JOURNAL OF COMPUTER AND SYSTEM SCIENCES 21, 203–217 (1980)

An O(EVlog²V) Algorithm for the Maximal Flow Problem

ZVI GALIL AND AMNON NAAMAD*

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

Received October 20, 1979; revised May 9, 1980

An $O(EV \log^2 V)$ algorithm for finding the maximal flow in networks is described. It is asymptotically better than the other known algorithms if $E = O(V^{2-\epsilon})$ for some $\epsilon > 0$. The analysis of the running time exploits the discovery of a phenomenon similar to (but more general than) path compression, although the "union find" algorithm is not used. The time bound is shown to be tight in terms of V and E by exhibiting a family of networks that require $\Omega(EV \log^2 V)$ time.[1]

INTRODUCTION

One of the well-known problems in combinatorial optimization is the problem of finding a maximum flow in a given network. Table 1 summarizes the history of the problem. For more details see [9]. All the bounds in Table 1 have been shown to be tight in [11], where a family of networks was constructed so that for every V and E, $V \leqslant E \leqslant V^2$, a network with approximately V vertices and E edges belongs to the family and requires the time (or space) which appears in the column of Table 1 by the corresponding algorithm. This family of "bad" networks is derived from one parametrized network.

TABLE 1

Solutions	Time	Space
Ford and Fulkerson (1956) [6, 7]	—	E
Edmonds and Karp (1969) [5]	E^2V	E
Dinic (1970) [4]	EV^2	E
Karzanov (1973) [13]	V^3	V^2
Malhotra *et al.* (1978) [14]	V^3	E
Cherkasky (1976) [3]	$V^2(E)^{1/2}$	E
Galil (1978) [9]	$V^{5/3}E^{2/3}$	E
The new algorithm	$EV \log^2 V$	E

* The first author was supported in part by the Israel Commission for Basic Research.

[1] We use V [E] for the set of vertices [edges] and for its size. Unless specified otherwise, logarithms are of base 2.

In this paper we present another algorithm for the max-flow problem. As is seen in Table 1 many of the algorithms take time $O(V^3)$ when $E = V^2$, while the new algorithm takes time $O(V^3 \log^2 V)$. But the latter is asymptotically superior whenever $E = O(V^{2-\epsilon})$ for any $\epsilon > 0$. In particular it is best for sparse networks ($E = O(V)$). Unlike some of the recent algorithms (e.g., Galil's or Cherasky's) the algorithm is quite silple. In order to understand it one need only know Dinic's algorithm. In fact it can be viewed as an efficient implementation of Dinic's algorithm.

Since the algorithm is based on Dinic's algorithm we will first explain Dinic's algorithm. Then we will present our algorithm. Correctness proof will be trivial, but the analysis of the running time will turn out to be quite involved. The most interesting part of this paper is the analysis of the running time of the algorithm. We have discovered a phenomenon similar to the path compression of the "union find" algorithm. This is surprising because no use is made of this algorithm even in an indirect way. Our initial attempts to come up with a reduction to the "union find" algorithm have failed. The reason for that was that the phenomenon mentioned above is more general.

We state a Combinatorial Lemma which implies as corollaries the time bound for our algorithm, and the known time bound for the "union find" algorithm without balancing but with path compression.

We then show that the time bound $O(EV \log^2 V)$ is tight. The "bad" netoworks of [11] are not bad enough for the new algorithm and require only $\Omega(EV \log V)$. We construct another family of networks that force the algorithm to run in time proportional to $EV \log^2 V$ for V and $V \leqslant E \leqslant V^2$. This family is derived from one parametrized network. (So for every such V and E the parameters can be fixed so that the numbers of vertices and edges are V and E, resp.) In the last section we consider the special case of planar networks.

Originally we came up with a slightly different algorithm [10]. Both versions may be considered as two implementations of the same basic idea. The algorithm described here is conceptually simpler. The original version is probably practically (but not asymptotically) better and we describe it in Appendix 1. A direct proof for the running time of a version very similar to our earlier version was discovered by Shiloach [16].

The Problem and the Subproblem

A *network* is a directed graph $G = (V, E)$ with two distinguished vertices s and t, and a positive real number $c(e)$; the *capacity* of e is associated with every edge e. Given a network, a flow is a real valued function $f\colon E \to R$ satisfying

(1) $0 \leqslant f(e) \leqslant c(e)$ for every e in E; and

(2) $\sum_{e \in \mathrm{in}(v)} f(e) = \sum_{e \in \mathrm{out}(v)} f(e)$ for every v, $v \neq s, t$, where

in(v) [out(v)] is the set of edges that enter [leave] v.

