Since not every arc is necessarily 0-fixed, the algorithm may terminate before all arcs are fixed.)

Theorem 4.5 *If refine reduces the error parameter ϵ by a constant factor, the total number of iterations of the while loop is $O(m \log n)$.*

Proof: Consider a time during the execution of the algorithm. During the next $O(\log n)$ iterations, either the algorithm terminates, or the error parameter is reduced by a factor of $2n$. In the latter case, Lemma 4.4 implies that an arc becomes fixed. If all arcs become fixed, the algorithm terminates in one iteration of the loop. Therefore the total number of iterations is $O(m \log n)$. ∎

Section 7 contains a version of our minimum-cost circulation algorithm that runs in $O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$ time. If the arc costs are huge, the strongly polynomial algorithm of Galil and Tardos [23], which runs in $O(n^2(\log n)(n \log n + m))$ time, is faster than our algorithm except on sparse graphs. Perhaps our algorithm can be improved to run in $O(n \log n)$ iterations, possibly by using some of the ideas in the Galil-Tardos paper. If this conjecture is true, our method would be competitive with that of Galil and Tardos on both sparse and dense graphs, and slower by a factor of $\log n$ on graphs of intermediate density.

```
procedure refine(ε, f, p);
    [initialization]
    ε ← ε/2;
    ∀(v, w) ∈ E do if c_p(v, w) < 0 then f(v, w) ← u(v, w);
    [loop]
    while ∃ a flow or price update operation that applies do
        select such an operation and apply it;
    return(ε, f, p);
end.
```

Figure 4: The generic *refine* subroutine.

5 A Generic Refinement Algorithm

In this section we describe an implementation of *refine* that is a generalization of our maximum flow algorithm [27,29]. We call this the *generic implementation.* The generic *refine* subroutine is a slight variant of the earlier minimum-cost flow algorithm of Bertsekas [4]. We use the subroutine in a different way, however. Whereas we use the subroutine repeatedly, each time to halve the error parameter ϵ, Bertsekas computes a minimum-cost flow by choosing an ϵ small enough that a single run of *refine* will convert an arbitrary circulation into an optimal circulation. Unfortunately, the latter approach results in only a pseudopolynomial-time algorithm, whereas we shall derive small polynomial time bounds for various versions of our algorithm.

As we have mentioned in Section 4.1, the input and output to *refine* is a triple $(\epsilon, f_\epsilon, p_\epsilon)$. The main effect of *refine* is to reduce ϵ by a factor of two.

The generic *refine* subroutine is described on Figure 4. It begins by halving ϵ and saturating every arc with negative reduced cost. This converts the circulation f into an ϵ-optimal pseudoflow (indeed, into a 0-optimal pseudoflow). Then, the subroutine converts the ϵ-optimal pseudoflow into an ϵ-optimal circulation by applying a sequence of *flow* and *price update operations*, each of which preserves ϵ-optimality. The idea of preserving ϵ-optimality in this context is due to Bertsekas [4].

The inner loop of our generic algorithm consists of repeatedly applying the two kinds of update operations described in Figure 5, in any order, until no such operation applies. Figure 6 illustrates the update operations in terms of kilter diagrams.

A *push* operation applies to a residual arc (v, w) of negative reduced cost, such that vertex v is active. It consists of pushing $\delta = \min\{e_f(v), u_f(v, w)\}$ units of flow from v to

push(v, w).
Applicability: v is active, $u_f(v, w) > 0$, **and** $c_p(v, w) < 0$.
Action: send $\delta = \min(e_f(v), u_f(v, w))$ units of flow from v to w.

relabel(v).
Applicability: v is active **and** $\forall w \in V\ u_f(v, w) > 0 \Rightarrow c_p(v, w) \geq 0$.
Action: replace $p(v)$ by $\min_{(v,w)\in E_f}(p(w) + c(v, w) + \epsilon)$.

Figure 5: The flow and price update operations.

w, thereby decreasing $e_f(v)$ and $f(w, v)$ by δ and increasing $e_f(w)$ and $f(v, w)$ by δ. The push is *saturating* if $u_f(v, w) = 0$ after the push and *nonsaturating* otherwise.

A *relabel* operation applies to an active vertex v that has no exiting residual arcs with negative reduced cost. It consists of increasing $p(v)$ to the highest value allowed by the ϵ-optimality constraints, namely $\min_{(v,w)\in E_f}\{c(v, w) + p(w) + \epsilon\}$. (If such a minimum is taken over the empty set, the problem is infeasible.)

The following lemmas, which generalize corresponding results of [4,29], give important properties of the update operations.

Lemma 5.1 *If the problem is feasible, a pseudoflow f is ϵ-optimal with respect to a price function p, and v is an active vertex, then either a push is applicable to some arc (v, w) or a relabeling is applicable to v.*

Proof: Obvious. ∎

Remark: If the problem is infeasible and we begin with any pseudoflow, then during the first iteration of *refine* there will be a vertex v with positive excess and no outgoing residual arcs. We can use this fact to detect infeasibility (instead of using a maximum flow subroutine as described in Section 4.1).

Lemma 5.2 *A push preserves the ϵ-optimality of a pseudoflow.*

Proof: Obvious. ∎

Lemma 5.3 *Suppose f is an ϵ-optimal pseudoflow with respect to a price function p and a relabeling is applied to a vertex v. Then the price of v increases by at least ϵ and the pseudoflow f is ϵ-optimal with respect to the new price function p'.*

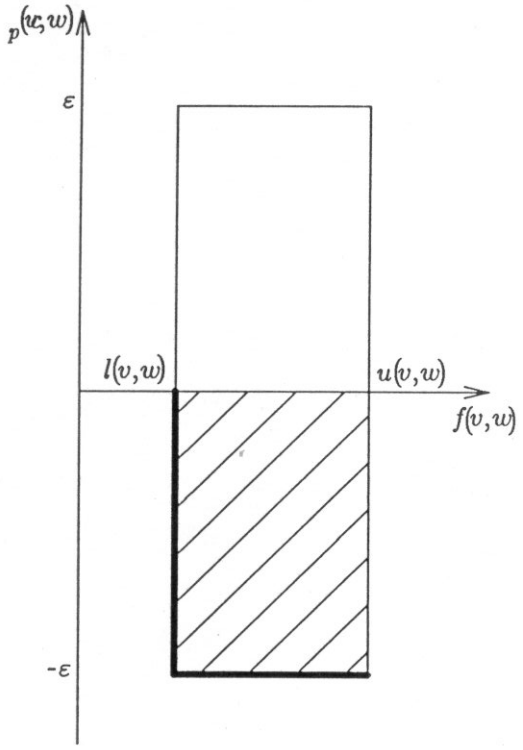


Figure 6: Operation $push(v, w)$ is applicable if $e_f(v) > 0$ and (v, w) is in the shaded rectangle (including the heavy curve). If $e_f(v) > 0$ but for all w *push* does not apply to (v, w), then *relabel* applies to v. Increasing the price of v corresponds to shifting the kilter diagram down. The absence of residual arcs in the shaded rectangle guarantees that the price of v increases by at least ϵ.

Proof: Before the relabeling, $c_p(v, w) \geq 0$ for all $(v, w) \in E_f$, i.e. $c_p(v, w) + p(w) \geq p(v)$ for all $(v, w) \in E_f$. Thus $p'(v) = \min_{(v,w)\in E_f} \{c(v, w) + p(w) + \epsilon\} \geq p(v) + \epsilon$.

To prove the second part of the lemma, observe that the only residual arcs whose reduced costs are affected by the relabeling are those of the form (v, w) or (w, v). Any arc of the form (w, v) has its reduced cost increased by the relabeling, preserving its ϵ-optimality constraint. Consider a residual arc (v, w). By the definition of p', $p'(v) \leq c(v, w) + p(w) + \epsilon$. Thus $c_{p'}(v, w) = c(v, w) - p'(v) + p(w) \geq -\epsilon$, which means that (v, w) satisfies its ϵ-optimality constraint. ∎

We can now establish the correctness of the *refine* subroutine, and hence of the entire algorithm.

Theorem 5.4 *If refine terminates, the pseudoflow f it returns is an ϵ-optimal circulation.*

Proof: The initial pseudoflow is ϵ-optimal. The update operations maintain ϵ-optimality by Lemmas 5.2 and 5.3. By Lemma 5.1, *refine* can terminate only if there are no active vertices, which, as noted in Section 2, implies that f is a circulation. ∎

Remark: One can obtain a variant of the *refine* subroutine by choosing a constant γ such that $0 \leq \gamma \leq \epsilon$, redefining *push* to be applicable when v is active, $u_f(v,w) > 0$, and $c_p(v,w) < -\gamma$, and redefining *relabel* to be applicable when v is active and $u_f(v,w) > 0$ implies $c_p(v,w) \geq -\gamma$. The bounds we derive on the original version of *refine* hold for this variant, with different constant factors depending on γ (see [26]). We have been unable to obtain any asymptotic improvements by choosing a value of γ other than zero.

Next we analyze the number of update operations that can take place during an execution of the generic implementation of *refine*. We begin with a few definitions. For a pseudoflow f and a price function p, we call an arc (v,w) *admissible* if $u_f\ (v,w) > 0$ and $c_p(v,w) < 0$. Note that a push operation is applicable to an arc (v,w) if and only if v is active and (v,w) is admissible. The *admissible graph* is the graph $G_A = (V, E_A)$ such that E_A is the set of admissible arcs.

As *refine* executes, the admissible graph changes. An important invariant is that the admissible graph remains acyclic, as was observed by Bertsekas [4] for the case when costs are integers and $\epsilon < 1/n$. To prove this fact for arbitrary real costs and arbitrary ϵ, we need the following lemma.

