Mathl. Comput. Modelling Vol. 16, No. 2, pp. 65–72, 1992
Printed in Great Britain.

0895-7177/92 $5.00 + 0.00

A NEW KARZANOV-TYPE $O(n^3)$ MAX-FLOW ALGORITHM

GARY R. WAISSI
School of Management, University of Michigan-Dearborn
Dearborn, Michigan 48128-1491, U.S.A.

(Received April 1991)

Abstract—A new algorithm is presented for finding maximal and maximum value flows in directed single commodity networks. The algorithm gradually converts a combination of blocking preflows and backflows to a maximal flow in the network. Unlike other maximal flow algorithms, the algorithm treats the network more symmetrically by attempting to increase flow on both the *ForwardStep* and the *BackwardStep*. The algorithm belongs to the so called phase algorithms, and is applied to Dinic-type layered networks. With an effort of at most $O(n^3)$ for maximum value flow, the algorithm ties with the fastest maximum flow algorithms in dense networks, where $m \approx n^2$, and can therefore be seen as a significant alternate technique. The algorithm is based on the Karzanov [1] algorithm, and shares features with the algorithm of Tarjan [2]. The first version of this algorithm was presented by the author in [3].

INTRODUCTION

This paper presents a new Karzanov-type [1] algorithm for finding maximal flows and maximum value flows in directed single commodity networks. The maximal flow is found in two phases. In the first phase, a layered network is generated using the Dinic algorithm [4], or any Dinic-type algorithm that generates and maintains acyclic layered networks. In the second phase, the proposed algorithm is applied to the layered network to find the maximal flow in that network. Repeated solving of maximal flows in such networks leads to finding the maximum value flow in an original network. We assume some familiarity with Dinic-type layered networks, as they are used in aiding the solution of maximal and maximum value flow problems.

In [4], Dinic showed that a maximum value flow problem requires in the worst case the generation of $n-1$ layered networks, where n is the number of nodes in the original network. Because of this relationship between the maximal flow in a Dinic network and the maximum value flow in an original network, it is sufficient, without loss of generality, to consider solving maximal flows in such Dinic-type networks.

We assume that, when using the two phase process, in the Phase I, a Dinic-type layered network has been generated. The presented algorithm is then, in the Phase II, applied to such a layered network. Only the requirements of layer structure and acyclicity are necessary for the presented algorithm. This makes it possible to use any algorithm that generates acyclic layered networks from an original network in the Phase I.

The next section outlines briefly the history of max-flow algorithms. In the third section the notation and background are discussed. The fourth section describes the algorithm. The fifth section gives the algorithm in detail. The sixth section discusses the computational aspects of the algorithm. In the seventh section, a small numerical example is given, and the section eight concludes the paper.

SUMMARY OF MAX-FLOW ALGORITHMS

Below, the history of max-flow algorithms is summarized with respect to the author(s), year of publication, reference and computational complexity. It has to be noted that the best max-flow

Ford and Fulkerson	(1956)[5]	$O(nmU)$
Dinic	(1970)[4]	$O(n^2m)$
Edmonds and Karp	(1972)[6]	$O(nm^2)$
Karzanov	(1974)[1]	$O(n^3)$
Cherkasky	(1977)[7]	$O(n^2\sqrt{m})$
Malhotra, Kumar and Maheswari	(1978)[8]	$O(n^3)$
Galil	(1980)[9]	$O(n^{\frac{5}{3}}m^{\frac{2}{3}})$
Galil and Naamad	(1980)[10]	$O(mn(\log n)^2)$
Sleator	(1980)[11]	$O(mn\log(n))$
Shiloach and Vishkin	(1982)[12]	$O(n^2\log(n))$
Sleator and Tarjan	(1983)[13]	$O(mn\log(n))$
Tarjan	(1984)[2]	$O(n^3)$
Goldberg	(1985)[14]	$O(n^3)$
Waissi (and this paper)	(1985)[3]	$O(n^3)$
Goldberg and Tarjan	(1986)[15]	$O(nm\log(\frac{n^2}{m}))$
Ahuja and Orlin	(1989)[16]	$O(nm + n^2\log(U))$

algorithms still require a computational effort of $O(n^3)$ for dense networks where $m \approx n^2$. All the algorithms below that offer some improvement computationally in sparse networks result actually to a worse than $O(n^3)$ complexity in dense networks.

