New Distance–Directed Algorithms for Maximum Flow and Parametric Maximum Flow Problems

J. B. Orlin
and
R. K. Ahuja

Sloan W.P. No. 1908-87 July 1987

New Distance–Directed Algorithms for Maximum Flow and Parametric Maximum Flow Problems

James B. Orlin and Ravindra K. Ahuja*
Sloan School of Managment
Massachusetts Institute of Technology
Cambridge, MA. 02139, USA

* On leave from Indian Institute of Technology, Kanpur 208 016, INDIA

New Distance–Directed Algorithms for Maximum Flow and Parametric Maximum Flow Problems

Abstract

Until recently, fast algorithms for the maximum flow problem have typically proceeded by constructing layered networks and establishing blocking flows in these networks. However, in the recent years, new "**distance-directed**" algorithms have been suggested that do not construct layered networks but instead maintain a "**distance label**" with each node. The distance label of a node is a lower bound on the length of the shortest augmenting path from the node to the sink. In this paper, we develop two new distance–directed augmenting path algorithms for the maximum flow problem. Both the algorithms run in $O(n^2m)$ time. We show that these algorithms are equivalent to Edmonds–Karp and Dinic's algorithm in the sense that they enumerate the same augmenting paths in the same sequence. Using a scaling technique, we improve the complexity of our distance–directed algorithms to O(nm log U), where U denotes the largest arc capacity. We also consider applications of these algorithms to unit capacity maximum flow problems and a class of parametric maximum flow problem.

Subject classification.

484. A New Distance-Directed Algorithm for Maximum Flows.

649. A New Distance-Directed Algorithm for Parametric Max Flows.

1. Introduction

In this paper, we suggest several new algorithms for the maximum flow and parametric maximum flow problems. The maximum flow problem is one of the most fundamental network problems and has been investigated extensively in the literature (for example, by Ford and Fulkerson 1956, Dinic 1970, Edmonds and Karp 1972, Karzanov 1974, Cherkasky 1977, Malhotra et. al 1978, Galil 1980, Galil and Naamad 1980, Sleator and Tarjan 1983, Gabow 1985, Goldberg 1985, Goldberg and Tarjan 1986, Ahuja and Orlin 1987). Efficient algorithms for computing maximum flows are important not only because they are applied directly to the analysis of traffic or communication networks, but are often employed in the subproblems of other network problems. Some of the network problems whose algorithms use the maximum flow algorithm as a subroutine are the time–cost tradeoff problem in CPM networks (Fulkerson 1961, Kelley 1961), the parametric network feasibility problem (Minieka 1972), the network design problem (Hu 1974) and the minimax transportation problem (Ahuja 1986). Moreover, it plays an important role in solving the minimum cost flow problem (Rock 1980, Tardos 1985, Bland and Jensen 1985). Recently, Goldberg and Tarjan (1987) have developed the fastest known algorithm for the minimum cost flow problem using an extension of their maximum flow algorithm as a major subroutine.

Ford and Fulkerson (1956) formulated the maximum flow problem and solved it using their augmenting path algorithm. Edmonds and Karp (1972) showed that by augmenting flows along shortest paths, the augmenting

path algorithm runs in time $O(nm^2)$ on networks with n nodes and m arcs. Independently, Dinic (1970) suggested an $O(n^2m)$ algorithm which proceeds by constructing shortest path networks (known as **layered networks**) and establishing blocking (or maximal) flows in these networks. Dinic's algorithm has been studied extensively by researchers who dramatically improved its worst case running time using sophisticated data structures. Most notable among these improvements is the 'dynamic trees' algorithm developed by Sleator and Tarjan (1983). Their algorithm runs in O(nm log n) steps. Gabow's (1985) scaling algorithm also uses Dinic's algorithm as a major subroutine.

Recently, Goldberg (1985) and Goldberg and Tarjan (1986) developed a new approach to solve the maximum flow problem that does not construct layered networks but instead maintains "**distance labels**". Informally, a distance label of a node is an integral lower bound on the length of the shortest augmenting path from that node to the sink. A distance label is called **exact** if it equals the length of the shortest augmenting path, and **approximate** otherwise. Distance labels have several advantages over layered networks. They are simpler to understand, easier to manipulate and have led to more efficient algorithms. We refer to algorithms that utilize distance labels as **distance–directed** algorithms. Goldberg (1985) developed the first distance–directed algorithm. Subsequently, Goldberg and Tarjan (1986) developed improved distance–directed algorithms. Currently, the fastest algorithm to solve the maximum flow problem, due to Ahuja and Orlin (1987a), is also a distance–directed algorithm. Its running time is $O(nm + n^2 \log U)$ steps, where U is an upper bound on the capacities of arcs directed from the source.

So far all the proposed distance–directed algorithms have been preflow based algorithms, similar in essence to Karzanov's (1974) algorithm. In this paper, we explore the use of distance labels in augmentation based algorithms, i.e., algorithms which consist of repetitively augmenting flows in augmenting paths from source to sink. We suggest two algorithms -- the first algorithm maintains distance labels approximately and the second algorithm maintains exact distance labels. The first algorithm has close resemblance to Dinic's algorithm and the second algorithm can be viewed as an improved implementation of Edmonds–Karp algorithm. We use a scaling technique to improve the worst case complexity of our algorithms and also consider their applications to unit capacity maximum flow problems and a class of parametric maximum flow problem.

Our motivation for the development of these methods is threefold. First, the distance–directed algorithms have certain computational advantages over the algorithms that construct layered networks. Second, our development of distance–directed algorithms helps to provide a unifying framework for maximum flow algorithms. Third, we provide additional supportive evidence for the computational power of distance–directed algorithms.

Our original motivation for conducting the research presented in this paper was to search for new augmenting path algorithms that might run more efficiently in practice than the best available algorithms. Dinic's algorithm provided a natural starting point since it is considered by many to be the most efficient augmenting path algorithm for solving the maximum flow problem (see Cheung 1980, Glover et. al 1979, 1984, and Imai 1983). Our

preliminary testing of the algorithms in Ahuja and Orlin (1987b) indicates that we have succeeded on this count. At the same time, we discovered both theoretical and practical improvements of the Edmonds–Karp algorithm.

Our results also reveal a common link between Dinic's and Edmonds–Karp algorithm. We show that the two distance–directed algorithms and the algorithms of Edmonds-Karp and Dinic are equivalent in the sense that they enumerate the same augmenting paths in the same sequence, and are different only in the manner in which the augmenting paths are obtained. If we consider the distance–directed algorithm with approximate distance labels as Dinic's algorithm, then we would say that we have found an efficient way to construct the layered networks dynamically. In particular, we dynamically construct those arcs of the layered networks which are traversed by Dinic's depth first search method. Our approach allows one to move from layered networks to working with the original network with no loss in the computational efficiency. Likewise, if we consider the distance–directed algorithm with exact distance labels as Edmonds–Karp algorithm, we would say that the distance labels allow us to locate the shortest augmenting paths in O(n) time on average instead of O(m) time. In other words, we have found an efficient data structure for locating the next shortest augmenting path quickly (in an amortized average case sense).

