J. Appl. Math. & Computing Vol. **15***(2004), No. 1 - 2, pp. 53 - 75*

SEQUENTIAL AND PARALLEL ALGORITHMS FOR MINIMUM FLOWS

ELEONOR CIUREA AND LAURA CIUPALĂ

Abstract. First, we present two classes of sequential algorithms for minimum flow problem: decreasing path algorithms and preflow algorithms. Then we describe another approach of the minimum flow problem, that consists of applying any maximum flow algorithm in a modified network. In section 5 we present several parallel preflow algorithms that solve the minimum flow problem. Finally, we present an application of the minimum flow problem.

AMS Mathematics Subject Classification : 90B10, 90C35, 05C35, 68R10.
Key words and phrases : Network flow, network algorithms, minimum flow problem, parallel algorithms.

1. Minimum flow problem

The literature on network flow problem is extensive. Over the past 50 years researchers have made continuous improvements to algorithms for solving several classes of problems. From the late 1940s through the 1950s, researchers designed many of the fundamental algorithms for network flow, including methods for maximum flow and minimum cost flow problems. In the next three decades, there are many research contributions concerning improving the computational complexity of network flow algorithms by using enhanced data structures, techniques of scaling the problem data etc. The algorithms designed in the 1990s did not improve essentially the algorithms already known at that time. The mini mum flow problem was not treated so often.

Received May 9, 2003. Revised August 28, 2003.

We consider a capacitated network $G = (N, A, l, c, s, t)$ with a nonnegative capacity $c(i,j)$ and with a nonnegative lower bound $l(i,j)$ associated with each arc $(i,j) \in A$. We distinguish two special nodes in the network G : a source node s and a sink node t.

A *flow* is a function $f : A \to R_+$ satisfying the next conditions:

$$f(i,N) - f(N,i) = \begin{cases} v, & i = s \\ 0, & i \neq s,t \\ -v & i = t \end{cases} \tag{1.a}$$

$$l(i,j) \leq f(i,j) \leq c(i,j),\ \forall (i,j) \in A, \tag{1.b}$$

for some $v \geq 0$, where

$$f(i,N) = \sum_{j \mid (i,j) \in A} f(i,j)$$

and

$$f(N,i) = \sum_{j \mid (j,i) \in A} f(j,i).$$

We refer to v as the *value* of the flow f.

The minimum flow problem is to determine a flow f for which v is minimized.

For solving the minimum flow problem we will use the following definitions: A *preflow* is a function $f : A \to R_+$ that satisfies (1.b) and

$$f(i,N) - f(N,i) \leq 0,\ \ \forall i \in N \setminus \{s,t\}. \tag{2}$$

The algorithms that we will describe in third section maintain a preflow at each intermediate stage.

For any preflow f, we define the *deficit* of the node i as

$$e(i) = f(i,N) - f(N,i), \quad \text{for all } \ i \in N. \tag{3}$$

We refer to a node i with $e(i) = 0$ as *balanced*. A preflow satisfying the condition

$$e(i) = 0, \quad \text{for all } \ i \in N \setminus \{s,t\}$$

is a flow. Thus, a flow is a particular case of preflow.

For the minimum flow problem, the *residual capacity* $r(i,j)$ of any arc $(i,j) \in A$, with respect to a given preflow f, is given by $r(i,j) = c(j,i) - f(j,i) + f(i,j) - l(i,j)$. By convention, if $(j,i) \notin A$ then we add arc (j,i) to the set of arcs A and we set $l(j,i) = 0$ and $c(j,i) = 0$. The residual capacity of the arc (i,j) represents the maximum amount of flow from the node i to node j that

can be cancelled. The network $G_f = (N, A_f)$ consisting only of the arcs with positive residual capacity is referred to as the *residual network* (with respect to preflow f).

A *cut* is a partition of the node set N into two subsets S and $\overline{S} = N - S$; we represent this cut using the notation $\left[S, \overline{S}\right]$. We refer to a cut $\left[S, \overline{S}\right]$ as a $s - t$ *cut* if $s \in S$ and $t \in \overline{S}$. We refer to an arc (i, j) with $i \in S$ and $j \in \overline{S}$ as a *forward arc* of the cut, and an arc (i, j) with $i \in \overline{S}$ and $j \in S$ as a *backward arc* of the cut. Let $\left(S, \overline{S}\right)$ denote the set of forward arcs in the cut, and let $(\overline{S}, S)$ denote the set of backward arcs.

For the minimum flow problem, we define the *capacity* $c\left[S, \overline{S}\right]$ of a $s - t$ cut $[S, \overline{S}]$ as the sum of the lower bounds of the forward arcs minus the sum of the capacities of the backward arcs. That is,

$$c\left[S, \overline{S}\right] = l\left(S, \overline{S}\right) - c\left(\overline{S}, S\right).$$

We refer to a $s - t$ cut whose capacity is the maximum among all $s - t$ cuts as a *maximum cut*.

Theorem 1.1 (Min-Flow Max-Cut Theorem). *If there exists a feasible flow in the network, the value of the minimum flow from a source node s to a sink node t in a capacitated network with nonnegative lower bounds equals the capacity of the maximum $s - t$ cut.*

This theorem can be proved in a manner similar to the proof of the Max-Flow Min-Cut Theorem.

We refer to a path in G from the source node s to the sink node t as a *decreasing path* if it consists only of arcs with positive residual capacity. Clearly, there is a one-to-one correspondence between decreasing paths in G and directed paths from s to t in G_f.