Lemma 5.5 *Immediately after a relabeling is applied to a vertex v, no admissible arcs enter v.*

Proof: Let (u,v) be a residual arc. Before the relabeling, $c_p(u,v) \geq -\epsilon$ by ϵ-optimality. By Lemma 5.3, the relabeling increases $p(v)$, and hence $c_p(u,v)$, by at least ϵ. Thus $c_p(u,v) \geq 0$ after the relabeling. ∎

Corollary 5.6 *Throughout the running of* refine, *the admissible graph is acyclic.*

Proof: Initially the admissible graph contains no arcs and is thus acyclic. Pushes obviously preserve acyclicity. Lemma 5.5 implies that relabelings also preserve acyclicity. ∎

Next we derive a crucial lemma.

Lemma 5.7 *Let f be a pseudoflow and f' a circulation. For any vertex v with $e_f(v) > 0$, there is a vertex w with $e_f(w) < 0$ and a sequence of distinct vertices $v = v_0, v_1, \ldots, v_{l-1}, v_l = w$ such that $(v_i, v_{i+1}) \in E_f$ and $(v_{i+1}, v_i) \in E_{f'}$ for $0 \leq i < l$.*

Proof: Let v be a vertex with $e_f(v) > 0$. Define $G_+ = (V, E_+)$, where $E_+ = \{(x,y) | f'(x,y) > f(x,y)\}$, and define $G_- = (V, E_-)$, where $E_- = \{(x,y) | f(x,y) > f'(x,y)\}$. Then $E_+ \subseteq E_f$, since $(x,y) \in E_+$ implies $f(x,y) < f'(x,y) \leq u(x,y)$. Similarly $E_- \subseteq E_{f'}$. Furthermore $(x,y) \in E_+$ if and only if $(y,x) \in E_-$ by antisymmetry. Thus to prove the lemma it suffices to show the existence in G_+ of a simple path $v = v_0, v_1 \ldots, v_l$ with $e_f(v_l) < 0$.

Let S be the set of all vertices reachable from v in G and let $\overline{S} = V - S$. (Set $\overline{S}$ may be empty.) For every vertex pair $(x,y) \in S \times \overline{S}$, $f(x,y) \geq f'(x,y)$, for otherwise $y \in S$. We have

$$
\begin{array}{rll}
0 & = \sum_{(x,y) \in (S \times \overline{S}) \cap E} f'(x,y) & \text{since } f' \text{ is a circulation} \\
& \leq \sum_{(x,y) \in (S \times \overline{S}) \cap E} f(x,y) & \text{holds term-by-term} \\
& = \sum_{(x,y) \in (S \times \overline{S}) \cap E} f(x,y) + \sum_{(x,y) \in (S \times S) \cap E} f(x,y) & \text{by antisymmetry} \\
& = \sum_{(x,y) \in (S \times V) \cap E} f(x,y) & \text{by definition of } \overline{S} \\
& = -\sum_{x \in S} e_f(x) & \text{by antisymmetry.}
\end{array}
$$

But $v \in S$. Since $e_f(v) > 0$, some vertex $w \epsilon S$ must have $e_f(w) < 0$. ∎

Using Lemma 5.7 we can bound the amount by which a vertex price can increase during an invocation of *refine*. The next lemma is similar to the lemma of Bland and Jensen [7] that bounds the number of maximum flow computations in a scaling step of their algorithm.

Lemma 5.8 *The price of any vertex v increases by at most $3n\epsilon$ during an execution of refine.*

Proof: Let $f_{2\epsilon}$ and $p_{2\epsilon}$ be the circulation and price functions on entry to *refine*. Suppose a relabeling causes the price of a vertex v to increase. Let f be the pseudoflow and p the price function just after the relabeling. Then $e_f(v) > 0$. Let $v = v_0, v_1, \ldots, v_l = w$ with $e_f(w) < 0$ be the vertex sequence satisfying Lemma 5.7 for f and $f' = f_{2\epsilon}$.

The ϵ-optimality of f implies

$$
-l\epsilon \leq \sum_{i=0}^{l-1} c_p(v_i, v_{i+1}) = p(w) - p(v) + \sum_{i=0}^{l-1} c(v_i, v_{i+1}) \tag{8}
$$

The 2ϵ-optimality of $f_{2\epsilon}$ implies

$$-2l\epsilon \leq \sum_{i=0}^{l-1} c_{p_{2\epsilon}}(v_{i+1}, v_i) = p_{2\epsilon}(v) - p_{2\epsilon}(w) + \sum_{i=0}^{l-1} c(v_{i+1}, v_i) \tag{9}$$

But $\sum_{i=0}^{l-1} c(v_i, v_{i+1}) = -\sum_{i=0}^{l-1} c(v_{i+1}, v_i)$ by cost antisymmetry. Furthermore $p(w) = p_{2\epsilon}(w)$ since a vertex with negative excess has the same price as long as its excess remains negative. Adding inequalities (8) and (9) and rearranging terms thus gives

$$p(v) \leq p_{2\epsilon}(v) + 3l\epsilon < p_{2\epsilon}(v) + 3n\epsilon.$$

∎

Now we count relabelings.

Lemma 5.9 *The number of relabelings during an execution of refine is at most* $3n(n-1)$.

Proof: Immediate from Lemmas 5.3 and 5.8. (At least one vertex is never relabeled; namely, the last remaining one with negative excess.) ∎

To count pushes, we amortize the saturating pulses over the relabelings and the nonsaturating pushes over the relabelings and the saturating pushes.

Lemma 5.10 *The number of saturating pushes during one execution of refine is at most* $3nm$.

Proof: For an arc (v, w), consider the saturating pushes along this arc. Before the first such push can occur, vertex v must be relabeled. After such a push occurs, v must be relabeled before another such push can occur. But v is relabeled at most $3n$ times. Summing over all arcs gives the desired bound. ∎

The bound in the next lemma and its proof are due to Ronald Rivest. (Our original bound was $O(n^4)$.)

Lemma 5.11 *The number of nonsaturating pushes during one execution of refine is at most* $3n^2(m+n)$.

Proof: For each vertex v, let $\Phi(v)$ be the number of vertices reachable from v in the current admissible graph G_A. Let $\Phi = 0$ if there are no active vertices, $\Phi = \sum\{\Phi(v)|v \text{ is active}\}$ otherwise. Throughout the running of *refine*, $\Phi \geq 0$. Initially $\Phi \leq n$, since G_A has no arcs.

Consider the effect on Φ of update operations. A nonsaturating push decreases Φ by at least one, since by Corollary 5.6 G_A is always acyclic. A saturating push can increase Φ by at most n, since at most one inactive vertex becomes active. If a vertex v is relabeled, Φ also can increase by at most n, since by Lemma 5.5 $\Phi(w)$ for $w \neq v$ can only decrease. The total number of nonsaturating pushes is thus bounded by the initial value of Φ plus the total increase in Φ throughout the algorithm, i.e. by $n + 3n^2(n-1) + 3n^2m \leq 3n^2(m+n)$. ∎

6 Efficient Sequential Implementation

The running time of *refine* depends upon the order in which basic operations are applied and on the details of the implementation, but any reasonable implementation will run in polynomial time. As a first step toward obtaining an efficient sequential implementation, we develop an implementation that runs in $O(n^2m)$ time. Then we improve the implementation to obtain an $O(n^3)$ time bound.

6.1 A Generic Implementation

To describe a straightforward implementation of the generic algorithm, we need some data structures to represent the network and the pseudoflow. We associate a positive direction with each edge $\{v, w\}$, and we store with each such edge the four values $u(v, w)$, $u(w, v)$, $f(v, w)$, and $c(v, w)$. Each vertex v has a list of the incident edges $\{v, w\}$, in fixed but arbitrary order. (Thus an edge $\{v, w\}$ appears in exactly two edge lists, the one for v and the one for w.) For each vertex v we maintain the excess $e_f(v)$ and a *current edge* $\{v, w\}$; the corresponding *current arc* (v, w) is the current candidate for a push out of v. Initially the current edge is the first edge on the edge list of v.

The generic implementation of the while loop in *refine* consists of repeating the following steps until there are no active vertices: select an active vertex v; apply to v the *push/relabel* operation. The *push/relabel* operation is described in Figure 7. Such an operation applies to an active vertex v with current edge $\{v, w\}$. It pushes flow through the arc (v, w) if this

```
push/relabel(v).
Applicability: v is active.
Action:        let {v, w} be the current edge of v.
               if u_f(v, w) > 0 and c_p(v, w) < 0 then push(v, w)
               else
                    if {v, w}is not the last edge on the edge list of v then
                              replace {v, w} as the current edge of v
                              by the next edge on the list
                    else begin
                              make the first edge on the edge list of v the current edge;
                              relabel(v);
                    end.
```

Figure 7: The push/relabel operation.

arc is eligible. If not, it replaces the current edge of v by the next edge on the edge list of v unless the current edge is the last edge on the list, in which case it makes the first edge on the list the current one and relabels v.

Lemma 6.1 *The push/relabel procedure uses the price update operation only when this operation is applicable.*

Proof: Suppose a vertex v is relabeled. Just before the relabeling, for each arc (v, w), either $c_p(v, w) \geq 0$ or $u_f(v, w) = 0$, because $p(v)$ has not changed since $\{v, w\}$ was most recently the current edge of v, $p(w)$ never decreases, and $u_f(v, w)$ cannot increase unless $c_p(w, v) < 0$, i.e. $c_p(v, w) \geq 0$. ∎

The generic implementation needs one additional data structure, a set S containing all active vertices. Initially S contains all vertices whose excess becomes positive during the initialization step of *refine*. Updating S takes only $O(1)$ time per *push/relabel* operation. (Such an operation requires possibly deleting one vertex from S and adding one vertex to S.)

