SOLVING MINIMUM-COST FLOW PROBLEMS BY SUCCESSIVE APPROXIMATION

Andrew V. Goldberg
Robert E. Tarjan

CS-TR-081-87

February 1987

Solving Minimum-Cost Flow Problems by Successive Approximation

*Andrew V. Goldberg**
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA 02139

Robert E. Tarjan†
Department of Computer Science
Princeton University
Princeton, NJ 08544
and
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

We introduce a framework for solving minimum-cost flow problems. Our approach measures the quality of a solution by the amount that the complementary slackness conditions are violated. We show how to extend techniques developed for the maximum flow problem to improve the quality of a solution. This framework allows us to achieve $O(\min(n^3, n^{5/3}m^{2/3}, nm\log n)\log(nC))$ running time.

1 Introduction

The minimum-cost flow problem is that of finding a feasible flow of minimum cost in a network with capacity constraints and edge costs. Extensive discussion of the problem and its applications appear in the books of Ford and Fulkerson [8], Lawler [22], Papadimitriou and Steiglitz [28], and Tarjan [35].

All known polynomial-time algorithms for the problem are based on the idea of *scaling*.[1] This idea was introduced by Edmonds and Karp [7], who used it to design the first polynomial-time algorithm for the problem. The Edmonds-Karp algorithm scales capacities. The first algorithm that scales costs is due to Röck [29].

Table 1 summarizes polynomial-time algorithms for the minimum-cost flow problem. The running times of the algorithms are given in terms of the number of vertices n, the number of edges m, the maximum absolute value of capacities U, and the maximum absolute value of costs C. When U (or C) appears in a bound, the capacities (or the costs) are assumed to be integer. When stating the running times, we assume the best time bounds known for the shortest path and maximum flow subroutines used by some of these algorithms: $O(m + n\log n)$ for the shortest path subroutine [10] and $O(nm\log(n^2/m))$ for the maximum flow subroutine [20]. Algorithms 1, 2, 5, 6, and 8 use capacity scaling and algorithms 3, 4, 7, and 9 use cost scaling. The algorithms 4, 5, 6, and 8 are strongly polynomial, i.e., their running time does not depend on U or C.

Since the running times of the algorithms in Table 1 are expressed in terms of different parameters, the algorithms cannot be compared directly. The previous algorithms 1-8 can be divided into the three comparable groups and ranked as follows. Algorithms 1 and 2 give the best capacity-dependent bound, algorithms 3 and 7 give the best cost-dependent bound, and algorithm 8 gives the best strongly polynomial bound. For most applications, however, one can assume that $U = n^{O(1)}$ and $C = n^{O(1)}$ [13]. Under these assumptions, the bound of $O(m(\log n)(m + n\log n))$ achieved by the capacity-scaling algorithms 1 and 2 is the best among previous algorithms.

In this paper we present a general approach to the minimum-cost flow problem. The approach combines methods for solving the maximum flow problem with

*Supported by a Fannie and John Hertz Foundation Fellowship and by the Advanced Research Projects Agency of the Department of Defense under the contract N00014-80-C-0622.
This paper constitutes a part of the author's research work towards his Ph.D. degree.

†Partially supported by the National Science Foundation under grant DCR-8605962.

[1] For applications of scaling to other network problems, see [13].

#	Date	Discoverer	Running Time	References
1	1972	Edmonds and Karp	$O(m(\log U)(m + n\log n))$	[7]
2	1980	Röck	$O(m(\log U)(m + n\log n))$	[29]
3	1980	Röck	$O(n(\log C)(nm\log(n^2/m)))$	[29]
4	1984	Tardos	$O(m^4)$	[34]
5	1984	Orlin	$O(m^2(\log n)(m + n\log n))$	[27]
6	1985	Fujishige	$O(m^2(\log n)(m + n\log n))$	[11]
7	1985	Bland and Jensen	$O(n(\log C)(nm\log(n^2/m)))$	[5]
8	1986	Galil and Tardos	$O(n^2(\log n)(m + n\log n))$	[16]
9	1987	Goldberg and Tarjan	$O(\min(nm\log n, n^{5/3}m^{2/3}, n^3)\log(nC))$	[19]

Table 1: Polynomial-time algorithms for the minimum-cost flow problem. Algorithm 9 is presented in this paper.

successive approximation techniques. We use our approach to construct algorithms for the problem with upper bounds of $O(nm\log(n)\log(nC))$, $O(n^{5/3}m^{2/3}\log(nC))$, and $O(n^3\log(nC))$, which significantly improves the best previous cost-dependent bound achieved by algorithms 3 and 7 in the table, as well as the best bound under the assumptions $U = n^{O(1)}$ and $C = n^{O(1)}$ discussed above. The algorithm gives the best bound on the complexity of the problem for $C = o(n^n)$ and $C = o(U^n/n)$. Our approach combines the ideas of Röck used to develop algorithm 3, of Bland and Jensen [5] used to develop algorithm 7, and of Bertsekas [3] used to develop an exponential-time algorithm for the problem. Our techniques are more powerful, however, and lead to more efficient algorithms.

The new algorithm starts by finding an approximate solution and then iteratively improves the current solution, each time doubling the quality of approximation by halving the error parameter ϵ. The inner loop subroutine that improves the approximation is based on generalizations of techniques for solving the maximum flow problem; one version of the inner loop is almost identical to the algorithm of Bertsekas [3]. When the error parameter is small enough, the current solution is optimal, and the algorithm terminates. To measure the quality of a solution, we use the notion of ϵ-optimality, which is related to the classical technique of perturbing a linear programming problem to avoid degeneracy (see, for example, [18]). The notion of ϵ-optimality is motivated by a relaxation of the complementary slackness conditions [4,34]. The termination condition used in our algorithm is due to Bertsekas [3].

The successive approximation approach can be viewed as a generalization of the cost-scaling approaches of Röck [29] and of Bland and Jensen [5]. However, our use of true costs throughout the algorithm simplifies its analysis and implementation.

A more extensive discussion of the results presented in this paper appears in [19].

2 Definitions and Notation

In this section we define the *minimum-cost circulation problem* and introduce the notation and terminology used throughout the paper. The minimum-cost circulation problem is a generalization of the maximum flow problem. As a special case of the linear programming problem, it is usually defined in linear programming terms. Although we use several theorems that have their roots in the theory of linear programming, most 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 the books and papers cited in the introduction.