Given a decreasing path P in G, we can decrease the current flow f in the following manner:

$$f(i,j) = \begin{cases} f(i,j) - r, & if \quad (i,j) \text{ is a forward arc in } P \\ f(i,j) + r, & if \quad (i,j) \text{ is a backward arc in } P \\ f(i,j), & if \quad (i,j) \text{ is not an arc in } P \end{cases}$$

where $r = \min\{r_1, r_2\}$, $r_1 = \min\{f(i,j) - l(i,j) \,|\, (i,j) \text{ is a forward arc in } P\}$, $r_2 = \min\{c(i,j) - f(i,j) \,|\, (i,j) \text{ is a backward arc in } P\}$. We refer to r as the residual capacity of the decreasing path P.

Theorem 1.2 (Decreasing Path Theorem). *A flow f^* is a minimum flow if and only if the residual network G_f contains no directed path from the source node to the sink node.*

This theorem can be proved in a manner similar to the proof of the Augmenting Path Theorem.

The minimum flow problem in a network can be solved in two phases:
(1) establishing a feasible flow
(2) from a given feasible flow, establish the minimum flow.

1.1. Establishing a feasible flow

The problem of determining a feasible flow consists in finding a function $f : A \to R_+$ that satisfies conditions (1.a) and (1.b). First, we transform this problem into a circulation problem by adding an arc (t, s) of infinite capacity and zero lower bound. This arc carries the flow sent from node s to node t back to node s. Clearly, the minimum flow problem admits a feasible flow if and only if the circulation problem admits a feasible flow. Because the se two problems are equivalent, we focus on finding a feasible circulation if it exists in the transformed network $\tilde{G} = \left(N, \tilde{A}, \tilde{l}, \tilde{c}, s, t\right)$, where

$$\begin{aligned}
&\tilde{A} = A \cup \{(t, s)\}\\
&\tilde{l}(i, j) = l(i, j), \text{ for every arc } (i, j) \in A\\
&\tilde{l}(t, s) = 0\\
&\tilde{c}(i, j) = c(i, j), \text{ for every arc } (i, j) \in A\\
&\tilde{c}(t, s) = \infty.
\end{aligned}$$

The feasible circulation problem is to identify a flow f satisfying the following constraints:

$$\tilde{f}(i, N) - \tilde{f}(N, i) = 0, \text{ for every node } i \in N, \tag{4.a}$$

$$\tilde{l}(i, j) \le \tilde{f}(i, j) \le \tilde{c}(i, j), \text{ for every arc}(i, j) \in A. \tag{4.b}$$

By replacing $\tilde{f}(i, j) = \tilde{f}'(i, j) + \tilde{l}(i, j)$ and $\tilde{c}(i, j) = \tilde{c}'(i, j) + \tilde{l}(i, j)$ in (4a) and (4b) we obtain the following transformed problem:

$$\tilde{f}'(i, N) - \tilde{f}'(N, i) = \tilde{b}(i), \text{ for every node } i \in N, \tag{5.a}$$

$$0 \le \tilde{f}'(i, j) \le \tilde{c}'(i, j), \text{ for every arc } (i, j) \in A \tag{5.b}$$

with the supplies/demands $\tilde{b}(\cdot)$ at the nodes defined by

$$\tilde{b}(i) = \tilde{l}(N, i) - \tilde{l}(i, N).$$

Clearly,

$$\sum_{i \in N} \tilde{b}(i) = 0.$$

We can solve this feasible flow problem by solving a maximum flow problem defined in a transformed network. We introduce two new nodes: a source node s' and a sink node t'. For each node i with $\tilde{b}(i) > 0$ we add a source arc (s', i) with capacity $\tilde{b}(i)$ and for each node i with $\tilde{b}(i) < 0$ we add a sink arc (i, t') with capacity $-\tilde{b}(i)$. Then we solve a maximum flow problem in this transformed network. If the maximum flow saturates all the source and the sink arcs, then the initial problem has a feasible solution (which is the restriction of the maximum flow that saturates all the source and sink arcs to the initial set of arcs A); otherwise it is infeasible.

1.2. Establishing a minimum flow

There are two approaches for solving maximum flow problem: (1) using augmenting path algorithms and (2) using preflow-push algorithms. The algorithms of both classes can be modified in order to obtain minimum flow algorithms. In the next two sections we present decreasing path algorithms and preflow algorithms.

2. Decreasing path algorithms

Any augmenting path algorithm for maximum flow problem can be modified in order to obtain a decreasing path algorithm for minimum flow problem.

We present the generic decreasing path algorithm. This algorithm is based on Decreasing path theorem 1.2. The algorithm begins with a feasible flow and proceeds by identifying decreasing paths and decreasing flows on these paths until the network contains no such path. The generic algorithm for the minimum flow problem is the following:

Generic Algorithm;
BEGIN
  let f be a feasible flow in network G;
  determine the residual network G_f;
  WHILE G_f contains a directed path from the source node s to

```
        the sink node t DO
  BEGIN
     identify a decreasing path P from the source node s to the
     sink node t;
     r := min{r(i,j) | (i,j) ∈ P};
     decrease r units of flow along P;
     update the residual network G_f;
  END
END.
```

Actually, the algorithm terminates with optimal residual capacities. From these residual capacities we can determine a minimum flow in several ways. For example, we can make a variable change: For all arcs (i,j), let $c'(i,j) = c(i,j) - l(i,j)$, $r'(i,j) = r(i,j)$ and $f'(i,j) = f(i,j) - l(i,j)$. The residual capacity of arc (i,j) is

$$r(i,j) = c(j,i) - f(j,i) + f(i,j) - l(i,j).$$

Equivalently,

$$r'(i,j) = c'(j,i) - f'(j,i) + f'(i,j).$$

Similarly,

$$r'(j,i) = c'(i,j) - f'(j,i) + f'(j,i).$$

We can compute the value of f' in several ways; for example

$$f'(i,j) = \max(r'(i,j) - c'(j,i), 0)$$

and

$$f'(j,i) = \max(r'(j,i) - c'(i,j), 0).$$

Converting back into the original variables, we obtain the following expressions:

$$f(i,j) = l(i,j) + \max(r(i,j) - c(j,i) + l(j,i), 0)$$

and

$$f(j,i) = l(j,i) + \max(r(j,i) - c(i,j) + l(i,j), 0).$$

Theorem 2.1 (Integrality Theorem). *If all arc capacities and lower bounds are integers and there exists a feasible flow, then the minimum flow problem has an integer minimum flow.*

Proof. If all arc capacities and lower bounds are integers, the feasible flow, that is established as it is shown above by solving a maximum flow problem, is an integer flow.