Theorem 6.2 *The generic implementation of refine runs in $O(n^2 m)$ time, giving an $O(n^2 m \min\{\log(nC), m \log n\})$ time bound for computing a minimum-cost circulation.*

Proof: The total number of passes through each edge list is $O(n)$ by Lemmas 5.3 and 5.8. The desired bound follows from Lemmas 5.9, 5.10, and 5.11. ∎

Discharge.
Applicability: v is active.
Action: apply *push/relabel* operations to v until v is relabeled or becomes inactive.

Figure 8: The *discharge* operation.

6.2 The Wave Implementation

To obtain a faster version of *refine*, we begin by somewhat restricting the freedom of choice in *push/relabel* operations. The restricted algorithm consists of repeatedly selecting an active vertex and applying to it the *discharge* operation, described in Figure 8. A discharge operation applies *push/relabel* operations to an active vertex until it is relabeled or becomes inactive, i.e. its excess drops to zero.

Remark: We can actually allow a certain amount of flexibility in the discharge operation. Specifically, after a vertex v is relabeled, additional *push/relabel* operations can optionally be applied to v, as long as v remains active. The bounds we shall derive, although stated for the original version of *discharge*, are valid for this more general version as well.

There remains the issue of the order in which to consider active vertices. The *first-in first-out* (FIFO) algorithm consists of maintaining the set of active vertices as a queue, repeatedly discharging the front vertex on the queue and adding newly active vertices to the rear of the queue. We might expect the first-in, first-out algorithm to be very efficient, since the analogous algorithm for the maximum flow problem requires $O(n^2)$ passes over the queue and runs in $O(n^3)$ time. Unfortunately, the first-in, first-out implementation of *refine* makes $\Theta(n^3)$ passes over the queue in the worst case, and we are unable to establish a bound on the total running time better than the $O(n^2m)$ bound of the generic implementation. The following lemma establishes an $O(n^3)$ bound on queue passes; the appendix describes a class of examples that take $\Omega(n^3)$ passes.

We define passes over the queue as follows. Pass one consists of the discharge operations applied to vertices added to the queue during the initialization of the pseudoflow. For $i \geq 2$, pass i consists of the discharge operations applied to vertices added to the queue during pass $i-1$.

Lemma 6.3 *During the execution of the first-in, first-out implementation of refine, there are $O(n^3)$ passes over the queue.*

Proof: As in the proof of Lemma 5.11, let $\Phi(v)$, for each vertex v, be the number of vertices reachable from v in the current admissible graph G_A. Let $\Phi = 0$ if there are no active vertices, $\Phi = \max\{\Phi(v)|v \text{ is active}\}$ otherwise. Consider the effect on Φ of a pass over the queue. If no relabelings take place during the pass, Φ drops by at least one, since the excess pushed from any vertex is pushed to vertices with smaller values of Φ. If one or more relabelings take place during the pass, Φ can increase by at most n, since it is always the case that $0 \leq \Phi \leq n$. The total number of passes is thus at most the number of relabelings plus the number of passes in which Φ decreases. The latter is at most n (the maximum possible initial value of Φ) plus $3n^2(n-1)$ (i.e. n times the maximum number of relabelings). This gives a bound of at most $3n(n-1) + n + 3n^2(n-1) \leq 3n^3$ on the total number of passes. ∎

Although the first-in, first-out approach fails to improve the efficiency of *refine*, a similar approach, proposed by Charles Leiserson (private communication) and independently by Bertsekas [4] (without some implementation details and the use of scaling), does produce a faster algorithm. We call this method the *wave* method since it resembles the wave algorithm for computing maximum network flows [53]. The wave method is a generalization of a version of our maximum flow algorithm that pushes the flow from an active vertex with the biggest distance label [29]; in the wave method, the vertices are processed in topological order with respect to the admissible graph.

The wave method, described in Figure 9, maintains a list L of all the vertices of G, in topological order with respect to the current admissible graph G_A, i.e. if (v, w) is an arc of G_A, v appears before w in L. The method also maintains a *current vertex* in L, which is the next candidate for discharging. Initially L contains the vertices of G in any order, and the current vertex is the first one on L. The method repeatedly scans L, applying the *discharge* operation to each active vertex encountered. When a discharge causes a relabeling to take place, the affected vertex v is moved to the front of L, but the next vertex examined is the one after the old position of v. The method terminates when there are no active vertices.

The key to analyzing the wave method is to observe that, because of the topological ordering of L, the method terminates immediately after a pass through L during which no relabelings occur.

Lemma 6.4 *The wave method terminates after $O(n^2)$ passes through L.*

Proof: Lemma 5.5 implies that the wave method maintains the invariant that L is topolog-

```
procedure wave;
    let L be a list of all vertices;
    let v be the first vertex on L;
    while ∃ an active vertex do begin
        if v is active then begin
            discharge(v);
            if the discharge has relabeled v then
                move v to the front of L;
        end;
        replace v by the vertex after the old position of v on L,
        or by the first vertex on L if v was previously last on L;
    end;
end.
```

Figure 9: The *wave* method.

ically ordered. If a pass over L occurs during which no relabelings occur, then the ordering of L does not change during the pass and every vertex is able to reduce its excess to zero. Thus no active vertices exist after the pass. ∎

Theorem 6.5 *The wave implementation of refine runs in $O(n^3)$ time, giving an $O(n^3 \min\{\log(nC), m \log n\})$ bound for finding a minimum-cost circulation.*

Proof: Lemma 6.4 implies that there are $O(n^3)$ nonsaturating pushes (one per vertex per pass) during an execution of *refine*. By Lemma 5.10 there are $O(nm)$ saturating pushes. The time taken by *refine* is $O(n^3)$ plus $O(1)$ per push, for a total of $O(n^3)$. ∎

The following order of processing vertices also leads to an $O(n^3)$ running time for *refine*: apply a *discharge* operation to the first active vertex on L, move this vertex to the front of L if it has been relabeled, and repeat until there are no active vertices. The only difference from the version of the *wave* subroutine described on Figure 9 is that when a vertex v is relabeled and moved to the beginning of L, the processing continues from the beginning of L (i.e. from the current position of v), rather than from the old position of v. We call this the *first-active* version of *refine*.

7 Use of Dynamic Trees

Through the use of sophisticated data structures, we can further reduce the running time of *refine* on sparse graphs. We shall describe how to implement the first-active version of

find-root(v):	Find and return the root of the tree containing vertex v.
find-size(v):	Find and return the number of vertices in the tree containing vertex v.
find-value(v):	Compute and return $g(v)$.
find-min(v):	Find and return the ancestor w of v of minimum value $g(w)$. In case of a tie, choose the vertex w closest to the root.
change-value(v, x):	Add real number x to $g(w)$ for each ancestor w of v. (We adopt the convention that $\infty + (-\infty) = 0$.)
link(v, w):	Combine the trees containing vertices v and w by making w the parent of v. This operation does nothing if v and w are in the same tree or if v is not a tree root.
cut(v):	Break the tree containing v into two trees by deleting the arc from v to its parent. This operation does nothing if v is a tree root.

Figure 10: Dynamic tree operations.

refine, described at the end of Section 6.2, so that it runs in $O(nm \log(n^2/m))$ time.

There are two bottlenecks in the implementation of this algorithm. First, we need a way to maintain L so that the first active vertex is easy to find. For this purpose we use a data structure based on finger search trees [39,54]. Second, we need a way to maintain the set of current edges so that the time per nonsaturating push is $o(1)$. For this purpose we use the dynamic tree data structure of Sleator and Tarjan [48,49]. The $O(nm \log(n^2/m))$ time bound of the resulting version of *refine* matches that of the dynamic tree implementation of our maximum flow algorithm [29], and its derivation is similar, except for the additional complication of maintaining L.

7.1 The Tree Method

We shall postpone to Section 7.3 the issue of maintaining L and focus on the use of dynamic trees. The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each vertex v of which has an associated value $g(v)$, which is either a real number or ∞. The structure supports the seven tree operations described in Figure 10. The total time for l tree operations, starting with a collection of single-vertex trees, is $O(l \log k)$, where k is an upper bound on the maximum number of vertices in a tree. (The implementation of dynamic trees discussed in [48,49] does not support *find-size* operations, but it is easily modified to do so, as discussed in [29].)

In our application, if *parent*(v) is the parent of a vertex v in a dynamic tree, then $\{v, \textit{parent}(v)\}$ is the current arc of v and $(v, \textit{parent}(v))$ is admissible, i.e. $u_f(v, \textit{parent}(v)) > 0$

```
Send(v).
Applicability: v is active.
Action:        while find-root(v) ≠ v and e(v) > 0 do begin
                   send δ ← min{e(v), find-value(find-min(v))} units of flow
                       along the tree path from v by performing change-value(v, −δ);
                   while find-value(find-min(v)) = 0 do begin
                       i ← find-min(v);
                       perform cut(i) followed by change-value(i, ∞);
                   end;
               end.
```

Figure 11: The Send operation.

and $c_p(v, parent(v)) < 0$. Not all admissible current arcs are tree arcs, however. The value $g(v)$ of a vertex v in its dynamic tree is $u_f(v, parent(v))$ if v has a parent and ∞ if v is a tree root. Initially, each vertex is in a one-vertex dynamic tree and has value ∞. We limit the maximum tree size to k, where k is a parameter to be chosen later, satisfying $2 \leq k \leq n$.