A *circulation network* is a directed graph $G = (V, E)$ with upper and lower capacity bounds and costs on edges. We denote the size of V by n and the size of E by m, and we assume that $m \geq n - 1$. We call an unordered pair $\{v, w\}$ such that $(v, w) \in E$ or $(w, v) \in E$ an *undirected edge* of G. For notational convenience, we extend the capacity functions and the cost function to all pairs of vertices. Let R denote the set of real numbers. The capacity bounds are given by functions $u : V \times V \to R$ and $l : V \times V \to R$ with the following *consistency* and *antisymmetry* constraints for all $(v, w) \in V \times V$:

$$l(v, w) \leq u(v, w), \tag{1}$$

$$u(v, w) = -l(w, v). \tag{2}$$

To extend the capacity functions to all pairs of vertices, we define

$$l(v, w) = u(v, w) = 0 \tag{3}$$

for all (v, w) such that $(v, w) \notin E$ and $(w, v) \notin E$.

A *pseudoflow*[2] is a function $f : V \times V \to R$ satisfying the following *capacity* and *flow antisymmetry* constraints for all $(v, w) \in V \times V$:

[2]The concept of a pseudoflow is different from the preflow concept of Karzanov [21] in that the flow conservation constraints are completely dropped.

$$l(v,w) \leq f(v,w) \leq u(v,w), \tag{4}$$

$$f(v,w) = -f(w,v) \tag{5}$$

A *circulation* is a pseudoflow that satisfies conservation constraints

$$\sum_{w \in V} f(v,w) = 0 \tag{6}$$

for all $v \in V$.

We assume that the costs of edges are given by a cost function $c : V \times V \rightarrow R$ that satisfies the following *cost antisymmetry* constraints for all $(v,w) \in V \times V$:

$$c(v,w) = -c(w,v). \tag{7}$$

We extend the cost function to pairs of vertices by defining $c(v,w) = 0$ for all (v,w) such that $(v,w) \notin E$ and $(w,v) \notin E$. The *cost of a circulation* f is given by the following expression:

$$\frac{1}{2} \sum_{(v,w) \in V \times V} c(v,w) f(v,w). \tag{8}$$

(The factor of $1/2$ appears because the sum counts the cost of the flow between each pair of vertices twice.) The *minimum-cost circulation* problem is to find a circulation of minimum cost (an *optimal circulation*).

Remark: We refer to equations (2), (5), and (7) as the *antisymmetry constraints*. One should think of a positive and a negative direction for each undirected edge of G, with the capacity and cost constraints given for the positive direction and derived for the negative direction using the antisymmetry constraints.

Another important concept that we shall use is that of *vertex prices*. Consider a vertex v and a real number x. Suppose that we add x to prices of all edges going into v and subtract x from prices of all edges going out of v. Because of the conservation constraints at v, the cost of f is not changed by this transformation (the cost of each unit of flow going into v increases by x, and the cost of each unit of flow going out of v decreases by x). Therefore the transformed problem is equivalent to the original one. A *price function* p is a function from V to R; the price of a vertex v is $p(v)$. The *reduced cost function* c_p is defined by $c_p(v,w) = c(v,w) - p(v) + p(w)$. In the linear programming formulation of the problem, prices are the dual variables.

Given a pseudoflow f, we define the *residual capacity function* $r_f : V \times V \rightarrow R$ by $r_f(v,w) = u(v,w) - f(v,w)$. The *residual graph* $G_f = (V, E_f)$ is the directed graph with vertex set V containing all edges with positive residual capacity: $E_f = \{(v,w) | r_f(v,w) > 0\}$. The *balance* $b_f(v)$ of a vertex v is the difference between the incoming and outgoing flows, i.e. the function $b_f : V \times V \rightarrow R$ defined by $b_f(v) = \sum_{u \in V} f(u,v)$. If f is a circulation, then $b_f(v) = 0$ for all v. Given a pseudoflow f, we say that a vertex v is *active* if $b_f(v) > 0$. Note that $\sum_{v \in V} b(v) = 0$ for any pseudoflow, so a pseudoflow is a circulation if and only if there are no active vertices.

We also need the following standard definitions. An *augmenting path (cycle)* is a simple path (cycle) in G_f. The *cost of a path (cycle)* is the sum of the costs of all edges on the path (cycle).

3 Optimality and Approximate Optimality

In this section we define the notion of an ϵ-optimal pseudoflow and show that for $\epsilon < 1/n$, an ϵ-optimal circulation is optimal.

The following theorem of Ford and Fulkerson [8] provides an optimality criterion for a circulation.

Theorem 3.1 *A circulation f is of minimum cost if and only if there exists a price function p such that $\forall (v,w) \in V \times V$,*

$$c_p(v,w) < 0 \Rightarrow f(v,w) = u(v,w). \tag{9}$$

The optimality conditions (9) are called *complementary slackness* conditions, and an edge (v,w) satisfying these conditions is said to be *in kilter*. We use the term kilter because of the relationship between our algorithm and the out-of-kilter method [12,25].

The following theorem [8] gives an optimality criterion that does not involve prices.

Theorem 3.2 *A circulation f is optimal if and only if the residual graph G_f contains no cycles of negative cost.*

To define ϵ-optimality, we use the notion of ϵ-relaxation of the complementary slackness conditions [4,34]. Given $\epsilon \geq 0$, we say that a pseudoflow f is ϵ-optimal with respect to a price function p if the following relaxations of the complementary slackness conditions hold: $\forall (v,w) \in V \times V$,

$$c_p(v,w) < -\epsilon \Rightarrow f(v,w) = u(v,w). \tag{10}$$

A pseudoflow f is *ϵ-optimal* if there exists a price function p with respect to which the pseudoflow f is ϵ-optimal. The antisymmetry constraints (2), (5), and (7) imply that if a circulation f is ϵ-optimal, then $c_p(v,w) > \epsilon \Rightarrow f(v,w) = l(v,w)$.

The following simple fact is used extensively in our presentation.

Lemma 3.3 *Suppose that pseudoflow f is ϵ-optimal with respect to a price function p. Then for any residual edge (v,w), we have $c_p(v,w) \geq -\epsilon$. (I.e., the cost of a residual edge cannot be too small.)*

The following theorem of Bertsekas [3] shows that if the costs are integers and ϵ is small enough, then an ϵ-optimal circulation is optimal.