We will prove the theorem by an induction argument applied to the number of decreases. Since the initial flow is an integer flow and all arc capacities and lower bounds are integers, the initial residual capacities are all integers. The flow decreased in any iteration equals the minimum residual capacity of the arcs, which form some directed path , which by the induction hypothesis is integer. Consequently, the residual capacities in the next iteration will again be integer.

Since the residual capacities $r(i,j)$, the lower bounds $l(i,j)$ and the arc capacities $c(i,j)$ are all integer, when we convert the residual capacities into flows by the method described previously, the arc flows $f(i,j)$ will be integer valued as well.

Since the residual capacities are integers, after each decrease the value of the flow becomes smaller with at least one unit. Since the value of the minimum flow cannot be smaller than the capacity of any cut, the algorithm will terminate in a finite number of iterations. □

Let $C = \max\{c(i,j) \,|\, (i,j) \in A\}$ and $L = \min\{l(i,j) \,|\, (i,j) \in A\}$.

Theorem 2.2. *The generic decreasing path algorithm runs in $O(nmC)$ time.*

Proof. The complexity of the algorithm is $\max(F(n,m,C,L), G(n,m,C,L))$, where $F(n,m,C,L)$ is the complexity of finding a feasible flow, and $G(n,m,C,L)$ is the complexity of establishing a minimum flow from a feasible flow.

In each iteration the algorithm performs a decrease. Obviously, each decrease requires $O(m)$ time because any arc is examined at most once. Therefore, $G(n,m,C,L)$ is $O(m)$ times the number of decreases. Since, the value of the feasible flow is at most nC, the value of the minimum flow is at least nL and after each decrease the value of the flow is smaller with at least one unit, the number of decreases is at most $nC - nL$, so $O(nC)$. Consequently, $G(n,m,C,L) = O(nmC)$.

Since the problem of determining a feasible flow is reduced to a maximum flow problem, if we solve this maximum flow problem using one of the augmenting path algorithms from Table 2.1, then the complexity of the generic decreasing path algorithm is $O(nmC)$. □

All the algorithms from Table 2.1 are augmenting path algorithms, i.e. algorithms which determine augmenting paths (by different rules) and then augment flows along these paths. The same transformation, that we made to generic augmenting path algorithm in order to obtain generic decreasing path algorithm, can be made to any of the other algorithm from Table 2.1. For each algorithm, we need to define the residual capacity of any arc (i,j) as the

maximum amount of flow from the node i to node j that can be cancelled: $r(i,j) = c(j,i) - f(j,i) + f(i,j) - l(i,j)$, as we did in the generic decreasing path algorithm.

Table 2.1

Augmenting path algorithm	Running time
Generic augmenting path algorithm	$O(nmC)$
Ford - Fulkerson labeling algorithm	$O(nmC)$
Gabow bit scaling algorithm	$O(nm \log C)$
Ahuja - Orlin maximum scaling algorithm	$O(nm \log C)$
Edmonds - Karp shortest path algorithm	$O(nm^2)$
Ahuja - Orlin shortest path algorithm	$O(n^2m)$
Dinic layered networks algorithm	$O(n^2m)$
Ahuja - Orlin layered networks algorithm	$O(n^2m)$

In view of this definition, any directed path from the source node s to the sink node t in the residual network corresponds to a decreasing path in the original network. But in view of the definition of the residual capacity for the maximum flow problem, any directed path from the source node s to the sink node t in the residual network corresponds to an augmenting path in the original network. Thus, both classes of algorithms have to establish directed paths from the source node s to the sink node t in the residual network. The algorithms for minimum flow problem will decrease the flow along these paths, instead of increasing it, as the algorithms for maximum flow problem do.

Consequently, the complexity of any minimum flow algorithm obtained by the transformation described previously will be equal to the complexity of the maximum flow algorithm from which it was obtained.

3. Preflow algorithms

3.1. Generic preflow algorithm

Before describing the generic preflow algorithm for the minimum flow problem, we introduce some definitions. In the residual network G_f, the *distance function* $d : N \to \mathcal{N}$ with respect to a given preflow f is a function from the set of nodes to the nonnegative integers. We say that a distance function is *valid* if it satisfies the following conditions:

$$d(s) = 0 \tag{6.a}$$

$$d(j) \leq d(i) + 1, \qquad \text{for every arc } (i,j) \in A_f. \tag{6.b}$$

We refer to $d(i)$ as the distance label of node i.

Lemma 3.1. (a) *If the distance labels are valid, the distance label* $d(i)$ *is a lower bound on the length of the shortest directed path from node* s *to node* i *in the residual network.*
(b) *If* $d(t) \geq n$, *the residual network contains no directed path from the source node to the sink node.*

Proof. (a) Let $P = (s = i_1, i_2, \ldots, i_k, i_{k+1} = i)$ be any path of length k from node s to node i in the residual network. The validity conditions imply that:

$$\begin{aligned} &d(i_2) \leq d(i_1) + 1 = d(s) + 1 = 1 \\ &d(i_3) \leq d(i_2) + 1 \leq 2 \\ &d(i_4) \leq d(i_3) + 1 \leq 3 \\ &\ldots \\ &d(i_{k+1}) \leq d(i_k) + 1 \leq k. \end{aligned}$$

(b) Since $d(t)$ is a lower bound on the length of the shortest path from s to t in the residual network and no directed path can contain more than $(n-1)$ arcs, imply that if $d(t) \geq n$, then the residual network contains no directed path from s to t. □

We say that the distance labels are *exact* if for each node $i, d(i)$ equals the length of the shortest path from node s to node i in the residual network.