The dynamic tree implementation of *refine*, which we shall call the *tree method*, uses tree operations to send flow along an entire tree path at a time, thereby avoiding explicitly dealing with nonsaturating pushes. The heart of the algorithm is the procedure *send*(v) defined in Figure 11, which pushes excess from a nonroot vertex v to the root of its tree, cuts arcs saturated by the push, and repeats these steps until $e(v) = 0$ or v is a tree root.

Instead of the *push/relabel* operation, the tree method uses the *tree push/relabel* operation, defined in Figure 12. Such an operation applies to an active vertex v that is the root of a dynamic tree. There are two main cases. The first case occurs if the current edge $\{v, w\}$ of v is such that (v, w) is admissible. If the trees containing v and w together have at most k vertices, these trees are linked by making w the parent of v, and then a *send*(w) is done. If these trees together contain more than k vertices, a *push*(v, w) is done followed by a *send*(w). The second case occurs if (v, w) is not admissible. In this case the current edge of v is updated and v is relabeled if the previous current edge was the last on the edge list of v. If v is relabeled, all tree arcs entering v are cut, thereby preserving the invariant that all tree arcs are admissible.

The tree method uses the modified form of *discharge* shown in Figure 13, called *tree-discharge*, which is the same as discharge except that *push-relabel* operations are replaced by *tree-push/relabel* operations, which are applied to an active vertex v until it becomes inactive.

tree-push/relabel(v).
Applicability: v is an active tree root.
Action: let $\{v, w\}$ be the current edge of v;
(1) if (v, w) is admissible **then**
(1a) if *find-size*(v) + *find-size*$(w) \leq k$ **then begin**
[make w the parent of v]
change-value$(v, -\infty)$;
change-value$(v, r_f(v, w))$;
link(v, w);
[push excess from v to w]
send(v);
end
(1b) else [*find-size*(v) + *find-size*$(w) > k$] **begin**
push(v, w);
send(w);
end
(2) else [(v, w) is not admissible]
(2a) if $\{v, w\}$ is not the last edge on the edge list of v **then**
replace $\{v, w\}$ as the current edge by the next edge on the list
(2b) else [$\{v, w\}$ is the last edge on the edge list of v] **begin**
make the first edge on the list the current one;
for every child u of v **do begin**
cut(u);
change-value(u, ∞);
end;
relabel(v);
end.

Figure 12: The *tree-push/relabel* operation.

The tree method itself is defined in Figure 14. It differs from the wave method in two respects: active vertices are processed in a different order, and *discharge* operations are replaced by *tree-discharge* operations.

It is important to realize that the tree method stores values of the pseudoflow f in two different ways. If $\{v, w\}$ is an edge that is not a tree edge, then $f(v, w)$ is stored explicitly, with $\{v, w\}$. If $\{v, w\}$ is a tree edge, with w the parent of v, then $u_f(v, w)$ is stored implicitly in the dynamic tree data structure as $g(v)$. Whenever a tree edge $\{v, w\}$ is cut, $g(v)$ must

tree-discharge(v).
Applicability: v is an active tree root.
Action: apply *tree-push/relabel* operations to v until v becomes inactive.;

Figure 13: The *tree-discharge* operation.

```
procedure tree;
    let L be a list of all vertices;
    while ∃ an active vertex do begin
        let v be the first active vertex on L;
        tree-discharge(v);
        if the discharge has relabeled v then
            move v to the front of L;
    end;
end.
```

Figure 14: The *tree* method.

be computed and $f(v, w)$ set to its correct current value. In addition, when the algorithm terminates, flow values must be computed for all edges remaining in dynamic trees.

Three observations imply the correctness of the tree method. First, the algorithm maintains the invariant that every tree arc is admissible. Second, the acyclicity of the admissible graph implies that the algorithm never attempts to link two vertices in the same dynamic tree. Third, a vertex v that is not a tree root can be active only in the middle of case (1) of a *tree-push/relabel* operation. To see this, note that only in this case does the algorithm create an active nonroot vertex, and this event is followed by a *send* operation, after which all the excess is on one or more roots.

Remark: If we let the maximum tree size k equal n and generalize the tree method to apply *discharge* operations to active vertices in any order, we obtain an $O(nm \log n)$ implementation of *refine*. See [26,29] for details of the analysis. This method does not need an elaborate data structure to select active vertices for processing, but its time bound is asymptotically worse than that of the tree method on non-sparse graphs.

Remark: The version of the tree method that processes active vertices in the same order as the wave method, i.e. by making passes through the list L, also runs in $O(nm \log(n^2/m))$ time, but it is slightly more complicated to implement.

7.2 Analysis of the Tree Method

In order to analyze the tree method, we introduce a bit of terminology. We say a vertex is *discharged* when a *discharge* operation is applied to it. We say a vertex is *activated* when it becomes active while it is a root vertex or it becomes a tree root while it is active. A vertex can only be activated during the initialization or in cases (1a) and (1b) of *tree-push/relabel*.

In case (1a), each push of flow from v during the operation *send* (v) can activate a vertex. In case (1b), the operation *push* (v, w) can activate w if w is a root. If w is not a root, each push of flow from w during the operation *send* (w) can activate a vertex; and, in addition, the last *cut* of the *send*, if it cuts the current arc leaving w, can activate w.

Lemma 7.1 *Between relabelings, the order of L does not change, and each vertex is discharged at most once.*

Proof: The only time the order of L is changed is when a vertex is relabeled. When a vertex is discharged, its excess is moved to vertices after it on L. The lemma follows. ∎

Lemma 7.2 *The maximum size of a dynamic tree is k. The number of dynamic tree operations is $O(nm)$ plus $O(1)$ per vertex activation.*

Proof: The test in case (1a) of *tree-push/relabel* guarantees that the size of a dynamic tree can never exceed k. Each *tree-push/relabel* operation does $O(1)$ tree operations plus $O(1)$ tree operations per *cut* operation (in invocations of *send* and in case (2b)). The total number of *cut* operations is at most the number of *link* operations, which is $O(nm)$ by Lemmas 5.3 and 5.8. (An operation *link*(v, w) can only occur once between relabelings of v.) Thus the number of tree operations is $O(nm)$ plus $O(1)$ per *tree-push/relabel*. During an operation *discharge*(v), every *tree-push/relabel* except the last does either a *link*, a saturating *push*, or a *relabel*, or changes the current edge of v. Thus there are $O(nm)$ tree operations plus $O(1)$ per *discharge*. The number of *discharge* operations equals the number of vertex activations. ∎

The next lemma, analogous to Lemma 5.2 of [29], is the heart of the analysis.

Lemma 7.3 *The number of vertex activations is $O(nm + n^3/k)$.*

Proof: There are at most $n - 1$ activations during the initialization. All others occur in cases (1a) and (1b) of *tree-push/relabel*. In either case the number of activations is at most one more than the number of cuts done during the invocation of *send* in the case. Each occurrence of (1a) results in a *link*. It follows that the number of activations is $O(nm)$ plus one per occurrence of (1b) that results in a nonsaturating push, an activation, and no cuts. Let us call such an occurrence of (1b) a *critical occurrence*.

For any vertex u, let T_u be the dynamic tree containing u and let $|T_u|$ be the number of vertices it contains. We say T_u is *small* if $|T_u| \leq k/2$ and *large* otherwise. At any time there are at most $2n/k$ large trees.

We shall charge each critical occurrence of (1b) to either a link, a cut, or a relabeling. We charge $O(1)$ occurrences to each link or cut and $O(n/k)$ to each relabeling. This gives a bound of $O(nm + n^3/k)$ on the total number of critical occurrences, and hence on the total number of activations. For ease in stating the argument, we shall assume that a "phantom" relabeling occurs at initialization.

Consider a critical occurrence (1b), in which a nonsaturating push from a vertex v to a vertex w occurs, followed by a *send* (w). Since this occurrence is critical, T_w does not change as a result of the *send* and the root of T_w is the only vertex activated by the occurrence. The condition on entry to case (1b) guarantees that T_v or T_w is large, giving us two possibilities to consider.

Suppose T_v is large. Vertex v is the root of T_v. The critical occurrence of (1b) removes all the excess from v, which means that such an occurrence can apply to a vertex v only once between relabelings. If T_v has changed since the last preceding relabeling, charge the critical occurrence to the link or cut that changed T_v most recently before the occurrence. The total number of such charges is at most one per link and two per cut. (A link forms one new tree; a cut, two.) If T_v has not changed since the last preceding relabeling, charge the occurrence to this relabeling. Since at any time, including just after a relabeling, there are at most $2n/k$ large trees, the total number of such charges is at most $2n/k$ per relabeling.

Suppose on the other hand that T_w is large. The critical occurrence activates the root of T_w, say r. A given vertex r can be activated at most once between relabelings. If T_w has changed since the last preceding relabeling, charge the occurrence to the link or cut that most recently changed T_w before the occurrence. The number of such charges is at most one per link and two per cut. If T_w has not changed, charge the occurrence to the last preceding relabeling. The number of such charges is at most $2n/k$ per relabeling. ∎

Lemma 7.4 *The running time of the tree method, not counting time spent maintaining L and finding vertices to discharge, is $O((nm + n^3/k)\log k)$.*

Proof: By Lemmas 7.2 and 7.3, the total time spent doing dynamic tree operations is $O((nm + n^3/k)\log k)$. In addition, $O(nm)$ time is spent in initialization, relabelings, and

maintenance of current edges. ∎

7.3 Representation of the Vertex List

The task remaining is to select a representation for the vertex list L and analyze the time needed to maintain it. To facilitate the finding of active vertices, we split L into sublists by dividing L just before each active vertex. Each sublist consists of a first vertex, called the *head* of the sublist, and zero or more inactive vertices. The head of the first sublist can be active or inactive; the second and subsequent sublists all have active heads. We represent L by a doubly linked list of these sublists.

