Acta Informatica 14, 221–242 (1980)

Acta Informatica

An $O(V^{5/3}E^{2/3})$ Algorithm for the Maximal Flow Problem*

Zvi Galil

Department of Mathematical Sciences, Computer Science Division, Tel-Aviv University, Ramat-Aviv, Tel-Aviv, Israel

Summary. A new algorithm for finding a maximal flow in a given network is presented. The algorithm runs in time $O(V^{5/3}E^{2/3})$, where V and E are the number of the vertices and edges in the network.

1. Introduction

One of the well-known problems in combinatorial optimization is the problem of finding the maximal flow in a given network ("max-flow" in short). The problem was posed and solved by Ford and Fulkerson [8]. (See also [7].) This paper presents the fifth improvement on the original solution. The interesting history of the problem is summarized in Table 1.

Table 1. The various solutions

Solution	Year	Time	Improvement	Space
Ford and Fulkerson	1956	–	–	E
Edmonds and Karp	1969	E^2V	–	E
Dinic	1970	EV^2	E/V	E
Karzanov	1973	V^3	E/V	V^2
Cherkasky	1976	$V^2E^{1/2}$	$(V^2/E)^{1/2}$	E
Galil	1978	$V^{5/3}E^{2/3}$	$(V^2/E)^{1/6}$	E

Our model of computation is the Ramdom Access Machine with unit cost [2]. The fourth column in Table 1 gives the factor of improvement of each solution upon its predecessor. These are always improvements because $V \leqq E \leqq V^2$.

* We use the notation $A=O(B)$ $[A=\Omega(B)]$ for $A \leqq cB$ $[A \geqq cB]$, and $A=\theta(B)$ for $c_1 B \leqq A \leqq c_2 B$ where c, c_1 and c_2 are positive constants. (The same constants for all the occurrences of this notation)

0001-5903/80/0014/0221/$04.40

All the bounds of Table 1 can be shown to be tight: in [9] we construct *one* family of network flow problems such that all the algorithms in Table 1 run in time proportional to the corresponding entry in the third column.

Table 1 does not include two recent solutions for the max-flow problem. In 1978, Malhotra, Pramodh Kumar and Maheshwari discovered another solution with time complexity $O(V^3)$ [13]. Their algorithm is very simple, but it has not improved the best known algorithm. Following the discovery of the algorithm described in this paper, the author together with A. Naamad discovered an $O(EV\log^2 V)$ algorithm [10]. This more recent algorithm is better for sparse networks. The algorithm given here is still best in a nontrivial range of E (in terms of V). In the concluding section we will give a more detailed comparison between these two algorithms.

Except the third algorithm, each algorithm started with the previous algorithm and showed how to execute faster some parts in it. Each improvement is nontrivial and it took at least one year each time to come up with the better solution.

Ford and Fulkerson posed the problem and gave the first solution which starts from some legal flow (usually the zero flow) and successively increments it using flow augmenting paths (*f.a.p.'s*). They proved the famous "max-flow min-cut theorem" that implies that when their algorithm (or any one of the improvements) halts then it has computed a maximal flow. They also showed that their algorithm terminates if all capacities are integral. They were not concerned with the number of steps of their algorithm. However, in the case of a maximal flow of size M and of integral capacities, their algorithm runs in time $O(ME)$. They observed that in the case of introducing several irrational capacities their algorithm may not terminate and what is worse, it may converge to the wrong flow: If the maximal flow is of size M, one can construct for any $k>0$ an example where their algorithm converges to a flow of size smaller than M/k. This weakness is not too bad because we usually deal with rational quantities in which case their algorithm terminates and correctly solves the problem. The main weakness of their algorithm is that even for integral capacities the bound $O(ME)$ cannot be improved, as is shown by a trivial example in [5]. So the number of steps cannot be bounded by a function that depends solely on the graph. Since we usually represent numbers in binary or decimal notation it follows that M can be very large and if we take into account that M can be represented by a number of size $\log_2 M$ then the original algorithm is exponential in the length of the input.

Edmonds and Karp fixed the first algorithm. They observed that by imposing a simple order on the search for an f.a.p. the weaknesses mentioned above simply disappear. They used the rule of "first labeled first scanned" which essentially imposes a "breadth first search" on the graph leading to the discovery of one of the shortest f.a.p.'s. In fact, those who implemented the original algorithm might have used this strategy unknowingly and thus guaranteed polynomial time bound. (Ford and Fulkerson did not specify the order in which the graph should be scanned.)

Independently of Edmonds and Karp, Dinic discovered too the idea of using the shortest f.a.p.'s first. But in addition he found a better way to use this idea.

His algorithm is divided into phases. In each phase an auxiliary network is constructed from the given network and the present flow. We call this network a layered network because its set of vertices is partitioned into disjoint subsets called layers $V_0, V_1, \ldots, V_k$, where $V_0 = \{s\}$, $V_k = \{t\}$ and every one of its edges goes from one layer to the next layer. Also, $v \in V_i$ if the shortest f.a.p. from s to v is of length i. Dinic then uses the layered network to find enough f.a.p.'s to t of length k until every path from s to t in the layered network contains a saturated edge. While Edmonds and Karp exhaust all shortest f.a.p.'s in time $O(E^2)$, Dinic does it with the help of the layered network in time $O(EV)$. The reason is that Edmonds and Karp start from scratch each time they look for a shortest f.a.p., while Dinic's layered network helps to keep the information needed for finding a shortest f.a.p. The number of phases of Dinic's algorithm is bounded by V since at the end of a phase no more f.a.p.'s of length k exist, and the shortest f.a.p. is of size larger than k.

The last three improvements improve the execution of a phase in Dinic's algorithm: Karzanov does it in time $O(V^2)$, Cherkasky in time $O(V\sqrt{E})$ and the new algorithm in time $O(VE)^{2/3})$. The corresponding times for the max-flow problem are $O(V^3)$, $O(V^2\sqrt{E})$ and $O(V^{5/3}E^{2/3})$ respectively because there can be at most V phases and the total time to construct the layered networks is $O(VE)$ ($O(E)$ per a layered network) is not of a larger order of magnitude.

Karzanov's innovation was to allow preflows in intermediate stages. Preflows are not legal flows in the sense that some vertices other than s or t do not satisfy the conservation rule: more amount of flow enters them than leaves them. These excesses of flow on some vertices are generated while pushing flow from layer to layer towards t. The excesses are later shifted one layer back and an attempt is made to push them forward through an alternative way. Karzanov beats Dinic because he discovers several f.a.p.'s at once. He pays a price for unsuccessful attempts when he has to reroute excesses he cannot push forward. But it turns out that this price is not too big.

Cherkasky refined Karzanov's solution in the following way. He partitions the layers into blocks of consecutive layers which we call here superlayers. He essentially applies Karzanov's algorithm to the superlayers. On the superlayers themselves Cherkasky uses Dinic's method. So surprisingly, by combining the methods of Dinic and Karzanov, Cherkasky came up with an algorithm better than both. Cherkasky beats Karzanov due to a sort of "divide and conquer" strategy: By using Karzanov's method to superlayers he reduces the price mentioned above in the case of rerouting excesses. On the other hand, the use of Dinic's approach, although inferior for the whole network, turns out to be good locally because we only need to go a relatively short distance (the depth of a superlayer) to find if we can push flow through a superlayer.

The new algorithm like Cherkasky's applies Karzanov's algorithm to superlayers but it uses different methods while working with the superlayers. For each superlayer we maintain a forest (the forest of the i-th superlayer is called FOREST_i). In FOREST_i an edge stands for a path in the layered network. FOREST_i enables us to save time by making shortcuts: we can walk along a (possibly long) path in one step. Although the main idea is quite simple, the

technical details of how to use the FOREST$_i$'s and how to maintain them are quite involved.

The correctness of the new algorithm and of Cherkasky's algorithm is intuitively clear but requires a proof. In both algorithms the computation of the running time is interesting: Various operations are charged to various accounts, sometimes in a quite subtle way.

In addition to improving the time complexities, the two last improvements improve the space complexity of Karzanov's algorithm from $O(V^2)$ to $O(E)$. So, the improvement of the new algorithm over Karzanov's algorithm is largest with respect to space and time when the graph is sparse ($E \approx V$).