We define the *layered network* as follows: The nodes are partitioned into layers $V_1, V_2, \ldots V_l$, where layer k contains the nodes whose exact distance labels equal k. Furthermore, for every arc (i,j) in the layered network $i \in V_k$ and $j \in V_{k+1}$ for some k. We say that f is a *blocking* preflow if the layered network contains no decreasing path.

We refer to an arc (i,j) from the residual network as an *admissible arc* if $d(j) = d(i) + 1$; otherwise it is *inadmissible.* We refer to a path from node s to node t consisting entirely of admissible arcs as an *admissible path.*

Lemma 3.2. *An admissible path is a shortest decreasing path from the source to the sink.*

Proof. Since every arc (i, j) in an admissible path P is admissible, the residual capacity of this arc and the distance labels of its end nodes satisfy the following conditions: (1) $r(i, j) > 0$
(2) $d(j) = d(i) + 1$.
Condition (1) implies that P is a decreasing path and condition (2) implies that if P contains k arcs then $d(t) = k$. Since $d(t)$ is a lower bound on the length of any path from the source to the sink in the residual network (from Lemma 3.1(a)), the path P must be a shortest decreasing path. □

We refer to a node i with $e(i) < 0$ as an *active* node. We adopt the convention that the source node and the sink node are never active.

The generic preflow algorithm for the minimum flow problem is the following:

Generic PreflowAlgorithm;

```
BEGIN
 let f be a feasible flow in network G;
 compute the exact distance labels d(·) in the residual
 network G_f;
 IF t is not labeled
    THEN f is a minimum flow
    ELSE
    BEGIN
       FOR each arc (i,t) ∈ A DO
                   f(i,t) := l(i,t);
       d(t) := n;
       WHILE the network contains an active node DO
       BEGIN
          select an active node j;
          IF the network contains an admissible arc (i,j)
             THEN pull g = min(−e(j), r(i,j)) units of flow from
                  node j to node i;
             ELSE d(j) = min{d(i) | (i,j) ∈ A_f} + 1
       END
    END
END.
```

A pull of g units of flow from node j to node i increases both $e(j)$ and $r(j, i)$ by g units and decreases both $e(i)$ and $r(i, j)$ by g units. We refer to the process

of increasing the distance label of node j, $d(j) = \min\{d(i) \mid (i,j) \in A_f\} + 1$, as a *relabel operation.*

This algorithm begins with a feasible flow and sends back as much flow, as it is possible, from the sink node to the source node. Because the algorithm decreases the flow on individual arcs, it does not satisfy the mass balance constraint (1.a) at intermediate stages. In fact, it is possible that the flow entering in a node exceeds the flow leaving from it. The basic operation of this algorithm is to select an active node and to send the flow entering in it back, closer to the source. For measuring closeness, we will use the distance labels $d(\cdot)$. Let j be a node with strictly negative deficit. If there exists an admissible arc (i,j), we pull the flow on this arc; otherwise we relabel the node j so that we create at least one admissible arc.

Theorem 3.3. *The generic preflow algorithm computes correctly a minimum flow.*

Proof. The algorithm terminates when the network does not contain any active nodes, therefore the current preflow is a flow. Since $d(t) := n$, the residual network contains no directed path from the source node to the sink node and Theorem 1.2 implies that the obtained flow is a minimum flow. □

Actually, the algorithm terminates with optimal residual capacities. From these optimal residual capacities, we can determine a minimum flow as we did in Section 2.

We refer to a pull of flow from node j to node i as a *canceling pull* if it deletes the arc (i,j) from the residual network; otherwise it is a *noncanceling pull.*

Theorem 3.4. *The generic preflow algorithm runs in $O(n^2m)$ time.*

Proof. To analyze the complexity of the generic preflow algorithm, we begin by establishing three important results:
(1) the total number of relabel operations is at most $2n^2$.
(2) the algorithm performs at most nm canceling pulls.
(3) the algorithm performs $O(n^2m)$ noncanceling pulls.

This results can be proved in a manner similar to the proof of complexity of the generic preflow algorithm for the maximum flow problem.

We can maintain the set L of active nodes organized as simply or doubly linked list, so that the algorithm can add, delete or select elements from it in $O(1)$ time. Consequently, it is easy to implement the generic algorithm in $O(n^2m)$ time. □

We suggest a practical improvement. We define a *minimum preflow* as a preflow with the minimum possible flow outgoing from the source node. The generic algorithm for minimum flow performs pull operations and relabel operations at active nodes until all the deficit reaches the source node or returns to the sink node. Typically, the algorithm establishes a minimum preflow long before it establishes a minimum flow. After establishing a minimum preflow, the algorithm performs relabel operations at active nodes until their distance label are sufficiently higher than n so it can pull flow back to the sink node. We can modify the generic algorithm in the following manner: we maintain a set N' of nodes that satisfy the property that the residual network contains no path from the source node to a node in N'. Initially, $N' = \{t\}$. We add nodes to N' in the following way: let $nb(k)$ be the number of nodes whose distance label is k. We can update $nb(\cdot)$ in $O(1)$ steps per relabel operation. Moreover, whenever $nb(k') = 0$ for some k', any node j with $d(j) > k'$ is disconnected from the set of nodes i with $d(i) < k'$ in the residual network. So, we can add any node j with $d(j) > k'$ to the set N'.