Given a flow f, the *value* of the flow is $\sum_{e \in \mathrm{out}(s)} f(e) - \sum_{e \in \mathrm{in}(s)} f(e)$. The max-flow problem is: Given a network, find a flow with maximal value.

All the algorithms mentioned in Table 1 with the exception of the first two use Dinic's approach by solving at most V instances of a sub-problem defined below. In fact this sub-problem can be found "hiding" also in the second solution.

A *layered network* is a network, the vertices of which are partitioned into layers $V_0 = \{s\}$, $V_1, \ldots, V_k = \{t\}$, with edges going only from one layer to the next layer. Given a flow f, an edge e is *saturated* if $f(e) = c(e)$; the flow is *blocking* if every directed path from s to t contains a saturated edge. The sub-problem (which is a phase in Dinic's algorithm) is: Given a layered network, find a blocking flow.

The new algorithm solves the sub-problem in time $O(E \log^2 V)$, and thus it solves the problem in time $O(EV \log^2 V)$.

Dinic's Algorithm and the New Algorithm

Dinic's algorithm performs a depth first search on the layered network. Each time we hit t a path from s to t is found and the maximum amount of flow is pushed through this path. All saturated edges (and there is at least one) are deleted and the search resumes from the tail of the deleted saturated edge closest to s. If we hit a dead-end, we delete the last edge, backtrack to its tail and continue on from it. We terminate when s becomes a dead-end (out(s) becomes empty).

The time bound is obviously $O(E + nk)$ where n is the number of paths found (recall that k is the length of the layered network) which is bounded by $O(VE)$ ($k \leqslant V$, $n \leqslant E$). In Fig. 1, assume that a path has been found and that the two marked edges are deleted. The search resumes at the vertex u. Dinic's algorithm "forgets" most of the information obtained in the previous searches. The fact that the part of the path from u to t has been traversed is not recorded and possibly big parts of this path will later be "rediscovered."

The new algorithm remembers the paths that are generated after deleting saturated edges (α_1 and α_2 in fig. 1). We call them *path fragments* or *PF*'s in short.

The main difficulty with this notion is the possibility that during the construction of a path we may break into a *PF* in the middle (Fig. 2). Then we will have to split that *PF* and concatenate its head to the path which we construct (which itself is a *PF*).

We now describe the algorithm ignoring some details of implementation that are included in the next section. The path constructed every minute is a *PF* which is denoted by PF_0. The vertex u points always to the last vertex of PF_0. The asterisk * will stand for the obvious adjustment of u in each case. At any moment there will be at most one *PF* going through or starting at any vertex v. (There may be several *PF*'s ending at v.) The edge in that *PF* which belongs to out(v) will be the first edge in out(v) that has not been

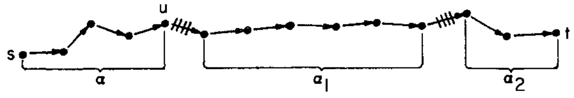

FIGURE 1

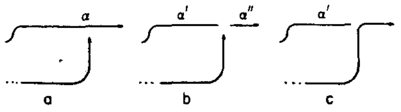

FIG. 2. (a) breaking into a PF α; (b) splitting α; (c) concatenating α'', the head of α.

deleted. We use an auxiliary PF, denoted by PF_1, which is always concatenated to PF_0.

```
1. $u \leftarrow s$; $PF_0 \leftarrow$ empty
2. repeat 3–8 until $s$ is a dead-end
3.     if $u$ is a dead-end delete last edge of $PF$ $*$
       otherwise
4.         if there is no $PF$ going through $u$ let $PF_1$ be the first edge in out($u$)
5.         else if there is a $PF$ that starts at $u$ let $PF_1$ be that $PF$
6.         otherwise—$u$ is in the middle of a $PF$ (Fig. 2)—split it and let $PF_1$ be
           its head
7.         concatenate $PF$ to $PF$ $*$
8.         if $u = t$ (Fig. 1) push flow, repeatedly split $PF_0$ at saturated edges
           and delete them, $PF_0 \leftarrow PF$ that starts at $s$ $*$
```