Procedure *Min-Cost*(V, E, l, u, c);

$\epsilon \leftarrow \max_{(v,w)\in E} |c(v,w)|$;
$f_\epsilon \leftarrow$ feasible circulation;
$\forall v,\ p_\epsilon(v) \leftarrow 0$;
while $\epsilon \geq 1/n$ **do**
    $\epsilon \leftarrow \epsilon/2$;
    $\langle f_\epsilon,\ p_\epsilon \rangle \leftarrow$ *Refine*$(f_{2\epsilon}, p_{2\epsilon}, \epsilon)$;
end;
return(f_ϵ);

end.

Figure 1: The minimum-cost flow algorithm.

Theorem 3.4 *Assume that all edge costs are integers. Then for any $0 \leq \epsilon < 1/n$, an ϵ-optimal circulation f is optimal.*

Proof: Consider a cycle in G_f. By Lemma 3.3, the ϵ-optimality of f implies that the cost of the cycle is at least $-n\epsilon$, which is greater than -1. Since the costs are integers, the cost of the cycle must be at least 0. The theorem then follows from the optimality criterion given by Theorem 3.2. ∎

4 High-Level Description of the Algorithm

Theorem 3.4 suggests the algorithm *Min-Cost* summarized in Figure 1. First, the algorithm finds a circulation (using a maximum flow algorithm) and sets the prices of all vertices to zero. The resulting circulation is C-optimal (recall that C is the largest absolute value of an edge cost). Then, the algorithm iteratively improves the approximation (using the *Refine* subroutine), until the error becomes less than $1/n$. At this point, the current solution is optimal.

Remark: The algorithm need not use a maximum flow subroutine in the initialization stage. It can start with any pseudoflow. If the problem is infeasible, we can discover this fact during the first execution of *Refine* because the increase in vertex prices will be greater than that allowed by the analysis below. We use the maximum flow subroutine only to be able to assume, without loss of generality, that the input problem is feasible, so that we do not have to worry about feasibility in our presentation.

Theorem 4.1 *Let $D(n, m)$ be the running time of the Refine subroutine. Then the minimum-cost flow algorithm runs in $O(D(n, m)\log(nC))$ time and returns a minimum-cost flow.*

Proof: Immediate from Theorem 3.4 and the above discussion. ∎

Procedure *Refine*(f, p, ϵ);

⟨⟨ initialization⟩⟩
$\forall (v, w) \in V \times V$ **do begin**
    if $c_p(v,w) > 0$ **then** $f(v,w) \leftarrow l(v,w)$;
    if $c_p(v,w) < 0$ **then** $f(v,w) \leftarrow u(v,w)$;
end;
⟨⟨ loop⟩⟩
while $\exists$ a basic operation that applies
    select a basic operation and apply it;
return(f, p);

end.

Figure 2: The generic Refine subroutine.

The inputs to the *Refine* subroutine are a circulation $f_{2\epsilon}$, a price function $p_{2\epsilon}$, and an error parameter ϵ, such that $f_{2\epsilon}$ is 2ϵ-optimal with respect to $p_{2\epsilon}$. The outputs of the subroutine are a circulation f_ϵ and a price function p_ϵ such that f_ϵ is ϵ-optimal with respect to p_ϵ. The subroutine begins by setting the flow through every out-of-kilter edge to the upper or lower bound to bring the edge into kilter. The resulting pseudoflow is ϵ-optimal (in fact, 0-optimal), but the conservation constraints (6) may be violated. Then, the pseudoflow is transformed into a circulation by applying a sequence of operations that preserve ϵ-optimality. This transformation uses generalizations of maximum flow techniques.

The cost scaling algorithms of Röck and of Bland and Jensen use a maximum flow algorithm in the inner loop. Since the *Refine* subroutine is a generalization of a maximum flow algorithm, our minimum-cost flow algorithm is similar to the cost scaling algorithms. The cost-scaling algorithms, however, halve the error in $O(n)$ iterations of the inner loop, whereas our algorithm halves the the error in a single iteration.

5 Generic Refine Subroutine

The generic *Refine* subroutine is a generalization of the generic maximum-flow algorithm of Goldberg and Tarjan [20]. An implementation of the generic subroutine using the *discharge* operation as described in Section 7 is almost identical to the earlier minimum-cost flow algorithm of Bertsekas [3], but we use the subroutine in a different way. In our algorithm the subroutine is used repetitively to reduce the approximation parameter ϵ by a factor of two each time. In the algorithm of Bertsekas, the value of ϵ is set to $1/(n+1)$ at initialization, so the algorithm terminates in one iteration — but may take exponential time.

The generic *Refine* subroutine is described in Figure 2. The subroutine forces all edges in kilter and then transforms

Push(v, w).
Applicability: v is active, $r_f(v, w) > 0$, **and** $c_p(v, w) < 0$.
Action: Send $\delta = \min(b_f(v), r_f(v, w))$ units of flow from v to w.

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

Figure 3: Push and update-price operations.

the resulting pseudoflow into an ϵ-optimal circulation. The *basic operations* used to manipulate the pseudoflow and the prices are *push* and *update-price*. These operations are described in Figure 3. A *push* through an edge (v, w) increases $f(v, w)$ and $b_f(w)$ by $\delta = \min(b_f(v), r_f(v, w))$ and decreases $f(w, v)$ and $b_f(v)$ by the same amount. For the purpose of the analysis, we distinguish between saturating and nonsaturating pushes. A push is *saturating* if $r_f(v, w) = 0$ after the push and *nonsaturating* otherwise. An *update-price* operation sets the price of a vertex v to the highest value allowed by the ϵ-optimality constraints.

The following lemmas give important properties of the *push* and *update-price* operations.

Lemma 5.1 *A pushing operation preserves the ϵ-optimality of a pseudoflow.*

Proof: Obvious. ∎

Lemma 5.2 *Suppose f is an ϵ-optimal pseudoflow with respect to a price function p and a price update operation 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 resulting price function p'.*

Proof: First we prove that the price of v increases by at least ϵ. The price updating operation changes the price of the vertex v from $p(v)$ to $p'(v)$ and does not affect prices of other vertices. Since a price updating operation is applicable to v, we have $c(v, w) - p(v) + p(w) \geq 0$ for all residual edges (v, w). Therefore, we have $c(v, w) + p(w) + \epsilon \geq p(v)$ for all edges $(v, w) \in E_f$, so the new price $p'(v) \geq p(v) + \epsilon$ by definition of the price updating operation.