We do not perform pull or relabel operations for nodes in N' and terminate the algorithm when all active nodes are in N'. At this point, the current preflow is a minimum preflow. By the flow decomposition theory, any preflow f can be decomposed into a sequence of at most $O(n + m)$ paths and cycles. Let S be such a set of augmenting paths and cycles . Let $S_0 \subseteq S$ be a set of paths which start at a deficit node and terminate at sink node; let f^0 be the flow contributed by these path flows. Then $f^* = f + f^0$ will be a minimum flow.

3.2. Several implementations of generic preflow algorithm

The generic preflow algorithm does not specify a rule for selecting active nodes. By specifying different rules we can develop many different algorithms, which can be better than the generic algorithm. For example, we could select active nodes in FIFO order, or we could always select the active node with the greatest distance label, or the active node with the minimum distance label, or the active node selected most recently or least recently, or the active node with the largest excess or we could select any of active nodes with a sufficiently large excess.

FIFO preflow algorithm for minimum flow

In an iteration, the generic preflow algorithm for minimum flow selects a node, say node i, and performs a canceling or a noncanceling pull, or relabels the node. If the algorithm performs a canceling pull, then node i might still

be active, but, in the next iteration, the algorithm may select another active node for performing a pull or a relabel operation. We can establish the rule that whenever the algorithm selects an active node, it keeps pulling flow from that node until either its deficit becomes zero or the algorithm relabels the node. We refer to a sequence of canceling pulls followed either by a noncanceling pull or a relabel operation as a *node examination.*

The FIFO preflow algorithm for minimum flow examines active nodes in FIFO order. The algorithm maintains the set L of the active nodes as a queue. It selects an active node i from the front of L, performs pulls from this node and adds newly active nodes to the rear of L. The algorithm terminates when the queue of active nodes is empty.

The FIFO preflow algorithm for the minimum flow problem is the following:

FIFO Preflow Algorithm;

```
BEGIN
 let f be a feasible flow in network G;
 compute the exact distance labels d(·) in the residual network
 G_f;
 IF t is not labeled
    THEN f is a minimum flow
    ELSE BEGIN
            L := ∅;
            FOR each arc (i,t) ∈ A DO
            BEGIN
               f(i,t) := l(i,t);
               IF (e(i) < 0) AND (i ≠ s)
                  THEN add i to the rear of L;
            END;
            d(t) := n;
            WHILE L ≠ ∅ DO
            BEGIN
               remove the node j from the front of the queue L;
               pull/relabel(j);
            END
    END
END.
```

```
PROCEDURE pull/relabel(j);

BEGIN
 select the first arc (i,j) that enters in node j;
 B := 1;
 REPEAT
    IF (i,j) is an admissible arc
       THEN BEGIN
               pull g = min(-e(j), r(i,j)) units of flow from node
               j to node i;
               IF (i ∉ L) AND (i ≠ s) AND (i ≠ t)
                  THEN add i to the rear of L;
            END;
    IF e(j) < 0
       THEN IF (i,j) is not the last arc that enters in node j
            THEN select the next arc (i,j) that enters in node j
            ELSE BEGIN
                    d(j) = min{d(i) | (i,j) ∈ A_f} + 1;
                    B:=0;
                 END;
 UNTIL (e(j) = 0) or (B = 0);
 IF e(j) < 0
    THEN add j to the rear of L;
END;
```

Theorem 3.5. *The FIFO preflow algorithm computes correctly a minimum flow.*

Proof. The correctness of the FIFO preflow algorithm follows from the correctness of the generic preflow algorithm (Theorem 3.3). □

Theorem 3.6. *The FIFO preflow algorithm runs in $O(n^3)$ time.*

Proof. This theorem can be proved in a manner similar to the proof of the complexity of the FIFO preflow algorithm for the maximum flow. □

Highest-Label Preflow Algorithm for Minimum Flow

The highest-label preflow algorithm always pulls flow from an active node with the highest distance label. Let $h = \max\{d(i) \mid i \text{ is active}\}$. The algorithm first

examines nodes with distance label equal to h and pulls the flow from these nodes to nodes with distance labels equal to $h-1$ and, from these nodes, to the nodes with distance labels equal to $h-2$ and so on until the algorithm relabels a node or it has exhausted all the active nodes. When it has relabeled a node, the algorithm repeats the same process. If the algorithm does not relabel any node during n consecutive node examinations, all the deficit reaches the source or the sink and the algorithm terminates. Since the algorithm performs at most $2n^2$ relabel operations, we obtain a bound of $O(n^3)$ on the number of node examinations. Since each node examination entails at most one noncanceling pull, the highest-label preflow algorithm performs $O(n^3)$ noncanceling pulls. Therefore, it runs in $O(n^3)$ time.

The highest-label preflow algorithm is the same as the FIFO preflow algorithm, with the set L implemented not as a queue, but as a priority queue.

3.3. The blocking flow algorithm

In this section we describe a method for determining a blocking flow for the minimum flow problem. It is known that a minimum flow problem in a general network can be solved by determining $O(n)$ blocking flows in layered networks. We focus on the problem of finding a blocking flow in a given layered network.

The algorithm begins with a feasible flow f. During initializations, the algorithm sends back as much flow as it is possible from the sink node to its neighbors, creating deficits at each of these neighbors. After initializations, f becomes a blocking preflow. The algorithm moves the flow deficits through the network while maintaining a blocking preflow, until eventually the blocking preflow becomes a blocking flow. The algorithm maintains a partition of the nodes into two states: blocked and unblocked. We say that an arc (i, j) is *admissible* if it has a positive residual capacity and node i is unblocked. The algorithm blocks node j when it discovers that none of the arcs entering in j is admissible. Once j is blocked, every path from s to j contains a saturated arc. Deficits from blocked nodes are returned towards the sink node by increasing the flow on appropriate outcoming arcs.