This unification of the simple augmenting path algorithm and the layered network approach also has pedagogical advantages. The distance based approach not only bridges the gap between Edmonds–Karp algorithm and Dinic's algorithm, but it helps to bridge the gap between these algorithms

and the distance-based preflow methods of Goldberg (1985), Goldberg and Tarjan (1986), and Ahuja and Orlin (1987a).

The paper is organized as follows. We present graph notations in Section 2. In Sections 3 and 4, we describe two distance–directed augmenting path algorithms for the maximum flow problem. The first algorithm uses approximate distance labels and the second algorithm maintains exact distance labels. Both the algorithms run in $O(n^2m)$ time on a network with n nodes and m arcs. We show the equivalence of these algorithms with the algorithms of Edmonds-Karp and Dinic in Section 5. In Section 6, we show that Gabow's scaling technique can reduce the complexity of these algorithms to O(nm log U), where U represents the largest integral capacity. The basic idea behind the scaling algorithms is to augment flows along paths with sufficiently large capacities. We next observe that if the number of capacitated arcs $p \leq m/2$, then the time bound of scaling algorithms can be improved to $O(nm \log_{m/p} U)$. This observation leads to an O(nm) algorithm for finding feasible flows in a sparse uncapacitated transportation problem under reasonable assumptions.

In Section 7, we consider special cases of the maximum flow problem. We show that the distance–directed algorithms can be used to obtain a maximum flow in unit capacity networks in $O(\min (n^{2/3}, m^{1/2}) m)$ time, and in unit capacity simple networks in $O(n^{1/2} m)$ time. These bounds are same as those obtained by Even and Tarjan (1975) for these problems and use similar techniques, but our proofs are simpler.

In Section 8, we consider a class of parametric maximum flow problems in which the capacities of some, say k, arcs emanating from the source node are increasing linear functions of a parameter λ and the objective is to determine the maximum flow value for all values of $\lambda \in (0, \infty)$. We show that Itai and Rodeh's (1985) algorithm, which utilizes layered networks and runs in $O(kn^2m)$ time, can be implemented in $O(n^2m)$ using distance labels. Indeed, the parametric maximum flow problem is solved with no increase in the worst case running time over the unparametrized problem. Independently, Gallo, Grigoriades and Tarjan (1987) have solved the problem in $O(nm \log (n^2/m)$ time by appropriately generalizing the Goldberg-Tarjan algorithm.

2. Notation

Let $G = (N, A)$ be a directed network with a positive integer capacity u_{ij} for every arc $(i, j) \in A$. Let $n = |N|$ and $m = |A|$. The source s and sink t are two distinguished nodes of the network. We assume without loss of generality that the network does not contain multiple arcs. We further assume that for every arc $(i, j) \in A$, an arc (j, i) is also contained in A, possibly with zero capacity. We define the arc adjacency list $A(i)$ of a node $i \in N$ as the set of arcs directed out of node i, i.e., $A(i) := \{(i, k) \in A : k \in N\}$.

Let $U = \max_{(i,j) \in A} \{u_{ij}\}$. The maximum flow problem is to determine a flow x of maximum flow value v. Mathematically, this problem can be stated as follows.

Maximize v,

subject to

$$\sum_{j \in N} x_{ji} - \sum_{j \in N} x_{ij} = \begin{cases} -v, & \text{if } i = s, \\ 0, & \text{if } i \neq s,t, \\ v, & \text{if } i = t, \end{cases} \quad \text{for all } i \in N,$$

$0 \leq x_{ij} \leq u_{ij}$, for each $(i, j) \in A$.

The **residual capacity** of any arc $(i, j) \in A$ with respect to a given flow x is given by $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. The network consisting of arcs with positive residual capacity only is referred to as the **residual network**.

A **distance function** $d: N \rightarrow Z^+$ for flow x is a function from the set of nodes to the non-negative integers. *We say that the distance function is valid if it also satisfies the following two conditions:*

C1. $d(t) = 0$

C2. $d(i) \leq d(j) + 1$, for every arc $(i, j) \in A$ with $r_{ij} > 0$.

We refer to d(i) as the **distance label** of node i. It is easy to demonstrate using induction that d(i) is a lower bound on the length of the shortest path from i to t in the residual network. If for each $i \in N$, the distance label d(i) equals the length of the shortest path from i to t in the residual network, then we call the distance labels **exact**; otherwise, we call d(i) **approximate**.

An arc (i, j) in the residual network is called **admissible** if it satisfies $d(i) = d(j) + 1$. An arc which is not admissible is called an **inadmissible** arc. The algorithms in this paper augment flows along paths consisting of admissible arcs only. It can be shown that a path from source to sink in the

residual network consisting only of admissible arcs is a shortest augmenting path.

3. Distance–Directed Algorithm with Approximate Distance Labels

In this section, we describe our first distance–directed algorithm for the maximum flow problem. This algorithm maintains approximate distance labels. The algorithm proceeds by performing depth first search of the residual network to identify shortest augmenting paths from source to sink. The distance labels are used to direct the search. We refer to this algorithm as DD1. Our presentation of the algorithm is similar to the presentation of Dinic's algorithm by Tarjan (1983).

The algorithm DD1 first performs a breadth first search of G starting with the sink node to compute the exact distance label d(i). At any general step, the algorithm maintains a path P from the source node to some node i* with the property that every arc in this path is **admissible**. We refer to node i* as the **current node** and path P as the **current path**. We store the current path using predecessor indices, i.e., pred(j) = i for each $(i, j) \in P$. The algorithm performs an **advance** step at the current node i* in which it attempts to find an admissible arc (i*,j*) directed from node i*. If such an arc is found, then it is added to the current path and the advance step is executed at node j*. Otherwise, d(i*) is increased, which makes the arc (pred(i*), i*) inadmissible. We thus perform a **retreat** step. In this step, arc (pred(i*), i*) is deleted from the current path and the advance step is executed at node pred(i*). Whenever the sink node is reached by the current path, a maximum possible flow is augmented on this path and the advance step is executed at the source. The

algorithm terminates when $d(s) \geq n$ indicating that there is no augmenting path from source to sink.

We use the following data structure to select an admissible arc in the advance step. We maintain with each node i a list, A(i), of arcs directed from it. Arcs in each list can be arranged arbitrarily, but the order once decided remains unchanged throughout the algorithm. Each node i has a **current-arc** (i, j) which is the current candidate for the next advance step. Initially, the current–arc of node i is the first arc in its arc list. This list is examined sequencially and whenever the current arc is found to be inadmissible for augmenting flow, the next arc in the arc list is made the current arc. When all arcs in A(i) have been examined, it is time to update the distance label of node i.