To prove the second claim of the lemma, consider an undirected edge $\{v, w\}$. ¿From the first part of the prove we know that $c_{p'}(v, w) < c_p(v, w)$ and by antisymmetry $c_{p'}(w, v) > c_p(w, v)$, so the constraint 10 for (w, v) remains satisfied. For (v, w), either $f(v, w) = u(v, w)$ and the constraint 10 does not apply, or $(v, w) \in E_f$ and therefore $c(v, w) - p'(v) + p'(w) = c(v, w) - p'(v) + p(w) \geq -\epsilon$ by the definition of the price update operation. ∎

Lemma 5.3 *If a pseudoflow f is ϵ-optimal with respect to a price function p and v is an active vertex, then either a pushing or a price updating operation is applicable to v.*

Proof: Obvious. ∎

The generic version of *Refine*, described in Figure 2, repetitively performs an applicable basic operation on the current pseudoflow f and price function p until no basic operation applies. We shall prove that the generic version of *Refine* is correct if it terminates, and then we shall prove termination.

Theorem 5.4 *If the generic Refine subroutine terminates, then the pseudoflow f output by the subroutine is an ϵ-optimal circulation.*

Proof: The subroutine terminates when there are no vertices with positive balance and therefore no vertices with negative balance, so f is a circulation. The ϵ-optimality of f follows from Lemmas 5.1 and 5.2 and from the fact that the pseudoflow computed during the initialization step of *Refine* is ϵ-optimal (in fact, 0-optimal). ∎

6 Analysis of the Generic Subroutine

In this section we analyze the generic *Refine* subroutine. The analysis is similar to the analysis of the generic maximum flow algorithm. We start by bounding the amount of increase in vertex prices during an execution of the generic subroutine.

Lemma 6.1 *Let f be a pseudoflow and let f' be a circulation. Then for any vertex v with a positive balance $b_f(v)$, there exists a vertex w with a negative balance $b_f(w)$ and a sequence of vertices $u_1, \ldots, u_{l-1}$ such that $(v, u_1, \ldots, u_{l-1}, w)$ is a simple path in G_f and $(w, u_{l-1}, \ldots, u_1, v)$ is a simple path in $G_{f'}$.*

Proof: Fix a vertex v with a positive balance. Let $G_+ = (V, E_+)$ where $E_+ = \{(i, j) | f'(i, j) > f(i, j)\}$ and let $G_- = (V, E_-)$ where $E_- = \{(i, j) | f'(i, j) < f(i, j)\}$. This definition implies that $E_+ \subseteq E_f$, because for any edge $(i, j) \in E_f$ we have $f(i, j) < f'(i, j) \leq u(i, j)$. Similarly, we have $E_- \subseteq E_{f'}$. Furthermore, if (i, j) is an edge in G_+, then (j, i) is an edge in G_- by antisymmetry. Therefore it is enough to show that if $b_f(v) > 0$, then there is a simple path $(v, u_1, \ldots, u_{l-1}, w)$ in G_+ such that $b_f(w) < 0$.

Assume by the way of contradiction that such a path does not exist, i.e. all vertices reachable from v in G_+ have nonnegative balance. Let S be the set of all vertices reachable from v in G_+ and let $\overline{S} = V - S$. The vertices in S have nonnegative balance, and since $v \in S$ and $b_f(v) > 0$, we have $\sum_{j\in S} b(j) > 0$. Also, for every pair of vertices $(i, j) \in S \times \overline{S}$, we have $f(i, j) \geq f'(i, j)$, for otherwise we would have $j \in S$.

Now we obtain a contradiction as follows:

$$\begin{aligned} 0 &= \sum_{(i,j)\in S\times\overline{S}} f'(i,j) \\ &\leq \sum_{(i,j)\in S\times\overline{S}} f(i,j) \\ &= \sum_{(i,j)\in S\times\overline{S}} f(i,j) + \sum_{(i,j)\in S\times S} f(i,j) \\ &= \sum_{(i,j)\in S\times V} f(i,j) \\ &= -\sum_{j\in S} b_f(j) \\ &< 0. \end{aligned}$$

∎

The following key lemma bounds the amount of increase in prices during an execution of the subroutine. This lemma is similar to the lemma of Bland and Jensen [5] that bounds the number of maximum flow computations in a scaling step.

Lemma 6.2 *The price of any vertex v increases by at most $3n\epsilon$ during an execution of Refine.*

Proof: Since prices of vertices with zero or negative balance cannot change, it is enough to prove the lemma for a vertex v that has a positive balance at some time t during the execution of *Refine*. Let f and p be the ϵ-optimal pseudoflow and price function at this point. Recall that $f_{2\epsilon}$ and $p_{2\epsilon}$ are the 2ϵ-optimal circulation and the price function at the beginning of the execution of *Refine*. Let $v, u_1, \ldots, u_{l-1}, w$ be a sequence of vertices satisfying Lemma 6.1. Applying the 2ϵ-optimality conditions to the edges on the path $P_{f_\epsilon} = (w, u_{l-1}, \ldots, u_1, v)$ in G_{f_ϵ}, we obtain

$$p_\epsilon(w) \leq p_\epsilon(v) + 2l\epsilon + \sum_{(i,j) \text{ on } P_{f_\epsilon}} c(i,j). \tag{11}$$

Applying the ϵ-optimality conditions to the edges on the path $P_f = (v, u_1, \ldots, u_{l-1}, w)$ in G_f, we obtain

$$\begin{aligned} p(v) &\leq p(w) + l\epsilon + \sum_{(j,i) \text{ on } P_f} c(j,i) \\ &= p(w) + l\epsilon - \sum_{(i,j) \text{ on } P_{f_\epsilon}} c(i,j). \end{aligned} \tag{12}$$

Vertex w has had a negative balance throughout the execution of *Refine* up to time t. The prices of vertices with negative balance do not change, so $p(w) = p_\epsilon(w)$. Combining this observation with inequalities (11) and (12), we get $p(v) \leq p_\epsilon(v) + 3l\epsilon \leq p_\epsilon(v) + 3n\epsilon$. ∎

Lemma 6.3 *The number of the price updates during an execution of Refine is at most $3n^2$.*

Proof: Immediate from lemmas 5.2 and 6.2. ∎

Lemmas 6.2 and 6.3 enable us to amortize the operations performed by the algorithm over increases in vertex prices.