The algorithm maintains a partition of the flow deficit into parts of deficit. A *part of deficit* is a maximal quantity of flow deficit that has moved in exactly the same way so far. A part of deficit p at a node i consists of an amount of deficit denoted by *size*(p); the node i is denoted by *pos*(p). We associate with a part of deficit p at node i a path from i to t, denoted by *path*(p), of arcs through which the deficit moved toward s, but not backward to t. During initializations, since the algorithm sends back the flow on every arc (i, t), it creates at each neighbor i of t a deficit p of size $f(i, t) - l(i, t)$ and with *path* (p) containing only the arc

(i, t). At each iteration, the algorithm selects a part of deficit p located at an active node and processes it in the following manner:

```
PROCEDURE process(p);

BEGIN
  j := pos(p);
  IF j is unblocked
     THEN IF there is an arc (i, j) in the residual network G_f and
     i is unblocked
        THEN BEGIN
                IF size(p) > r(i, j)
                   THEN BEGIN
                           create a new part of deficit p';
                           path(p') = path(p);
                           size(p') = size(p) − r(i, j);
                           size(p) = r(i, j);
                        END;
                pos(p) = i;
                path(p) = path(p) ∪ {(i, j)};
                r(i, j) = r(i, j)− size(p);
                e(i) = e(i)− size(p);
                e(j) = e(j)+ size(p);
             END;
        ELSE mark j as blocked;
  IF j is blocked
     THEN BEGIN
             let (j, k) be the last arc from path(p);
             pos(p) = k;
             path(p) = path(p) \ {(j, k)};
             r(j, k) = r(j, k)+ size(p);
          END;
END;
```

Let $j := pos(p)$. If j is not blocked, the algorithm tries to move a part of flow deficit p toward the source node. If there are no admissible arcs entering in j, then j becomes blocked; otherwise the algorithm selects an admissible arc, say (i, j). If $size(p) > r(i, j)$ then the part of deficit p is split into two parts of deficit. One part of deficit, of size equal to $size(p) - r(i, j)$, gets a new name p'.

The other one, of size equal to $r(i,j)$, retains the name p. The part of deficit p' remains at node j to be processed later and the part of deficit p moves to node i. If $size(p) \leq r(i,j)$ then the entire part of deficit p moves to node i. Finally, if the part of deficit p has not moved (i.e. node j is blocked) then the part of deficit p is returned to the node k from which it first reached j.

Theorem 3.7. *The blocking flow algorithm determines correctly a blocking flow in a layered network.*

Proof. The algorithm ends when there are no active nodes, i.e. when all the deficits are reached either the source node or the sink node. Consequently, at the end of the algorithm, the preflow becomes a flow. Moreover, this flow is a blocking flow because the preflow from which it was obtained was a blocking preflow. (After initializations, the algorithm obtains a blocking preflow and, during its execution, the algorithm maintains a blocking preflow and converts it into a blocking flow by moving flow deficits through the network.) □

Lemma 3.8. *The total number of parts of deficit created during an execution of the blocking flow algorithm is $O(m)$.*

Proof. First, we show that each increase in the number of parts of deficit corresponds to an arc saturation. Parts of deficit created during initializations are charged to the saturation of the corresponding arcs. A part of deficit created by splitting in procedure *process* is charged to the saturation of arc (i,j) in the same execution of the procedure. Thus, each new created part of deficit corresponds to an arc saturation. Since each arc becomes saturated at most once, the number of parts of deficit created by the algorithm is at most m. □

Lemma 3.9. *A part of deficit p moves from one node to another $O(n)$ times.*

Proof. A part of deficit p moves toward the sink node from node $j \neq \{s,t\}$ once j is blocked. After this, j is not in $path(p)$ and p never visits j again. Since a part of deficit can visit a node j at most two times, once when it moves toward the source node and once when it moves toward the sink node, it implies that a part of deficit can move $O(n)$ times. □

We define the phases of the algorithm as follows. Initializations are phase 1. Phase $i (i > 1)$ begins at the end of phase $i-1$ and ends as soon as every part of deficit that existed at the end of phase $i-1$ and every part of deficit created

by splitting since the end of the phase $i-1$ has moved at least one step. Since every part of deficit moves at least once during each phase, we have established the following result:

Lemma 3.10. *The number of phases of the blocking flow algorithm is* $O(n)$.

To obtain an efficient implementation of the algorithm, we maintain the set of parts of deficit located at active nodes as a queue and the path of each part of deficit as a stack of arcs. When a part of deficit moves toward the source node an arc is pushed on the top of the stack. When a part of deficit moves toward the sink node it moves to the tail of the top-of-stack arc and we pop the stack. Persistent stacks allow the push, pop and copy operation (necessary at part of deficit splitting) in constant time. Thus, we have established the following result:

Theorem 3.11. *The blocking flow algorithm for layered networks implemented using persistent stacks runs in* $O(nm)$ *time and* $O(nm)$ *space.*

We can find a minimum flow in a general network in the following manner: we start with a feasible flow, we determine the layered network, we find a blocking flow in the layered network and than we update the flow and we determine a new layered network. We repeat this process until the new constructed layered network contains no directed path from the source node to the sink node, consequently, the current flow is a minimum flow. It is known that we can determine a minimum flow by solving $O(n)$ blocking flow problems in layered networks. Thus, we can find a minimum flow in a general network in $O(n^2m)$ time and $O(nm)$ space.

4. The minmax algorithm

We can solve the minimum flow problem by determining a maximum flow from the sink node to the source node in the residual network. This approach is based on the following idea: the aim of the minimum flow problem is to send as few flow as possible from the source node to the sink node, that is opposite to the aim of the maximum flow problem. The algorithm that we will present computes a minimum flow in the following manner: knowing a feasible flow, we determine a minimum flow from the source node to the sink node by establishing a maximum flow in the residual network from the sink node to the source node. For determining a maximum flow from the sink to the source, we can use any

maximum flow algorithm, including preflow algorithms. The minimum flow algorithm is the following:

Minmax Algorithm;
BEGIN
 let f be a feasible flow in network G;
 determine the residual network G_f;
 establish a maximum flow f^* from t to s in G_f;
 f^* is a minimum flow from the source node s to the sink node t;
END.