A formal description of the algorithm is given below followed by its analysis.

initialize. Perform breadth first search of the residual network starting with the sink node to compute the exact distance labels d(i). Let $P = \emptyset$ and $i = s$. Go to **advance(i)**.

advance (i). Starting with the current–arc of node i, scan arcs in A(i) in order until either (1) the end of arc list is reached, or (2) an admissible arc (i, j) is found. In case 1, perform **relabel (i)** and go to **retreat (i)**. In case 2, set pred(j) : = i and $P: = P \cup \{(i,j)\}$. If $j = t$ then go to **augment**; else replace i by j and repeat **advance(i)**.

augment. Let Δ be the minimum residual capacity among arcs in P. For each (i, j) $\in$ P, substract Δ from r_{ij} and add Δ to r_{ji}. Set P: = ø, i: = s and go to **advance(i)**.

relabel(i). Update $d(i) = \min_{(i,j) \in A(i)} \{d(j) + 1: r_{ij} > 0\}$ and set the current–arc of node i as the first admissible arc in A(i). If $d(s) \geq n$, then stop.

retreat(i). If i = s then go to **advance(i)**; else delete (pred(i), i) from P, replace i by pred(i) and go to **advance(i)**.

We now show that the above algorithm correctly computes a maximum flow in $O(n^2m)$ time. Lemma 1 and Lemma 2 are implicit in the paper by Goldberg and Tarjan (1986) in the context of preflow based methods. We include the proofs here for the sake of completeness.

Lemma 1. The algorithm DD1 maintains valid distance labels at each step. Moreover, at each relabel step the distance label of a node strictly increases.

Proof. We show that the algorithm maintains valid distance labels at every step by performing induction on the number of augment and relabel steps. The initialize step constructs valid exact distance labels. Assume inductively that the distance function is valid prior to a step, i.e., satisfies the validity conditions C1 and C2. A flow augmentation on arc (i, j) might delete this arc from the residual network, but this does not affect the validity of the distance function. Augmentation on arc (i, j) may create an additional arc (j, i) with $r_{ji} > 0$ and an additional condition $d(j) \leq d(i) + 1$ needs to be satisfied. This validity condition remains satisfied since $d(i) = d(j) + 1$ by the property of the augmenting path. During a relabel step, the new distance label

of node i is $d'(i) := \min\{d(j) + 1: (i, j) \in A(i) \text{ and } r_{ij} > 0\}$, which is consistent with the validity conditions. The relabel step is performed when there is no arc $(i, j) \in A(i)$ with $d(i) = d(j) + 1$ and $r_{ij} > 0$. Hence $d(i) < \min\{d(j) + 1: (i, j) \in A(i) \text{ and } r_{ij} > 0\} = d'(i)$, thereby proving the second part of the lemma. □

Theorem 1. The algorithm DD1 correctly computes a maximum flow.

Proof. The algorithm terminates when $d(s) \geq n$. Since $d(s)$ is a lower bound on the length of the shortest augmenting path from s to t, it implies that there is no augmenting path from source to sink. This is the classical termination criteria for the maximum flow algorithm.

It is easy to show the presence of a minimum cutset when $d(s) \geq n$. Let n_k denote the number of nodes with distance label equal to k for $0 \leq k \leq n$. Note that n_{k^*} must be zero for some $k^* \leq n - 1$ as $\sum_{k=0}^{n-1} n_k \leq n - 1$. Let $S := \{i \in N: d(i) > k^*\}$ and $T := \{i \in N: d(i) < k^*\}$. Both the sets S and T are non–empty as $s \in S$ and $t \in T$. Consider the cutset $Q := \{(i,j) \in A : i \in S \text{ and } j \in T\}$. By construction, we have $d(i) > d(j) + 1$ for all $(i, j) \in Q$. By property C2, $r_{ij} = 0$ for each $(i, j) \in Q$. Hence Q is a minimum cutset and the current flow is maximum. □

Lemma 2. (a) Each distance label increases at most n times. Consequently, the total number of relabel steps is at most n^2. (b) The number of augment steps is at most $nm/2$.

Proof. Each relabel(i) step increases $d(i)$ by at least one. After at most n relabels of node i, $d(i) \geq n$. Then this node is never picked up during advance step since $d(s) < n$ and for every node k in the current path $d(k) < d(s)$.

Thus a node is relabeled at most n times and the total number of relabel step is bounded by n^2.

Each augment step saturates at least one arc, i.e., decreases its residual capacity to zero. Suppose that arc (i, j) becomes saturated at some iteration (at which d(i) = d(j) + 1). Then no more flow can be sent on (i, j) until flow is sent back from j to i (at which d'(j) = d'(i) + 1 $\geq$ d(i) + 1 = d(j) + 2). Hence, between two consecutive saturations of arc (i, j), d(j) increases by at least 2 units. The arc (i, j) can become saturated at most n/2 times and the total number of arc saturations is no more than nm/2. ❑

Theorem 2. The algorithm DD1 runs in $O(n^2m)$ time.

Proof. The step relabel(i) is performed O(n) times and each execution requires $O(|A(i)|)$ time. The total time spent in relabel steps is thus $O\left(\sum_{i \in N} n\,|A(i)|\right) = O(nm)$. Each retreat step requires O(1) effort and is executed $O(n^2)$ times, resulting in $O(n^2)$ total effort. The number of flow augmentations is O(nm) and each augmentation takes O(n) time. The total time spent in augment steps is, therefore, $O(n^2m)$. Further, the advance step is executed $O(n^2m)$ times, since after at most n consecutive advance steps either a relabel step is performed or a flow augmentation is done. The time taken by an advance(i) step is O(1) plus the time spent in replacing current–arc by next–arc while finding an admissible arc. After $|A(i)|$ such replacements for node i, relabel(i) occurs. Thus, the total number of these replacements is bounded by $\sum_{i \in N} n\,|A(i)| = O(nm)$. The time taken by all advance steps is $O(n^2m + nm) = O(n^2m)$. The theorem now follows. ❑

The proof of Theorem 1 also suggests an efficient alternative termination condition for the algorithm DD1. The termination criteria of $d(s) \geq n$ is satisfactory from a worst case analysis, but may not be so efficient in practice. We have observed empirically (Ahuja and Orlin (1987b)) that the algorithm spends too much time in relabelings, a major portion of which is done, after the maximum flow has already been established. The algorithm can be sped up by detecting the presence of a minimum cutset earlier. We accomplish this by maintaining the number of nodes n_k with distance label equal to k, for $0 \leq k \leq n$. The algorithm updates this array after every relabel operation and terminates whenever a gap in this array is found, i.e., a zero in between two non-zero elements. The nodes across the gap constitute a minimum cutset.

4. Distance–Directed Algorithm with Exact Distance Labels

In this section, we describe a variant of the algorithm DD1 which maintains exact distance labels instead of approximate distance labels. The resulting algorithm proceeds by maintaining a shortest path tree and augmenting flows on the unique tree path from source to sink. We refer to this algorithm as **DD2**.

A **directed in–tree** (rooted at sink) is a tree in which every node other than the sink has exactly one out–going arc. Such a tree has a unique directed path from every node in the tree to the sink. A **shortest path tree** is a directed in–tree in which the unique path from every node to t is a shortest path in the residual network. The following lemma gives a characterization of the shortest path tree.