Lemma 6.4 *The number of the saturating push operations during an execution of Refine is at most $5nm$.*

Proof: For any undirected edge $\{v, w\}$, consider saturating pushes from v to w and from w to v. Consider a saturating push from v to w. In order to push flow from v to w again, the subroutine must first push from w to v, which cannot happen until the price of w increases by at least ϵ. Similarly, $p(v)$ must increase by at least ϵ between saturating pushes from w to v. By charging all saturating pushes from v to w (except for the first one) to price increases of v and applying Lemma 6.2, we can bound the number of pushes that use the undirected edge $\{v, w\}$ by $2+3n \leq 5n$ (assuming $n \geq 2$). Summing over all undirected edges gives the desired bound. ∎

We define an admissible edge and an admissible graph as follows. Given a pseudoflow f and a price function p, we say that an edge (v, w) is *admissible* if $(v, w) \in E_f$ and $c_p(v, w) < 0$. Note that if (v, w) is an admissible edge and v is active, then a pushing operation is applicable to (v, w). For a given price function p, the *admissible graph* G_A is the graph of all admissible edges: $G_A = (V, E_A)$, where $E_A = \{(v, w) \in E_f | c_p(v, w) < 0\}$.

The admissible graph changes during the execution of the subroutine. We show, however, that the admissible graph is always acyclic.

Lemma 6.5 *Immediately after a price update operation has been applied to a vertex v, there are no admissible edges entering v.*

Proof: Before the price update, the reduced cost of any residual edge entering v is at least $-\epsilon$. The price update increases this cost by at least ϵ. Therefore after the price update, the cost of any edge entering v is nonnegative. ∎

Using Lemma 6.5, we can easily prove the following fact.

Lemma 6.6 *At any time during the execution of the generic Refine subroutine, the admissible graph G_A is acyclic.*

The next lemma bounds the number of nonsaturating pushes. This lemma has been strengthened to its current form with the help of Ron Rivest.

Lemma 6.7 *The number of the nonsaturating pushing operations in an execution of Refine is $O(n^2m)$.*

Proof: For each vertex v, let $h(v)$ be the number of vertices reachable from v in the current admissible graph G_A. Define the potential function Φ by $\Phi = \sum\{h(v) | v \text{ is active}\}$. (We define $\Phi = 0$ if there are no active vertices). It follows that $0 \leq \Phi \leq n$.

Consider the changes in Φ caused by operations performed during the execution of the generic subroutine. After the initialization $\Phi \leq n$. A nonsaturating push decreases Φ by at least one. Φ can increase only because of a saturating push or because of a price update. In either case Φ increases by at most n; in the latter case this follows from Lemma 6.5. The total number of nonsaturating pushes is bounded by the initial value of Φ plus the total increase in Φ throughout the execution of the generic subroutine, which is $O(n^2m)$. ∎

The following theorem bounds the number of basic operations in the generic *Refine* subroutine.

Theorem 6.8 *The generic Refine subroutine terminates after $O(n^2m)$ basic operations.*

Proof: Immediate from Lemmas 5.1, 5.2, 6.3, 6.4, and 6.7. ∎

Remark: The definition of admissible edges as unsaturating edges of negative cost can be changed to require admissible edges to have cost at most γ, for any fixed constant $-\epsilon < \gamma \leq 0$. This change affects only the constant factors of the above analysis. Choosing a value of γ strictly less than zero, such as $-\epsilon/2$, may be desirable from a theoretical or practical standpoint, but justification of this choice would require further analysis or experimentation.

7 Sequential Implementation

The running time of the generic subroutine depends upon the order in which the basic operations are applied and on the details of the implementation, but it is clear that any reasonable sequential implementation will run in polynomial time. As a first step toward obtaining an efficient sequential implementation, we shall describe a simple refinement of the subroutine that runs in $O(n^2m)$ time. Then we shall describe an implementation that runs in $O(n^3)$ time. In the next section, we shall describe implementation of the generic subroutine that uses the dynamic tree data structure and runs in $O(nm \log n)$ time.

We need some data structures to represent the network and the pseudoflow. We associate a positive direction and the following four values with each undirected edge $\{v, w\}$: $l(v, w)$, $u(v, w)$, $c(v, w)$, and $f(v, w)$. Each vertex v has a list of the incident undirected edges $\{v, w\}$ in a fixed order. Each edge $\{v, w\}$ appears in exactly two lists, the one for v and the one for w. Each vertex v has a balance $b_f(v)$ and a *current edge* $\{v, w\}$, which is the current candidate for a push out of v. Initially the current edge is the first edge on the edge list of v. After the initialization, the refined subroutine repeatedly selects an active vertex and applies the *push/update* operation described in Figure 4 to this vertex until there are no active vertices.

We need to show that *push/update* uses the price update operation correctly.

Lemma 7.1 *The push/update procedure uses a price updating operation only when this operation is applicable.*

Proof: The *push/update* procedure applies the price updating operation only to active vertices with positive balance. Just before the price update, for each edge (v, w) either $c_p(v, w) \geq -\epsilon/4$ or $r_f(v, w) = 0$, because $p(v)$ has not changed since $\{v, w\}$ was a current edge, $r_f(v, w)$ cannot increase unless $c_p(v, w) > -\epsilon/4$, and $p(w)$ never decreases. The lemma follows from the definition of the price updating operation. ∎

The refined subroutine needs one additional data structure, a set Q containing all vertices with positive balance. Initially Q contains vertices whose balance has been made positive during the initialization. Maintaining Q takes only $O(1)$ time per *push/update* operation. (Such an operation applied to an edge $\{v, w\}$ may require adding w to Q and/or deleting v.)

Push/update(v).
Applicability: v is active.
Action:
```
Let {v, w} be the current edge of v;
if push(v, w) is applicable then apply it
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 edge list of v
    else begin
      make the first edge on the edge list of v
            the current edge;
      apply Update-Price(v);
    end.
```

Figure 4: The push/update operation.

Theorem 7.2 *The refined algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating pushing step, for a total of $O(n^2m)$ time.*

Proof: Let $v \in V$ and let Δ_v be the number of edges on the edge list of v. A price update of v requires a single scan of the edge list of v. By the proof like that of Lemma 6.3, the total number of passes through the edge list of v is at most $6n + 1$: one for each of the at most $3n$ price updates of v, one before each price update as the current edge runs through the list, and one after the last price update. Every *push/update* operation selecting v either causes a push, changes the current edge of v, or increases $p(v)$. The total time spent in *push/update* operations selecting v is $O(n\Delta_v)$ plus $O(1)$ time per push out of v. Summing over all vertices and applying Lemmas 6.4 and 6.7 gives the theorem. ∎

In the generic maximum flow algorithm [20], a first-in, first-out ordering of the operations equivalent to the *push/update* operations leads to improved sequential and parallel time bounds. So far, we are unable to prove these bounds for a similar implementation of the minimum-cost flow algorithm. The first author conjectures, however, that these bounds hold. See [19] for details on the first-in, first-out implementation.