NOTATION AND BACKGROUND

The Problem

The finding of a single commodity maximum value flow in a directed network implies that we are attempting to send as much flow as possible from one or more origin nodes to one or more destination nodes.

It suffices, without loss of generality, to consider the maximum value flow problem in directed single source single sink networks. The problem may be formulated and solved as a linear program (LP). However, in this paper we develop a fast polynomial network flow phase algorithm for this problem.

Notation

Let an original, directed, single commodity network be $G = (N, A, \lambda, c, s, t)$. Here N denotes the set of nodes, A the set of arcs, λ a vector $(\lambda_{i,j} : (i,j) \in A)$ of lower bounds for flows on the arcs, c a vector $(c_{i,j} : (i,j) \in A)$ of capacities for flows on the arcs, and s, t the specified source and sink nodes. Each arc in the network is an ordered pair (i,j), $i \neq j$, with $i, j \in N$. Nodes i, j are called the tail and head nodes of arc (i,j). The arc (i,j) is available for shipping the commodity from node i to node j.

Flow Feasibility and Flow Conservation

A flow vector in G is a vector $f = (f_{i,j} : (i,j) \in A)$ where $f_{i,j}$ denotes the units of commodity shipped from node i along the arc $(i,j) \in A$ to node j. Given the flow vector $f = (f_{i,j})$ in G, define

$$f(\text{in}, i) = \sum(f_{j,i} : \text{over } j \text{ such that } (j,i) \in A),$$
$$f(i, \text{out}) = \sum(f_{i,j} : \text{over } j \text{ such that } (i,j) \in A).$$

The flow vector f is said to be a feasible flow vector if

$$f(s, \text{out}) - f(\text{in}, s) = f(\text{in}, t) - f(t, \text{out}),$$
$$f(\text{in}, i) = f(i, \text{out}), \qquad \text{for all } i \in N, \text{ and } i \neq s \text{ or } t,$$
$$\lambda \leq f \leq c.$$

The first two constraints are called the flow conservation constraints, and the third is the flow bound constraint. When the second constraint holds for some node, say node i, then the flow vector f satisfies flow conservation at node i.

If f is a feasible flow vector in G, the common value of the quantities in the first equation is called the value of the flow vector f, and denoted by $v(f)$.

The Acyclic Layer Network

An acyclic network with a layer- or level-structure is generated from an original network G using the algorithm presented in [3], or the Dinic Algorithm [4]. Any other algorithm that generates and maintains the acyclicity and the layer structure can be used with the proposed algorithm. Many algorithms [1–3,7–12,17–20] use the Dinic-type layered network algorithms to solve one phase of the problem. There are other algorithms, e.g., Goldberg [14], Goldberg and Tarjan [15], and Ahuja and Orlin [16], that do not use the Dinic algorithm's layered network concept, but instead use distance labels to direct flows closer to the sink. The algorithm of Goldberg and Tarjan [15] was called in [16] a "preflow-push algorithm". The "excess-scaling algorithm" of Ahuja and Orlin [16] improves this preflow-push algorithm.

Let $H = (V, R, 0, \kappa, s, t)$ be an acyclic layered network constructed from the network G using for example the Dinic algorithm. Here V denotes the set of nodes, R the set of arcs, zero lower bound for flow on the arcs, κ the vector of capacities for flows on the arcs, s the source node, t the sink node. It is assumed that V is partitioned into mutually exclusive sets $L_0, L_1, \cdots, L_{\text{num}}$ with $L_0 = \{s\}$, and $L_{\text{num}} = \{t\}$, such that for each $(i,j) \in R$, $i \in L_r$ and $j \in L_p$ for some r and $p \geq r+1$. The sets of nodes $L_0, L_1, \cdots, L_{\text{num}}$ are the layers in H, and *num*, the number of layers, is called the length of the network H. Each arc in R joins a node in a layer to a node in a higher layer. The partitioning of the network H into such layers can always be done because the network H is acyclic.

