AN $0(n^2m^{1/2})$ DISTRIBUTED MAX-FLOW ALGORITHM

John M. Marberg

January 1987
CSD-870002

An $O(n^2m^{1/2})$ Distributed Max-Flow Algorithm †

John M. Marberg
Eli Gafni

Computer Science Department
University of California
Los Angeles, CA 90024

Technical Report CSD-870002
January 1987

Abstract

A distributed algorithm for finding maximum flow in a network is presented. The message complexity of the algorithm is $O(n^2m^{1/2})$, where n and m denote the number of nodes and the number of links in the network. This is an improvement by a factor of $\frac{n}{m^{1/2}}$ upon the previous best complexity.

† Work supported by NSF grant DCR-84-51396 and by IBM grant D840622.

1. Introduction

Consider an asynchronous point-to-point communication network. Each node is an independent processor. The processors communicate by sending messages over the links of the network. One approach to the design of algorithms for this computation model is to take a centralized (single-processor) algorithm for the problem at hand and adapt it to the distributed environment. Inherent parallel properties of the algorithm can then be fully exploited, whereas the implicit centralized control can be replaced with synchronization. It this paper we present a distributed algorithm for maximum flow that is based on Cherkasky's centralized method [3, 9].

Finding maximum flow in a network is a well-known problem in combinatorial optimization. It has been extensively studied, resulting in a plethora of centralized algorithms (see [11] for the most current survey of results). In recent years, a number of distributed algorithms for the problem have been developed. Table 1 summarizes these works. Message and time complexities are expressed in terms of the parameters n and m, denoting, respectively, the number of nodes and the number of links in the network.

Algorithm	Messages	Time	Centralized Method
Cheung [4]	$O(n^2m^2)$	$O(n^3m)$	Edmonds & Karp [6]
Segall [13]	$O(nm^2)$	$O(n^2m)$	Dinic [5]
Marberg & Gafni [12]	$O(n^3)$	$O(n^3)$	Shiloach & Vishkin [15]
Awerbuch [1]	$O(n^3)$	$O(n^2 \log n)$	Shiloach & Vishkin [15]
Goldberg & Tarjan [11]	$O(n^3)$	$O(n^2 \log n)$	Goldberg & Tarjan [11]
This Paper	$O(n^2m^{1/2})$	$O(n^2m^{1/2})$	Cherkasky [9]

Table 1. Survey of Distributed Max-Flow Algorithms

All the algorithms listed in Table 1 are based on the approach mentioned above, that is, they are distributed implementations of some centralized method. The algorithm presented in this paper is the most efficient in the number of messages, improving by a factor of $\frac{n}{m^{1/2}}$ the previous best result.

The distributed max-flow algorithms of Awerbuch [1] and Goldberg and Tarjan [11] have better time complexity than our algorithm. This is since the methods upon which these algorithms are based are highly parallel-oriented. Cherkasky's method, on the other hand, has a critical part which seems to be inherently sequential, hence the time complexity of our algorithm is the same as the message complexity.

The max-flow algorithms in [1, 11] were actually designed for a synchronous network, then superimposed with a global synchronization mechanism [2] that simulates each synchronous step on the asynchronous network. Interestingly, we have not been able to employ this strategy in our algorithm without compromising the complexity. Instead, we use local synchronization mechanisms that are tailored for the specific needs of the algorithm.

The remainder of the paper is organized as follows. Section 2 defines the max-flow problem and our notation. Section 3 describes Cherkasky's algorithm. The distributed implementation is presented in Section 4. Concluding remarks are given in Section 5.

2. The Max-Flow Problem

A *flow network* consists of a connected directed graph with no self loops or parallel edges, and two distinguished nodes called *source* and *target*, denoted s and t, respectively. Each edge $e=v \rightarrow w$ is associated with a positive value $c(e)$ called the *capacity* of the edge, and a non-negative value $f(e)$ called the *flow* of the edge. The flow is outgoing from v and incoming into w.