Lemma 3. A directed in–tree T with respect to a flow x is a shortest path tree if and only if there exists numbers d(i) satisfying the following conditions:

C3. $d(i) = d(j) + 1$ and $r_{ij} > 0$, for each $(i, j) \in T$; and

C4. $d(i) \leq d(j) + 1$, for each $(i, j) \in A$ with $r_{ij} > 0$.

Proof. Define the length of every arc (i, j) in the residual network as 1. The above conditions are then equivalent to the optimality conditions of the shortest path problem (Lawler 1976). ❑

The algorithm DD2 maintains a shortest path tree at every step. The initial tree is constructed by performing a breadth first search of the residual network starting at the sink node. The algorithm then repeatedly performs two steps: augment and update–tree. In the augment step, a maximum possible flow is sent in the tree path from s to t. The flow augmentation decreases the residual capacity of some arcs in the path to zero thereby making them inadmissible (violating condition C3). The update–tree step replaces these inadmissible arcs by admissible arcs, one by one, and at the same time keeps the condition C4 satisfied. The algorithm uses the same data structure as in the algorithm DD1. A detailed description of the algorithm is given below.

initialize. Perform breadth first search of the residual network starting with the sink to determine the initial distance labels d(i) and the tree of shortest paths T. Go to **augment**.

augment. Let P be the unique path in T from source to sink and Δ be the minimum residual capacity among arcs in P. For each $(i, j) \in P$, subtract Δ

from r_{ij} and add Δ to r_{ji}. Let B denote the set of arcs whose residual capacities are reduced to zero by flow augmentation. Go to **update–tree.**

update–tree. If $B = \emptyset$, then go to **augment.** Otherwise delete an arc (i, j) from B. Starting with the current–arc of node i, scan arcs in A(i) until either (1) an admissible arc (i, k) is found, or (2) end of the arc list is reached. In case 1, replace (i, j) by (i, k) in T and repeat update–tree. In case 2, relabel node i as $d(i) = \min_{(i,j) \in A(i)} \{d(j) + 1: r_{ij} > 0\}$ and set the current-arc of node i as the first admissible arc in A(i). There are two possibilities to consider.

(a) If $d(i) \geq n$, then delete node i and arc (i, j) from T. If node s is deleted, then STOP, else repeat update–tree.

(b) If $d(i) < n$, then replace (i, j) by the current–arc of node i in T. Set $B = B \cup \{(p, i): (p, i) \in T\}$ and repeat update–tree.

Lemma 4. The algorithm DD2 maintains a valid tree of shortest paths at every step (except possibly during the execution of update-tree).

Proof. We use induction over the number of augment and update–tree steps to show that the algorithm maintains a tree of shortest paths. By construction, the initial tree is a tree of shortest paths. It is already shown in Lemma 1 that flow augmentation keeps the condition C4 satisfied. The flow augmentation, however, reduces the residual capacity of some arcs to zero thus making them inadmissible. The update–tree step picks up these inadmissible arcs one by one and replaces them by admissible arcs. A relabel operation of node i makes all tree–arcs incident on node i inadmissible and

they are added to the set of inadmissible arcs. Eventually, all tree–arcs are admissible and we again get a tree of shortest paths. ❑

Theorem 3. The algorithm DD2 correctly computes a maximum flow in $O(n^2m)$ time.

Proof. The algorithm terminates when $d(s) \geq n$ implying that there is no augmenting path from source to sink. The minimum cutset can be constructed in the same way as in the algorithm DD1. The augment step is performed O(nm) times, since each execution saturates at least one arc and there can be O(nm) saturations. The augment step takes O(n) time per execution and $O(n^2m)$ in total. The update–tree step, too, is performed O(nm) times since it follows an augment step. The bottleneck operations in the update–tree step are the time spent in scanning arcs to find admissible arcs, and the time needed to relabel nodes. As in Theorem 2, both of these operations take O(nm) time. The theorem now follows. ❑

5. Relationship to Edmonds-Karp and Dinic's Algorithms

In this section, we point out the relationship of our distance–directed algorithms with the algorithms of Edmonds and Karp (1972) and Dinic (1970). We show that all of these algorithms are equivalent in the sense that they enumerate the same augmenting paths in the same sequence, and they are different only in the manner in which these augmenting paths are obtained.

To begin with, we need some additional notations. We assume that arcs in A are ordered in an **arbitrary** manner, but this ordering remains fixed throughout the algorithm. The index o(i, j) indicates the sequence number of

an arc (i, j) $\in$ A in this order and is called its **arc–order**. We further assume that the order of the arcs in A(i) are compatible with the order or arcs in A in that the arcs in A(i) are arranged in the increasing arc–order values for each i $\in$ N. The result of this section are based on the assumption that whenever arcs in A(i) are scanned by any algorithm, they are scanned in the above order.

We say that a vector **a** is lexicographically smaller than another vector **b** if the first non–zero component of **a** – **b** is negative. We denote it as **a** $\{$ **b**. We say that a path P is lexicographically smaller than another path Q, if path P taken as a sequence of arc-order values of arcs in the path is lexicographically smaller than the corresponding sequence for path Q. An augmenting path from source to some node i is called the **least lexicographic** path if it is of shortest length and is lexicographically smaller than all other shortest length paths from s to i in the residual network. We denote the least lexicographic path of node i by **PATH(i)**. We intend to show that the algorithms of Edmonds and Karp and Dinic, as well as the algorithms DD1 and DD2 always generate least lexicographic paths and augment flows in these paths. This establishes the equivalence of these algorithms.

The Edmonds–Karp algorithm determines the shortest path from source to sink using breadth first search, i.e., by labeling nodes reachable from the source and examining the labeled nodes in the first–in–first–out (FIFO) order. The algorithm iteratively determines nodes at distance k (or in layer k) and examines the arc adjacency lists of these nodes to determine nodes in layer k + 1. The algorithm associates with each labeled node i a unique augmenting path which can be determined by backtracking up to the source node. We show inductively that this path is the least lexicographic path to that node and nodes in a layer are determined in the order so that

their least lexicographic paths are lexicographically increasing. We perform induction on the number of nodes examined by the algorithm. Suppose that a node i in layer k is examined and let $j_1, j_2, \ldots, j_w$ be the newly labeled nodes, in order, in layer k + 1. Then, PATH(j_l):= PATH(i) $\cup\{(i, l)\}$, for each l: = 1, . . . , w. Clearly, PATH(j_1) $\{$ PATH(j_2) $\{ \ldots \{$ PATH(j_w), since arcs in A(i) are scanned in increasing arc–order values. Further, by the induction hypothesis and the fact that nodes are examined in the FIFO order, all nodes at level k that are yet to be examined have their least lexicographic paths lexicographically greater than PATH(i). Hence, all paths of lengths k+1 generated by examining these nodes will be lexicographically greater than each of the paths PATH(j_1), PATH(j_2), . . ., PATH(j_l). This shows that our claim is true with respect to the last examined node.