DESCRIPTION OF THE ALGORITHM

Suppose that the feasible flow vectors in H are denoted by $g = (g_{i,j})$. The algorithm can be initiated with zero or any other known feasible flow vector in H. Suppose that g is a feasible or nonfeasible flow vector in H. Define $g_{\text{in},i}$ and $g_{i,\text{out}}$ for a node, say i, as follows:

$$g(\text{in}, i) = \sum (g_{j,i} : \text{ over } j \text{ such that } (j,i) \in R)$$

$$g(i, \text{out}) = \sum (g_{i,j} : \text{ over } j \text{ such that } (i,j) \in R)$$

g is said to have a PREFLOW at node $i \in V$ if $g_{\text{in},i} > g_{i,\text{out}}$.
g is said to have a BACKFLOW at node $i \in V$ if $g_{\text{in},i} < g_{i,\text{out}}$.
g is said to be a balanced flow at node $i \in V$ if $g_{\text{in},i} = g_{i,\text{out}}$.

The proposed algorithm finds a maximal flow from a specified source node to a specified sink node. It uses *ForwardStep* and *BackwardStep*. The algorithm, starting from a feasible flow, attempts to increase flow from a specified source node to a specified sink node during both *Steps*.

During the *ForwardStep* the flow is gradually increased from the source on all arcs out of the source by processing, or pushing, the flow through the network from the first layer to the last layer. The *BackwardStep* reverses the process. During a *BackwardStep*, the flow is gradually increased on all arcs into the sink by processing the network from the sink to the source, from the last layer to the first layer, in reverse order, and "pushing" artificial flow through the network.

Both steps are basically identical with the only difference being that the *ForwardStep* is applied to the network from source to sink in an increasing order of layer numbering, whereas the *BackwardStep* is applied from sink to source in a decreasing order of layer numbering. The steps are applied in an alternating order.

The flow may be infeasible after both the *ForwardStep* and the *BackwardStep*. During each of the steps unbalanced nodes of the previous step are balanced by reducing flow to regain feasibility. If the flow was increased during the *ForwardStep* so that the flow into a node, say i, is greater

than the flow out of node i, then node i is balanced in the subsequent *BackwardStep* by reducing flow into node i. If the flow was increased during the *BackwardStep* so that the flow out of a node, say i, is greater than that into node i, then the node i is balanced in the subsequent *ForwardStep* by reducing flow out of node i. The sequences of *ForwardStep* and *BackwardStep* are repeated until no flow increase occurs and all nodes become balanced.

All nodes in H are initially unblocked. The algorithm processes the nodes starting from source and increases the flow along arcs $(s,i) \in R$ to capacity. At this stage, node i, initially unblocked and balanced, becomes unblocked and unbalanced. Next, the algorithm attempts to balance node i by increasing the flow along arcs $(i,j) \in R$ out of node i. If the flow cannot be increased along such arcs, then it is said that the balancing attempt has failed for node i in the *ForwardStep*. Similarly, a balancing attempt can fail in the *BackwardStep*.

A node $i \in V$, where the balancing attempt has failed in the *ForwardStep*, and hence $g_{\text{in},i} > g_{i,\text{out}}$, is assigned a flow conservation status PF-blocked, called preflow blocked. Once a PF-blocked node becomes balanced it is called a PF-blocked-balanced node. A node $i \in V$ where $g_{\text{in},i} = g_{i,\text{out}}$ is assigned a flow conservation status BAL, and it is called balanced. A node $i \in V$, where the balancing attempt has failed in the *BackwardStep*, and hence $g_{\text{in},i} < g_{i,\text{out}}$, is assigned a flow conservation status BF-blocked, called backflow blocked. Once a BF-blocked node becomes balanced it is called a BF-blocked-balanced node.