Let V denote the set of nodes, and E the set of edges of the graph. Also, let $INFLOW(v)$ be the sum of all flows incoming into node v, and $OUTFLOW(v)$ the sum of all flows outgoing from v. A legal flow in the network must satisfy the following conditions.

1. $0 \le f(e) \le c(e)$ for all $e \in E$.
2. $INFLOW(v) = OUTFLOW(v)$ for all $v \in V - \{s, t\}$.

The value of the network flow is defined as the quantity $OUTFLOW(s) - INFLOW(s)$. The largest possible flow value is called the *maximum flow* of the network. The *max−flow* problem is to assign flows to the edges of the network such that maximum flow is achieved.

A flow network is *layered* if it has the following properties.

1. V is the union of disjoint sets $L_0, L_1, \ldots, L_k$ called *layers*.
2. $L_0=\{s\}$; $L_k=\{t\}$.
3. For any edge $e=v \rightarrow w \in E$, $v \in L_i$ and $w \in L_{i+1}$ for some $0 \le i \le k-1$.
4. Each node $v \in V$ is on a directed path from s to t.

Clearly, a layered network is acyclic. We say that layer L_i is *downstream* form any layer L_j, $j<i$. Similarly, L_i is *upstream* form any layer L_j, $j>i$.

We also use the following terminology. The *residual capacity* of an edge e is the quantity $c(e)-f(e)$. If the residual capacity is 0, e is said to be *saturated.* A node v is *closed* if every directed path from v to t contains a saturated edge; otherwise v is *open.* An edge $e=v \rightarrow w$ is *closed* if w is closed or if e is saturated; otherwise e is *open.* A flow in the network is *maximal* if s is closed (note that a maximal flow is not necessarily maximum).

An *augmenting path* from node v to node w is a directed path from v to w in which all edges are open. The capacity of the augmenting path is the smallest residual capacity of any of its edges.

During the execution of the max-flow algorithm, condition 2 in the definition of legal flow may be temporary violated at some nodes. Specifically, it can be that $INFLOW(v)>OUTFLOW(v)$. The quantity $INFLOW(v)-OUTFLOW(v)$ is called the *excess flow* of v. When a node other than s and t has nonzero excess flow, it is *unbalanced*; otherwise it is *balanced.* Nodes s and t are always considered balanced, regardless of any excess flow.

3. Cherkasky's Algorithm

Our description of Cherkasky's algorithm [3] follows Galil [9]. The algorithm is based on a general scheme by Dinic [5], in which maximum flow is achieved by iteratively constructing a layered network from the original network, finding maximal flow in the layered network, then updating the flows in the original network. We assume the reader is familiar with Dinic's scheme (see also [7]), hence our discussion will focus only on finding maximal flow in a layered network.

Some of the layers in the layered network are designated by the algorithm as *special.* We will show later how to select the special layers. At this point it is only important to know that the first and last layers (L_0 and L_k) are special, and that the distance between any two consecutive special layers is upper-bounded by the parameter x.

Two consecutive special layers and all the non-special layers between them comprise a *super layer.* We denote the super layers by $SL_1, SL_2, \ldots, SL_{k'}$, in order of their distance from the source. The special layers located at the upstream and downstream ends of SL_i are called the *rear layer* and the *front layer* of SL_i, denoted RL_i and FL_i, respectively. Notice that by definition $FL_i = RL_{i+1}$.

At the beginning of the maximal flow computation, all edge-flows in the layered network are 0, and node s contains a (dummy) excess flow of $\sum\limits_{s \to v} c'(s \to v)$ units, where $c'(e)$ denotes the capacity of edge e in the layered network.

The algorithm uses two procedures, called $PUSH(i)$ and $BALANCE(i)$. $PUSH(i)$ moves as much excess flow as possible from open nodes in RL_i, via SL_i, to open nodes in FL_i. When $PUSH(i)$ is invoked, all the nodes in layers downstream from RL_i are balanced. Upon termination of the procedure, all the nodes in RL_i that are still unbalanced are closed. Also, some open nodes in FL_i may become unbalanced.