We next consider Dinic's algorithm as presented by Tarjan (1983). We outline the algorithm here. We refer the reader to Tarjan (1985) for a complete description of the algorithm. Dinic's algorithm performs a depth first search of the layered network to identify an augmenting path from source to sink. Note that limiting the search to the layered network is not really restrictive since any path in the residual network, but not in the layered network, is not the shortest augmenting path and will not be discovered by any of these algorithms.

At any general step, Dinic's algorithm maintains a path from the source to some node i in the layered network. We refer to node i as the **currrent node** and its path as the **current path**. The current path is stored using predecessor indices. The algorithm performs either an **advance** step or a **retreat** step at current node. The advance step at node i is performed by scanning its arc list to identify an arc (i, j) such that node j is in the next layer

and is unblocked. If no arc is found, then a retreat step is performed. In this step, node i is labeled as "blocked" and an advance step is performed at pred(i).

We claim that the current path to the current node i is the least lexicographic path PATH(i) and the current node is the node satisfying PATH(i) $\{$ PATH(l) for all unblocked nodes l at its layer. We prove this by performing induction on the length of the current path. Each advance step at node i adds an arc, say (i, j) , of smallest arc–order value among admissible arcs in A(i). Node j then becomes the new current node and PATH(i) $\cup$ {(i,j)} is the new current path. Clearly, the new current path is lexicographically smaller than the path PATH(i) $\cup$ {(i, l)} for each (i, l) $\in$ A(i) - {(i, j)}. By the induction hypothesis, it is also lexicographically smaller than any path from the source to any node in layer k + 1 that does not pass through node i in layer k. This shows that advance step preserves the induction hypothesis. Each retreat step reduces the length of the current path and obviously does not affect the validity of our claim.

The algorithm DD1 is very much similar to Dinic's algorithm except that it operates on the residual network instead of the layered network. The algorithm DD1, too, identifies the least lexicographic paths from source to sink and augments flows in these paths. To show this, we may use the following induction hypothesis: the current path to the current node i is the least lexicographic path and current node is the node i satisfying PATH(i) $\{$ PATH(l) among all nodes l with distance label equal to d(i). It can be easily verified that the advance and retreat steps keep this hypothesis satisfied.

The algorithm DD2 does not necessarily enumerate the same augmenting paths as other algorithms if it is not initialized properly. This algorithm identifies the first augmenting path by breadth first search starting from sink, whereas Edmonds–Karp algorithm performs breadth first search from source to find the first augmenting path. These two paths need not be same, even if arc in A(i) are scanned in increasing arc-order values. However, if the algorithm DD2 is initiated with a tree T for which every arc $(i, j) \in T$ is an admissible arc with smallest arc order value in A(i), then it goes through the same sequence of iterations as other algorithms. Such a tree can be obtained as follows: Perform breadth first search from sink to initialize the distance labels. Mark the sink node, while all other nodes are unmarked. Then, iteratively, select an unmarked node i, find the admissible arc (i, j) of smallest arc-order value, mark node i and make node j the predecessor of node i. Repeat this basic step until all nodes are marked.

Thus the algorithm starts with a tree whose every arc (i, j) is an admissible arc of least arc–order value in A(i). This property is preserved by the algorithm since during the update-tree step, arcs in A(i) are scanned in the increasing arc-order values. It is easy to show, again using induction, that the unique path in the tree from source to sink is the least lexicographic path. Let P be the tree path and P* be the least lexicographic path from source to sink . Clearly, $|P| = |P^*|$. Assume inductively that the first k arcs of P and P* are same. Let the k-th arc be (p, i), and let (i, j) be the admissible arc of smallest arc-order value in A(i). By assumption, (i, j) is in P. Also, (i, j) must be in P*, else it would contradict that P* is the least lexicographic path. Hence the first k+1 arcs of P and P* are same.

The results given above establish the equivalence of the distance-directed algorithms with the algorithms of Edmonds-Karp and Dinic in the sense of augmenting path enumeration. The algorithms, however, differ substantially in several computational aspects, such as the manner in which these paths are obtained or the time needed to enumerate paths. The Edmonds-Karp algorithm requires O(m) time to enumerate a path, whereas other algorithms require O(n) time on average. Dinic's algorithm maintains layered networks whereas the distance-directed algorithms maintain distance labels. Dinic's algorithm runs better in worst case when applied to unit capacity networks. However, it is not clear to us whether the distance-directed algorithms applied as such will result in improved algorithms for these networks. As we show in Section 7, that to obtain improved distance-directed algorithms for unit capacity networks, we need to resort to a two-phase technique.

6. Scaling Versions of Distance-Directed Algorithms

In this section, we consider scaling versions of the algorithms DD1 and DD2. We show that using a scaling technique, the complexity of the algorithms DD1 and DD2 can be improved from $O(n^2m)$ to $O(nm \log U)$. The resulting algorithms are, in spirit, similar to Gabow's (1985) scaling algorithm for the maximum flow problem, but different in several computational aspects.

The basic idea behind the scaling algorithms is as follows. Let a **Δ-optimal flow** denote a flow which does not admit any augmenting path of capacity at least Δ. Clearly, a zero flow is Δ-optimal for every $\Delta > U$, and an integral Δ–optimal flow for any $\Delta < 1$ is a maximum flow. The scaling algorithm starts with a Δ-optimal flow with $\Delta = 2^{\lceil \log U \rceil}$ and at each

iteration replaces Δ by $\Delta/2$ until $\Delta < 1$. Given a Δ-optimal flow, the scaling algorithm obtains a $\Delta/2$-optimal flow by augmenting flows in paths of capacity at least $\Delta/2$, until no more augmenting paths exist. (Gabow's algorithm sent flow of value exactly $\Delta/2$.) An iteration during which Δ remains unchanged is called a **scaling iteration**. Clearly, there are $\lceil \log U \rceil + 1$ scaling iterations.

We now show that each scaling iteration can discover at most 2m augmenting paths. Consider a Δ–optimal flow. Let $v(\Delta)$ denote the current flow value and v^* denote the maximum flow value. Further, let X be the set of nodes that admit augmenting paths from source of capacity at least Δ. Since the flow is Δ-optimal, $t \notin X$ and the cutset $(X, N - X)$ separates the source from the sink. The residual capacity of every arc in the cutset $(X, N - X)$ is less than Δ and there are at most m arcs in the cutset. Consequently, $v^* - v(\Delta) \leq m\Delta$. Each augmenting path in the next scaling iteration increases the flow value by at least $\Delta/2$ units and this can happen at most 2m times. (A more careful analysis bounds the number of augmentations at m.)

The scaling algorithm employs the algorithm DD1 or DD2 as a subroutine to transform a Δ–optimal flow into a $\Delta/2$–optimal flow. The only change that is called for in the algorithms DD1 and DD2 is that the arcs with residual capacities less than $\Delta/2$ are considered as "non-existent" (they are treated the same as arcs with no capacity). The distance labels too are defined with respect to arcs with $r_{ij} \geq \Delta/2$. The bottleneck operation in the algorithms DD1 or DD2 is the augmentation time. Since there are at most 2m augmenting paths per scaling iteration, the augmentation time is O(nm).