In this paper we assume that the reader is familiar with the original algorithm and its first three improvements. In the second section we give a formal definition of the problem, mainly to introduce our notation. Then we describe the subproblem that we solve, that is the equivalent of a phase in Dinic's algorithm. Then we describe in short Karzanov's algorithm. In addition to the original papers the reader can find the details of the algorithms by Dinic and Karzanov in a nice expository paper by S. Even [6]. In Sect. 3 we describe in detail our version of Cherkasky's algorithm. In Sect. 4 we present our new algorithm. We have included Cherkasky's algorithm for the sake of completeness since many of the proofs will carry over to our algorithm. Also, Cherkasky's paper is currently not available in English. The appendix contains a glossary for the various definitions in the algorithms of Sect. 3 and Sect. 4.

We assume that the reader is familiar with the well-known data structures [12], and we usually omit the details about their implementation. A special use will be made of doubly linked lists. We will have doubly linked lists of edges. Obviously an edge can be a member of two or three lists. Also, given an edge we can check if it is in a certain list, or delete it in a constant amount of time. The edges will stay in the same place in memory. Only the appropriate links will be changed.

2. Preliminaries

The *max-flow problem* is defined as follows. A *network* is a directed graph $G=(V, E)$, where V is the set of vertices that includes two distinguished vertices s and t, and E the set of edges, together with a *capacity function* that assigns to each edge e a real number $c(e)>0$, the *capacity* of e. For each vertex v in V we define $in(v)$ [$out(v)$] as the set of edges that enter [leave] v. A *legal flow* (or a flow in short) is a function $f: E \rightarrow R$ satisfying two constraints: (1) the *capacity constraint:* $0 \leqq f(e) \leqq c(e)$ for every e in E; and (2) the *conservation rule:*

$$excess(v) \equiv \sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) = 0$$

for every vertex v other than s or t. The quantity $excess(t)$ $(=-excess(s))$ is called the *value of the flow.* The *max-flow problem* is: given a network, find a legal flow f with maximal value among legal flows.

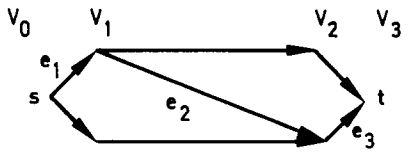

Fig. 1

We will solve in this paper only a subproblem that arises in solving the max-flow problem and is equivalent to a phase in Dinic's algorithm: *A layered network* is a network whose vertices are partitioned into disjoint sets called layers, $V_0, V_1, \ldots, V_k$, where $V_0=\{s\}$, $V_k=\{t\}$ and if $e=(u,v)$ is an edge in the network then $u\in V_i$ and $v\in V_{i+1}$ for some $0\leqq i\leqq k-1$. (By definition of a network each edge has a positive capacity.) Given a legal flow in the layered network, we say that a vertex v is *blocked* if all paths from v to t contain a *saturated* edge (i.e. its flow equals its capacity).

The subproblem we will deal with is: given a layered network, find a legal flow on it in which s is blocked. Note that such a flow is not necessarily maximal: Consider Fig. 1 with all capacities 1, and flow 1 in e_1, e_2, e_3 and 0 elsewhere. The source s is blocked but the flow is not maximal.

We describe in short Dinic's and Karzanov's algorithms and in detail Cherkasky's algorithm and the new algorithm. All solve the subproblem described above. Our presentation differs slightly from the original algorithms so that these algorithms fit better in a unified framework. We first explain some details that are common to all these algorithms.

The layered network is represented by the lists $out(v)$ for all vertices in V. (In the sequal $out(v)$ and $in(v)$ will refer to the corresponding sets of edges in the layered network.) Each list $out(v)$ is ordered according to an arbitrary but fixed order. The algorithm will consider $out(v)$ according to this order. During the execution, the algorithm will sometimes *close* some edges. At any time and for every vertex v a pointer will point to the first *open* (non-closed) edge in $out(v)$. In $out(v)$ all closed edges appear first, and initially all edges are open. When we close an edge the corresponding pointer advances to the next (open) edge. If the list $out(v)$ is exhausted v is said to be *closed*. Otherwise v is *open*. The vertex t will always be considered as an open vertex. This will cause no confusion because we never push flow from t. An *open path* in the layered network is a path that contains only open vertices and edges.

Dinic solved the subproblem by starting from s and using a "depth first search" on the open edges towards t. If an open path is found, the flow is increased on it as much as possible and we close the edges on the path that become saturated. If we reach a dead end (a closed vertex) we close the last edge and back up one vertex and continue from there. So at least once per k steps we close an edge, hence the number of steps in $O(kE)=O(VE)$.

Karzanov's solutions allows *preflows* which like flows satisfy the capacity constraint and unlike flows satisfy a relaxed conservation rule, namely $excess(v)\geqq 0$ for $v\neq s,t$. A vertex v, $v\neq t$, with $excess(v)>0$ is a *positive vertex*. We exclude t because its positive excess is legal. A vertex v with $excess(v)=0$ is

said to be *balanced*. The algorithm alternates between pushing the flow from the closest layer to t with positive vertices as far as possible, possibly generating new positive vertices, and balancing positive vertices that are generated during these pushes. (We *balance* a vertex v by reducing enough flow on edges in $in(v)$ so that its excess becomes 0.)

Pushing flow is achieved by a repeated call to a routine PUSH(i) with increasing i. PUSH(i) pushes the flow from V_i to V_{i+1}. It considers in turn each open positive vertex in V_i. For such a vertex v it considers in turn each open edge in $out(v)$. Assume the edge considered is e. If e leads to a closed vertex we close it and consider the next (open) edge in $out(v)$. Otherwise we push through e the maximal possible amount of flow. (The two constraints that exist are the current excess of v and the amount needed to saturate the edge.) If e becomes saturated we close it and consider the next (open) edge in $out(v)$. So after dealing with v, either v becomes balanced or v becomes closed. During PUSH(i) we record the history of the flow increments: for every vertex u we maintain a stack HISTORY(u) and when we increment the flow in the edge (v, u) we add on top of HISTORY(u) the pair (v, δ), where δ is the amount of the increment. The routine PUSH(i) is successful if some flow is pushed to V_{i+1}. If it is successful we call PUSH($i+1$) and so on.

For balancing we call a routine BALANCE(i), where V_i is the closest layer to t with positive vertices all of which must be closed. BALANCE(i) uses the stacks to shift back flow from V_i to V_{i-1}. It balances all positive vertices in V_i that are all closed. The balance is successful if some open vertices in V_{i-1} have become positive. In this case BALANCE(i) is followed by PUSH($i-1$). Otherwise it is followed by BALANCE($i-1$).

$i \leftarrow 0$

PLOOP: PUSH(i)
If the push was successful and $i+1<k$ then [$i \leftarrow i+1$; go to PLOOP]
If there are no positive vertices STOP
Otherwise, let i be the maximal number of a layer that contains positive vertices

BLOOP: BALANCE(i)
$i \leftarrow i-1$
if $i=0$ STOP
If the balance was successful go to PLOOP
Otherwise go to BLOOP.

By maintaining a stack that contains in increasing order the numbers of layers with positive vertices we can always find immediately which layer to balance next. The obvious details are left to the reader. Karzanov proved the following lemmas.

Lemma 2.1. *If a vertex v becomes closed, then it is blocked from this point on.*

Lemma 2.2. *We balance each vertex at most once.*

Lemma 2.2 implies that the PLOOP and BLOOP can repeat at most V times, and thus the algorithm terminates. Since upon termination all vertices

must be balanced, the final preflow is a legal flow. Since s is closed after the first PUSH(0) it will be blocked at the end of the algorithm by Lemma 2.1. Thus the algorithm is correct.

We now compute the running time of the algorithm. The number of flow decrements on edges is bounded by the number of flow increments because we balance each vertex at most once and use the history of increments in the stacks to decrease the flow. Thus it suffices to compute the number of flow increments. There can be at most E times in which edges become closed. When we push flow from v in V_i only the last edge considered in $out(v)$ does not necessarily become closed. Thus the total number of times we consider an edge and not close it is bounded by V^2. (At most one per vertex per execution of the PLOOP.) Hence the total time is $O(V^2)$ since $E \leqq V^2$.

3. Cherkasky's Algorithm

We choose some layers and call them *special layers* and their vertices *special vertices*. The way we choose them will be described later. The only facts that matter are the following: The first and last layers are special, so s and t are special vertices. The number of special vertices is m and the distance between two consecutive special layers is bounded by x. A part of the layered network which consists of two consecutive special layers and all layers in between them is called a *superlayer*. The superlayers are numbered from 1 to say k' in the obvious way. When we deal with the i-th superlayer SL_i we assume that it contains the layers $V_p, V_{p+1}, \ldots, V_q$. V_q is called the *front* layer of SL_i and V_p the *rear* layer of SL_i. A vertex v belongs to SL_i if it is in any layer in SL_i other than its rear layer. (So the vertices of V_p belong to SL_{i-1}.) The algorithm will use two routines PUSH and BALANCE that we now describe.