$BALANCE(i)$ attempts to reroute as much excess flow as possible from closed nodes in FL_i to open nodes in that layer, via alternative paths in SL_i. Any remaining excess flow of closed nodes in FL_i is then returned to closed nodes in RL_i. When $BALANCE(i)$ is invoked, all nodes in layers downstream from FL_i are balanced, and all unbalanced nodes in the network are closed. Upon termination of the procedure, all closed nodes in FL_i are balanced. Also, some open nodes in FL_i and some closed nodes in RL_i may become unbalanced.

The general scheme for the maximal-flow computation is shown in Figure 1. It should be noted that the scheme differs slightly from that of Cherkasky. Specifically, our scheme iterates in the loop labeled PLOOP until $i = k'$, whereas in the original scheme the loop is exited as soon as the current $PUSH(i)$ fails to push any flow to FL_i. In other words, in our version, some calls to $PUSH$ are redundant because there is no excess flow to push. The reason for using the modified scheme is that it lends itself easier to distributed implementation. However, there is no change in the complexity of the centralized implementation.

```
        $i$:=1;
PLOOP:  **while** $i \le k'$ **do**
                    $PUSH(i)$;
                    $i$:=$i$+1
                **end**;
        **if** there are no unbalanced nodes
        **then stop** (* flow is maximal *)
        **else do**
                $i$:=max$\{ j \mid FL_j$ contains unbalanced nodes $\}$;
                $BALANCE(i)$;
                $i$:=$i$+1;
                **goto** PLOOP
            **end**
```

Figure 1. Finding Maximal Flow in a Layered Network

$PUSH(i)$ works by iteratively finding an augmenting path from some unbalanced open node in RL_i via internal layers of SL_i to an open node in FL_i, then increasing the flow on each edge of the path by the capacity of the path. This continues until either all excess flow of nodes in RL_i has been pushed to FL_i, or there are no more augmenting paths. Augmenting paths are constructed using depth-first-search (DFS), similar to Dinic's algorithm [5]. Additional implementation details of $PUSH(i)$ are given in [9].

Let us denote the layers of SL_i as $RL_i = L_{p_i}, L_{p_i+1}, \ldots, L_{q_i-1}, L_{q_i} = FL_i$. $BALANCE(i)$ starts by returning the excess flow from L_{q_i} to L_{q_i-1}, thereby balancing all the nodes in L_q and unbalancing some nodes in L_{q_i-1}. On which edges flow is to be reduced, and by how much, is determined from flow-history records that are maintained for each node (see [9]). It should be noted that flow reduction does not result in reopening any previously closed node or edge.

After all excess flow is returned one layer backward, an attempt is made to reroute the flow to open nodes in FL_i by means of augmenting paths from L_{q_i-1}. Excess flow in L_{q_i-1} that cannot be rerouted is then returned to L_{q_i-2}, from where rerouting to FL_i via augmenting paths is again attempted. The process continues iteratively, going backwards layer after layer, until all the excess flow of closed nodes in FL_i has been either rerouted to open nodes in FL_i or returned to some (closed) nodes in RL_i. We call the task of rerouting flow from a given internal layer to the front layer a *micropush*, because of the resemblance to *PUSH*. Further implementation details of $BALANCE(i)$ are given in [9].

We now calculate the time complexity of the maximal flow computation. Recall that x is an upper bound on the distance between two consecutive special layers. Also, let us denote by f the total number of nodes in all special layers, and by f_i the number of nodes in special layer FL_i. The following claims have been proved in [9].