THE DATA STRUCTURE

Each path fragment is implemented by a 2–3 tree [1]. The ordered leaves of the (tree corresponding to a) PF correspond to the edges of the PF. So, a node[2] n in the tree corresponds to a path $\alpha(n)$ in the network: $\alpha(n)$ consists of the edges corresponding to the leaves of the subtree rooted at n. With each node n we associate two fields $f(n)$ (flow) and $c(n)$ (capacity). $f(n)$ is the flow pushed in the corresponding path that has not been added to the subtrees. Consequently, if an edge e is in a PF, then $f(e) = \sum f(m)$, where the sum is over nodes m on the path from the leaf corresponding to e to the root. This is a different way to implement the idea of big edges introduced in [9]. (Big edges stand for paths and were used to send flow directly as long as no (small) edge on the path became saturated.) $c(n)$ is the minimal residual capacity (i.e., capacity minus flow) of an edge on the path corresponding to n ignoring flows associated with proper ancestors of n. Thus

$$c(n) = \min_{e \in \alpha(n)} \left[c(e) - \sum_{m \in \beta(n,e)} f(m) \right], \qquad \text{where } \alpha(n) \text{ is as above and } \beta(n, e)$$

is the path from the leaf corresponding to e to n. Whenever we consider the capacity field of a node n, either n is a root of a PF, or all its proper ancestors have flow fields zero.

[2] We refer to a node in the tree that represents a PF, and to a vertex in the layered network.

In both cases $c(n)$ is the minimal residual capacity of an edge in $\alpha(n)$. In particular $c(n) = 0$ exactly when there is a saturated edge in $\alpha(n)$.

It follows from the definitions that $c(n) = \min c(n_i) - f(n)$, where the n_i's are the sons of n. Consequently, except for leaves we do not need the flow field ($f(n) = \min c(n_i) - c(n)$). We will not use this observation in the sequel.

We sometimes *transfer flow* from a node n to its sons n_i. This is achieved by $(c(n_i), f(n_i)) \leftarrow (c(n_i) - f(n), f(n_i) + f(n))$ and $(c(n), f(n)) \leftarrow (c(n), 0)$.

The operations which we perform on *PF*'s are concatenate, split, and delete edges. The corresponding known operations on 2–3 trees [1] can be easily modified to support each one of these operations in time $O(\log V)$. The modifications (given below) are needed to update the two fields associated with each node.

Assume we concatenate PF_1 to PF_0 and (the tree corresponding to) PF_0 has larger height. The root of PF_1 becomes a son of a node n on the path from the root of PF_0 to the leaf corresponding to its last edge. But the proper ancestors of n in PF_0 may have flows associated with them. These flows have to be eneventually added to the edges of PF_0, but have nothing to do with the edges of PF_1. So, when we go down the tree in the search for n we transfer the flows of the nodes on the path to their sons. The case that the tree PF_1 has larger height is analogous.

When we split a *PF* at a certain leaf we first have to go along the path from the root to this leaf and transfer flows of the nodes on the path to their sons. When we delete the last edge of a *PF* we first have to go up the tree and compute its flow, and update $c(n)$ on the way.

For every vertex v we have a pointer, which (if defined) points to the unique leaf (in a *PF*) corresponding to (the currently first) edge in out(v). Consequently, the if's asked in steps 4, 5, 6 can be easily answered.

Pushing flow in step 8 is performed by $(c(r), f(r)) \leftarrow (0, f(r) + c(r))$, where r is the root of PF_0. Locating saturated edges is achieved by going down the tree and transferring flow of a node to its sons for all nodes with (new) capacity field zero. Exactly all ancestors of leaves corresponding to saturated edges are visited and in the end all leaves with capacity field zero represent the saturated edges.

To make the algorithm complete we need to add at its end another step that transfers all flows of nodes in the remaining *PF*'s down the trees. This is needed so that edges that have not been deleted will have the correct flow, and it can be achieved by deleting all these edges.

The Analysis of the Running Time

In computing the running time we can ignore the time needed to answer the if's in steps 4, 5, 6 because it is subsumed by the $O(\log V)$ needed for the following concatenation. We can also ignore the $O(\log V)$ time per locating a saturated edge because it is subsumed by the $O(\log V)$ needed to delete it. Consequently, the time bound is proportional to $\log V$ times the number of operations on *PF*'s in step 3, 6, 7, and 8.

The total time needed to delete edges is $O(E \log V)$ because we delete at most E edges.