Next we describe an implementation of the subroutine that runs in $O(n^3)$ time. This implementation is due to Charles Leiserson. We call this implementation the *wave subroutine* because of its similarity to the maximum flow algorithm described in [36]. When applied to the generic maximum flow algorithm of [20] the wave method yields an $O(n^3)$-time algorithm as well.

The wave implementation does not maintain a queue of active vertices, but instead maintains a list L of *all* vertices, and preserves the invariant that L is topologically ordered

Discharge(v).
Applicability: v is active.
Action: **Repeat**
Apply *push/update*(v);
until $b_f(v) = 0$ **or** $p(v)$ increases;
if $p(v)$ has increased **then**
move v to the beginning of L;

Figure 5: The discharge operation.

with respect to the admissible graph G_A (i.e., for any two distinct vertices v and w, if w is reachable from v in G_A, then w appears after v on L).

Initially, the list L contains the vertices of G in arbitrary order. The implementation makes passes over the list, applying the *discharge* operation described in Figure 5 to active vertices. The *discharge* operation consists of applying the *push/relabel* operation to an active vertex until the excess becomes zero or the price of the vertex increases. In the latter case, the vertex is moved to the beginning of L, but the processing of L continues from the previous position of this vertex. The subroutine terminates when there are no active vertices.

The key to the analysis of the wave subroutine is the observation that, because of the topological ordering of vertices on the list L, if no price update occurs during a pass over the vertex list, the subroutine terminates.

Lemma 7.3 *The number of passes over the vertex list in the execution of the wave implementation of the Refine subroutine is* $O(n^2)$.

Proof: First we show by induction on the number of basic operations that the wave subroutine maintains a topological ordering of the vertices with respect to the admissible graph G_A (except in the middle of the *discharge* operation). The basis is trivial, because immediately after initialization there are no admissible edges. A push preserves the topological ordering, because this operation cannot create a new admissible edge. Immediately after the price of a vertex is changed, the vertex is moved to the beginning of the list L. The resulting ordering is topological by Lemma 6.5.

Next we show that if the vertex prices do not change during a pass over the vertex list, the subroutine terminates. This implies that there is at most one pass during which the price function does not change. The total number of passes is therefore at most $3n^2 + 1$, by Lemma 6.3.

Suppose that the price function does not change during a pass. Then no price updating operations are performed during this pass, and therefore the ordering of the vertices on the list L does not change and every vertex is able to get rid of all of its excess. Since the vertices are processed in topological order, no vertex has excess after the pass, and the algorithm terminates. ∎

The $O(n^2)$ bound on the number of passes allows us to prove an $O(n^3)$ bound on the running time of the wave subroutine.

Theorem 7.4 *The wave subroutine runs in* $O(n^3)$ *time.*

Proof: The work done by *update-price* and saturating *push* operations can be bounded by $O(nm)$ by a proof like that of Theorem 7.2. The number of nonsaturating *push* operations is $O(n^3)$ because there are $O(n^2)$ passes and at most one nonsaturating push per vertex per pass. Finally, operations on the vertex list L require $O(n)$ time per pass, for a total of $O(n^3)$ time. ∎

8 Use of Dynamic Trees

We obtain an $O(nm \log n)$ implementation of the generic subroutine by using the dynamic tree data structure of Sleator and Tarjan [32,33,35]. This data structure allows us to maintain a set of vertex-disjoint rooted trees in which each vertex v has an associated real value $g(v)$, possibly ∞ or $-\infty$. We regard a tree edge as directed toward the root, i.e. from child to parent. We denote the parent of a vertex v by $p(v)$. We adopt the convention that every vertex is both an ancestor and a descendant of itself. The tree operations we shall need are described in Figure 6.

The total time for a sequence of l tree operations starting with a collection of single-vertex trees is $O(l \log n)$, since n is the maximum tree size. (The implementation of dynamic trees presented in [33,35] does not support *find-size* operations, but it is easily modified to do so. See [20].)

In our application the edges of the dynamic trees are a subset of the current edges of the vertices. The current edge $\{v, w\}$ of a vertex $v \in V - \{s, t\}$ is *eligible* to be a dynamic tree edge (with $p(v) = w$) if $d(v) = d(w) + 1$ and $r_f(v, w) > 0$. Not all eligible edges are tree edges, however. The value $g(v)$ of a vertex v in its dynamic tree is $r_f(v, p(v))$ 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 ∞.

By using appropriate tree operations we can push flow along an entire path in a tree, either causing a saturating push or moving flow excess from some vertex in the tree all the way to the tree root. The dynamic tree operations are charged to the price updates and saturating pushes in such a way that each price update and each saturating push is charged a constant number of times. Therefore the total number of charges is $O(nm)$ and, since each dynamic tree operation costs $O(\log n)$, the running time of the dynamic tree subroutine is $O(nm \log n)$.

The details of the improved subroutine, which we call the *dynamic tree subroutine*, are as follows. The heart of the subroutine is the procedure *send*(v) which pushes excess from a nonroot vertex v to the root of its tree, cuts edges saturated by the push, and repeats these steps until $b_f(v) = 0$ or v is a tree root.

At the top level, the dynamic tree subroutine is exactly the same as the simple implementation of the generic sub-

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 all ancestors 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 edge from v to its parent. This operation does nothing if v is a tree root.

Figure 6: Dynamic tree operations.

routine described in the previous section. However, we replace the *push/update* operation with the *tree-push/update operation.*

A *tree-push/update* operation applies to a vertex v with positive balance 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 eligible for a pushing operation. In this case we link the trees containing v and w by making w the parent of v and do a send operation from v. The second case occurs if the edge $\{v, w\}$ is not eligible for a pushing operation. In this case we update the current edge of v and update the price of v if necessary. If the price of v is increased, we cut all tree edges entering v, to maintain the invariant that all dynamic tree edges are admissible.

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

Two observations imply that the dynamic tree subroutine is correct. First, any edge $\{v, w\}$ that is in a dynamic tree is admissible. By Lemma 6.6, in case (1) of *tree-push/update*, vertices v and w are in different trees, and the algorithm never attempts to link a dynamic tree to itself. Second, a vertex v that is not a tree root can have positive balance only in the middle of case (1) of a *tree-push/update* operation. To see this, note that only in this case does the algorithm add flow to a nonroot vertex, and this addition of flow is followed by a send operation that moves the nonroot excess of flow to one or more roots.