Claim 1. For each given i, $BALANCE(i)$ is invoked at most f_i times. Overall, *BALANCE* is invoked at most $f-2$ times. ■

Claim 2. The loop labeled PLOOP is entered at most $f-1$ times. Hence, *PUSH* is invoked a total of $O(f^2)$ times. ■

Claim 3. The total number of elementary steps involved in finding augmenting paths and pushing flow forward is $O(f^2x+mx)$. ■

Claim 4. The total number of elementary steps involved in reducing (returning) flow on edges is $O(mx)$. ■

From these claims, the time complexity of finding maximal flow in a layered network is $T=O(f^2x+mx)$. We now describe a method to select the special layers so that T is minimized. Our method is different from Cherkasky's original, the reason being that the latter does not have an efficient distributed implementation.

For $0 \le i \le x-1$, let $V_i = \bigcup_{j=i \bmod x} L_j$. In other words, the set V_i is the union of the layers that are at distance $0, x, 2x, 3x, \cdots$ from layer L_i. Clearly, there exists some $0 \le i' \le x-1$ such that $|V_{i'}| \le \frac{n}{x}$. The special layers are those that comprise $V_{i'}$ (plus, of course, layers L_0 and L_k). Given any reasonable representation of the layered network, i' can be easily determined in $O(n)$ time, i.e., at no added cost.

Hence, we have $f \leq |V_{i'}|+2=O(\frac{n}{x})$. Substituting for f in the complexity, we get $T=O(\frac{n^2}{x}+mx)$. This expression is minimized when $x=\lfloor \frac{n}{m^{1/2}} \rfloor$. Consequently, $T=O(nm^{1/2})$.

Dinic's scheme uses at most $n-1$ iterations of layered network construction followed by finding maximal flow. A layered network can be constructed in $O(m)$ time using breadth-first-search (BFS). The total time complexity of Cherkasky's max-flow algorithm is therefore $O(n^2m^{1/2})$.

4. Distributed Implementation

4.1. Preliminaries

The distributed computation model we use is an asynchronous point-to-point communication network. Each node of the network is an independent processor. Each link provides a bidirectional communication channel between the two nodes it connects. Processors communicate by sending messages over the links. Messages incur finite but unbounded transmission delay.

There are two performance measures: message complexity — the number of message transmissions during the computation; and time complexity — the time it takes to perform the computation. In calculating the complexity we assume that each message arrives within one unit of time after it has been sent, and that local computation time is negligible. Also, each message is limited to $O(\log \max\{n, c\})$ bits, where c denotes the maximum link capacity.

To define the flow problem on the communication network, each link is associated with a capacity and a direction, known to both nodes on which the link is incident. Notice that messages can be sent in both directions of a link, regardless of the flow direction.

The model assumes no central control or global clock. Hence, we must provide a means of synchronization among nodes. We use a mechanism called PIF (Propagation of Information with Feedback) [14], described next.

Let S be a set of nodes connected by a spanning tree rooted at a given node r. Each node knows which of its incident links leads to the parent and which to a child. A PIF on S is a distributed protocol which enables r to broadcast a message to all the nodes in S and then verify that each node has indeed received the message. The protocol consists of two phases, referred to as *shout* and *echo*.

In the shout phase, the broadcast message is propagated from r top-down over the spanning tree. Each node, upon receiving the message from its parent, sends it to all its children. In the echo phase, an acknowledgement message (*ack*, in short) is propagated bottom-up from the leaves to the root. A leaf sends an ack to its parent spontaneously upon receiving the broadcast message. An internal node waits until it has received an ack from each of its children, then propagates the ack to its parent. When r has received an ack from all its children, it knows that the broadcast message has been received by all nodes. It is easy to see that the complexity of the PIF protocol is $O(|S|)$ messages and time.

Among the many applications of PIF are: termination detection; signaling; and summation. In *termination detection*, the root node r is responsible for detecting that each node in S has completed a given task (e.g., a phase of an algorithm). To this end, r initiates a PIF, broadcasting a request for termination detection. A node propagates an ack to its parent only after having completed the given task (and received ack from all its children). Thus, when the PIF terminates, r knows that all the nodes have completed the task.