Thus each scaling iteration requires O(nm) time and the algorithm runs in O(nm log U) time. We summarize our discussion with the following result.

Theorem 4. Scaling versions of algorithms DD1 and DD2 run in O(nm log U) time. ❑

The scaling approach can also be regarded as an improved version of Edmonds and Karp (1972) maximum capacity augmentation algorithm. Edmonds and Karp showed that if flow is augmented along paths with maximum augmenting capacity, then O(m log U) augmenting paths are enumerated. We have shown above that if flow is augmented along paths with **sufficiently large** capacity, but **not** necessarily maximum, we again generate O(m log U) augmenting paths. Whereas determining a maximum capacity augmenting path requires $O(m \log_n U)$ time (Gabow 1985), determining a path with sufficiently large capacity requires O(m) time using simple breadth first search , or O(n) time on average in the algorithms DD1 or DD2.

We now show that the complexity of the scaling algorithm can be improved when a subset of arcs are capacitated while other arcs have infinite capacities. It is, however, assumed that the maximum flow value is finite. This problem arises, for instance, while finding a feasible flow in a sparse uncapacitated transportation problem. Determining statistical security of a tabular data can also be reduced to this problem (see Gusfield 1984).

The speed-up of the algorithm is obtained by using a scaling factor higher than 2 depending upon the number of capacitated arcs. We use a scale factor of $\beta = \max\{2, \lceil m/p \rceil\}$, where p is the number of capacitated arcs. In the k-th scaling iteration, the scaling algorithm identifies augmenting

paths of capacity at least $\lceil U/\beta^{k-1} \rceil$ and augments flow in these paths. After $\log_\beta U + 1$ scaling iterations, the algorithm obtains a maximum flow. The following theorem analyses the complexity of the modified scaling algorithm.

Theorem 5. Let $\beta = \lceil m/p \rceil$ and assume $\beta \geq 2$. Then the modified scaling algorithm runs in $O(nm \log_\beta U)$ time.

Proof. Consider a Δ-optimal flow. Let X be the set of nodes reachable from s by augmenting paths of capacity at least Δ. The number of arcs in the cutset $(X, N - X)$ are no more than p, since no arc with infinite capacity can be in the cutset. Thus the total residual capacity of arcs in this cutset is at most $p\Delta$. Each augmenting path in the next scaling iteration decreases the residual capacity of this cutset by at least $\Delta/\beta \geq \frac{p\Delta}{m}$ and this can happen at most m times. Each scaling iteration, therefore, takes O(nm) time and the algorithm runs in $O(nm \log_\beta U)$ time. ❑

The scaling algorithm yields an almost optimum algorithm for a class of feasibility problems in uncapacitated sparse transportation networks. For this problem if $U = O(n^{(O(1))})$ and $m = (n^{1+\varepsilon})$ for some $\varepsilon > 0$, the algorithm takes $O(nm \log_{n^\varepsilon} n) = O(nm)$ time. This time bound matches the best known time bound due to Ahuja and Orlin (1987) for solving this class of problems.

7. Application to Unit Capacity Networks

A network is called a **unit capacity network** if the capacity of every arc in the network is one. A unit capacity network is called a **simple network** if either the indegree or outdegree of each node is at most one. Determining maximum flows in unit capacity networks is important in many situations including that of determining connectivity of a network

(Even and Tarjan 1975) or solving the bipartite matching problem. In this section, we show that the distance-directed algorithms can be used to determine the maximum flow in unit capacity networks in $O(\min(m^{1/2}, n^{2/3})\, m)$ time, and in simple network in $O(n^{1/2}\, m)$ time. These bounds are the same as those obtained by Even and Tarjan (1975) using layered networks. Our analysis is, however, simpler and the algorithms are likely to perform better in practice, too. The analysis here is not particular to distance-directed algorithms and can be used to prove the time bounds for Dinic's algorithm as applied to unit capacity networks.

Our algorithms are two-phase algorithms. In the first phase, we apply either the algorithm DD1 or the algorithm DD2 with the modification that we stop examining a node whenever its distance label is greater than or equal to d^* for a suitable choice of d^*. This phase sends most of the flow from source to sink. In the second phase, we successively identify augmenting paths from source to sink using breadth first search and augment flows in these paths.

We first note that the complexity of the first phase is $O(d^*m)$. The bottleneck operation in the algorithms DD1 and DD2 is the flow augmentation time. For the unit capacity networks, flow augmentation on an arc immediately saturates that arc. Consequently, no arc is examined more than $O(d^*)$ during flow augmentations, giving a bound of $O(d^*m)$. Other operations too take $O(d^*m)$ time, since we stop examining a node whenever its distance label exceeds d^*. Let v' denote the flow value at the end of first phase and v^* denote the maximum flow value. We will choose d^* so that $v^* - v' = O(d^*)$. This choice implies that the second phase identifies $O(d^*)$

augmenting paths and thus runs in O(d*m) time. This time is the same as that of the first phase.

We first consider the unit capacity (non-simple) networks. For this class of networks $d^* = \min\{2n^{2/3}, m^{1/2}\}$. Let us first examine the case when $d^* = 2n^{2/3}$. Let $V_k := \{i \in N: d(i) = k\}$, $k = 0, 1, \ldots, d^*$. We refer to V_k as the set of nodes in the k-th layer. Clearly, $d(s) \geq d^*$. Note that $\sum_{k=1}^{d^*} |V_k| \leq n - 1$, since the sink does not belong to the considered layers. It follows from the conditions C1 and C2 that there are no arcs (i, j) in the residual network with $i \in V_k$, $j \in V_l$ and $k > l + 1$. Hence, the set of arcs going from V_k to V_{k-1} for each k, $1 \leq k \leq d^*$, form a cutset in the residual network.

We claim that there must exist some layer k, for some $1 \leq k \leq d^*$ such that $|V_k| \leq n^{1/3}$ and $|V_{k-1}| \leq n^{1/3}$. For, if not, then every alternate layer must have nodes no less than $n^{1/3}$ and the total number of nodes in the layers $1, \ldots, d^*$ would be $n^{1/3} \cdot d^*/2 = n$ leading to a contradiction. The residual capacity of the cutset defined by the layers V_k and V_{k-1} is at most $|V_k| \cdot |V_{k-1}| \leq n^{2/3}$. Thus, $v^* - v' \leq n^{2/3}$.

Next, consider the case when $d^* = m^{1/2}$. Again, there must exist a cutset defined by the layers V_k, V_{k-1} for some k, with $1 \leq k \leq d^*$, whose residual capacity is no more than $m^{1/2}$. This follows from the fact that there are $m^{1/2}$ cutsets and each arc contributes the residual capacity of at most one unit to at most one such cutset. Thus, $v^* - v' \leq m^{1/2}$. We have shown the following result.