Theorem 8.1 *The dynamic tree implementation of the Refine subroutine runs in $O(nm \log n)$ time.*

Proof: First we bound the number of *link* and *cut* operations. The number of *link* operations is at most $5nm$ by a proof like that of Lemma 6.4. The number of *cut* operations is at most the number of *link* operations. The total number of *link* and *cut* operations is $O(nm)$. It is straightforward to establish that there are $O(1)$ tree operations per *cut* or *link*, from which the theorem follows. ∎

Corollary 8.2 *The implementation of the minimum-cost flow algorithm that uses the dynamic tree version of the Refine subroutine runs in $O(nm(\log n)\log(nC))$ time.*

Proof: Immediate from Theorems 4.1 and 8.1. ∎

9 Blocking Refine Subroutine

In this section we present an alternative approach to the design of the *Refine* subroutine, based on Dinic's approach to the maximum flow problem [6], which uses blocking flows. A blocking flow is defined for flow networks rather than for circulation networks. In a flow network all lower bounds on capacity are zero and there are two distinguished vertices, a *source* s and a *sink* t. A *flow* is a pseudoflow with zero balance for all vertices except s and t. A *blocking flow* is a flow with no forward augmenting path, i.e. a flow for which every path from s to t in the network contains a saturated edge. We use blocking flows only in the context of layered and acyclic networks. A network is *acyclic* if the underlying directed graph is acyclic. An equivalent definition is to say that a network is acyclic if its vertices can be labeled by integers in such a way that for every edge (v, w), we have *label*(v) > *label*(w). A network is *layered* [6] if its vertices can be labeled by integers in such a way that for every edge (v, w), we have *label*(v) = *label*(w) + 1.

There are many algorithms for finding a blocking flow in a layered network. Although a layered network is a special case of an acyclic network, most of these algorithms work in the more general acyclic case as well, achieving the same complexity bounds. In particular, the algorithms described in [21,24,36] can be used to find a blocking flow in an acyclic network in $O(n^2)$ time, and the algorithm described in [31] can be used to find a blocking flow in $O(m \log n)$ time. Galil's algorithm [14] finds a blocking flow in a layered network in $O(n^{2/3}m^{2/3})$ time. This algorithm can be modified to work on acyclic networks within the same time bound [15].

The Shiloach-Vishkin algorithm [30] can also be modified to find a blocking flow in an acyclic network in $O(n \log n)$ time on a PRAM using $O(n)$ processors and $O(n^2)$ memory.

We show how to implement the *Refine* subroutine using at most $3n$ blocking flow computations. The blocking flow variant of the subroutine, summarized in Figure 7, starts by

Procedure *Refine*(f, p, ϵ);

```
⟨⟨initialization⟩⟩
∀(v, w) ∈ V × V do begin
        if c_p(v, w) > 0 then f(v, w) ← l(v, w);
        if c_p(v, w) < 0 then f(v, w) ← u(v, w);
end;
⟨⟨ loop⟩⟩
while f is not a circulation do block(f, p, ε);
return(f, p);
```

end.

Figure 7: The blocking flow Refine subroutine.

bringing all edges into kilter. Then the subroutine repetitively performs the *block* operation described in Figure 8 until f is a circulation.

The *block* operation consists of two steps. In the first step, the vertices are partitioned into two sets, S and $\overline{S}$. The set S contains the vertices reachable from some vertex with positive balance in the admissible graph G_A (defined in Section 6), and the set $\overline{S}$ contains all remaining vertices. Then prices of vertices in S are increased by ϵ. The second stage constructs an acyclic auxiliary network, finds a blocking flow f' in the auxiliary network, and augments the pseudoflow f by f' (i.e. $f(v,w) \leftarrow f(v,w) + f'(v,w)$ for each $(v,w) \in V \times V$). The auxiliary network $G_{aux} = (V \bigcup \{s,t\}, E_{aux})$ is constructed from the admissible graph G_A by adding a source s, a sink t, and edges (s,v) of capacity $b_f(v)$ for every vertex $v \in V$ with $b_f(v) > 0$ and edges (w,t) of capacity $-b_f(w)$ for every vertex $w \in V$ with $b_f(w) < 0$. The capacities of edges $(v,w) \in E_A$ are defined to be $r_f(v,w)$. Note that augmenting f by a flow f' in the auxiliary network results in a valid pseudoflow.

Correctness of the blocking flow subroutine follows from the following lemma.

Lemma 9.1 *At the beginning and at the end of each execution of block, pseudoflow f is ϵ-optimal with respect to p, $S \supseteq \{v | b_f(v) > 0\}$, and $\overline{S} \supseteq \{v | b_f(v) < 0\}$. At the beginning and at the end of steps 1 and 2, the admissible graph G_A is acyclic.*

Proof: Omitted. ∎

The following lemma is a generalization of the lemma bounding the number of phases in a layered network algorithm for the maximum flow problem. This lemma is also similar to Lemma 6.2 and the lemma that bounds the number of maximum flow computations in a scaling step of the Bland-Jensen algorithm [5].

Lemma 9.2 *The number of blocking flow computations in an execution of the blocking flow Refine subroutine is at most $3n$.*

Procedure *block*(f, p, ϵ);

```
⟨⟨step 1⟩⟩
S ← {w | ∃v such that b(v) > 0
  and w is reachable from v in G_A};
S̄ ← V − S;
∀v ∈ S do p(v) ← p(v) + ε;
⟨⟨ step 2⟩⟩
construct G_aux = (V ⋃ {s, t}, E_aux);
find a blocking flow f' in G_aux;
augment f by f';
return(f, p);
```

end.

Figure 8: The block operation. The running time of the operation depends on the blocking flow subroutine used.