1. PUSH(i): When PUSH(i) is called there are positive vertices only in special layers V_z $z \leqq p$. In V_p there are positive vertices all of which are open. The other positive vertices are all closed. The routine tries to push the flow through SL_i to open vertices in V_q. At the end of a call all vertices in V_p that are still positive become closed. PUSH(i) is *successful* if at the end some flow was pushed to V_q, so some open vertices in V_q become positive.

2. BALANCE(i): When BALANCE(i) is called there are positive vertices in V_q and all positive vertices are closed and are in special layers V_z, $z \leqq q$. The routine tries to reroute the excesses of flow from the closed positive vertices in V_q to open vertices in V_q. At the end of a call all positive vertices in V_z, $z \leqq p$ are still closed and all positive vertices in V_q (if any) are open. BALANCE(i) is successful if at the end some flow was rerouted to V_q, so some open vertices in V_q become positive. In case it fails all excesses in V_q are brought back to V_p. A successful BALANCE(i) may shift back to V_p some of the excesses in V_q. In both cases BALANCE(i) may generate new closed positive vertices in V_p.

Ignoring the details of these routines the new algorithm is essentially Karzanov's algorithm applied to superlayers.

$i \leftarrow 1$

PLOOP: PUSH(i)
If the push was successful and $i < k'$ then [$i \leftarrow i+1$; go to PLOOP]
If there are no positive vertices STOP
Otherwise let i be the maximal number of a superlayer whose front layer contains positive vertices
BLOOP: BALANCE(i)
If the balance was successful then [$i \leftarrow i+1$; go to PLOOP]
Otherwise: if $i=1$ then STOP else [$i \leftarrow i-1$; go to BLOOP]

Fig. 2. The macro-algorithm of Cherkasky's algorithm

The PLOOP pushes flow as far as it can. The BLOOP tries to reroute the flow and if it fails the excesses are shifted to the previous special layer and a new attempt to reroute the flow is started. Assuming the PUSH(j) [BALANCE(j)] indeed does what we described above for $1 \leqq j \leqq k'$ then one can show that the initial conditions for PUSH(i) [BALANCE(i)] hold whenever it is called. (The proof is by induction on the time of the call.) Note that finding the number of the superlayer closest to t whose front layer contains positive vertices can be easily handled by using a stack as in Karzanov's algorithm. We now give the details of PUSH(i) and BALANCE(i).

PUSH(i) looks for open paths from positive open vertices in V_p to open vertices in V_q in a manner very similar to Dinic's algorithm. Assume it considers a positive open vertex v in V_p. It constructs an open path starting at v. If it fails i.e. it reaches a closed vertex (possibly in V_q) it closes the last edge in the path constructed so far backs up to the last vertex and continues from it. If it succeeds, let $\delta(v)$ be the excess of v and $\varepsilon(\pi) = \min_{e \in \pi}(c(e) - f(e))$, where π is the path that was just found. There are two cases: 1. $\delta(v) < \varepsilon(\pi)$ and 2. $\delta(v) \geqq \varepsilon(\pi)$. In the first case the flow is increased along π by $\delta(v)$ and v becomes balanced. In the second case the flow is increased along π by $\varepsilon(\pi)$ and we close the edges that become saturated and there is at least one such edge. [Flow is increased in an edge e by increasing $f(e)$]. The searches for open paths from v are continued until either v becomes balanced or v becomes closed. Sooner or later one of these will happen and then the next positive and open vertex in V_p is dealt with and so on. Finally all the vertices in V_p that are still positive must be closed and the positive vertices in V_q must be open. Fact 1 follows immediately from the construction of PUSH(i).

Fact 1. For every path except possibly the last one that is constructed from a positive vertex in V_p we close at least one edge.

In addition to increasing the flow in the edges of SL_i, PUSH(i) records the changes in a way slightly different from Karzanov's algorithm. For every SL_i, the number of times PUSH(i) was executed, ℓ_i, is maintained. For every vertex v in SL_i we maintain a variable *pushnumber*(v) which is the number of the last call to PUSH(i) during which or after which some flow was pushed into v. For every

edge $e=(u,v)$ in SL_i we maintain a variable *newflow*(e) that contains the total increment of flow on e since the ℓ-th call to PUSH(i), where *pushnumber*$(v)=\ell$. (By total increment we mean that it takes into account also flow reductions on e.) For every vertex v in SL_i we also maintain a doubly linked list $\Delta(v)$, where $\Delta(v)=\{e \mid e \in in(v)$, *newflow*$(e)>0\}$. When we push δ units of flow in an edge $e=(u,v)$ in SL_i there can be several cases: (1) *pushnumber*$(v)=\ell_i$ and $e\in\Delta(v)$: we increase *newflow*(e) by δ; (2) *pushnumber*$(v)=\ell_i$ and $e\notin\Delta(v)$: we insert e into $\Delta(v)$ and set *newflow*(e) to δ; (3) *pushnumber*$(v)<\ell_i$: We first set *newflow*(e') to 0 for every e' in $\Delta(v)$ while emptying $\Delta(v)$, and then update *pushnumber*(v) to ℓ_i, insert e into $\Delta(v)$ and set *newflow*(e) to δ. We charge the cost of deleting an edge from $\Delta(v)$ to the operation of inserting it into $\Delta(v)$. Consequently, the following fact holds:

Fact 2. The cost of increasing the flow in an edge is constant.

BALANCE(i) starts with balancing V_q. For every positive vertex v in V_q (which must be closed), $\Delta(v)$ and *newflow*(e)'s for e's in $\Delta(v)$ are used to reduce the flow that enters v until it is balanced. We will show below (in Fact 5) that all *newflow*(e) for e's in $\Delta(v)$ will suffice for balancing a positive vertex v in V_q. The flow reduction is done by taking one element of $\Delta(v)$ at a time and decreasing the flow on the corresponding edge until v becomes balanced. So, for every edge e considered in balancing a vertex v except possibly the last one, the complete increment of flow *newflow*(e) is cancelled and e is deleted from $\Delta(v)$. This process is repeated until all vertices in V_q are balanced. As a result some vertices in V_{q-1} become temporarily positive and we next balance V_{q-1}.

Inductively, after balancing V_{y+1}, $p\leqq y<q$ we generate positive vertices in V_y and we balance V_y as follows. First we consider the open positive vertices which we call *microsources*. We execute on them a *micropush* of flow that is essentially the same as the push of flow that is done in PUSH(i) from open positive vertices in V_p. We search for open paths from microsources to open vertices in V_q. After each successful search the flow is increased in the path found with pushnumber ℓ_i. Intuitively this is because a flow pushed in the last PUSH(i) is now rerouted. Fact 1 still holds for microsources. When the micropushes end, if either there are no more positive vertices in V_y (all excesses have been rerouted) or $p=y$, then BALANCE(i) ends. Otherwise ($y>p$ and there are positive vertices in V_y all of which are closed), the excesses of flow are shifted back to V_{y-1} as was done in balancing V_q and we balance V_{y-1} in the same way. Fact 3 below follows immediately from the construction. Facts 4 and 5 need some explanation.

Fact 3. We balance a (not necessarily special) vertex v only if it is closed.

Fact 4. In all edges in which flow is reduced while balancing v, except possibly the last one, flow will not be reduced any more.

Proof. By the construction above all these edges except possibly the last one are deleted from $\Delta(v)$. Because by Fact 3 v is closed, we will never push more flow through v and thus the deleted edges will never appear in $\Delta(v)$, so flow will not be reduced on these edges. □

Fact 5. When we balance a (not necessarily special) vertex v in BALANCE(j), it suffices to consider *newflow*(e) of edges e in $\Delta(v)$.

Proof. Intuitively, when PUSH(i) was called for the last (ℓ_i-th) time all layers V_z with $z>p$ were balanced, so the flow beyond V_p had reached t; thus the current excess of v is completely due to flow increments since the last call to PUSH(i).

The proof is by induction on the time T_1 that we balance v. Obviously, the claim holds vacuously for $T_1=0$. So, we assume it holds for all $T<T_1$, and show that it holds for T_1. We assume that we balance v in BALANCE(i) at time T_1. (Otherwise, there is nothing to prove.)