Theorem 6. The two-phase algorithm obtains maximum flow in a unit capacity network in $O(\min(m^{1/2}, 2n^{2/3})\, m)$ time. □

A stronger result can be proved for the simple networks.

Theorem 7. The two-phase algorithm obtains maximum flow in a simple network is $O(n^{1/2} m)$ time.

Proof. Apply the algorithm with $d^* = n^{1/2}$ and consider the resulting layers of nodes V_k, $1 \le k \le n^{1/2}$. Let the layer V_p have the smallest number of nodes. Then $|V_p| \le n^{1/2}$, else it would imply that the number of nodes in all layers are strictly greater than n. Since each node has either indegree or outdegree at most one, it allows at most one unit of additional flow from source to sink. As all additional flow must pass through the layer V_p, we get $v^* - v' \le n^{1/2}$. The theorem now follows. □

8. Application to Parametric Maximum Flows

In this section, we consider a special, yet practically important, class of parametric maximum flow problems. In this problem, some or all arcs emanating from the source node have capacities that are increasing linear functions of a parameter λ, while other arcs have fixed capacities. We wish to determine the maximum flow in the network for all values of $\lambda \in (0, \infty)$. We refer to this problem as the **Source Parametric Maximum Flow (SPMF)** problem. This problem arises in information retrieval from a large shared database (Eisner and Severance 1976), program module decomposition (Stone 1977), and scheduling transmission in a network (Itai and Rodeh 1985). (See Gallo et. al 1987 for additional application of this problem. They consider the problem in which arcs directed to the sink are also parametrized.)

The SPMF problem has been solved by Itai and Rodeh (1985) in $O(kn^2m)$ time using the proportional augmentation algorithm, where k is

the number of arcs with parametric capacities. Proportional augmentation is a technique for augmenting flows on a number of paths from different origins to the same destination in a proportionate manner until one of the arcs in these paths gets blocked. Itai and Rodeh's algorithm proceeds by creating at most kn layered networks and constructing blocking flows in these networks by proportional augmentation. Gusfield (1986) has reduced the SPMF problem to solving at most k maximum flow problem. Recently, Gallo, Grigoriadis and Tarjan (1987) have solved this problem in O(nm $\log(n^2/m)$ time. In this section, we show that Itai and Rodeh's algorithm can be improved by a factor of k if we use distance labels instead of layered networks. This result is based on incorporating proportional augmentation technique in the algorithms DD1 and DD2. Our algorithms run in $O(n^2m)$. This bound is not as attractive as that of Gallo, Grigoriadis and Tarjan (1987), but we anticipate our algorithms to be very useful in practice.

We refer to an arc $(s, j) \in A$ as a source arc. Let the capacity of each arc $(i, j) \in A$ be $u_{ij} + \lambda u^*_{ij}$ with $u^*_{ij} > 0$ for some source arcs only. Initially, a maximum flow problem is solved with u_{ij}'s as arc capacities. Let $(W, \bar{W})$ be the minimum cutset for which $|\bar{W}|$ is least among all minimum cutsets. If $(W, \bar{W})$ does not contain any source arc (s, j) with $u^*_{sj} > 0$, then it remains a minimum cutset for all $\lambda \in (0, \infty)$. Otherwise, the capacity of the cutset $(W, \bar{W})$ increases as λ increases and more flow can be sent from source to sink. Let $\{j \in \bar{W} : u^*_{sj} > 0\}$. The slope of the maximum flow function at $\lambda = 0$ is

$$\sum_{j \in \bar{S}} c^*_{sj}.$$

We now describe a method for the SPMF problem based on the algorithm DD2. First, the source node and all of the source arcs are deleted from the networks. Then, a breadth first search of the residual network is performed starting at the sink node to initialize the distance labels and to form the tree of shortest paths T spanning the nodes in $\bar{W}$. The algorithm repeatedly performs the proportional augmentation step and the update-tree step. The update-tree step is same as described in the algorithm DD2, whereas the proportional augmentation step is a generalization of the augment step in the following manner. The proportional augmentation sends αu^*_{sj} flow from each $j \in \bar{S}$ to the sink on the tree arcs, and the value of α is chosen so that at least one tree arc becomes saturated. A detailed description of this step is given below.

proportional augmentation.

(a) Set $\pi_j := u^*_{sj}$, for each $j \in \bar{S}$ and 0, otherwise. Set $\beta_{ij} := 0$, for each $(i,j) \in T$, and set $\bar{T} := T$.

(b) If the sub-tree $\bar{T} = \emptyset$, then go to (c); otherwise select a leaf node i of $\bar{T}$. Let j: = pred(i). Set $\beta_{ij} := \pi_i$ and update $\pi_j := \pi_i + \pi_j$. Delete (i, j) from the sub-tree $\bar{T}$ and repeat (b).

(c) Let $\Delta = \min_{(i,j) \in T} \left\{ \frac{r_{ij}}{\beta_{ij}} : \beta_{ij} > 0 \right\}$. For each $(i, j) \in T$, subtract $\beta_{ij} \Delta$ from r_{ij} and add $\beta_{ij} \Delta$ to r_{ij}. Let B denote the set of arcs whose residual capacities are reduced to zero by flow augmentation.

The above step can be easily implemented in O(n) time. The algorithm performs a update-tree step after every proportional augmentation. The

nodes whose distance labels exceed $n - 1$ are not considered further. The set $\bar{S}$ and the cutset $(W, \bar{W})$ may have to be updated if distance label of some node in $\bar{S}$ exceeds $n - 1$. This updating is done at most n times since this reduces $|\bar{S}|$ by at least one unit. The algorithm terminates when $|\bar{S}| = 0$. The slope of the maximum flow function at any step is given by

$$\sum_{j \in \bar{S}} u^*_{sj}.$$

This algorithm is same as the algorithm DD2 except that the simple augmentation of flow is replaced by a proportional augmentations. Both types of augmentation take O(n) time and saturate at least one arc in the residual network. The complexity of the proportional flow algorithm is thus $O(n^2m)$. We have obtained the following result.

Theorem 8. The proportional flow algorithm utilizing distance labels solves the SPMF problem in $O(n^2m)$ time.

We can also use the algorithm DD1 instead of DD2 in the proportional augmentation algorithm in the following manner. Initially, the sink node is marked while all other nodes are unmarked. The algorithm picks up an unmarked node $i \in \bar{S}$ and repeatedly performs advance and retreat steps until either a marked node is reached or $d(i) \geq n - 1$. In the former case, all nodes in the path from i to the marked nodes are marked, and in the later case node i is deleted from $\bar{S}$. This process is repeated until all nodes in $\bar{S}$ are marked. The marked nodes now define a subtree T spanning nodes in $\bar{S}$ and some other nodes on which the proportional augmentation, described above, is performed. All nodes except the sink are

unmarked and this procedure is repeated until $\bar{S} = \emptyset$. The complexity of this algorithm can be shown to be $O(n^2m)$.