A node which became PF-blocked in the *ForwardStep* will be balanced in the following *BackwardStep*. A node which became BF-blocked in the *BackwardStep* will be balanced in the following *ForwardStep*.

A node can be in two states: unblocked or blocked. An unblocked node can be unbalanced or balanced. A blocked node can be

PF-blocked-unbalanced or PF-blocked-balanced
BF-blocked-unbalanced or BF-blocked-balanced

An unblocked node $i \in V$ can become PF- or BF-blocked. A blocked node becomes balanced in the subsequent *Step*, but it never again becomes unblocked. Once a blocked node becomes balanced it remains balanced until the end of the algorithm. It has to be pointed out that once a node has been assigned a flow conservation status, the flow conservation status remains unchanged until the end of the algorithm. Hence, a PF-blocked node, balanced or unbalanced, cannot change to a BF-blocked node, or vice versa, during the algorithm.

The algorithm uses *ForwardStep* and *BackwardStep* over the acyclic layered network. During the *ForwardStep*, the flow is increased through the network as much as possible, except through arcs $(i,j) \in R$ where j is blocked. It starts from $s \in L_0$ and saturates all arcs $(s,i) \in R$. Then it moves to layer L_1 to node $i \in V$ which is unbalanced and attempts to balance it by increasing flow out of node i. If the balancing attempt of node $i \in V$ fails, and hence $g_{\text{in},i} > g_{i,\text{out}}$, then node i is made PF-blocked. If during the *ForwardStep* BF-blocked nodes are encountered, they are balanced by reducing the flow out of node i. This can be always done because the flow out of node i can be reduced to zero on all arcs going out of node i if necessary.

During the *BackwardStep*, starting from node t, flow is increased on unsaturated arcs $(i,t) \in R$ to capacity where node i is not blocked. As the procedure moves one layer backwards towards the source s then for an unbalanced, unblocked node i an attempt is made to balance it by increasing flow along the incoming arcs $(p,i) \in R$. If the balancing attempt fails, and hence $g_{\text{in},i} < g_{i,\text{out}}$, then node i is made BF-blocked. If during the *BackwardStep* PF-blocked nodes are encountered, they are balanced by reducing the flow on the incoming arcs. This can be always done because the flow into the node i can be reduced to zero on all incoming arcs if necessary.

After both the *ForwardStep* and the *BackwardStep*, the network is *pruned* if all nodes are balanced, and hence, the flow is feasible. The *pruning* consists of eliminating all saturated arcs, and all nodes, say $i \neq s,t$, where either all arcs incident into, or out of, the node i have been eliminated. When this repetitive process leads to elimination of all arcs incident at s or t, then there are no flow augmenting chains, or FACs, in the network and the current feasible flow vector is maximal.

THE ALGORITHM

ForwardStep

STEP 1. Increase flow on all arcs $(s,i) \in R$, out of the source s, to capacity, except on arcs $(s,i) \in R$ where node i is blocked.

STEP 2. Suppose that the last layer processed is L_{p-1}. Go to the next layer L_p and attempt to balance all unblocked unbalanced nodes, where the flow into the node is greater than the flow out of the node, and all BF-blocked nodes.

Suppose that the node i is unblocked and unbalanced with $g_{\text{in},i} > g_{i,\text{out}}$. Attempt to balance node $i \in L_p$ by increasing flow out of node i along arcs $(i,j) \in R$, where j is not blocked, one arc at a time in any order. If the attempt fails, and flow out of node i cannot be increased further along any arc (i,j), and hence $g_{\text{in},i} > g_{i,\text{out}}$, then make node i PF-blocked.
Suppose that node $j \in L_p$ is BF-blocked. Balance node j by decreasing the flow on arcs $(j,k) \in R$ out of node j until node j is balanced. Reduce flow on arcs (j,k) first where node k is not blocked.