Consider the operations on L needed in the tree method. Finding the first active vertex on L requires examination of the heads of the first two sublists and selection of the first of these that is active. After a vertex v is relabeled, it must be moved to the front of L. We can do this as follows. When v is relabeled, it is active. If v is not already on the front of L, we delete it from its current sublist (of which it is the head), concatenate what remains of this sublist with the preceding sublist, and either insert v into the front of the first sublist if its head is inactive, or make v into a one-element sublist on the front of L, if the head of the old first sublist is active.

List L must also be updated when vertices are activated and discharged. (Recall from Section 7.2 that a vertex is activated when it is a root and becomes active or when it is active and becomes a root.) We can do this as follows. When a vertex v is activated, we split the sublist containing v into two sublists, with v the head of the second one. Immediately after a vertex is discharged, we concatenate the sublist of which it is the head with the preceding sublist, if any.

In summary, we need the following four operations on sublists:

1. *Initialize* a sublist containing a given sequence of vertices.
2. *Access* the head of a given sublist.
3. *Split* the sublist containing a given vertex just before that vertex and return the two resulting sublists.
4. *Concatenate* two sublists and return the resulting sublist.

Deleting the head of a sublist is a special case of splitting; adding a new head to the front of a sublist is a special case of concatenation.

We desire a sublist representation that achieves the following time bounds for the four operations:

1. $O(l \log l)$ to initialize a sublist containing l vertices.
2. $O(1)$ to access the head of a sublist.
3. $O(1)$ to split a sublist just after a given vertex.
4. $O(1 + \log \min\{l_1, l_2\})$ to concatenate two sublists containing l_1 and l_2 vertices, respectively.

It suffices that these bounds be *amortized*, i.e. that the total time for an arbitrary sequence of operations starting with no sublists be no greater than the sum of the time bounds of the operations.[2] Let us call a sublist representation *suitable* if it achieves the desired bounds in the amortized sense.

Lemma 7.5 *With a suitable sublist representation, the total time needed to maintain L and to find vertices for discharging is $O((nm + n^3/k) \log k)$, where k is the maximum dynamic tree size.*

Proof: The time to initialize the sublists of which L is comprised is $O(n \log n)$. The time to find the first active vertex on L is $O(1)$; by Lemma 7.3, this operation occurs $O((nm + n^3/k) \log k)$ times. The number of splitting operations is $O(1)$ per vertex activation and $O(1)$ per relabeling, for a total time bound of $O((nm + n^3/k) \log k)$. There are at most two concatenations per relabeling. One takes $O(1)$ time; the other, $O(\log n)$. The total time for concatenations accompanying relabelings is thus $O(n^2 \log n)$.

It remains to bound the time for concatenations that occur between relabelings as a result of discharge operations. We shall show that the total concatenation time during an interval in which no relabelings occur is $O((n/k) \log k)$ plus $O(\log k)$ per vertex activation, from which an overall $O((nm + n^3/k) \log k)$ bound on concatenation time follows.

Consider a time interval I during which no relabelings occur. There is one concatenation per *discharge* operation during I. An operation *discharge*(v) results in the concatenation of the sublist containing v, say $S(v)$, with the preceding sublist. Observe that during interval I successively concatenated sublists $S(v), S(w), \ldots$ are vertex-disjoint. Call a sublist $S(v)$

[2] The second author's survey paper [51] contains a thorough discussion of amortization.

small if it contains at most k vertices and *large* otherwise. The time for concatenating small sublists is $O(\log k)$ per concatenation, i.e. $O(\log k)$ per vertex activation. Let $v_1, v_2, \ldots, v_l$ be the vertices for which $S(v_1), S(v_2), \ldots, S(v_l)$ are large when concatenated during I. Then $l \leq n/k$, and the total time for concatenating these sublists is

$$O(\sum_{i=1}^{l} \log(1 + |S(v_i)|)) \leq l \log(1 + n/l) \leq (n/k) \log(1 + k).$$

(The worst case occurs when all sublists are of the same size.) Thus the total time for concatenating large sublists, summed over all intervals containing no relabelings, is $O((n^3/k) \log k)$. ∎

Theorem 7.6 *With a suitable sublist representation, the tree method runs in* $O(nm \log(n^2/m))$ *time, giving an* $O(nm \log(n^2/m) \min\{\log(nC)), m \log n\})$ *bound for finding a minimum-cost circulation.*

Proof: Choose $k = 2n^2/m$ and apply Lemmas 7.4 and 7.5. ∎

A suitable way to represent sublists is to use finger search trees [39,54]. Such trees give the following amortized time bounds for the four sublist operations (see [54]):

1. $O(l \log l)$ to initialize a sublist containing l vertices.
2. $O(1)$ to access the head of a sublist.
3. $O(1 + \log \min\{l_1, l_2\})$ to split a sublist into sublists of sizes l_1 and l_2, or to concatenate two sublists of sizes l_1 and l_2.

These are not quite the desired bounds. In particular, we want each splitting operation to take $O(1)$ time rather than $O(1 + \log \min\{l_1, l_2\})$. Fortunately we can charge the splitting time to concatenations, thereby reducing the amortized time per split to $O(1)$ while increasing the amortized time per concatenation by only a constant factor.

The argument uses the idea of a *potential function* [51]. We define the *potential* of a sublist of size l to be $-c(1 + \log_2 l)$, where c is the constant in the amortized time bound for splitting. We define the *total potential* of a collection of sublists to be the sum of their individual potentials. We define the *nominal time* of a sublist operation to be its

amortized time bound plus the net increase in total potential it causes. For any sequence of sublist operations, the sum of the nominal times equals the sum of the amortized time bounds plus the final total potential minus the initial total potential. In our application, the initial total potential is zero (there are no sublists initially) and the final potential is at least $-cn \log n$. Thus for any sequence of sublist operations the sum of the amortized time bounds is $O(n \log n)$ plus the sum of the nominal times. The nominal time to initialize a sublist of l items is $O(l \log n)$. (The initial potential is zero; the final potential is negative.) The nominal time to access the head of a sublist is $O(1)$. (There is no change in potential.) The nominal time for splitting is $O(1)$. (When sublists of sizes l_1 and l_2 are formed by a split, the potential change is $c(-1 - \log_2 l_1 - 1 - \log_2 l_2 + 1 + \log(l_1 + l_2)) \leq -c \log \min\{l_1, l_2\}$.) The nominal time for a concatenation is at most a constant factor times its amortized time bound. (When sublists of sizes l_1 and l_2 are concatenated, the potential change is $c(-1 - \log(l_1 + l_2) + 1 + \log l_1 + 1 + \log l_2) \leq c(1 + \log \min\{l_1, l_2\}.)$) Thus we have the following result.

Lemma 7.7 *Finger search trees are a suitable representation of sublists.*

8 Refinement Using Blocking Flows

It is natural to ask how much harder the minimum-cost circulation problem is than the maximum flow problem. Based on the results of Sections 4-7, one might conjecture that the minimum-cost circulation problem can be solved in $O(\min\{\log(nC), m \log n\})$ iterations of a maximum flow subroutine. Although we are unable to prove this conjecture, we show in this section that $O(n \min\{\log(nC), m \log n\})$ iterations of a blocking flow subroutine suffice to compute a minimum-cost circulation. The maximum flow algorithm of Dinic [11], as well as the many maximum flow algorithms based on his approach, e.g. [35,38,46,53], find a maximum flow by solving $O(n)$ blocking flow problems. Thus our result shows that the time to solve a minimum-cost circulation problem is only $O(\min\{\log(nC), m \log n\})$ times greater than the time to find a maximum flow using Dinic's approach.

To describe our method we need some standard definitions. Let $G = (V, E)$ be a directed graph with a capacity function u and with two distinguished vertices, a *source* s and a *sink* t. A pseudoflow on G is a *flow* if every vertex except s and t has zero excess. Observe that if f is a flow, $e_f(t) = -e_f(s)$. The *value* of a flow f is $e_f(t)$. A flow is *maximum* if $e_f(t)$ is as large as possible. A flow f is *blocking* if any path from s to t in G contains at least

one saturated arc, i.e. an arc (v,w) such that $u_f(v,w) = 0$. A maximum flow is blocking, but not conversely. A directed graph is *acyclic* if it contains no cycles. It is *layered* if its vertices can be assigned integer layers in such a way that $layer\,(v) = layer\,(w) + 1$ for every arc (v,w). A layered graph is acyclic but not conversely.

The main step of our method is the computation of a blocking flow on an acyclic network. Many of the known blocking flow algorithms are stated for layered networks, but they apply to acyclic networks as well (e.g. [35,38,53]). The blocking flow algorithms of Galil [22] and Shiloach and Vishkin [46] can be extended from layered to acyclic networks without affecting their asymptotic time bounds (see [26]).