In *signaling*, r is responsible for notifying all the nodes in S to start a given task. Again, this is done by a PIF from r; only the shout phase is necessary. Where applicable, signaling and termination detection for the same task can be combined into one PIF. The shout phase initiates the task, and the echo phase verifies termination.

Finally, in *summation*, the task is to evaluate the expression $a_1 \oplus a_2 \oplus \cdots \oplus a_{|S|}$, where each a_i is a local value of some node in S, and $\oplus$ is a commutative and associative operator (e.g., "+", "max", etc.). This is accomplished as follows. The root r initiates a PIF. In the echo phase, a leaf sends its local value a_i to the parent (as part of the ack message). An internal node sums up (using $\oplus$) the values received form the children and its own local value, then propagates the result to the parent. When r has received values from all its children, it computes the total sum.

4.2. Description of the Algorithm

As in Section 3, we focus only on finding maximal flow in a layered network. Each node knows its layer number and which incident links lead to upstream or downstream neighbors. We assume that a spanning tree of the layered network, rooted at the target node t, has also been constructed. Also, each node knows the values n and x.

The first task is to select the special layers. Node t initiates x summation protocols to find the number of nodes in each subset $V_0, \ldots, V_{x-1}$ (recall that $V_i = \bigcup_{j=i \bmod x} L_j$). In the i'th summation, the local value used by nodes of V_i is 1, whereas all other nodes use local value 0. Upon completion of the summations, node t finds the index i' such that $|V_{i'}|$ is minimum and broadcasts it to all nodes. Each node can then determine to which super layer(s) it belongs, and whether or not it is in a special layer.

$PUSH(i)$ is implemented as follows. Each node in layer RL_i is responsible for finding the augmenting paths originating from itself. An augmenting path can be found by a DFS process. This is simply a token that is routed over SL_i in DFS order. At each point, the token carries the capacity of the path between its origin node and its current location. When the token reaches a closed node it backtracks on the last link, marking the link as closed. When the token reaches layer FL_i, it backtracks to the origin over the augmenting path, increasing the flows on the links by the capacity of the path.

Once a link has been closed, it is never traversed again by any token. To avoid contention among tokens of different parallel paths, nodes forward only one token at a time over each link. If tokens have been forwarded on all open links of a given node, additional tokens arriving at that node are delayed until some token backtracks. A node in RL_i ends its part in $PUSH(i)$ when it has no more excess flow or when it becomes closed.

In the centralized algorithm, $PUSH(i+1)$ is invoked after $PUSH(i)$ has terminated. In the distributed implementation, we simulate this implicit order of events by means of synchronization. For this purpose, a spanning forest is constructed in each super layer, with the tree roots located in the rear layer. This can be done by applying the MST algorithm of Gallager, Humblet and Spira [10] on each super layer.

Assume $PUSH(i)$ is in progress. It is easy to see that a sufficient condition for a node in $RL_{i+1}(=FL_i)$ to start $PUSH(i+1)$ is that all nodes in its connected component in SL_i have terminated their part in $PUSH(i)$. To this end, each root of the spanning forest of SL_i initiates a PIF to detect the termination of $PUSH(i)$ in its component. A node in RL_i delays its ack until all its augmenting paths have been completed (i.e., all tokens have returned). All other nodes can send ack immediately. Upon detecting termination, the roots, which are in RL_i, send a signal to the nodes in RL_{i+1} (using a second PIF) to start $PUSH(i+1)$.

This synchronization scheme propagates downstream from one super layer to the next, thereby implementing the loop labeled PLOOP (see Figure 1). When the start signal reaches node t, the entire loop has completed (notice that $SL_{k'}$ consists of exactly one connected component).

Node t then initiates a summation protocol to find the index of the highest front layer with unbalanced nodes. Unbalanced nodes in FL_i use as their local value for the summation the index i; balanced nodes use local value 0. The summation operator is $\oplus$=max. The result, say index i', is broadcast over the network, thereby signaling the nodes in $FL_{i'}$ to start $BALANCE(i')$.