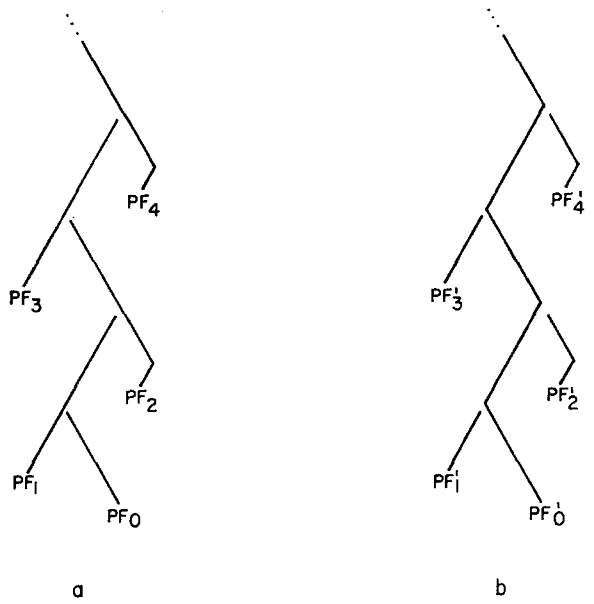

FIG. 3. Before and after the construction of the path.

Let K be the number of times we enter into another PF in the construction of PF_0 (either splitting it as in step 6 or entering into its first vertex as in step 5). The number of times we concatenate PF's is at most $K + E$. (The term E is due to step 4.) The number of times we split a PF is at most $K + E$. (The term E is due to step 8.) Consequently, the order of the time bound is $E \log V + K \log V$. We will show that $K = O(E \log V)$. This will imply that the time bound for the sub-problem is $O(E \log^2 V)$ and for the max-flow problem is $O(EV \log^2 V)$.

Figure 3 above describes part of a search for a path. It is quite possible that during the construction of one path we will enter many new PF's on the way (as many as k) but while PF_1, PF_2, PF_3 were at "distance" m, $m - 1$, $m - 2$, resp. from a "main highway" before the construction of the path, all will be at distance *one* after the construction. (A "main highway" is either a PF that enters t or a PF that enters a dead end. The "distance" is the number of PF's changed on the way.) This is exactly what happens in path compression in the "union find" algorithm: the various nodes on the way are at distances m, $m - 1$, $m - 2 \cdots$ from the root but all become attached to the root. Viewed differently, this phenomenon means that while occasionally we may have to spend a long time, this time is not completely wasted because it is a sort of investment for the future.

THE COMBINATORIAL LEMMA. *Consider a process that modifies a set of pairs S, $S \subseteq \{(p, q) \mid 1 \leqslant p < q \leqslant N\}$. Initially the set need not be empty but it contains at most one pair (p, q) for every p. The process consists of M stages. During a stage some pairs may leave S and at its end some pairs may enter S. The leaving pairs constitute a chain $((p_1, p_2), (p_2, p_3) \cdots)$. An entering pair (p, r) must satisfy* (1) *no pair (p, q) is currently in S, and* (2) *if (p, q) was in S and left in the same stage as (p', q'), then $r > q'$. Under these conditions the number of pairs that left S is $O((M + N)\log(M + N)/\log(\lceil(M + N)/N\rceil)$).*

The proof of the Lemma is identical to the direct proof of Corollaty 1 below given in [17]. (Tarjan attributes the result to Paterson [15].) The conditions of the lemma are sufficient for the proof to go through. In fact the condition that the leaving pairs must constitute a chain can be weakened. For completeness we give the proof of the Lemma in Appendix 2.

Corollary 1. *The time of the "union find" with path compression (without balancing) with m finds and n elements is $O((m + n)\log(m + n)/\log(\lceil(m + n)/n\rceil)$).*

Proof. Perform all unions first and consider partial finds [17] on the resulting tree. Number the nodes of the tree so that a node has a number larger than the numbers of its sons. The set S of pairs consists of the edges of the tree. The stages are the finds. The second condition on an entering pair follows from the fact that the partial finds are derived from finds, i.e., we need always go up to the root at the time of the find. ▌

Corollary 2 (or Theorem 1). *The time bound of our algorithm is $O(EV \log^2 V)$.*

Proof. First let us number the *PF*'s. PF_0—the path we construct—does not have a number. The only case where *PF*'s get new numbers is immediately after deleting saturated edges and splitting PF_0: Consider Fig. 1. If N_0 is the last number given to a *PF*, then α_1 and α_2 are given the numbers $N_0 + 1$ and $N_0 + 2$, respectively. (α is not given a number because it becomes the new PF_0.) It follows that $N \leqslant E$, where N is the last number given to a *PF*. *PF*'s may change: if the case described in Fig. 2 occurs, then α' gets the number of α. (α'' does not need a number because it is concatenated to PF_0.)

PF number *p points at PF* number q if the former ends in a vertex that belongs to the latter but is not its last vertex. For example, PF_i points at PF_{i+1} in Fig. 3.