8.1 The Blocking Flow Method

We shall describe an implementation of *refine* that reduces ϵ by a factor of two by computing $O(n)$ blocking flows. This method, called the *blocking flow method*, consists of *refine* as described in Section 5 (Figure 4) with the original loop replaced as described in Figure 15. The new loop repeatedly modifies the current pseudoflow until it is a circulation. To modify the pseudoflow, the method first partitions the vertices of G into two sets S and $\overline{S}$, such that S contains all vertices reachable in the current admissible graph G_A from vertices of positive excess. Vertices in S have their prices increased by ϵ. Next, an auxiliary network N is constructed by adding to G_A a source s, a sink t, an arc (s,v) of capacity $e_f(v)$ for each vertex v with $e_f(v) > 0$, and an arc (v,t) of capacity $-e_f(v)$ for each vertex with $e_f(v) < 0$. An arc $(v,w) \in E_A$ has capacity $u_f(v,w)$ in N. A blocking flow b on N is found. Finally, the pseudoflow f is replaced by the pseudoflow $f'(v,w) = f(v,w) + b(v,w)$ for $(v,w) \in E$.

The correctness of the blocking flow method follows from the next lemma.

Lemma 8.1 *The set S computed in the inner loop contains only vertices v with $e_f(v) \geq 0$. At the beginning of an iteration of the loop, f is an ϵ-optimal pseudoflow with respect to the price function p. Increasing the prices of vertices in S preserves the ϵ-optimality of f. The admissible graph remains acyclic throughout the algorithm.*

Proof: The proof is by induction on the number of iterations of the inner loop. At the beginning of an iteration of the loop, there is no path in G_A from a vertex v with positive excess to a vertex w of negative excess; for the second and subsequent iterations this follows from the addition to f of a blocking flow during the previous iteration. Increasing the

```
procedure refine(ε, f, p);
    [initialization]
    ε ← ε/2;
    ∀(v, w) ∈ E do if c_p(v, w) < 0 then f(v, w) ← u(v, w);
    [loop]
    while f is not a circulation do begin
        S ← {v ∈ V | ∃u ∈ V such that e_f > 0 and v is reachable from u in G_A};
        ∀v ∈ S, p(v) ← p(v) + ε;
        let N be the network formed from G_A by adding a source s, a sink t,
            an arc (s, v) of capacity e_f(v) for each v ∈ V with e_f(v) > 0, and
            an arc (v, t) of capacity −e_f(v) for each v ∈ V with e_f(v) < 0;
        find a blocking flow b on N;
        ∀(v, w) ∈ E_A, f(v, w) ← f(v, w) + b(v, w);
    end;
    return(ε, f, p);
end.
```

Figure 15: The blocking *refine* subroutine.

price of every vertex in S by ϵ preserves the ϵ-optimality of f, since before the increase every residual arc (v, w) with $v \epsilon S, w \epsilon \overline{S}$ has $c_p(v, w) \geq 0$. The only time new arcs become admissible is as a result of such a price increase. This increase can add new admissible arcs from S to $\overline{S}$ but will make all arcs from $\overline{S}$ to S non-admissible. Thus the admissible graph remains acyclic. ∎

Remark: If we are given a price function p with respect to which some circulation is optimal, then we can find an optimal circulation f by means of a single maximum flow computation. The method resembles the blocking flow approach. We begin by constructing the pseudoflow f that saturates all negative-cost arcs and is zero on all other arcs. Then we construct the subgraph of the admissible graph containing all zero-cost arcs. To this graph, we add a source s, a sink t, an arc (s, v) of capacity $e_f(v)$ for each vertex v with $e_f(v) > 0$, and an arc (v, t) of capacity $-e_f(v)$ for each vertex v with $e_f(v) < 0$. Finally, we find a maximum flow f' in this network and use f' to augment f. (The optimality of p implies that f' will saturate all arcs leaving s and all arcs entering t.) ∎

8.2 Analysis of the Blocking Flow Method

The following lemma bounds the number of blocking flow computations. It is analogous to the bound of Bland and Jensen [7] on the number of maximum flow computations in a

scaling step of their algorithm.

Lemma 8.2 *The number of iterations of the inner loop in the blocking flow implementation of refine is at most* $3n$.

Proof: The inner loop never changes a vertex excess from nonnegative to negative or from nonpositive to positive. Thus all vertices with negative excess never have their price changed, and all vertices of positive excess have had their price increased by $i\epsilon$ after i iterations of the inner loop. The same proof as that of Lemma 5.8 shows that no vertex of positive excess can have a price change exceeding $3n\epsilon$. Hence there are at most $3n$ iterations. ∎

Theorem 8.3 *The blocking flow implementation of refine runs in* $O(nB(n,m))$ *time, giving an* $O(nB(n,m)\min\{\log(nC), m\log n\})$ *bound for finding a minimum-cost circulation, where* $B(n,m)$ *is the time needed to find a blocking flow in an acyclic network with* n *vertices and* m *arcs.*

Proof: Immediate from Lemma 8.2. ∎

Theorem 8.3 implies that the blocking flow approach yields implementations of *refine* with running times of $O(n^3)$, $O(n^{5/3}m^{2/3})$, and $O(nm\log n)$, based on the known bounds for computing a blocking flow in an acyclic network, namely $O(n^2)$ [35,38,46,53], $O(n^{2/3}m^{2/3})$ [22], and $O(m\log n)$ [47,48]. The algorithms of Shiloach and Vishkin [46] and Galil [22] must be extended to handle acyclic networks instead of layered networks, but such extension does not affect their asymptotic running times.

In the next section we shall develop an $O(m\log(n^2/m))$-time blocking flow algorithm, which in combination with Theorem 8.3 yields an $O(nm\log(n^2/m)\min\{\log(nC), m\log n\})$ time bound for computing a minimum-cost circulation. This matches the bound obtained in Section 7.

The blocking flow approach also yields efficient parallel and distributed minimum-cost circulation algorithms. These use the Shiloach-Vishkin parallel blocking flow algorithm, which runs in $O(n\log n)$ time using n processors and $O(nm)$ memory [46]. Vishkin (private communication, 1986) has reduced the memory requirement of the algorithm from the published bound of $O(nm)$ to $O(n^2)$. Previous results known to us are as follows. The Shiloach-Vishkin algorithm can be used in the cost-scaling methods of Röck [45] and Bland

and Jensen [7] to yield an $O(n^3(\log n)\log(nC))$-time n-processor parallel algorithm. The blocking flow approach improves this bound by a factor of n. Bertsekas [4] has proposed a "chaotic" algorithm for the minimum-cost circulation problem that converges in a finite number of steps in a distributed model.

Theorem 8.4 *The blocking flow approach yields a minimum-cost circulation algorithm running in $O(n^2(\log n)\min\{\log(nC), m\log n\})$ time using n processors and $O(n^2)$ memory. These bounds are valid for either the PRAM computation model* [14] *without concurrent writing or the DRAM model* [37].

The same method gives good bounds in distributed models of parallel computation [2,24]. In the statement of results below, Δ_v denotes the degree of vertex v; each vertex v has a processor p_v and communication links with all adjacent vertices.

Theorem 8.5 *The blocking flow approach yields a minimum-cost circulation algorithm for the synchronous distributed model running in $O(n^2\min\{\log(nC), m\log n\})$ time using $O(n^3\min\{\log(nC), m\log n\})$ total messages and $O(n\Delta_v)$ memory per processor p_v. In the asynchronous distributed model, the time increases to $O(n^2(\log n)\min\{\log(nC), m\log n\})$; the bounds on message complexity and memory remain the same.*

Proof: The bounds for the synchronous model follow from the observation that the Shiloach-Vishkin blocking flow algorithm runs on the distributed model in $O(n^2)$ time using $O(n^3)$ messages and $O(n\Delta_v)$ memory per processor. The bounds for the asynchronous model follow using the synchronization protocol of Awerbuch [2], which increases the time bound by a factor of $O(\log n)$. ∎

Remark: A disadvantage of the Shiloach-Vishkin algorithm is the greater than linear (i.e. $\Omega(n^2)$) memory requirement. The generic *refine* method has a parallel version that processes all active vertices in parallel (see [26]). It uses only a linear amount of memory. Unfortunately, the running time bound for this algorithm is worse than the running time bound for the Shiloach-Vishkin algorithm. It is still possible, however, that a variation of this algorithm runs in the same time as the Shiloach-Vishkin algorithm and uses a linear amount of memory. See [26] for a discussion of implications of this conjecture. Although the theoretical bound on the running time of the parallel version of the generic method is no better than the time bound of the best sequential version, it is still possible that the parallel version will perform well in practice.

8.3 Finding a Blocking Flow

In this section we propose a new sequential algorithm for finding a blocking flow in an acyclic network. The algorithm combines ideas in the wave algorithm [53] with the data structures discussed in Section 7. It runs in $O(m \log(n^2/m))$ time. Since the new algorithm closely resembles the dynamic tree implementation of *refine*, we shall merely sketch the ideas and omit the analysis.

Let $G = (V, E)$ be an acyclic graph with source s, sink t, and capacity function u. The algorithm maintains a preflow f on G, initially such that $f(s, v) = u(s, v)$ for $v \in V$ and $f(v, w) = 0$ for $v, w \in V - \{s\}$. Each vertex v is in one of two states, *blocked* or *unblocked*. An unblocked vertex can become blocked, but not conversely. Initially all vertices are unblocked. A vertex $v \notin \{s, t\}$ is *active* if $e_f(v) > 0$.

The algorithm makes use of the three operations *push*, *pull*, and *block*. A *push* operation applies to an arc (v, w) such that v is active, v and w are unblocked, and $u_f(v, w) > 0$. It consists of increasing $f(v, w)$ by $\min\{e_f(v), u_f(v, w)\}$. A *pull* operation applies to an arc (v, w) such that w is active and blocked and $f(v, w) > 0$. It consists of decreasing $f(v, w)$ by $\min\{e_f(w), f(v, w)\}$. A *block* operation applies to an unblocked vertex v such that, for every arc (v, w), either $u_f(v, w) = 0$ or w is blocked. It consists of making v blocked.