For establishing the correctness of the algorithm we must introduce first some definitions and a theorem with respect to the maximum flow problem:

For the maximum flow problem, the *residual capacity* $R(i,j)$ of any arc $(i,j) \in A$, with respect to a given preflow f, is given by $R(i,j) = c(i,j) - f(i,j) + f(j,i) - l(j,i)$. The residual capacity of the arc (i,j) represents the maximum additional flow that can be sent from the node i to node j using the arcs (i,j) and (j,i). The network G_f consisting only of the arcs with positive residual capacity is referred to as the *residual network* (with respect to preflow f).

Theorem 4.1. *The minmax algorithm computes correctly a minimum flow.*

Proof. Let f^* be the flow at the end of the algorithm. Therefore, f^* is a maximum flow from t to s. In view of the Augmenting Path Theorem, there is no directed path from t to s in the residual network. This implies that there is a $t-s$ cut $\left[X, \overline{X}\right]$ with

$$R(i,j) = 0, \ \forall (i,j) \in \left(X, \overline{X}\right).$$

Equivalently,

$$c(i,j) - f(i,j) + f(j,i) - l(j,i) = 0, \ \forall (i,j) \in \left(X, \overline{X}\right).$$

In view of condition (1b), this implies that

$$f(i,j) = c(i,j), \ \forall (i,j) \in \left(X, \overline{X}\right)$$

and

$$f(j,i) = l(j,i), \ \forall (j,i) \in \left(\overline{X}, X\right).$$

Equivalently,

$$r(j,i) = c(i,j) - f(i,j) + f(j,i) - l(j,i) = 0, \ \forall (j,i) \in \left(\overline{X}, X\right). \qquad (7)$$

Since $\left[X, \overline{X}\right]$ is a $t - s$ cut, $\left[\overline{X}, X\right]$ is a $s - t$ cut. Using (7), we obtain that residual network G_{f^*} contains no directed path from the source node s to the sink node t. In view of Decreasing path theorem, f^* is a minimum flow. □

Theorem 4.2. *The complexity of the minimum flow algorithm is the complexity of the maximum flow algorithm used for determining a feasible flow and for establishing a maximum flow from t to s.*

5. Parallel algorithms for minimum flow problem

In section 3, we have presented sequential preflow algorithms for minimum flow that we obtained by modifying known preflow algorithms for maximum flow. The same transformations can be applied to any parallel version of these algorithms. If we modify Shiloach and Vishkin's parallel algorithm for maximum flow we obtain a parallel algorithm for minimum flow that can be implemented with n processors and runs in $O(n^2 \log n)$ time and $O(n^2)$ space. If we consider two parallel implementations of highest-label preflow algorithm for maximum flow made by Andrew V. Goldberg, we obtain two parallel p reflow algorithms for minimum flow problem that use $p = O(\sqrt{m})$ processors. These processors-efficient implementations are obtained by assigning the work to processors in such a way that most processors are busy most of the time. To reduce the space requirement, sophisticated data structures are used. The first parallel implementation of highest-label preflow algorithm for minimum flow uses a dynamic array and runs in $O\left(n^2 \log(m/n + p)(\sqrt{m}/p)\right)$ time within $O(m + n \log n)$ space. The second one uses the parallel cardinality stack structure, which is based on the parallel 2-3 trees and runs in $O(n^2 \log n(\sqrt{m}/p))$ time within $O(m + n)$ space.

5.1. A parallel implementation of the blocking flow algorithm

In this section we present a parallel implementation of the algorithm for finding a blocking flow for the minimum flow problem in a layered network. The parallel implementation works in pulses; at each pulse, every part of deficit, including those arising by splitting, moves either toward the source node or toward the sink node or both. Thus each pulse completes at least one phase. The parallel implementation begins with a feasible flow f and consists in the following five steps:

STEP 1. (initialize) For each arc (i,t) create a part of deficit at node i of size $f(i,t) - l(i,t)$ and having the stack containing only arc (i,t) and then set $f(i,t) = l(i,t)$.

Block node t and unblock all other nodes.

STEP 2. (push deficits toward the source node) For each unblocked node $i \notin \{s,t\}$ do in parallel the following: Arbitrarily order the parts of deficit located at node j, say $p_1, p_2, \ldots, p_k$ and admissible arcs, say $(i_1,j),(i_2,j),\ldots,(i_q,j)$. For $y = \overline{1,k}$ compute a cumulative size $S(y) = \sum_{x=1}^{y} size(p_x)$. For $y = \overline{1,q}$ compute a cumulative residual capacity $R(y) = \sum_{x=1}^{y} r(i_x,j)$. Assign the part of deficit p_x to the admissible arcs (i_y,j) as follows:

(1) If $S(x-1) \geq R(y-1)$ and $S(x) \leq R(y)$ then assign all parts of deficit p_x to (i_y,j).

(2) If $S(x-1) \geq R(y-1)$ and $S(x) > R(y)$ then assign an amount of $\max\{0, R(y) - S(x-1)\}$ of part of deficit p_x to (i_y,j).

(3) If $S(x-1) < R(y-1)$ and $S(x) > R(y)$ then assign an amount of $r(i_y,j)$ of part of deficit p_x to (i_y,j).

(4) If $S(x-1) < R(y-1)$ and $S(x) \leq R(y)$ then assign an amount of $\max\{0, S(x) - R(y-1)\}$ of part of deficit p_x to (i_y,j).