9. Summary and Conclusions

We have considered several distance-directed augmentation procedures for the maximum flow problem. In a number of cases, we have shown that the distance-directed procedures can replace layered networks in augmentation based algorithm. In this way, we have improved Edmonds-Karp maximum flow algorithm, as well as their "greatest augmenting path algorithm." We have also shown some important connections between the Edmonds-Karp algorithm and Dinic's algorithm. In particular, they enumerate the same augmenting paths and in the same order.

In addition to improving the worst case complexity of several algorithms, distance-directed approaches are also very flexible computational tools. One can use them to create layered networks implicitly, and always maintain the exact layered network as in DD2. Alternatively, one can use them to create only part of the layered network as in DD1. Moreover, by maintaining additional information such as the number of nodes whose distance label is k for each k (or possibly maintaining the sum of the residual capacities of admissible arcs directed from a node at distance k) one can terminate the algorithm earlier, and possibly fine-tune the algorithm. Finally, this approach does not require the creation of the layered network, which in and of itself is a time and space consuming task.

Finally, we believe that the distance-directed approaches here nicely complement the distance-directed approaches of Goldberg (1985), Goldberg and Tarjan (1986), and Ahuja and Orlin (1987a). As such, they may serve a pedagogical role in the development of network flow theory.

Acknowledgments

This research was supported in part by Presidential Young Investigator Grant 8451517-ECS of the National Science Foundation, and by grants from Analog Devices, Apple Computer Inc., and Prime Computer.

References

Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA. 1974.

Ahuja, R.K., "Algorithms for the Minimax Transportation Problem", *Naval Research Logistics Quarterly*, 33(1986) 725-739.

Ahuja, R.K. and J.B. Orlin, "A Fast and Simple Algorithm for the Maximum Flow Problem", Sloan W.P. No. 1905-87, Sloan School of Management, M.I.T., Cambridge, MA. 1987a.

Ahuja, R.K. and J.B. Orlin, "Numerical Investigation with New Maximum Flow Algorithms," Work in Progress, Sloan School of Management, M.I.T., Cambridge, MA. 1987b.

Bland, R.G. 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.

Cherkasky,R.V., "Algorithms of Construction of Maximal Flow in Networks with Complexity $O(n^2\sqrt{m})$ operations", *Mathematical Methods of Solution of Economical Problems*, 7(1977) 112-125.

T. Cheung, "Computational Comparison of Eight Methods for the Maximum Network Flow Problem", *ACM Transactions on Mathematical Software*, 6(1980) 1-16.

E.A. Dinic, "Algorithms for Solution of a Problem of Maximum Flow in Networks with Power Estimation", *Soviet Mathematics Doklady* 11(1980) 1277-1280.

Edmonds, J. and R.M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of ACM* 19(1972), 248-264.

Eisner, M.J. and D.G. Severance, "Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases", *Journal of ACM* 23(1978), 619-635.

Even,S. and R.E. Tarjan, "Network Flow and Testing Graph Connectivity," *SIAM Journal of Computing*, 4(1975), 507-518.

Ford,L.R., Jr., and D.R. Fulkerson, "Maximal Flow Through a Network", *Canadian Journal of Mathematics* 8(1956), 399-404.

Fulkerson, D.R., "A Network Flow Computation for Project Cost Curve", *Management Science*, 7(1961) 167-178.

Gabow, H.N., "Scaling Algorithms for Network Problems", *Journal of Computer and Systems Sciences* 31(1985), 148-168.

Gabow, H.N. and R.E. Tarjan, "Faster Scaling Algorithms for Network Problems," Research Report, Computer Science Dept., Princeton University, Princeton, NJ, 1987.

Galil, Z., "An $O(n^{5/3} m^{2/3})$ Algorithms for the Maximal Flow Problem", *Acta Informatica* 14(1980), 221-242.

Galil, Z. and A. Naamad, "An $O(nm \log^2 n)$ Algorithm for the Maximal Flow Problem", *Journal of Computer and Systems Sciences* 21 (1980) 203-217.

Gallo, G., M.D. Grigoriadis and R.E. Tarjan, "A Fast Parametric Flow Algorithm," Research Report, Dept. of Computer Science, Princeton University, Princeton, NJ 1987.

Glover, F., D. Klingman, J. Mote and D. Whitman, "Comprehensive Computer Evaluation and Enhancement of Maximum Flow Algorithms," Research Report CCS356, Center for Cybernetic Studies, University of Texas, Austin, 1979.

Glover, F., D. Klingman, J. Mote and D. Whitman, "A Primal Simplex Variant for the Maximum Flow Problem," *Naval Research Logistics Quarterly* 31(1984) 41-61.

Goldberg, A.V., "A New Approach to the Maximum Flow Problem," Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., Cambridge, MA., 1985.

Goldberg, A.V. and R.E. Tarjan, "A New Approach to the Maximum Flow Problem", *Proceedings of the Eighth Annual ACM Symposium on the Theory of Computing*,1986.

Goldberg, A.V. and R.E. Tarjan, "Solving Minimum Cost Flow Problem by Successive Approximations", *Proceedings of the Ninth Annual ACM Symposium on the Theory of Computing*,1987.

Gusfield, D., "A Graph Theoretic Approach to Statistical Data Security", Technical Report Yale/DCS/TR-326, Department of Computer Science, Yale University, 1984.

Gusfield, D., "On Scheduling Transmission in a Network", Yale/DCS/TR-481, Department of Computer Science, Yale University, 1986.

Hu, T.C., "Optimum Communication Spanning Trees", *SIAM Journal of Computing* 3 (1974), 188-195.

Imai, H., "On the Practical Efficiency of Various Maximum Flow Algorithms", *Journal of Operations Research Society of Japan* 26(1983), 61-82.

Itai, A. and M. Rodeh, "Scheduling Transmission in a Network," *Journal of Algorithms* 6 (1985), 409-429.

Karzanov, A.V., "Determining the Maximal Flow in a Network by the Method of Preflows", *Soviet Mathematics Doklady* 15 (1974), 434-437.

Kelley, J.R., Jr., "Critical Path Planning and Scheduling: Mathematical Basis," *Operations Research* 9(1961), 296-320.

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

Malhotra, V.M., M.P. Kumar and S.N. Maheshwari, "An $O(n^3)$ Algorithm for Finding Maximum Flows in Networks", *Information Processing Letters* 7(1978), 277-278.

Mineka, E., "Parametric Network Flows", *Operations Research* 20 (1972), 1162-1167.

Rock, H., "Scaling Techniques for Minimal Cost Network Flows", *Discrete Structures and Algorithms*, Ed. V. Page, Carl Hanser, München (1980), 181-191.

Sleator, D.D. and R.E. Tarjan, "A Data Structure for Dynamic Trees", *Journal of Computer and System Sciences* 24 (1983), 362-391.

Stone, H.S., "Critical Load Factor in Two-Processor Distributed Systems", *IEEE Transactions on Software Engineering* SE-4 (1978), 254-258.

Tardos, E., "A Strongly Polynomial Minimum Cost Circulation Algorithm", *Combinatorica 5* (1985), 247-255.

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