The *generic blocking flow algorithm* consists of performing *push*, *pull* and *block* operations in any order until no vertex is active, at which time the current preflow is a blocking flow. The proof of correctness resembles the proof of correctness of the generic version of *refine* presented in Section 5. The generic algorithm can be implemented in a straightforward way to run in $O(nm)$ time. This result is analogous to the $O(n^2 m)$ time bound for the generic version of *refine* derived in Section 6. In the analysis, vertex blockings play the same role as relabelings do in the analysis of *refine*. There are at most $n - 2$ vertex blockings.

We improve the algorithm by introducing the *discharge* operation. A discharge operation applies to an active vertex v. The operation applies a *push*, *block*, or *pull* operation to v, whichever applies, and repeats this until v becomes inactive. The *first-active* blocking flow algorithm, similar to the first-active version of *refine*, uses a list L to determine the order in which to process active vertices. List L contains all vertices of G, initially in topological order. The algorithm consists of repeating the following three steps until there are no active vertices: Select the first active vertex on L, say v. Apply a discharge operation to v. If the discharge has made v blocked, move v to the front of L. This algorithm runs in $O(n^2)$ time.

The key observation is that each vertex can be discharged at most once between vertex blockings.

We improve the algorithm further by implementing it using finger search trees to represent L and dynamic trees to represent the set of arcs eligible for pushing and pulling. The details and the analysis are similar to those of the tree version of *refine* presented in Section 7. The running time of this algorithm is $O(m \log(n^2/m))$. The factor of n savings over the time bound of the tree version of *refine* comes from the fact that there are $O(n)$ vertex blockings instead of $O(n^2)$ relabelings.

This algorithm in combination with Theorem 8.3 gives the following result:

Theorem 8.6 *The blocking flow approach to the minimum cost circulation problem has an* $O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$*-time implementation.*

9 Practical Improvements

In this section we discuss the practicality of our approach. Initial practical experience with scaling algorithms for the minimum-cost flow problem was disappointing. As a result, such algorithms were condemned as impractical in the literature, and simplex-based algorithms continued to be used in practice. This situation did not change even though new scaling algorithms were developed. Recently, however, Bland and Jensen conducted a study to compare their scaling algorithm with simplex-based codes [7]. They found their algorithm competitive, and concluded that the practicality of scaling algorithms should be investigated further. Their results are especially promising since they used the maximum flow algorithm of Malhotra, Promodh Kumar, and Maheshwari [38] as a subroutine. This algorithm, though simple to code, is likely to be slower in practice than other, newer maximum flow algorithms, e.g. [1,29]. The results of Bland and Jensen also agree with the results of Bateson [3], who compared Gabow's scaling algorithm for the assignment problem [19] with the Hungarian method.

We are extremely optimistic that our approach will yield highly practical algorithms. Our optimism is based on the work of Bland and Jensen discussed above, as well as the first author's experience [26] with sequential and parallel implementations of our maximum flow algorithm. (See also [42].) A careful experimental study is needed, however, before a claim of practicality can be made with certainty.

We suggest several heuristics that may improve the practical performance of the algo-

rithm. We expect that in practice the extra overhead needed to make the algorithm strongly polynomial is likely to make the algorithm slower, and should be omitted. Instead, we can avoid calling *refine* until the current error parameter is within a factor of two of minimum. That is, after halving ϵ but before calling *refine*, we do a shortest path computation as described in Section 3.1 to discover whether the current circulation is ϵ-optimal. (If not, the shortest path algorithm will find a negative-cost cycle.) If so, we halve ϵ again and repeat. If not, we call *refine*. With this method, the bound on the number of iterations is still $O(\log(nC))$, but some of the iterations require only a shortest path computation.

Another improvement is to delete arcs when their flows become fixed, as suggested by the lemma of Tardos discussed in Section 4.2. In order to delete arcs whose flows can be fixed, we modify the capacities after every iteration of the outer loop by subtracting the current circulation. This transforms the problem so that the current circulation is the zero circulation. We then delete every arc (v, w) whose reduced cost has absolute value at least $n\epsilon$. This modified algorithm needs only $O(m)$ extra time per iteration of the outer loop.

Our approach to the minimum-cost flow problem allows a great degree of flexibility. For example, the *refine* subroutine is used to reduce the error parameter by a factor of two, but it can instead be used to reduce the error parameter ϵ by any factor (at the cost of increasing the time bound by a related factor). Although fine-tuning this factor does not improve the asymptotic running time of the algorithm, it will affect practical performance. Also, as in the case of the maximum flow algorithm, alternative orderings of the update operations *push* and *relabel* may result in better performance.

Another potential way of improving the performance of the generic *refine* subroutine would be to use shortest path computations during the execution of the subroutine to update the price function. This improvement would be similar to using breadth-first search to update distance labels in our maximum flow algorithm [29]. One would have to be careful, however, to assure that these updates of the price function preserve the acyclicity of the admissible graph.

10 Remarks

In this section we discuss possible theoretical improvements and extensions of our results. The most obvious open question is whether the *refine* procedure can be modified so that it runs faster. The analogy with our maximum flow algorithm suggests the possibility of obtaining an $O(n^2 \log U + nm)$ bound for a suitable version of *refine*, based on the

$O(n^2 \log U + nm)$-time maximum flow algorithm of Ahuja and Orlin [1]. Unfortunately, the example in the appendix suggests that some new idea will be needed if this bound is to be obtained. Another question is whether the strongly polynomial framework of Section 4.2 can be improved to reduce the number of iterations of *refine* by a factor of m/n. Such an improvement would make our approach competitive with the algorithm of Galil and Tardos on both sparse and dense graphs.

Although all the versions of *refine* that we have considered here reduce the error parameter by a constant factor, one can consider other implementations of *refine* that reduce the error parameter by a factor that depends on the parameters of the input problem. In our recent paper [28], we describe an implementation of *refine* that reduces the error parameter by a factor of $(n-1)/n$. The resulting running time bounds are competitive with the bounds of this paper.

Our general approach is to scale costs. It would be interesting to see if there is a similar method that scales capacities, or that scales costs and capacities simultaneously. Such a double scaling approach might lead to improved complexity bounds.

Both the maximum flow problem and the shortest path problem are special cases of the minimum-cost flow problem. Our result gives a bound for the minimum-cost flow problem only a logarithmic factor larger than the sum of the best known bounds for these subproblems. A nice extension would be a reduction of the minimum-cost flow problem to the solution of a logarithmic number of maximum flow and shortest path problems.

A more general question is whether the scaling approach can be used to efficiently solve other network optimization problems. Natural candidates include problems involving flows with gains and multicommodity flow problems.

If all edges have unit capacities, all pushes performed by the generic algorithm are saturating, and the generic minimum-cost circulation algorithm runs in $O(nm \log(nC))$ time. Using the blocking flow approach presented in Section 8, Harold Gabow and the second author [20,21] have obtained improved algorithms for solving a minimum-cost flow problem on a network with small capacities. Such a problem can be solved in $O(n^{2/3}U^{1/3}m \log(n^2/m) \log(nC))$ time, assuming that the graph contains no multiple arcs. If $U = O(1)$, the bound becomes $O(\min\{m^{1/2}, n^{2/3}\}m \log(nC))$. The approach also gives an algorithm for the assignment problem (bipartite weighted matching) running in $O(n^{1/2}m \log(nC))$ time. The method extends to the nonbipartite weighted matching problem, giving a bound of $O((n\alpha(m,n) \log n)^{1/2}m \log(nC))$, where $\alpha(m,n)$ is a functional in-

verse of Ackermann's function. In this case the algorithm becomes much more complicated, because sophisticated data structures are needed to implement it efficiently. The best previously bound is $O(n^{3/4}m \log C)$ for both bipartite and nonbipartite weighted matching [18,19]. The new bounds for weighted matching are close to the best known bound, $O(n^{1/2}m)$, for maximum cardinality matching [31,40]. The new nonbipartite weighted matching algorithm also specializes in the case of maximum cardinality matching to yield an $O(n^{1/2}m)$-time algorithm that is somewhat simpler than that of Micali and Vazirani [40].

Appendix. A Hard Network for the First-In, First-Out Algorithm

The first-in, first-out (FIFO) implementation of *refine*, described in Section 6.2, can make $\Omega(n^3)$ passes over its queue. A network on which this can happen is shown in Figure 16. The network consists of two paths, $x_n, x_{n-1}, \ldots, x_1$ and $y_1, y_2, \ldots, y_n$, and three extra arcs, (x_1, y_1), (y_n, x_n), and (x_n, x_1). The arcs have the following residual capacities and costs, where h is a sufficiently large integer ($h = n^2$ will do).

$$\begin{array}{lll} u_f(x_i, x_{i-1}) = h, & c(x_i, x_{i-1}) = 1 & \text{for } 2 \leq i \leq n \\ u_f(y_i, y_{i+1}) = h, & c(y_i, y_{i+1}) = -2 & \text{for } 1 \leq i \leq n-1 \\ u_f(x_1, y_1) = h-1, & c(x_1, y_1) = 0 & \\ u_f(y_n, x_n) = h, & c(y_n, x_n) = -2 & \\ u_f(x_n, x_1) = h, & c(x_n, x_1) = 1 & \end{array}$$

All reverse arcs (e.g. (x_i, x_{i+1})) have a residual capacity of zero.