Fact. If PF number p points at PF number q at a certain moment but not at a previous moment, then the latter is a new PF (i.e., *the number q has just been given*).

We now show that the conditions of the Lemma hold. Let $S = \{(p, q) \mid PF \text{ number } p$ points at PF number $q\}$. By the Fact, $1 \leqslant p < q \leqslant N$. The process that changes S is our algorithm. A stage starts each time we visit an edge for the first time (step 4). Thus the number of stages $M \leqslant E$. That the leaving pairs constitute a chain follows from Fig. 3. (PF_i pointed at PF_{i+1} at the beginning of the stage but not at its end.) The first condition on entering pairs is obvious. (No two *PF*'s can go through the same vertex.) The second condition follows from the Fact. Since $M, N \leqslant E$ it follows from the conclusion of the Lemma that the number of pairs that left S is $O((M + N)\log(M + N)/\log(\lceil(M + N)/N\rceil) = O((M + N)\log(M + N)) = O(E \log V)$. But it follows from Fig. 3 that except for the first time in a stage, whenever we enter another *PF* in the construction of PF_0 at least one pair leaves S. Thus K, the total number of times we enter another *PF*, is bounded above by E + the number of pairs that left $S = O(E \log V)$. ▌

EXAMPLES REQUIRING $\Omega(EV\log^2 V)$ TIME

In this section we show that the upper bound that we derived in the previous section is tight. Given V and E, $V \leqslant E \leqslant V^2$, we construct a network with $O(V)$ vertices and $O(E)$ edges that forces the algorithm to run $\Omega(EV \log^2 V)$ steps.

Assume that part of the layered network is the tree described in Fig. 4. Each arc in the tree is a simple path of length r, where r is a parameter that will be determined later.

Assume that when a path constructed passes through V_p for the ith time it enters the tree in leaf number i mod 8. It is easy to see that after the eighth visit the complete tree has been visited and in any following visit three *PF*'s will be split (in step 6). Each of these splits will cost log r because each *PF* has length of at least r.

The tree in Fig. 4 is denoted by $T_{3,r}$ because it has "depth" 3. One can similarly define the tree $T_{d,r}$ with 2^d leaves and number them in such a way that after each leaf is visited once, every path will cost $\sim d \log r$ because there will be d splits of *PF*'s of length at least r.

The layered network that arises in our examples is described in Fig. 5. It consists of one copy of $T_{d,r}$ with $d = \lceil \log V^{1/2} \rceil$ and $r = \lceil V^{1/2} \rceil$, and a complete bipartite graph with sides $\hat{V}_1$ and $\hat{V}_2$ that contain $E^{1/2}$ vertices each. The two edges that are incident with s and t stand for two simple paths of length $\theta(V)$. The only difference distinguishing the layered networks arising in the different phases will be that these two paths will become longer. The end vertex of the path starting at s is connected to all the vertices of $\hat{V}_1$ and the ith vertex in $\hat{V}_2$ is connected only to leaf number i mod 2^d of $T_{d,r}$. Without loss of generality the number of leaves 2^d ($\leqslant V^{1/2}$) divides $E^{1/2}$. (Otherwise we take fewer vertices in $\hat{V}_1$ and $\hat{V}_2$.)

The edges in the bipartite graph have capacity one and are called *bottlenecks*. Some edges on the two paths mentioned above will have finite but large capacity and will not become saturated during the phase. (Two will become saturated at the end of the phase.) All the other edges have infinite capacity.

One can easily verify that the numbers of vertices and edges in this layered network are $O(V)$ and $O(E)$, respectively. Also, it is quite clear that there are E successful paths which saturate the E bottlenecks, and the ith path goes through leaf number i mod 2^d of $T_{d,r}$. (Note that the order according to which the edges in out(v) are arranged, for each vertex v, is important.) Consequently, the time required per phase is $\sim Ed \log l = \Omega(E \log^2 V)$.

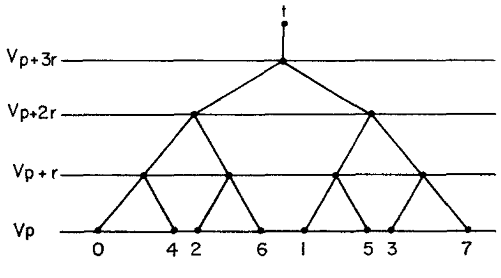

FIG. 4. $T_{3,r}$.