STEP 3. Repeat Step 2 for all unblocked-unbalanced nodes i with $g_{\text{in},i} > g_{i,\text{out}}$, and BF-blocked-unbalanced nodes j, and all layers in increasing order of the layer numbering until the sink t is reached.

STEP 4. If there are no unbalanced nodes, then *prune* the network. If the *pruning* eliminates all arcs incident at the source s or the sink t, terminate. The current feasible flow vector is maximal flow in the acyclic network. Otherwise go to the *BackwardStep*.

BackwardStep

STEP 1. Increase flow on all arcs $(i,t) \in R$ directed into the sink t to capacity, except on arcs $(i,t) \in R$ where node i is blocked.

STEP 2. Suppose that the last layer processed is L_{r+1}. Go to layer L_r and attempt to balance all unblocked-unbalanced nodes where the flow out of the node is greater than the flow into the node, and all PF-blocked nodes.

Suppose that the node i is unblocked and unbalanced with $g_{\text{in},i} < g_{i,\text{out}}$. Attempt to balance node $i \in L_r$ by increasing flow into node i along arcs $(q,i) \in R$, where

$$q \in \{\cup(L_p : p \leq r-1)\}$$

is not blocked, one arc at a time in any order. If the attempt fails, and flow into node i cannot be increased further along any arc (q,i), and hence $g_{\text{in},i} < g_{i,\text{out}}$, then make node i BF-blocked.
Suppose that node $j \in L_r$ is PF-blocked. Balance node j by decreasing the flow on arcs $(h,j) \in R$ into node j until node j is balanced. Reduce flow on arcs (h,j) first where node h is not blocked.

STEP 3. Repeat Step 2 for all unblocked-unbalanced nodes i with $g_{\text{in},i} < g_{i,\text{out}}$, and PF-blocked-unbalanced nodes j, and all layers in decreasing order of the layer numbering until the source s is reached.

STEP 4. If there are no unbalanced nodes, then *prune* the network. If the *pruning* eliminates all arcs incident at the source s or the sink t, terminate; the current feasible flow vector is maximal flow in the acyclic network. Otherwise go to the *ForwardStep*.

DISCUSSION

The validity proof follows the same steps as that of Tarjan [2]. The only difference is that here the algorithm is applied to increase flow starting both from source s and sink t. When a node is assigned a flow conservation status, PF- or BF-blocked, it means that the node cannot handle more flow out of the node or into the node. In subsequent steps the blocked nodes are balanced, and flow increase is applied on alternate paths. When all arcs incident at a node lead to blocked nodes no flow increase is possible. When no flow increase is possible, and all nodes are balanced, the algorithm halts. The maximal (blocking) flow has been found.

The algorithm gradually converts a combination of blocking preflows and backflows to a maximal flow in the network. Unlike other maximal flow algorithms the algorithm treats the network more symmetrically by attempting to increase flow on both the *ForwardStep* and the *BackwardStep*.

THEOREM. *The worst case computational complexity of the maximal flow algorithm is* $O(n^2)$.

PROOF. In a special case when the flow is increased from source to sink it may happen that no nodes become blocked. If this occurs then all nodes are balanced after the *ForwardStep*, and all arcs out of the source node are at capacity, i.e., the flow is maximal after only the *ForwardStep*.

Beyond this special case the following must be considered. At the beginning of each *ForwardStep* the only unbalanced nodes are those which became unbalanced during the previous *BackwardStep*. Those nodes are BF-blocked-unbalanced. Similarly, at the beginning of each *BackwardStep* the only unbalanced nodes are those that became unbalanced during the preceding *ForwardStep*. Those nodes are PF-blocked-unbalanced. Every sequence of *ForwardStep* and *BackwardStep* either blocks or balances at least one node. During a *ForwardStep* at least one node is PF-blocked or a BF-blocked node is balanced. During each *BackwardStep* at least one node is BF-blocked or a PF-blocked node is balanced. A PF-blocked-unbalanced node can only become a PF-blocked-balanced node. Similarly a BF-blocked-unbalanced node can only become a BF-blocked-balanced node. Once a PF- or BF-blocked node is balanced, then arcs incident at those nodes are not used for flow increase in subsequent steps of the algorithm. There are at most $(n-2)$ repetitions of the *ForwardStep* and the *BackwardStep*, and at most $\sum(i$: for all i from 1 to $n-2) = n^2$ balancing attempts.