Proof: Note that the balance of a vertex can become positive only during the initialization step of the subroutine and the subroutine terminates when there are no vertices with a positive balance. Thus there must be a vertex v whose balance is positive during each execution of step 1. By Lemma 9.1, the price of v increases by ϵ during such a step, and does not change otherwise. Prices of vertices with a negative balance do not change. By an argument similar to the proof of Lemma 6.2, the price of a vertex cannot increase by more then 3ϵ. Therefore the number of phases is at most $3n$. ∎

Theorem 9.3 *Let $B(n,m)$ be the running time of an algorithm that finds a blocking flow in acyclic networks. Then the minimum-cost flow algorithm runs in $O(nB(n,m)\log(nC))$ time.*

Proof: Immediate from Theorem 4.1 and Lemma 9.2. ∎

Corollary 9.4 *The minimum-cost flow problem can be solved in $O(\min(nm\log n, n^{5/3}m^{2/3}, n^3)\log(nC))$ time.*

Proof: The lemma follows from Theorem 9.3 and using the blocking flow algorithms described in [21,14,31] (extended to acyclic networks as discussed above). ∎

10 Parallel and Distributed Implementations

In this section we discuss parallel and distributed implementations of the minimum-cost flow algorithms described earlier. Related work on parallel and distributed algorithms for network flow problems includes a paper of Shiloach and Vishkin [30], and a paper of Bertsekas [3]. Shiloach and Vishkin give an $O(n^2 \log n)$ time parallel algorithm for the maximum flow problem, which can be combined with the

cost scaling algorithms [5,29] to obtain an $O(n^3 \log(n) \log C)$ time parallel algorithm for the minimum-cost flow problem. Bertsekas [3] exhibits a "chaotic" algorithm for the minimum-cost flow problem that converges in a finite number of steps in a distributed model.

First we discuss parallel and distributed implementations of the minimum-cost flow algorithm which are based on the parallel version of the Shiloach-Vishkin blocking flow algorithm [30]. We call the algorithm obtained by using this parallel blocking flow algorithm to implement *Min-Cost* the *blocking algorithm.* In PRAM [9] and DRAM [23] models of parallel computation the results of Section 9 and [30] imply the following theorem.

Theorem 10.1 *The blocking algorithm runs in* $O(n^2(\log n)\log(nC))$ *parallel time using* n *processors and* $O(n^2)$ *memory.*

The following theorem gives the performance of the blocking algorithm in the synchronous distributed model of parallel computation [1,17]. In the statement of this theorem, Δ_p denotes the degree of processor p in the network. The proof of this theorem follows from the results of Section 9 and [30].

Theorem 10.2 *In the synchronous distributed model, the blocking algorithm runs in* $O(n^2(\log nC))$ *time using* $O(n\Delta_p)$ *memory per processor* p *and* $O(n^3 \log(nC))$ *messages.*

In the asynchronous distributed model [1,17], the synchronization protocol of [1] can be used to implement the blocking algorithm. The resulting bounds on message and memory complexity of the algorithm are the same as in Theorem 10.2 and the resulting time bound is greater then the bound given by the theorem by a factor of $\log n$.

The above parallel and distributed algorithms require a large amount of memory. Alternative implementations of the minimum-cost flow algorithm using a parallel version of the generic *Refine* subroutine (similar to the parallel version of the generic maximum flow algorithm [20]) use a linear amount of memory. The time bounds we can prove for these implementations, however, are worse then the sequential time bounds of this paper. The first author conjectures that the actual running time of these implementations are better (i.e. same as the time bounds in Theorems 10.1 and 10.2). See [19] for more details about this conjecture and for an alternative asynchronous implementation of *Refine.*

11 Remarks

The concluding remarks concern practicality and potential improvements and extensions of the minimum-cost flow algorithm.

We believe that our approach will yield highly practical algorithms. Our belief is based on the work of Gabow [13] and of Bateson [2] who have shown that scaling algorithms are practical; on the experimental results of Bland and Jensen [5] on their cost-scaling algorithm for the minimum-cost flow problem; on experience with implementations of the Goldberg-Tarjan maximum flow algorithm [19,26]; and on the experimental results of Bertsekas and Tseng with an implementation of their minimum-cost flow algorithm [4]. Experimental study is needed, however, before the practicality claim can be made with certainty.

The approach presented in this paper allows a great degree of flexibility. For example, the *Refine* subroutine can be modified to reduce the error parameter ϵ by a factor different from the factor of two as in our description. Fine-tuning of this factor changes the constant factors of the running time, and therefore the practical performance of the algorithm. Also, as in the case of the maximum flow algorithm, a different ordering of the basic operations in the generic *Refine* subroutine may result in a better performance.

The algorithm can start with any initial price function such that the absolute value of the difference between the prices of any pair of adjacent vertices is at most C (or even $O(n^k C)$ for a constant k). In fact, in a practical implementation the initial price function should be obtained by using a shortest path subroutine. *Refine* can also use a shortest path subroutine to update the price function at intermediate points of the execution. Another heuristic improvement is to set ϵ to the largest amount of violation of the complementary slackness conditions before each call to *Refine.*

Our algorithm works essentially by scaling costs. It would be interesting to see if a similar approach could be made to work by scaling costs and capacities simultaneously. A possible approach is to order operations of the generic subroutine by the change in value (i.e. cost-capacity product) caused by these operations.

Another interesting open question is the existence of a strongly polynomial $O(n^3 \log^k n)$ (or $O(nm \log^k n)$, if one is able to take advantage of sparsity) algorithm for the minimum-cost flow problem. A possible approach would combine the techniques of [11,16,27,34] with the techniques described in this paper.

Acknowledgments

We would like to thank Charles Leiserson, Serge Plotkin, Ron Rivest, and David Shmoys for many helpful suggestions and for comments on preliminary versions of this paper.

References

[1] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32:804–823, 1985.

[2] C. A. Bateson. Performance comparison of two algorithms for weighted bipartite matchings. 1985. M.S. thesis, University of Colorado, Boulder, Colorado.

[3] D. P. Bertsekas. Distributed asynchronous relaxation methods for linear network flow problems. In *Proc. 25th IEEE Conference on Decision and Control, Athens, Greece*, 1986.

[4] D. P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *O. R. Journal*, 1986. (To appear).

[5] 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.

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

[7] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

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

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

[10] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses. In *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pages 338–346, 1984.

[11] 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.

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

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

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

[15] Z. Galil. Personal communication. 1987.

[16] 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.

[17] 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(1):66–77, 1983.

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

[19] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., 1987.

[20] 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.

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

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

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

[24] 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.

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

[26] A. T. Ogielski. Integer optimization and zero-temperature fixed point in Ising random-field systems. *Physical Review Lett.*, 57(10):1251–1254, 1986.

[27] 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.

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

[29] H. Röck. Scaling techniques for minimal cost network flows. In V. Page, editor, *Discrete Structures and Algorithms*, Carl Hansen, Munich, 1980.

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

[31] 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.

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

[33] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

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

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

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