Suppose *refine* is given this network with the zero flow, the zero price function, and $\epsilon = 1$. First, each arc (y_i, y_{i+1}) is saturated, as is the arc (y_n, x_n). This results in an excess of h at vertex x_n and an excess of $-h$ at vertex y_1. Next (with unfavorable tiebreaking) the excess of h moves from x_n along the path to x_1, in the process increasing the potential of each of the vertices $x_2, x_3, \ldots, x_n$ to two. After the potential of x_1 is raised to one, $h-1$ of the excess flows to y_1, where it cancels all but one unit of negative excess. The potential of x_1 then is increased to two. The current state is shown in Figure 17; the only remaining positive excess, of one unit, is on x_1.

Now the unit of excess moves repeatedly around the cycle $x_1, x_2, \ldots, x_n, x_1$. The first time this happens, the price of x_n goes up by two. The second time, the price of x_{n-1}

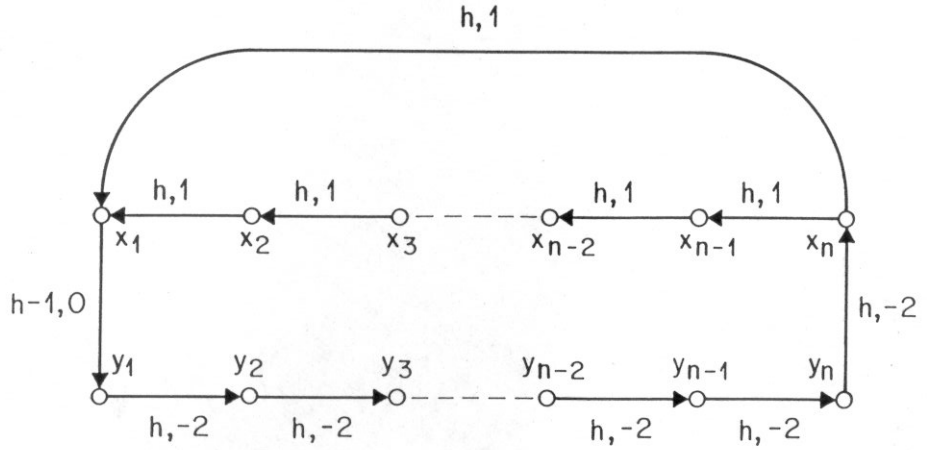


Figure 16: A hard network for the first-in, first-out algorithm.

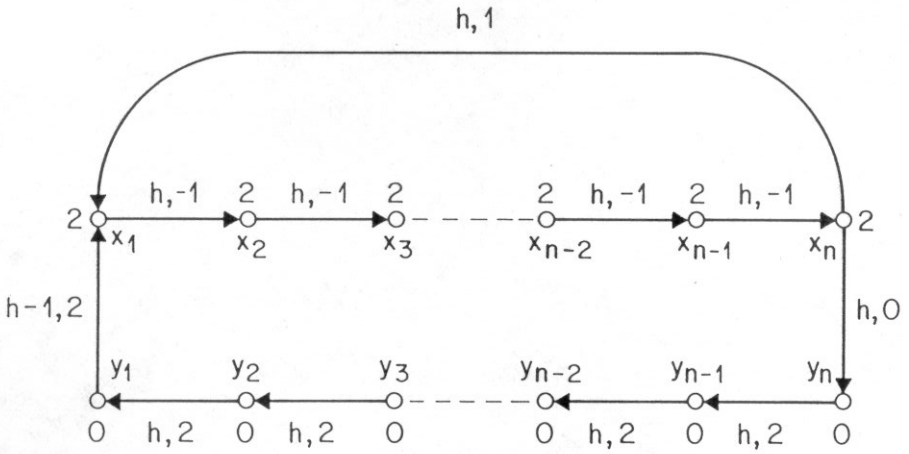


Figure 17: The residual graph after $h-1$ units of excess are canceled. The reduced costs and vertex prices are shown. Vertex x_1 has one unit of excess; vertex y_1, one negative unit of excess.

goes up by two. After n iterations, all of $x_1, x_2, \ldots, x_n$ have increased by two in price. Then the process repeats. The unit of excess will take occasional excursions along the path $x_n, y_n, y_{n-1}, \ldots, y_1$, but it will return to x_n and continue going around the cycle. Finally, after $\Omega(n^2)$ traversals of the cycle, the unit of excess will succeed in traversing the entire path $x_n, y_n, y_{n-1}, \ldots, y_1$, and *refine* will terminate. Since there are n passes over the queue of the FIFO algorithm for each traversal of the cycle, the total number of FIFO passes is $\Omega(n^3)$.

Acknowlegements

We thank Harold Gabow, Charles Leiserson, Serge Plotkin, Ronald Rivest, and David Shmoys for many extremely helpful suggestions, ideas and comments. We also thank Robert Bland, who proposed the interpretation of the generic algorithm in terms of kilter diagrams.

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., 1987.

[2] B. Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32:804–823, 1985.

[3] C. A. Bateson. Performance comparison of two algorithms for weighted bipartite matchings. 1985. M.S. thesis, Department of Computer Science, University of Colorado.

[4] D. P. Bertsekas. *Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems.* Technical Report LIDS-P-1986, Lab. for Decision Systems, M.I.T., September 1986. (Revised November, 1986).

[5] D. P. Bertsekas and J. Eckstein. Distributed asynchronous relaxation methods for linear network flow problems. In *Proc. IFAC '87, Munich, Germany*, 1987. (to appear).

[6] D. P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Oper. Res.*, (to appear).

[7] R. G. Bland and D. L. Jensen. *On the Computational Behavior of a Polynomial-Time Network Flow Algorithm.* Technical Report 661, School of Operations Research and Industrial Engineering, Cornell University, 1985.

[8] R. G. Busacker and P. J. Gowen. *A Procedure for Determinimg a Family of Minimal-Cost Network Flow Patterns.* Technical Report 15, O.R.O., 1961.

[9] R. G. Busacker and T. L. Saaty. *Finite Graphs and Networks: An Introduction with Applications.* McGraw-Hill, New York, NY., 1965.

[10] R. B. Dial. Algorithm 360: shortest path forest with topological ordering. *Comm. ACM*, 12:632–633, 1969.

[11] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[12] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

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

[14] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pages 338–346, 1984. (To appear in J. Assoc. Comput. Mach.).

[16] S. Fujishige. A capacity-rounding algorithm for the minimum-cost circulation problem: a dual framework of the tardos algorithm. *Math. Prog.*, 35:298–308, 1986.

[17] D. R. Fulkerson. An out-of-kilter method for minimal cost flow problems. *SIAM J. Appl. Math*, 9:18–27, 1961.

[18] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 90–100, 1985.

[19] H. N. Gabow. Scaling algorithms for network problems. *J. of Comp. and Sys. Sci.*, 31:148–168, 1985.

[20] H. N. Gabow and R. E. Tarjan. A faster scaling algorithm for general weighted matching. (To appear).

[21] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. (To appear).

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

[23] Z. Galil and E. Tardos. An $O(n^2 \log n(m + n \log n))$ min-cost flow algorithm. In *Proc. 27th IEEE Symp. of Foundations of Computer Science*, pages 1–9, 1986.

[24] R. G. Gallager, P. A. Humblet, and P. M Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.

[25] S. I. Gass. *Linear Programming: Methods and Applications.* McGraw-Hill, 1958.

[26] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[27] A. V. Goldberg. *A New Max-Flow Algorithm.* Technical Report MIT/LCS/ TM-291, Laboratory for Computer Science, M.I.T., 1985.

[28] A. V. Goldberg and R. E. Tarjan. *Finding Minimum-Cost Circulations by Canceling Negative Cycles.* Technical Report MIT/LCS/TM-333, Laboratory for Computer Science, M.I.T., 1987. Also available as Technical Report CS-TR 107-87, Department of Computer Science, Princeton University.

[29] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 136–146, 1986. (To appear in J. Assoc. Comput. Mach.).

[30] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 7–18, 1987.

[31] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[32] P. A. Jensen and J. W. Barnes. *Network Flow Programming.* J. Wiley & Sons, 1980.

[33] W. S. Jewell. *Optimal Flow through Networks.* Technical Report 8, M.I.T., 1958.

[34] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23:309–311, 1978.

[35] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dok.*, 15:434–437, 1974.

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

[37] C. Leiserson and B. Maggs. Communication-efficient parallel graph algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.

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

[39] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, Berlin, 1984.

[40] S. Micali and V. V. Vazirani. An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs. In *Proc. 21st IEEE Symp. on Found. of Comp. Sci.*, pages 17–27, 1980.

[41] G. J. Minty. Monotone networks. *Proc. Roy. Soc. London*, A(257):194–212, 1960.

[42] A. T. Ogielski. Integer optimization and zero-temperature fixed point in Ising random-field systems. *Physical Review Lett.*, 57:1251–1254, 1986.

[43] J. B. Orlin. *Genuinely Polynomial Simplex and Non-Simplex Algorithms for the Minimum Cost Flow Problem.* Technical Report No. 1615-84, Sloan School of Management, MIT, December 1984.

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

[45] H. Röck. Scaling techniques for minimal cost network flows. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 181–191, Carl Hansen, Münich, 1980.

[46] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[47] D. D. Sleator. *An $O(nm \log n)$ Algorithm for Maximum Network Flow.* Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, Stanford, CA, 1980.

[48] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[49] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32:652–686, 1985.

[50] E. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[51] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Math.*, 6:306–318, 1985.

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

[53] R. E. Tarjan. A simple version of Karzanov's blocking flow algorithm. *Operations Research Letters*, 2:265–268, 1984.

[54] R.E. Tarjan and C.J. Van Wyk. An $O(nlog \log n)$-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, (to appear).

[55] R. A. Wagner. A shortest path algorithm for edge-sparse graphs. *JACM*, 23:50–57, 1976.