Let T_0 be the time of the call to the (successful) ℓ_i-th call to PUSH(i). An edge (w,u) in SL_j is said to be *important* at time T, $T_0 \leqq T \leqq T_1$, if *pushnumber*(u) $=\ell_j$. Recall that a vertex is in SL_j if it is in a layer of SL_j other than the rear layer of SL_j. We show that at any time T, $T_0 \leqq T \leqq T_1$ and for every vertex u in SL_i:

$$(1)\quad excess(u) \leqq \sum_{\substack{e \in in(u) \\ e \text{ is important}}} newflow(e) - \sum_{\substack{e \in out(u) \\ e \text{ is important}}} newflow(e).$$

Consequently, if we balance a vertex u in SL_i at time T, $T_0 \leqq T \leqq T_1$, then *pushnumber*(u)$=\ell_i$ (because if *pushnumber*(u)$<\ell_i$, then no edge in $in(u)$ is important and by (1) *excess*(u)$=0$) and all edges in $in(u)$ are important. So by (1) it suffices to decrease newflows of edges in $\Delta(u)$ to balance u. Hence, Fact 5 follows from (1) with $u=v$ and $T=T_1$.

First note that we have equality in (1) ($0=0$) at T_0 because at the time of the ℓ_i-th call to PUSH(i) *excess*(u) was zero. Also, each call to PUSH($i+1$) between T_0 and T_1 (and there can be several such calls) preserves (1) because ℓ_{i+1} is increased by 1 and as a result only the right hand side of (1) for u in V_q can increase. (The second term is set to 0 because the important edges in $out(u)$ stop being important.) If we increase the flow in $in(u)$ [$out(u)$] by δ in a push of flow, then both sides of (1) increase [decrease] by δ. If as a result of balancing we decrease the flow in $in(u)$ [$out(u)$] by δ, at time T then by the induction hypothesis only newflows of important edges are decreased and thus both sides of (1) decrease [increase] by δ. Therefore, all the changes between T_0 and T_1 preserve (1). □

We now prove correctness of the algorithm and then compute its running time. Lemma 3.1 (that will be used later) and its proof justify the closing of edges in the PUSH(i)'s and in the BALANCE(i)'s. Its proof is identical to Karzanov's proof of Lemma 2.1. Recall that a vertex v is blocked if every path from v to t contains a saturated edge.

Lemma 3.1. *If a (not necessarily special) vertex v becomes closed, then it is blocked from this point on.*

Proof. Induction on the distance of v from t.

Basis. t is always open so the basis holds vacuously.

Induction step. Assume v is in V_{z-1} and the claim holds for layers V_y, $y \geqq z$. Consider the moment when v becomes closed. All edges going from v to V_z have been closed. Take one such edge e that leads to a vertex u. It was closed either

because u was closed and by induction hypothesis u would be blocked from that moment on, or because e became saturated. The flow on e can be reduced at some time after it was saturated (possibly before v becomes closed). But it is reduced when we balance u, and by Fact 3, u is closed and will remain blocked from that point on. So in all cases v will be blocked from now on. □

By the proof above, if an edge is closed and is not saturated it must lead to a closed vertex which must be blocked. That is why it is useless to increment flow in such an edge. Lemma 3.2 is the analog of Lemma 2.2 and its proof is similar to Karzanov's proof.

Lemma 3.2. *We balance a special vertex v at most once.*

Proof. Assume $v \in V_q$, the front layer of SL_i, so we balance v in BALANCE(i). When we balance v for the first time v is closed and no more flow will be pushed later through v. We show that flow will not be reduced in any edge $e=(v,u)$ leaving v, so v will never become positive again and we will not balance it again. We balance v when BALANCE(i) is called, so at this moment all layers V_y, $y>q$, are balanced. Assume we balance u at a later time. Since $u \in V_{q+1}$, there must be a call to PUSH($i+1$) between this time and the time when we balance u. By Fact 5, flow will not be reduced in e because flow is not incremented in e in that PUSH($i+1$) or later. □

Lemma 3.2 implies that the BLOOP is executed at most $m-1$ times. This implies that the algorithm terminates because one can easily show that each BLOOP or PLOOP must terminate. Upon termination all layers must be balanced, and thus the final-preflow is a legal flow. Since s is closed after the first PUSH(1) it must be (by Lemma 3.1) blocked from that point on. Hence, upon termination a flow from s to t is found in which s is blocked, which is a correct solution to the subproblem.

By Lemma 3.2 flow is reduced at most once on edges that enter special vertices. However this is not true in general for other edges: The way increments of flow are recorded implies that the algorithm does not remember the order of increments as was the case in Karzanov's algorithm. (Each increment is added to *newflow*(e), but the history of increments is forgotten.) So when we balance a vertex v the flow that is shifted back is not necessarily the most recent flow that was pushed into v. Since BALANCE(i) can be called several times before PUSH(i) is called (consider a successful BALANCE(i) that reroutes flow to open vertices in V_q that later become closed and must be balanced), it is possible that flow will be reduced several times on the same edge. [By Fact 5 it is impossible that flow will be reduced on the same edge in SL_i before and after a call to PUSH(i).] Also, it is possible that a vertex can be microsource more than once and by the discussion above it can be a microsource more times than the number of edges that leave it. Lemma 3.3 that will be proved later estimates $y \equiv$ the total number of occurrences of microsources and $z \equiv$ the total number of reductions of flow on edges. Recall that x is an upper bound on the number of layers in a superlayer.

Lemma 3.3. $y=O(E)$ *and* $z=O(xE)$.

We now compute the time complexity of the algorithm. First we compute the time to push flow forward in calls to PUSH(i) or BALANCE(i) for all i's. The total number of times a construction of path leads to a closing of an edge is bounded by E. By Fact 1 that holds also for microsources, the total number of times a construction of path does not lead to a closing of an edge is bounded by m^2+y. (Recall that the PLOOP can be executed at most m times.) Thus because of Fact 2 the total time for pushing flow forward is $O((m^2+y+E)x)$.

Since Fact 2 holds also in the case of decreasing the flow in an edge, the total time is $T=O((m^2+y+E)x+z)$ and by Lemma 3.3 $T=O((m^2+E)x)$. We now show how to choose the special layers to minimize T: Divide the layers into blocks of $[x/2]$ consecutive layers each. From each block choose a layer with the minimal number of vertices to be a special layer in addition to the layers of s and t. Obviously the distance between consecutive special layers is at most x. By our choice of the special layers, $[x/2]m_j \leqq V_j$ where V_j [m_j] is the number of vertices [special vertices] in the j-th block. So by summing up these inequalities $[x/2]\cdot(m-2)\leqq V$, so $xm\leqq 3V$. Hence $T=O(mV+Ex)$ where $xm\leqq 3V$. This is minimal for $m=O(\sqrt{E})$ and $x=O(V/\sqrt{E})$ and in this case $T=O(V\sqrt{E})$.

By choosing $x=k$, the layer-number of t, we get Dinic's algorithm. By choosing $x=1$ we get an algorithm very similar to Karzanov's. The only differences are: (1) BALANCE(i) includes in addition to shifting flow to V_{i-1}, an attempt to reroute the flow back to V_i and if successful it is followed by PUSH(i) and not by PUSH($i-1$). (2) The increments are recorded differently. Karzanov records all increments and Cherkasky records only newflows. As a result the algorithm obtained from Cherkasky's algorithm by choosing $x=1$ uses only $O(E)$ space, while Karzanov's algorithm uses $O(V^2)$ space.

Proof of Lemma 3.3. Consider a call to BALANCE (i) and conside the subgraph $\bar{G}=(\bar{V},\bar{E})$ of SL_i defined by $\bar{E}=$all edges on which flow is reduced during this call to BALANCE(i) and $\bar{V}=$all vertices that are incident with edges in $\bar{E}$. Note that (1) flow is reduced exactly once on edges of $\bar{E}$ during the call; (2) all microsources of this call belong to $\bar{V}$; (3) a vertex can be a microsource at most once during the call; and (4) if $v\in\bar{V}$, and $v\notin V_q$, then there is at least one edge in $\bar{E}$ that leaves v. We partition the edges of $\bar{E}$ into two sets: the *good edges* and the *bad edges*. An edge e is good if the flow is reduced on e in this call for the first time or for the last time during the whole algorithm and it is bad otherwise. (So an edge e is bad in a call to BALANCE(i) if flow on it is reduced in this call, was reduced in a previous call and will be reduced in a later call). The total number of occurrences of good edges during the whole algorithm is at most $2E$ (each edge can be good at most twice). For every vertex v in $\bar{V}$ that is not in V_q we define a *standard edge* in $\bar{E}$ by choosing an arbitrary edge in $\bar{E}$ that leaves v. We also define a *standard path* that corresponds to v as follows: Using standard edges we go from v in the direction of the special layer V_q until we hit a good edge which is the last edge in the standard path. We must eventually reach a good edge because by Lemma 3.2 all edges in $\bar{E}$ that enter V_q must be good. We also choose a standard path corresponding to a bad edge $e=(u,v)$ to be e followed by the standard path corresponding to v.