$BALANCE(i)$ is implemented as follows. To perform flow reduction on edge $e=v \rightarrow w$, node w sends a message to v containing the amount of flow to be reduced. For purpose of synchronization (see below), an acknowledgement is sent from v to w. To perform a micropush, we use DFS tokens, similar to $PUSH$.

In the centralized algorithm, reduction followed by micropush is performed iteratively layer after layer. We need a mechanism to synchronize the distributed iterations, similar to the mechanism in $PUSH$. To this end, we construct a spanning forest in each pair of adjacent layers $L_j \cup L_{j+1}$, $0 \le j \le k-1$, with the roots located in L_{j+1}.

Assume a micropush from some level L_j in SL_i is in progress. The roots of the spanning forest of $L_{j-1} \cup L_j$ initiate a PIF to detect termination. A node in L_j delays its ack until all its augmenting paths have been completed and all flow reduction messages it sent to L_{j-i} have been acknowledged. Upon detecting termination, the roots (which are in L_j) signal the nodes in L_{j-1} to start their micropush.

The synchronization scheme propagates upstream layer after layer within SL_i. When the start signal reaches layer RL_i, a PIF is initiated by the roots of the spanning forest of SL_i, to detect termination of all the micropushes of the super layer (recall that the roots of SL_i are in RL_i). Subsequently, a signal to start $PUSH(i+1)$ is propagated to RL_{i+1}, thereby initiating another pass over PLOOP.

4.3. Complexity

Essentially, every elementary step that is involved in pushing or returning of flow is implemented by a message. Following Claims 3 and 4 (see Section 3), the number of messages used in flow-related activities is $O(\frac{n^2}{x}+mx)$. The delays that are imposed on DFS tokens due to contention among augmenting paths create bottlenecks during *PUSH* and micropush. This renders the progress of the tokens, in the worst case, completely serial. Consequently, the time spent on flow-related computation is also $O(\frac{n^2}{x}+mx)$.

Other activities in the network include the PIFs that are used in selecting the special layers. These incur a total cost of $O(nx)$ messages and time. In addition, before each *BALANCE*, two PIFs are performed to find the index of highest layer with unbalanced nodes and to propagate that index. By Claim 2, there are $f-2$ executions of *BALANCE*. Hence these PIFs incur a total cost of $O(nf)=O(\frac{n^2}{x})$ messages and time.

We now calculate the costs of synchronization. Each execution of *PUSH*(i) requires two PIFs over SL_i, which cost $O(|SL_i|)$ messages and time. Consequently, each pass over PLOOP has a synchronization cost of $O(\sum_{j=1}^{k'}|SL_j|)=O(n)$ messages and time. Following Claim 2, the total synchronization cost of all executions of *PUSH* is $O(nf)=O(\frac{n^2}{x})$ messages and time.

Each execution of *BALANCE*(i) entails two PIFs on each pair of adjacent layers in SL_i and two PIFs on the entire super layer, for a cost of $O(|SL_i|)$ messages and time. Following Claim 1, the synchronization of all executions of *BALANCE* requires a total of $O(\sum_{j=1}^{k'}f_i|SL_i|)=O(nf)=O(\frac{n^2}{x})$ messages and time.

Finally, there are setup costs for the synchronization, namely the construction of the spanning forests. A spanning forest in a subnetwork with p nodes and q links can be constructed in $O(q+p\log p)$ messages and $O(p\log p)$ time, using the MST algorithm of Gallager, Humblet and Spira [10]. Since each node in the layered network participates in at most four different spanning trees, the total cost of constructing all the spanning forests is $O(m+n\log n)$ messages and $O(n\log n)$ time.

Summing up all the costs, the complexity of the maximal flow computation is $O(\frac{n^2}{x}+mx+n\log n)$ messages and time. This is minimized by $x=\lfloor\frac{n}{m^{1/2}}\rfloor$, yielding a complexity of $O(nm^{1/2})$.