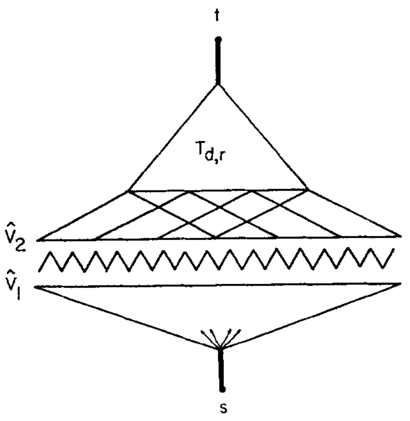

FIG. 5. The layered network.

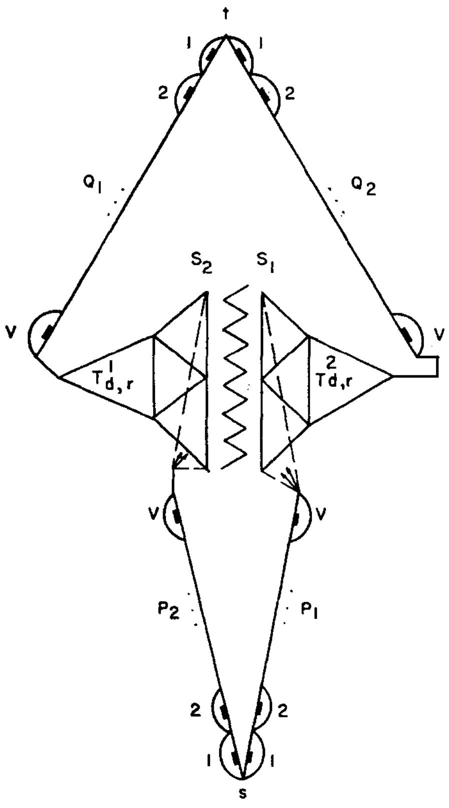

FIGURE 6

The parametrized network that requires $\Omega(EV \log^2 V)$ time by our algorithm is given in Fig. 6. It contains two copies of $T_{d,r}$ (with d and r as above), and a complete bipartite graph with sides S_1 and S_2 of size $E^{1/2}$. Each side is connected to the leaves of the corresponding copy of $T_{d,r}$ in the same way $\hat{V}_2$ was connected above to the leaves of $T_{d,r}$.

The mechanism used to force the "bad" layered network of Fig. 5 to repeat in $\Omega(V)$ phases is similar to the one used in [11] for similar purposes. Therefore we omit some of the details (e.g., the exact constants mentioned below). The network contains four similar sections P_1, P_2, Q_1, Q_2. Each section contains V *units*. A unit has a short part and a long part. Both are simple paths of different constant length. In the short part there is an edge called the *bottleneck of the unit*. The bottleneck of unit number i has capacity iE. The bottlenecks (the edges in the bipartite graph) have capacity one. All other edges have infinite capacity. The only difference between the sections is that P_2 and Q_2 have in addition to the V units also an additional simple path of constant length. The end vertex of P_1 [P_2] is connected to all vertices of S_1 [S_2].

In the first phase the layered network will contain the short parts of the units in P_1 and Q_1. $T^1_{d,r}$ S_1 and S_2 will play the role of $T_{d,r}$ $\hat{V}_1$ and $\hat{V}_2$. A path from P_2 [Q_2] will also appear but will be disconnected from t because the shortest path through P_1 [through Q_1] from s to S_2 [from S_2 to t] is shorter than the corresponding path through P_2 [Q_2]. Omitting these redundant paths we see that the layered network in the first phase is essentially the one in Fig. 5.

In the second phase the picture is reversed: Due to the fact that the short parts of units number 1 in P_1 and Q_1 become saturated at the end of the first phase, it is now shorter to use the shortest paths in P_2 and Q_2 with $T^2_{d,r}$ S_2 and S_1 as $T_{d,r}$ $\hat{V}_1$ and $\hat{V}_2$. The second phase will push back the E units of flow in the bottlenecks and when it ends the bottlenecks of the units number 1 in P_2 and Q_2 will become saturated. Phases number 3, 5,..., $2V - 1$ [4,..., $2V$] will be similar to phase number 1 [2]. Consequently, the time required will be $\Omega(EV \log^2 V)$. Note that the network of Fig. 6 has only $O(V)$ vertices and edges more than the one in Fig. 5, and therefore it has also $O(V)$ vertices and $O(E)$ edges.