Consider the standard paths that correspond to all vertices in $\bar{V}$. If two standard paths have a common edge, then one of them contains the other. The reason is that standard paths cannot split because there is exactly one standard edge for each vertex in $\bar{V}-V_q$; they cannot merge because by Fact 4 it is impossible that two bad edges will enter the same vertex. (Standard paths can meet but not continue together.)

Consider a standard path of a microsource and let v be an internal vertex on the path. Thus, the edge that belongs to the path and enters v is a bad edge; so the flow on it was reduced in a previous BALANCE(i) because we balanced v. Hence v was closed and hence cannot be now a microsource. Therefore the standard paths of all microsources are all edge-disjoint. We correspond to a microsource v the occurrence of the good edge at the end of its standard path. Because of the edge-disjointness of the standard paths of the microsources and the fact that the total number of occurrences of good edges is $2E$ we derive that $y \leqq 2E$.

To get a bound for z we take all standard paths that correspond to bad edges of $\bar{G}$ and correspond to a bad edge the occurrence of the good edge at the end of its standard path. Since these paths cannot split or merge, any good edge can correspond to at most x bad edges and thus there are at most $2xE$ occurrences of bad edges altogether. Hence $z=O(xE)$. □

4. The New Algorithm

The new algorithm is derived from almost the same macro-algorithm of Fig. 2. It implements differently PUSH(i) and BALANCE(i). For each SL_i it maintains an auxiliary graph called FOREST$_i$ that consists of a collection of vertex-disjoint trees with roots in the front layer V_q of SL_i and the leaves are exactly all open vertices in the rear layer V_p of SL_i. The vertices of FOREST$_i$ are vertices of SL_i and are called *junctions*. An edge (u, v) in FOREST$_i$ corresponds to a directed path in SL_i from u to v. An edge in FOREST$_i$ is called a *big edge* and the edges in SL_i are called *small edges*. All vertices that are incident with small edges that belong to the big edges are marked and are called the *marked vertices*. Junctions have a double mark to distinguish them from the other marked vertices. Whenever we use FOREST$_i$ it always satisfies two conditions:

1) all the marked vertices are open, and

2) all its small edges are open and are not saturated. If at a certain point one of these conditions will not hold we will modify FOREST$_i$ so that the new FOREST$_i$ will satisfy these conditions.

The role of FOREST$_i$ is to expedite the push of flow in case no edges are closed. In each call to PUSH(i) we will scan once FOREST$_i$ and use it to push flow. In case no edge becomes saturated the cost will be $O(m_i)$ (m_i is the number of special vertices in the rear layer of SL_i) hence a total of $O(m^2)$. (There can be at most m calls to PUSH(i).) In all other cases FOREST$_i$ will be changed and we will close at least one small edge per work of at most $O(x)$. Hence the total time will be bounded by $O(m^2+Ex)=O(m^2+EV/m)$. The optimal m will be $(EV)^{1/3}$ and thus the time bound for a phase in Dinic's algorithm will be $O((EV)^{2/3})$ and

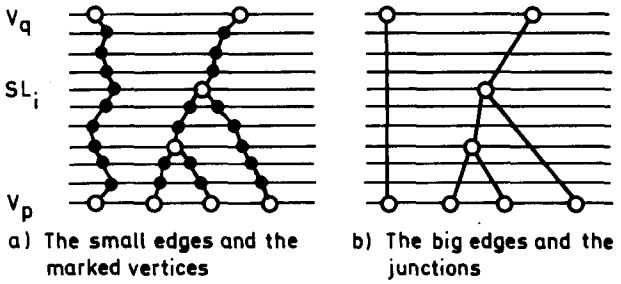

Fig. 3. FOREST(i) (all edges big and small are directed upwards). Small edges in SL_i other than those that correspond to big edges, and the unmarked vertices are not drawn

for the whole algorithm $O(V^{5/3}E^{2/3})$. The choice of the optimal m is achieved by choosing the optimal x to be $(V^2/E)^{1/3}$ and as in the previous algorithm we take blocks of $[x/2]$ layers choosing from each a layer with the minimal number of vertices to be a special layer.

We now describe in detail FOREST_i and the operations we perform on it. Next we show how to execute PUSH(i) and then how to execute BALANCE(i). Finally we prove correctness and show that the time bound above holds.

A big edge $\hat{e}$ has a list LIST($\hat{e}$) of the small edges it represents, and each small edge has a pointer to its big edge. A big edge $\hat{e}$ has two kinds of flows associated with it: *oldflow*($\hat{e}$) and *newflow*($\hat{e}$). The latter is flow added to it since the last call to PUSH(i) and the former is flow added to it before. Flows associated with a big edge (old and new) represent flows in the corresponding small edges that have not yet been added to them. These small edges may have already flow in them. Using the pointer to the big edge the total amount of flow in a small edge e *totalflow*(e) can be easily computed: If e belongs to $\hat{e}$ *totalflow*(e) is the sum of the current flow in e plus *oldflow*($\hat{e}$) plus *newflow*($\hat{e}$). A big edge $\hat{e}$ has a (residual) capacity $c(\hat{e}) = \min_{e \in \text{LIST}(\hat{e})} [c(e) - \textit{totalflow}(e)]$. Each junction $\hat{v}$ will have a doubly linked list LIST($\hat{v}$) of the big edges that enter $\hat{v}$. Junctions may become temporarily positive during PUSH(i), but those that are not special vertices will become balanced at the end of the call. We maintain for every junction $\hat{v}$ *excess*($\hat{v}$), and also *degree*($\hat{v}$), the number of big edges entering $\hat{v}$. As in the previous algorithm, each small edge e will have the variable *newflow*(e), and each vertex v will have the variable *pushnumber*(v) and the list $\Delta(v)$ associated with it. If $e \in \text{LIST}(\hat{e})$, then *newflow*($e$) *does not* include the increments included in *newflow*($\hat{e}$). Whenever we increase or decrease the flow in a small edge e we do it in the same way as in Cherkasky's algorithm.

The macro-algorithm for the new algorithm is given in Fig. 4. We construct FOREST_i in the obvious way by executing a depth first search on the open edges of SL_i starting from each open vertex in V_p in turn and marking the vertices on the way until either 1) we reach a marked vertex or reach an open vertex in V_q; or 2) we reach a closed vertex. In the first case we generate a new big edge and possibly a new junction if this vertex is not already one and continue with the next open vertex in V_p. In the second case we close the last

small edge on the path, unmark the last vertex, back up one vertex and continue from there. One can easily verify that the construction of FOREST_i with all its associated lists takes at most time bounded by $O(m_i x + r_i)$, where r_i is the number of small edges that we close during the construction. We demolish FOREST_i by traversing all its small edges and updating their flow according to the flows associated with their big edges. Demolishing FOREST_i takes at most $O(m_i x)$ steps. Hence except for a term that equals $O(r)$, where $r = \sum r_i$ is the total number of small edges that we close in the construction, the total time of constructing and demolishing all FOREST_i's is bounded by $O(mx) \leqq O(V)$, which we can ignore in computing the total time bound. The additional term mentioned above will be included in the final time bound in the term $O(Ex)$. The latter term will be derived from the fact that we close at most E small edges and the cost per edge is $O(x)$. (In this case the cost per edge is just $O(1)$.)