A layered network can be constructed in $O(nm^{1/2})$ messages and time using Frederickson's BFS algorithm [8]. As Dinic's scheme uses $n-1$ iterations of layered network construction followed by maximal flow computation, the total complexity of our distributed max-flow algorithm is $O(n^2m^{1/2})$ messages and time.

5. Conclusion

Since the centralized complexity of Cherkasky's algorithm is $O(n^2m^{1/2})$, our distributed implementation of it optimal in the number of messages. Yet, the time complexity is essentially as high as can be, namely equal to the message complexity. The critical part of the algorithm, which causes the slow progress, is the construction of augmenting paths. We have not been able to implement multiple DFS processes in parallel without contention among tokens. It seems that the augmenting path mechanism is inherently sequential.

Our results improve the message complexity of the max-flow problem by a factor of $\frac{n}{m^{1/2}}$. It is an open question whether further improvement is possible. There exist several max-flow algorithms (see [11]) which have better centralized complexity than Cherkasky's algorithm. However, these algorithms use sophisticated data structures to make "shortcuts" in the network. It seems that such mechanisms cannot be implemented distributively.

References

[1] Awerbuch, B., “Reducing Complexities of the Distributed Max-Flow and Breadth-First-Search Algorithms by Means of Network Synchronization,” *Networks* **15**, 4 (1985), pp. 425-437.

[2] Awerbuch, B., “Complexity of Network Synchronization,” *JACM* **32**, 4 (Oct. 1985), pp. 804-823.

[3] Cherkasky, B.V., “Algorithm of Construction of Maximal Flow in Networks with Complexity of $O(V^2E^{\frac{1}{2}})$ Operations,” *Mathematical Methods of Solution of Economical Problems* **7** (1977), pp. 117-125. (In Russian).

[4] Cheung, T., “Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation,” *IEEE Trans. Software Engineering* **SE-9**, 4 (July 1983), pp. 504-512.

[5] Dinic, E.A., “Algorithm for Solution of the Problem of Maximal Flow in a Network with Power Estimation,” *Soviet Math. Dokl.* **11** (1970), pp. 1277-1280.

[6] Edmonds, J. and R.M. Karp, “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems,” *JACM* **19**, 2 (April 1972), pp. 248-264.

[7] Even, S., *Graph Algorithms*, Computer Science Press, Rockville, Maryland, 1979.

[8] Frederickson, G.N., “A Single Source Shortest Path Algorithm for a Planar Distributed Network,” in *Proceedings 2nd Ann. Symp. on Theoretical Aspects of Computer Science*, 1985, pp. 143-150.

[9] Galil, Z., “An $O(V^{5/3}E^{2/3})$ Algorithm for the Maximal Flow Problem,” *Acta Informatica* **14**, 3 (Sept. 1980), pp. 221-242.

[10] Gallager, R.G., P.A. Humblet, and P.M. Spira, “A Distributed Algorithm for Minimum Weight Spanning Trees,” *ACM TOPLAS* **5**, 1 (Jan. 1983), pp. 66-77.

[11] Goldberg, A.V. and R.E. Tarjan, “A New Approach to the Maximum Flow Problem,” in *Proceedings 18th ACM Symp. on Theory of Computing*, 1986, pp. 136-146.

[12] Marberg, J.M. and E.M. Gafni, "An $O(N^3)$ Distributed Max-Flow Algorithm," in *Proceedings 1984 Conf. on Information Sciences and Systems*, Princeton Univ., Princeton, NJ, 1984, pp. 478-482.

[13] Segall, A., "Decentralized Maximum-Flow Protocols," *Networks* **12**, 3 (1982), pp. 213-230.

[14] Segall, A., "Distributed Network Protocols," *IEEE Trans. on Info. Theory* **IT-29**, 1 (Jan. 1983).

[15] Shiloach, Y. and U. Vishkin, "An $O(n^2 \log n)$ Parallel Max-Flow Algorithm," *J. of Algorithms* **3**, 2 (June 1982), pp. 128-146.