Planar Networks—A—Known Result for Free

Consider the special case in which the network is s–t planar. (It is planar even with an additional edge connecting s and t.) Berge [2] described an $O(V^2)$ algorithm (which is essentially due to Ford and Fulkerson [8]) that solves the max-flow problem (not the sub-problem) using essentially Dinic's approach. (The interesting part is that this works for the problem itself.) He always sends flow through the uppermost path. Itai and Shiloah [12] have found a clever way to implement this algorithm in $O(V \log V)$. We get (for free) an alternative $O(V \log V)$ algorithm by using our algorithm instead of Dinic's to implement Berge's algorithm, and by showing that the number of times we break into a *PF* is $O(V)$ in this case. (Recall that $E = O(V)$ for planar graphs.)

Consider Fig. 7 that describes a possible breaking into a *PF* in a vertex u. It contains (from top down) the additional edge from s to t, the last path from s to t that passed

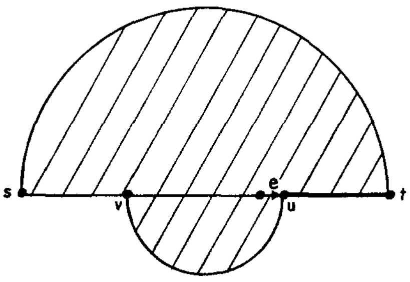

FIGURE 7

through u, and the part of the current path from the last vertex v before u which belonged also to the other path. Let e be the edge in the old path that enters u.

We charge this breaking into a PF to e. Any edge e can be charged at most once, because all future paths will not enter the shaded area. (We use an upper path first.) Therefore the number of times we enter into another PF while constructing PF_0 (i.e., K) is $O(E) = O(V)$, and the time bound is $O(V \log V)$.

We suspect that $K = O(V)$ in the case that the network is planar but not s–t planar. If this is true, then we have an $O(V^2 \log V)$ algorithm for max-flow in planar networks. In [12] there is a quite complicated $O(V^2 \log V)$ algorithm for *undirected* planar networks. In case of directed planar networks it is still unknown how to use the planarity and the best algorithm is the one presented here whose time bound is $O(V^2 \log^2 V)$.

CONCLUSION

There is no known lower bound for the max-flow problem. However, if one uses Dinic's approach (and all the algorithms in Table 1 except the first one use it), then $\Omega(EV)$ is a lower bound. So we are pretty close to making an optimal use of Dinic's approach.

The discovery of a common phenomenon in two quite different algorithms seems to us the most important contribution of this paper, especially in light of the lack of such common phenomena in the theory of analysis of algorithms. There may be a neater way to state the Combinatorial Lemma and we are still trying to find it. Hopefully, it will turn out to be useful in analysing run times of other algorithms.

APPENDIX 1: THE ORIGINAL VERSION

We will refer to the edges of the layered network as small edges. The information will be maintained in *big edges* that are defined as follows: a big edge $\hat{e}$ is either a small edge $e = (u, v)$ with *length*$(e) = 1$, *source*$(e) = u$ and *destination*$(e) = v$, or it is composed of two big edges $\hat{e}_1$ and $\hat{e}_2$ such that *source*$(\hat{e})$ = *source*$(\hat{e}_1)$, *destination*$(\hat{e})$ = *destination*$(\hat{e}_2)$, *destination*$(\hat{e}_1)$ = *source*$(\hat{e}_2)$, and *length*$(\hat{e})$ = *length*$(\hat{e}_1)$ + *length*$(\hat{e}_2)$. In the latter case $\hat{e}_1$

and $\hat{e}_2$ are the ***components*** of $\hat{e}$ and $\hat{e}$ is the ***bigger edge*** of $\hat{e}_i$, $i = 1, 2$. If $\hat{e}_1$ and $\hat{e}_2$ are the components of $\hat{e}$, then either *destination*($\hat{e}$) $= t$ or *length*($\hat{e}_1$) $=$ *length*($\hat{e}_2$) $=$ a power of two (or both). A big edge $\hat{e}$ represents (in an obvious way) a path in the layered network. It can be viewed as a binary tree which will always be complete unless *destination*($\hat{e}$) $= t$. A big edge $\hat{e}$ belongs to a big edge $\hat{f}$ if the tree corresponding to $\hat{e}$ is a subtree of the tree corresponding to $\hat{f}$. At any given time and for any vertex v, at most one small edge in *out*(u) (the first edge in *out*(u) that has not been deleted) will be a big edge. A big edge $\hat{e}$ will have two additional fields associated with it: *flow*($\hat{e}$) and *capacity*($\hat{e}$). These fields have exactly the same meaning as the flow and capacity fields of nodes in *PF*'s. For every vertex v the algorithm will maintain *largest*(v)—the largest big edge $\hat{e}$ with *source*($\hat{e}$) $= v$.