For $i = 1, \ldots, k'$ construct FOREST_i

$i \leftarrow 1$

PLOOP: PUSH(i)
If the push was successful and $i < k'$ then [$i \leftarrow i+1$; go to PLOOP]
If there are no positive vertices go to LSTOP
Otherwise let i be the maximal number of a superlayer whose front layer contains positive vertices

BLOOP: BALANCE(i)
If the balance was successful then [$i \leftarrow i+1$; go to PLOOP]
Otherwise: if $i = 1$ then go to LSTOP else [$i \leftarrow i-1$; go to BLOOP]

LSTOP: For $i = 1, \ldots, k'$ demolish FOREST_i STOP

Fig. 4. The macro-algorithm for the new algorithm

During the execution of the algorithm FOREST_i will keep changing. We now describe two primitive operations that will be used: *delete*($\hat{e}$) and *reconnect*($\hat{u}$). *Delete*($\hat{e}$) deletes the big edge $\hat{e}$ from FOREST_i. We delete a big edge either because it is saturated and this because at least one small edge on it has become saturated, or because it leads to a closed vertex. We delete $\hat{e}$ by going along the path that corresponds to $\hat{e}$ (by using LIST($\hat{e}$)) unmarking the internal vertices and *updating the flow* by adjusting the flow on the small edges (adding to it *oldflow*($\hat{e}$)+*newflow*($\hat{e}$)) and adjusting *newflow*(e) of e's in LIST($\hat{e}$) by adding the increment *newflow*($\hat{e}$) as in the previous algorithm. We close all the small edges that become saturated and in case $\hat{v}$ is closed we close the last small edge (that enters $\hat{v}$). So we close at least one small edge in any case. Then we update *degree*($\hat{v}$) by decreasing it by one. If it becomes 1 and $\hat{v}$ is not in V_q, then $\hat{v}$ stops being a junction.

When this is the case, we will replace the big edges $\hat{e}_1$ and $\hat{e}_2$ by the big edge $\tilde{e}$. This is done by visiting LIST($\hat{e}_1$) and LIST($\hat{e}_2$) updating the flow in the small edges as was done above, then concatenating LIST($\hat{e}_1$) and LIST($\hat{e}_2$) to create LIST($\tilde{e}$) and finally setting $c(\tilde{e}) = \min(c(\hat{e}_1), c(\hat{e}_2))$, $newflow(\tilde{e}) = oldflow(\tilde{e}) = 0$,

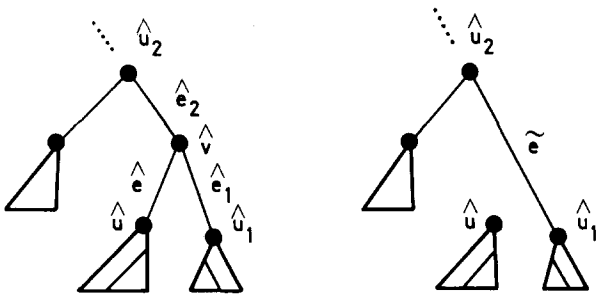

Fig. 5. From left to right - deleting $\hat{e}$, $\hat{v}$ stops being a junction. From right to left - reconnecting $\hat{u}$, $\hat{v}$ is a new junction

and replacing $\hat{e}_2$ by $\tilde{e}$ in LIST($\hat{u}_2$). Note that the cost of a deletion of a big edge is proportional to x.

After deleting a big edge $\hat{e}=(\hat{u},\hat{v})$ *reconnect*($\hat{u}$) tries to reconnect $\hat{u}$ to the forest. The reconnection is done similarly to the construction of the original forest. We simply use the small edges to search for open paths closing the last small edge if we hit a dead end (a closed vertex). The *reconnect* operation is successful if we finally hit a marked vertex or an open vertex in V_q, and we have again a forest. In this case we generate a new big edge and we may generate a new junction: Consider Fig. 5 from right to left. The big edge $\tilde{e}$ is broken into $\hat{e}_1$ and $\hat{e}_2$, the capacity of each is computed by going along the lists but flows *are not* transferred to small edges. The *reconnect* may fail if $\hat{u}$ becomes closed. In this case $\hat{u}$ stops being a junction. The time of reconnect is $O(\ell+x)=O((\ell+1)x)$, where ℓ is the number of small edges that are closed during its execution.

We are now ready to describe PUSH(i). PUSH(i) consists of two stages. In the first stage we traverse FOREST$_i$ via the big edges from the leaves up pushing flow up as much as possible. For each junction $\hat{v}$ we keep a counter that counts down from *degree*($\hat{v}$) to 0. Each time we scan a big edge that enters $\hat{v}$ the corresponding counter is decreased by one. The junction $\hat{v}$ is visited only when its counter is 0 and then the big edge that leaves $\hat{v}$ is scanned. So the traversal of FOREST$_i$ takes $O(m_i)$ time.

The big edges that are going to be deleted will be kept in a stack called STACK, so that the one on top of STACK will be deleted first. During the traversal of FOREST$_i$ all big edges that become saturated are put in STACK. The second stage of PUSH(i) is a loop in which we alternately, delete a big edge from FOREST$_i$ (and from STACK) and reconnect the forest.

PUSH(i) is essentially applying Karzanov's algorithm (actually Cherkasky's version of Karzanov's algorithm that uses less space) to FOREST$_i$ that keeps changing during the execution of the loop. In addition to the push of flow in the first stage, flow will be pushed up as much as possible after each successful *reconnect*($\hat{u}$) with *excess*($\hat{u}$)>0. In case *reconnect*($\hat{u}$) fails and $\hat{u}\notin V_p$, we will balance $\hat{u}$ and then insert all the big edges in LIST($\hat{u}$) into STACK. In order to balance junctions we need a *third* kind of flow on the big edges, that we denote by *verynewflow*($\hat{e}$). This quantity will represent part of *newflow*($\hat{e}$) that we do not know whether it has been pushed to V_q or not. We update *verynewflow*($\hat{e}$) as we

update *newflow*($\hat{e}$) except in one case: Whenever we delete a big edge $\hat{e}=(\hat{u},\hat{v})$ and $\hat{v}$ is open we set *verynewflow*($\tilde{e}$) to zero for every big edge $\tilde{e}$ above $\hat{e}$ in FOREST$_i$. This update of the verynewflows is charged to the corresponding *delete*($\hat{e}$) which still costs $O(x)$. When we balance a junction $\hat{u}$ we use *verynewflow*($\hat{e}$) of the big edges $\hat{e}$ in LIST($\hat{u}$) to balance $\hat{u}$. We will prove in Fact 6 that it will always be possible to balance a junction in this way. Note that in PUSH(i) only newflow is pushed; so newflow [verynewflow] corresponds to flow [newflow] in Cherkasky's version of Karzanov's algorithm.

STACK will be maintained as a collection of doubly linked lists *list*$_j$ for $p \leqq j < q$. *List*$_j$ consists of the big edges in STACK that leave from vertices in V_j. Obviously it takes a constant amount of time to insert a big edge into STACK or to delete a big edge from STACK. By maintaining a counter that counts the number of the big edges in STACK we can check if STACK is empty in constant time. If $j_1 > j_2$ *list*$_{j_1}$ will be on top of *list*$_{j_2}$ in STACK. The only costly operation will be to find $\hat{e}$ – the topmost big edge in STACK in case STACK is not empty. This edge is the first big edge in the topmost nonempty *list*$_j$ in STACK. So, to find $\hat{e}$ may cost $O(x)$ operations. This cost is added to the cost of *delete*($\hat{e}$) which remains $O(x)$.

If one of the two cases that is described in Fig. 5 occurs, then *delete* [*reconnect*] will update STACK as follows. In case of *delete*($\hat{e}$), $\hat{e}=(\hat{u},\hat{v})$ and $\hat{v}$ stops being a junction, then if $\hat{e}_1$ is in STACK ($\hat{e}_2$ cannot be in STACK because it would have been deleted before $\hat{e}$), then it is deleted from STACK and $\tilde{e}$ is inserted into STACK. In case of *reconnect*($\hat{u}$), if the *reconnect* is successful and the new junction breaks a big edge $\tilde{e}$ that is already in STACK, then *reconnect* will delete $\tilde{e}$ from STACK and will add at least one of the two new edges $\hat{e}_1$ and $\hat{e}_2$ to STACK. We insert $\hat{e}_1$ [or $\hat{e}_2$] into STACK either if it is saturated or if it leads to a closed vertex.

After a successful *reconnect*($\hat{u}$) if *excess*($\hat{u}$) >0, then flow is pushed up through big edges as much and as up as possible until we either hit V_q or a big edge that is in STACK. The big edges that become saturated are inserted into STACK. Due to the way we maintain STACK the only case that we hit a big edge in STACK is when the reconnect breaks a big edge as in Fig. 5 and in addition $\hat{e}_2$ is inserted into STACK (because $\tilde{e}$ was in STACK). In this case we push flow only in the new big edge $\hat{e}$. Therefore, including the time to push flow after a successful *reconnect*, the time bound of *reconnect*($\hat{u}$) is still $O((\ell+1)x)$ as above.