Split any part of deficit assigned to more than one arc into two or more parts of deficit, one per assigned arc, each of size equal to the amount of the original part of deficit assigned to the arc. Each of these new parts of deficit inherits the assignment of the corresponding amount of the old part of deficit, as well as a copy of the stack of the old part of deficit. For each admissible arc (i_y,j) decrease $f(i_y,j)$ by the sum of the sizes of the parts of deficit assigned to (i_y,j) and move each such parts of deficit to i_y, pushing (i_y,j) to its stack. If all arcs (i_y,j) are now saturated, mark j to be blocked.

STEP 3. (block nodes) Block every node marked to be blocked in Step 2.

STEP 4. (push deficits toward the sink node) For each blocked node $j \notin \{s,t\}$ do in parallel the following:

For each part of deficit p at j, let (j,v) be the top arc on $stack(p)$. Pop $stack(p)$, increase $f(j,v)$ by $size(p)$ and move p to v.

STEP 5. (loop) If every part of deficit is at s or t then stop; otherwise go to Step 2.

By using standard techniques of parallel computation, we can implement each step of the algorithm to run in $O(\log n)$ time on a m-processor PRAM. Consequently, by Lemma 3.10, we have established the following theorem:

Theorem 5.1. *The parallel implementation of the blocking flow algorithm runs in $O(n \log n)$ time and $O(nm)$ space.*

We can determine a minimum flow in a general network by solving $O(n)$ blocking flow problems in layered networks. Thus, by using the parallel implementation of the blocking flow algorithm, we can find a minimum flow in a general network in $O(n^2 \log n)$ time and $O(nm)$ space.

6. An application of the minimum flow problem

In this section we present the machine setup problem. A job shop needs to perform p tasks on a particular day. It is known the start time $\pi(i)$ and the end time $\pi'(i)$ for each task $i, i = 1, \ldots, p$. The workers must perform these tasks according to this schedule so that exactly one worker performs each task. A worker cannot work on two jobs at the same time. It is known the setup time $\pi_2(i, j)$ required for a worker to go from task i to task j. We wish to find the minimum number of workers to perform the tasks. We can formulate this problem as a minimum flow problem in the network $G = (N, A, l, c, s, t)$, where $N = N_1 \cup N_2 \cup N_3 \cup N_4$, $N_1 = \{s\}, N_2 = \{i \,|\, i = 1, \ldots, p\}$, $N_3 = \{i' \,|\, i' = 1, \ldots, p\}$, $N_4 = \{t\}, A = A_1 \cup A_2 \cup A_3 \cup A_4, A_1 = \{(s, i) \,|\, i \in N_2\}, A_2 = \{(i, i') \,|\, i, i' = 1, \ldots, p\}, A_3 = \{(i', j) \,|\, \pi'(i') + \pi_2(i', j) \leq \pi(j)\}, A_4 = \{(i', t) \,|\, i' \in N_3\}, l(s, i) = 0, c(s, i) = 1, (s, i) \in A_1, l(i, i') = 1, c(i, i') = 1, (i, i') \in A_2, l(i', j) = 0, c(i', j) = 1, (i', j) \in A_3, l(i', t) = 0, c(i', t) = 1, (i', t) \in A_4$. We solve the minimum flow problem in the network $G = (N, A, l, c, s, t)$ and the value of the minimum flow is the minimum number of workers that can perform the tasks.

References

1. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows. Theory, al gorithms and applications*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1993.
2. R. Ahuja, T. Magnanti and J. Orlin, *Some Recent Advances in Network Flows*, SIAM Review **33** (1990), 175-219.
3. R. Ahuja and J. Orlin, *A Fast and Simple Algorithm for the Maximum Flow Problem*, Operation Research **37** (1988), 748-759.

4. R. Ahuja, J. Orlin and R. Tarjan, *Improved Time Bounds for the Maximum Flow Problem.* SIAM Journal of Computing **18** (1988), 939-954.
5. A. V. Goldberg, *Processor-efficient implementation of a maxim um flow algorithm*, Information Processing Letters **38** (1991), 179-185.
6. A. V. Goldberg, *A New Max-Flow Algorithm*, MIT, Cambridge, 1985.
7. A. V. Goldberg and R. E. Tarjan, *A New Approach to the Maximum Flow Problem*, Journal of ACM **35** (1988), 921-940.
8. A. V. Goldberg and R. E. Tarjan, *A Parallel Algorithm for Fin ding a Blocking Flow in an Acyclic Network*, Information Processing Letters **31** (1989), 265-271.
9. A. V. Karzanov, *Determining the Maximum Flow in a Network by the Method of Preflows*, Soviet. Math. Dokl. **15** (1974), 434-437.
10. Y. Shiloach and U. Vishkin, *An* $O(n^2 \log n)$ *parallel max- flow algorithm*, J. Algorithms **3** (1982) 128-146.

Dr. Eleonor Ciurea is a full professor at the "Transilvania" University of Braşov, Department of Computer Science. He has been an academic of this University for over 25 years. He is a member of the Scientists' Associations of Romania, of the Society of Mathematical Sciences of Romania and of the American Mathematical Soci ety. He also is a reviewer for the abstracting journals: *Zentralblatt Fur Mathematik* and *Mathematical Reviews.* His field of research is algorithmic graphs (particularly dynamic network flow). He has published articles addressing the topic of dynamic flow in France, Belgium, USA, England, Korea etc.

Department of Computer Science *Transilvania* University 50, Iuliu Maniu 2200 Braşov, Romania
e-mail: eciurea@unitbv.ro

Laura Ciupală is an assistant at the "Transilvania" University of Braşov, Department of Computer Science. Her field of research is theory of graphs (particularly network flows). S he has published articles addressing the topic of minimum flow in Moldavia and Romania.

Department of Computer Science *Transilvania* University 50, Iuliu Maniu 2200 Braşov, Romania
e-mail: cciupala@yahoo.com