The flow on an arc (i,j) is increased in the *ForwardStep* if and only if j is unblocked. The flow on an arc (i,j) is increased in the *BackwardStep* if and only if i is unblocked. Similarly, the flow on an arc (i,j) is decreased in the *ForwardStep* if and only if node i is BF-blocked. The flow on an arc (i,j) is decreased in the *BackwardStep* if and only if node j is PF-blocked.

Hence, the flow first increases on an arc and then decreases. The increase of flow either saturates the arc or stops when the respective node is balanced. The decrease of flow either reduces the flow in an arc to zero or stops when respective node is balanced.

In each sequence of the *ForwardStep* and *BackwardStep*, one node is blocked or a blocked node is balanced. Consider the r^{th} sequence of such steps over the acyclic network. Suppose that during this sequence a blocked node is balanced. Suppose also that there are b_r arcs incident at that node. Each time a blocked node is balanced, the node itself and arcs incident to it are not considered for further flow increase in subsequent steps of the algorithm. Hence, clearly $\sum(b_r$: for all r over $n) \leq m$. The total effort required by the algorithm is hence at most $O(m+n^2) = O(n^2)$.

COROLLARY. *The algorithm requires at most* $O(n^3)$ *effort for maximum value flow in a non-acyclic directed network* G.

PROOF. The maximal flow in an acyclic network requires at most $O(n^2)$ effort. The Dinic algorithm [4] reduces the maximum value flow problem to solving at most $n-1$ maximal flow problems in acyclic networks. Hence, the maximum value flow in a original network G can be found in at most $(n-1)n^2$ steps, i.e., the effort is at most $O(n^3)$.

This complexity is the same as with the fastest maximum flow algorithms in dense networks, where $m \approx n^2$. The implementation here is different than that of Karzanov's maximal flow algorithm. The algorithm clearly treats the entire network more symmetrically. Because of this it is likely to produce maximal flows of higher value than other maximal flow algorithms in each acyclic network, and hence, is likely to run faster in practice.

EXAMPLE

Figure 1 shows a small acyclic network presented in [3]. In Figure 2, the maximal flow algorithm is applied to the network. After three *Steps*, two *ForwardSteps* and one *BackwardStep*, all nodes in the example network are balanced and the flow is maximal. The maximal flow is also maximum value flow in the original network.

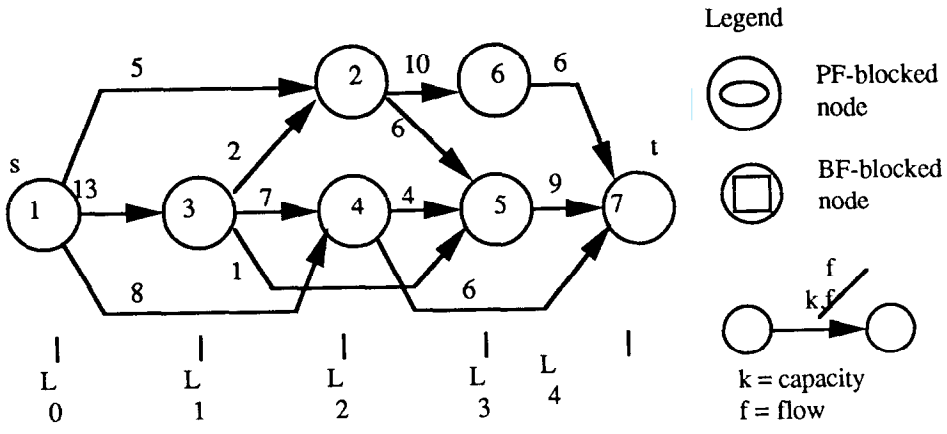

Figure 1. Example network.