If *reconnect*($\hat{u}$) fails, and $\hat{u} \notin V_p$, we balance $\hat{u}$ by decreasing *verynewflow*($\hat{e}$) and *newflow*($\hat{e}$) for big edges $\hat{e}$ in LIST($\hat{u}$). Then all big edges in LIST($\hat{u}$) are inserted into STACK.

PUSH(i)
Scan FOREST$_i$ from leaves up
Push new flow up and execute (1)–(4)
(1) Set *oldflow*($\hat{e}$)←*oldflow*($\hat{e}$)+*newflow*($\hat{e}$)
(2) *Set newflow*($\hat{e}$) and *verynewflow*($\hat{e}$) to the amount of flow being pushed
(3) Update *excess*($\hat{v}$) and $c(\hat{e})$ on the way
(4) If $\hat{e}$ becomes saturated ($c(\hat{e})=0$) put it in STACK
LOOP: If STACK is empty terminate

Otherwise let $\hat{e}=(\hat{u}, \hat{v})$ be on top of STACK
Pop $\hat{e}$ from STACK
Delete$(\hat{e})$
If $\hat{v}$ is open set *verynewflow*$(\tilde{e})\leftarrow 0$ for all big edges above $\hat{e}$ in FOREST$_i$
Reconnect$(\hat{u})$
If *reconnect* is successful and *excess*$(\hat{u})>0$ then do:
    Push flow up through big edges until you either hit V_q or a big edge that is in STACK
    Update as in (2)′ (see below), (3) and (4).
    (2′) Add to *newflow*$(\hat{e})$ and *verynewflow*$(\hat{e})$ the amount of flow being pushed
if *reconnect* fails and $\hat{u}\notin V_p$ do:
    Consider in turn each big edge $\hat{e}$ in LIST$(\hat{u})$
    Use *verynewflow*$(\hat{e})$ to decrease *newflow*$(\hat{e})$ and *verynewflow*$(\hat{e})$
    Repeat until $\hat{u}$ becomes balanced
    Put every big edge from LIST$(\hat{u})$ in STACK if it is not there already
Go to LOOP

Fact 6. Whenever we balance a junction $\hat{u}$ it suffices to consider verynewflows of big edges in LIST$(\hat{u})$ at the time of the balancing.

Intuitively, the other flows had been pushed to V_q and did not cause the current excess at $\hat{u}$. Fact 6 follows immediately from the following claim:

Claim. At any moment in the execution of the loop and for every junction that is not a special vertex

$$(1)\quad excess(\hat{u}) \leqq \sum_{e\in \mathrm{LIST}(\hat{u})} verynewflow(\hat{e}) - verynewflow(\hat{e}_{\hat{u}})$$

where $\hat{e}_{\hat{u}}$ is the big edge that leaves $\hat{u}$.

Proof. Just before the execution of the loop and for a newly generated junction (in a *reconnect*) we have equality in (1) $(0=0)$. If we push δ units of flow in $(\hat{w}, \hat{v})$ then both sides of (1) for $\hat{u}=\hat{v}$ $[\hat{u}=\hat{w}]$ are increased [decreased] by δ. Whenever we balance a junction $\hat{u}$, (1) implies that decreasing the verynewflows in LIST$(\hat{u})$ suffice to balance $\hat{u}$. If we decrease the flow in $(\hat{w}, \hat{v})$ by δ units of flow, then both sides of (1) for $\hat{u}=\hat{v}$ $[\hat{u}=\hat{w}]$ are decreased [increased] by δ. There are three cases in which we set *verynewflow*$(\tilde{e})$ to zero, where $\tilde{e}=(\hat{v}, \hat{w})$: (a) A big edge $\hat{e}$ below $\tilde{e}$ in FOREST$_i$ was deleted; (b) $\tilde{e}$ is a new big edge that was generated as a result of deleting a big edge $\hat{e}$ (as in Fig. 5); and (c) $\tilde{e}$ is a new big edge as a result of a successful *reconnect*$(\hat{v})$. Obviously, (1) still holds for $\hat{u}=\hat{v}$ in all these cases because only its right hand side can increase. If $\hat{w}\notin V_q$, then (1) still holds for $\hat{u}=\hat{w}$: In case (c) because nothing changes in (1). In cases (a) and (b), note that $excess(\hat{w})=0$ because otherwise $\hat{e}_{\hat{w}}$ would have been on top of $\hat{e}$ in STACK. (We push more flow up unless the big edge is in STACK.) Also, *verynewflow*$(\hat{e}_{\hat{w}})$ is set to zero; so after the change the left hand side of (1) is zero and the right hand side is nonnegative and (1) holds for $\hat{u}=\hat{w}$. Finally, if we delete a big edge $\hat{e}=(\hat{v}, \hat{w})$, then if $\hat{w}\notin V_q$ is still a junction (1) holds for $\hat{u}=\hat{w}$ as in cases (b) and (c) above; and that it holds for $\hat{u}=\hat{v}$ after a successful *reconnect*$(\hat{v})$ was shown in (c) above. So all the changes preserve (1). □

It is possible to show that maintaining the order in STACK (although all big edges in it are eventually deleted), and using a third kind of flow (*verynewflow*($\hat{e}$)) are both necessary so that we would be able to balance a junction by decreasing flow on big edges only.

Note that a nonspecial vertex can stop being a junction in two occasions: (1) a failure of reconnect, and (2) as in Fig. 5. In the first case we balance the vertex, and in the second case the vertex must be balanced (otherwise $\hat{e}$ would not have been on top of STACK). So PUSH(i) starts with positive vertices in V_p and ends with possibly positive (that must be open) vertices in V_q, no positive vertices in between V_p and V_q and possibly positive (that must be closed) vertices in V_p.

We now show that the total time of PUSH(i) is bounded by $O(m_i + xr)$ where r is the number of small edges that we closed during the execution of PUSH(i): The scanning takes time $O(m_i)$. Taking into account the following push of flow up the tree in FOREST$_i$ the cost of each *reconnect* is still $O((\ell+1)x) = \ell O(x) + O(x)$, where ℓ is the number of small edges that we closed during the *reconnect*. The $O(x)$ is attributed to the big edge whose deletion caused the reconnect and each of the $\ell O(x)$'s is attributed to the corresponding closing of a small edge. The cost of deleting a big edge is still $O(x)$ after adding the $O(x)$ of the *reconnect* (and the other charges added to it before), and it is attributed to the small edge that became closed as a result. The cost of balancing a junction $\hat{u}$ is bounded by the length of LIST($\hat{u}$). Hence the cost of decreasing flow in the big edges is bounded by the cost of increasing it in them. So this cost can be ignored in computing the order of the time bound.

We now describe BALANCE(i): Some of the junctions in V_q might become closed during a previous PUSH or BALANCE in SL_{i+1}. If this is the case all big edges that enter a closed junction in V_q are put in STACK. Then a sequence of deletions and reconnections as in PUSH(i) reconstructs FOREST$_i$. We *do not* balance the positive junctions in V_q, so no flow is pushed during the reconstruction of FOREST$_i$. After all roots of FOREST$_i$ are open junctions, a process almost identical to BALANCE(i) of the previous algorithm is executed. There is only a small difference in handling microsources: Assume we have a microsource u and we push flow. If we do not hit a marked vertex, then it is the same as the previous algorithm. Otherwise we reach a marked vertex v (possibly $v=u$) and we use the small edges in FOREST$_i$ to reach V_q. The capacities of the big edges on the way are updated and flows on these big edges are transferred to the small edges. If as a result some big edges are saturated, using STACK after some deletions and reconnections as in PUSH(i) (but without pushing flow) the balancing is resumed. Note that when we reduce the flow on a small edge it cannot belong at that time to any big edge because all marked vertices must be open.

The total time bound for all BALANCE(i)'s is $O(z + yx + \ell_b x)$, where z is the number of times flow of a small edge is decreased, y is the number of occurrences of microsources and ℓ_b is the total number of small edges that we closed during all the BALANCE(i)'s.

Our new algorithm is *in principle* the same as Cherkasky's algorithm. The only difference is that FOREST$_i$ dictates sometimes the order of pushing the flow and expedites sometimes PUSH(i) because occasionally we push flow

through a big edge (i.e. a path of small edges) at once. Lemma 3.1, 3.2 and 3.3 hold for the new algorithm because they do not depend on the order flow is pushed from V_p to V_q by PUSH(i). So correctness is derived as before, and by Lemma 3.2 PUSH(i) can be executed at most m times and thus the total time bound for all PUSH(i)'s is $O(m^2+\ell_p x)$, where ℓ_p is the total number of small edges that we closed during all the PUSH(i)'s. Therefore, the total time bound for the new algorithm is $T=O(z+yx+Ex+m^2)=O(m^2+Ex)$ (by Lemma 3.3). The rest, how to choose special vertices to get the optimal m and x, has been explained already.