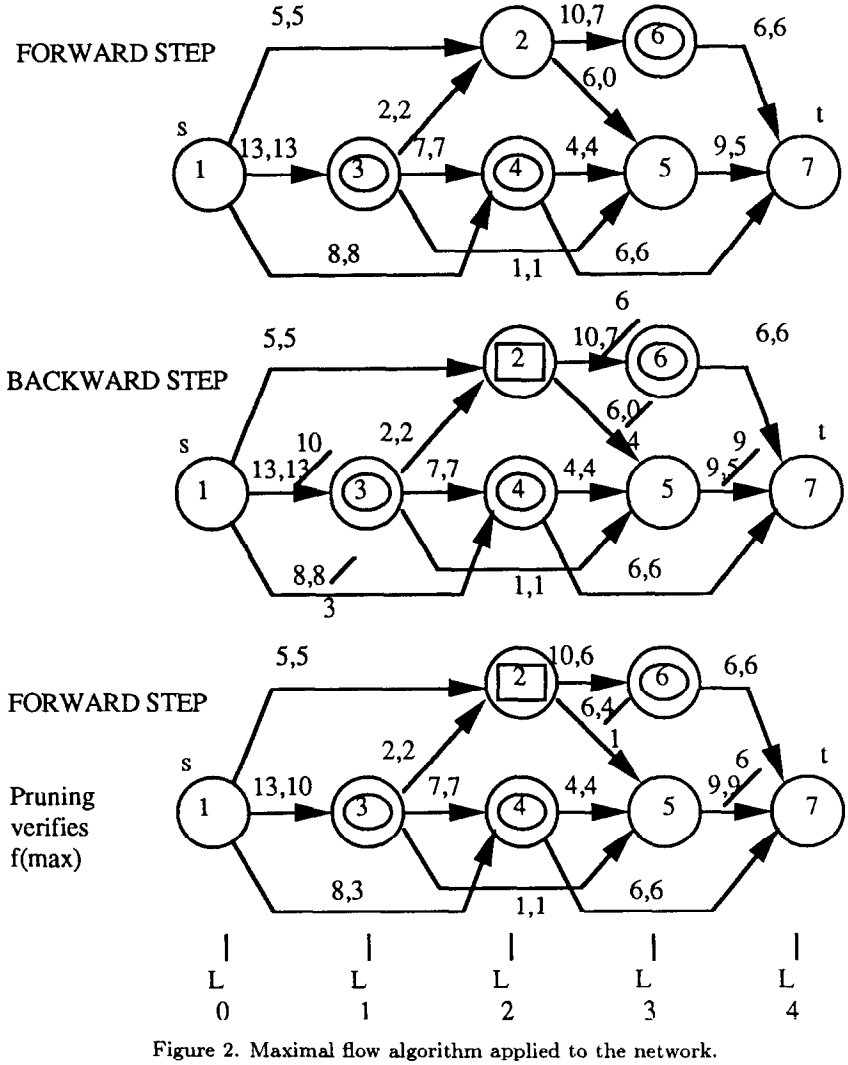

Figure 2. Maximal flow algorithm applied to the network.

CONCLUSION

The algorithm gradually converts a combination of blocking preflows and backflows to a maximal flow in the network. Unlike other maximal flow algorithms the algorithm treats the network more symmetrically by attempting to increase flow in repeated *ForwardStep* and *BackwardStep*.

During the *ForwardStep*, the flow is gradually increased from the source on all arcs out of the

source by processing, or pushing, the flow through the network from the first layer to the last layer, layer by layer. During a *BackwardStep*, the flow is gradually increased on all arcs into the sink by processing the network from the sink to the source in reverse order, and "pushing" artificial flow through the network. The algorithm requires an effort of at most $O(n^2)$ for maximal flow in acyclic networks. This ties with the fastest maximal flow algorithms in terms of the computational effort required in the worst case in dense networks with $m \approx n^2$.

In maximum flow applications, the algorithm solves one phase of the problem. When the Dinic algorithm is used to solve the Phase I, i.e., the acyclic network generation, and the proposed algorithm is used to find the maximal flow in the Phase II, then the maximum flow in the original network can be found with an effort $O(n^3)$ in the worst case. This, again, is a tie with the fastest maximum value flow algorithms with respect to the computational effort in the worst case, also in dense networks.

REFERENCES

1. V.A. Karzanov, Determining the maximal flow in a network by the method of preflows, *Soviet Math. Doklady* **15**, 434–437 (1974).
2. R.E. Tarjan, A simple version of Karzanov's blocking flow algorithm, *Operations Research Letters* **2**, 265–268 (1984).
3. G.R. Waissi, Acyclic Network Generation and Maximal Flow Algorithms for Single Commodity Flow, Ph.D. Dissertation, The University of Michigan, Ann Arbor, Michigan, (1985).
4. E.A. Dinic, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Doklady.* (English translation by R.F. Rinehart) **11**, 1277–1280 (1970).
5. L.R. Ford and D.R. Fulkerson, Maximal flow through a network, *Canad. J. Math.* **9**, 210–218 (1957).
6. J. Edmonds and R.M. Karp, Theoretical improvement in algorithmic efficiency for network flow problems, *Journal of ACM* **19**, 248–264 (1972).
7. B.V. Cherkasky, Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations, *Math. Meth. Solution Econ.*, (in Russian)(English transl. in [18], by Galil), 117–125 (1977).
8. V.M. Malhotra, M.P. Kumar and S.N. Maheswari, An $O(V^3)$ algorithm for finding maximum flows in networks, *Information Processing Letters* **7**, 277–278 (1978).
9. Z. Galil, An $O(V(5/3)E(2/3))$ algorithm for maximal flow problem, *Acta Informatica* **14**, 221–242 (1980).
10. Z. Galil and A. Naamad, An $O(EV(\log V)^2)$ algorithm for the maximal flow problem, *Journal of Computer and System Sciences* **21**, 203–217 (1980).
11. D.D. Sleator, An $O(mn\log(n))$ algorithm for maximum network flow, Tech. Rep. STAN-CS-80-831 (1980).
12. Y. Shiloach and U. Vishkin, An $O(n^2\log(n))$ parallel max-flow algorithm, *Journal of Algorithms* **3**, 128–146 (1982).
13. D.D. Sleator and R.E. Tarjan, A data structure for dynamic trees, *Journal of Computer Sciences* **26**, 362–391 (1983).
14. A.V. Goldberg, *A New Max-Flow Algorithm,* Technical Report MIT/LCS/TM-291, Laboratory of Computer Science, MIT, Cambridge, Massachusetts, (1985).
15. A. Goldberg and R.E. Tarjan, A new approach to the maximum flow problem, Presented at the Proceedings of the *18th ACM Symposium on the Theory of Computing*, pp. 136–146, (1986).
16. R.K. Ahuja and J.B. Orlin, A fast and simple algorithm for the maximum flow problem, *Operations Research* **37**, 748–759 (1989).
17. Z. Galil, A new algorithm for the maximal flow problem, Presented at the Proceedings of the *19th IEEE Symposium on Foundations of Computer Science*, pp. 231–245, (1978).
18. F. Glover, D. Klingman, J. Mote and D. Whitman, A comprehensive computer evaluation and enhancement of maximum flow algorithms, *Applications of Management Science* **3**, 109–175 (1983).
19. K.G. Murty, *Network Programming*, Prentice-Hall, NJ (to appear).
20. R.E. Tarjan, Data structures and network algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics* **44** (1983).
21. L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, N.J., (1962).
22. L.R. Ford and D.R. Fulkerson, A simple algorithm for finding maximal network flows and an application to the Hitchcock problem, *Canad. J. Math.* **9**, 210–218 (1957).
23. C. Martel, A comparison of phase and nonphase network flow algorithms, *Networks* **19**, 691–705 (1989).