As for space complexity, Cherkasky's algorithm needs $O(E)$ space because it uses a fixed number of variables per vertex and per edge. The new algorithm uses in addition a fixed number of variables per junction and per big edge. So, by reusing the space the increase is $O(m_i)$ per SL_i, or a total of $O(m)=O(V)$. Therefore, our algorithm uses linear space too. Note that Karzanov's algorithm may need space proportional to V^2 because there can be about V stacks HISTORY(u) of total length up to $\theta(V^2)$.

5. Conclusion

We have presented a new algorithm for the max-flow problem. This algorithm is the fifth improvement on the original algorithm by Ford and Fulkerson. It is not clear when this theoretical improvement is useful in practice. The overhead due to the FOREST$_i$'s and due to Cherkasky's machinery is not negligible.

Prior to the discovery of the more recent algorithm [10], the algorithm described here was always asymptotically best and asymptotically better than the other algorithms except for very dense graphs. (For $E=\theta(V^2)$ it ties with Karzanov and Cherkasky's algorithm.) The largest fraction of improvement was $V^{1/6}$ for $E=\theta(V)$. The discovery of the $O(EV\log^2 V)$ algorithm has changed the picture completely. The algorithm described here is now best only for $V^2/\log^6 V \leqq E \leqq V^2$, and the largest improvement is when $E=\theta(V^2/\log^4 V)$. In this case the factor of improvement on Cherkasky's algorithm and on the more recent algorithm is $(\log V)^{2/3}$.

The $O(E^{2/3}V^{5/3})$ algorithm can be viewed as a fast implementation of Cherkasky's algorithm making use of a special data structure - FOREST$_i$. FOREST$_i$'s were graphs, the edges of which were paths in the original graph. They were useful in making some shortcuts and expedited the push of flow. We hope that similar constructs will be found useful for improving other algorithms too.

One of the tricks for making algorithms more efficient is finding a way not to lose information and by doing so saving additional work to recompute this information if needed. Dinic's layered network helps to keep the information needed for finding the shortest f.a.p.'s, while Edmonds and Karp's algorithm starts from scratch each time it looks for a shortest f.a.p.

Similarly, FOREST$_i$ is the information passed from one call to PUSH(i) to the next call, while in Cherkasky's algorithm no use is made of any information obtained in a call to PUSH(i) in later calls.

Acknowledgments. I am indebted to M. Ben Ari and A. Itai for reading the manuscript and for many suggestions. Thanks also to N. Megiddo for providing me with a Hebrew translation of Cherkasky's paper.

The work of the author was supported in part by the Israel Commission for Basic Research and by NSF grant no. MCS 78-25301.

References

1. Adelson-Velsky GM, Dinic EA, Karzanov AV (1975) Flow algorithms. (in Russian). Nauka, Moscow
2. Aho AV, Hopcroft JE, Ullman JD (1974) The design and analysis of computer algorithms. Reading, Mass Addison-Wesley
3. Cherkasky BV (1977) Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations (in Russian). Mathematical Methods of Solution of Economical Problems 7: 117-125
4. Dinic EA (1970) Algorithm for solution of a problem of maximal flow in a network with power estimation. Soviet Math Dokl 11: 1277-1280
5. Edmonds J, Karp RM (1972) Theoretical improvement in algorithmic efficiency for network flow problems. JCAM 19: 248-264
6. Even S (1976) The max-flow algorithm of Dinic and Karzanov: An exposition, MIT, LCS, TM-80
7. Ford LR, Fulkerson DR (1962) Flows in networks. Princeton University Press, New Jersey
8. Ford LR, Fulkerson DR (1956) Maximal flows through a network. IRE Trans on Inform Theory, IT-2:117-119
9. Galil Z (1980) On the theoretical efficiency of various network flow algorithms. IBM Report, RC7320, September 1978, Theor Comput Sci (in press)
10. Galil Z, Naamad A (1980) Network flow and generalized path compression. Proceedings 11th Annual ACM Symposium on Theory of Computing, May 1979, 13-26. To appear in J Comput System Sci as: An $O(EV\log^2 V)$ algorithm for the maximal flow problem
11. Karzanov AV (1974) Determining the maximal flow in a network by the method of preflows. Soviet Math Dokl 15:434-437
12. Knuth DE (1968) The art of computer programming. Vol 1 (Fundamental algorithms). Addison-Wesley Reading, Mass
13. Malhotra VM, Pramodh Kumar M, Maheshwary SN (1978) An $O(V^3)$ algorithm for finding the maximal flow in networks. Information Processing Lett 7:277-278

Received September 25, 1979; revised March 18, 1980

Appendix

Glossary

balance a vertex v	– reduce the flow in $in(v)$ until v is balanced.
balanced vertex	– a vertex v with $excess(v)=0$.
big edge	– an edge in FOREST_i which represents an open path in SL_i.
$c(e)$	– the capacity of a (small) edge in the layered network.
$c(\hat{e})$	– the capacity of a big edge $\hat{e}$, which equals the minimal residual capacity of a small edge on the corresponding path.
closed edge	– an edge that has been closed by the algorithm either because it became saturated or because it led to a closed vertex.
closed vertex	– a vertex v, $v \neq t$, such that all edges in $out(v)$ are closed.
$degree(\hat{v})$	– the number of big edges entering a junction $\hat{v}$.

$\Delta(v)$ - a doubly linked list containing all edges e in $in(v)$ with $newflow(e)>0$.
$excess(v)$ - $\sum_{e\in in(v)} f(e) - \sum_{e\in out(v)} f(e)$
$excess(\hat{v})$ - the sum of $newflow(\hat{e})$ for $\hat{e}$ in LIST($\hat{v}$) minus $newflow$ of the big edge that leaves $\hat{v}$. (The corresponding term for $oldflow$'s is zero.)
$f(e)$ - the flow in the edge e.
front layer of SL_i - the layer in SL_i closest to t.
$in(v)$ - the list of edges that enter v in the layered network.
junction - a vertex in FOREST$_i$.
ℓ_i - the number of the calls to PUSH(i).
LIST($\hat{e}$) - the list of the small edges on the path corresponding to a big edge $\hat{e}$.
LIST($\hat{u}$) - the list of the big edges entering a junction $\hat{u}$.
list$_j$ - the list of big edges in FOREST$_i$ which start at V_j.
m - the number of special vertices.
m_i - the number of vertices in the rear layer of SL_i.
marked vertex - a vertex on a path that corresponds to a big edge.
micropush - a push of flow from a microsource.
microsource - an open vertex that becomes positive during BALANCE(i).
$newflow(e)$ - the total flow increment in a (small) edge $e=(u, v)$ in SL_i since the last call to PUSH(i), that pushed flow into v.
$newflow(\hat{e})$ - the total flow increment in a big edge $\hat{e}$ in FOREST$_i$ since the last call to PUSH(i).
open edge - an edge that has not been closed by the algorithm; an open edge cannot be saturated and cannot lead to a closed vertex.
open vertex v - a vertex v such that either $v=t$ or there are open edges in $out(v)$.
open path - a path consisting of open edges and open vertices.
$out(v)$ - the list of edges that leave v in the layered network.
positive vertex - a vertex v, $v\neq t$, with $excess(v)>0$.
$pushnumber(v)$ - the number of the last PUSH(i) that pushed flow into v.
rear layer of SL_i - the layer in SL_i closest to s. Its vertices belong to SL_{i-1}.
saturated edge - an edge e with $f(e)=c(e)$.
SL_i - the i-th superlayer.
small edge - an edge in the layered network
special vertex - a vertex in a special layer.
special layer - a layer chosen by the algorithm as special.
superlayer - the part of the layered network between two consecutive special layers.
$totalflow(e)$ - $f(e)+oldflow(\hat{e})+newflow(\hat{e})$, if $e\in$LIST($\hat{e}$) and $f(e)$ otherwise.
$verynewflow(e)$ - part of $newflow(\hat{e})$ used for balancing junctions; modified like $newflow(\hat{e})$ except for the case when it is set to zero when a big edge below it is deleted.
V_p - the rear layer of SL_i.
V_q - the front layer of SL_i.
x - the bound on the number of layers in a superlayer.