The algorithm will use two primitive operations. The first composes a new big edge $\hat{e}$ from $\hat{e}_1$ to $\hat{e}_2$ (that satisfies the conditions mentioned above) and the second breaks a big edge $\hat{e}$ into its components and transfers to them the flow of $\hat{e}$. The flow transfer is exactly the same as the flow transfer from a node to its sons in a *PF*. It is obvious how to implement these two operations in a constant amount of time. We will also delete a small edge e from the layered network. The deletion will include setting *largest*(*source*(e)) to undefined. We will sometimes *visit* a small edge e. It will mean making it a big edge and setting *largest*(*source*(e)) to e.

We now give the macro algorithm. The asterisks denote steps that are going to be refined later.

```
1.  Initialize and repeat 2–4:
2*. Find a path from $s$ to $t$.
3*. If there is no such path, transfer flows to small edges and stop.
4*. Push flow along path and destroy all saturated edges.
```

Finding a path (2): we use a stack STACK to store a chain of big edges that represent the path being constructed. The vertex u is the last vertex of this path. The word edge will stand for big edge.

```
A.   $u \leftarrow s$; STACK $\leftarrow$ empty;
     repeat $B - J$:
B.    If $u$ is a dead-end then do:
C.     If STACK is empty, stop (the search has failed).
D*.    Otherwise destroy top-most edge in STACK.
E.    Else do: if largest($u$) is undefined visit the first small edge in out($u$)
F.    $\hat{e} \leftarrow$ largest($u$)
G*.   if bigger edge of $\hat{e}$ is defined, destroy all big edges to which $\hat{e}$ belongs.
H*.   insert $\hat{e}$ into STACK and try to modify STACK.
I.    $u \leftarrow$ destination($\hat{e}$)
J.    if $u = t$, stop (a path has found).
```

D is executed by repeatedly breaking the topmost edge in STACK replacing it by its components until the topmost edge is a small edge, in which case it is deleted, and u is set to its source. G is executed by first finding all the ancestors of $\hat{e}$ in the tree and then breaking them from the root down. Trying to modify STACK in H means repeatedly composing the two topmost edges in STACK as long as they have equal length or the one on top, $\hat{e}$, satisfies *destination*$(\hat{e}) = t$.

Transferring flow to small edges at the end (step 3) is performed by visiting every vertex in the layered network, setting $\hat{e} \leftarrow$ *largest*(v) and destroying all big edges to which $\hat{e}$ belongs as in $2G$. (Note that the order of the visits is not important.)

Pushing flow in step 4 is achieved by setting $f(\hat{e}) \leftarrow c(\hat{e})$ and $c(\hat{e}) \leftarrow 0$ for the unique edge $\hat{e}$ in STACK. Destroying saturated edges is executed by going down the tree that corresponds to $\hat{e}$ and breaking all saturated big edges on the way.

This version does not use *PF*'s, but *PF*'s appear implicitly in it. It maintains large pieces of *PF*'s (the big edges). These pieces are usually paths of length which is a power of two. The only exceptions are paths that end at t. We could get rid of these exceptions by first adding some layers so that the number of layers is one plus a power of two. Thus, one can view this version as implementing *PF*'s by binomial queues.

The version given in the paper is conceptually simpler because it implements a *PF* as one unit. On the other hand the above version seems to be slightly faster because in many cases the manipulations of the big edges require only a few operations even if the big edges involved represent long paths. For example, to split a big edge exactly in the middle requires only constant number of operations no matter how long the path it represents, while splitting a *PF* of length l always costs $\log l$. On the other hand the version using big edges seems to be too rigid to be useful for the s–t planar case.

The analysis of the running time is similar to the one given in the paper. Also, similar examples show that the bound is tight for this version. One has to make sure that the vertices of $T_{d,r}$ in Fig. 5 (where the splits occur) belong to odd numbered layers. As a result, most of these splits (which in this version are replaced by step $2G$) will involve destroying $\log r$ big edges. (Because if u is in an odd numbered layer *length*(*largest*$(u)) = 1$.)

APPENDIX 2: THE PROOF OF THE COMBINATORIAL LEMMA

Let F be the set of pairs that left S. By the conditions on entering pairs, a pair cannot leave S twice. So our goal is to estimate the size of F. Let $b = \lceil (M+N)/N \rceil$ and let $z = \lceil \log_b N \rceil + 1$. For $1 \leqslant i \leqslant z$, let $M_i = \{(p, q) \in F \mid \lceil q/b^{i-1} \rceil > \lceil p/b^{i-1} \rceil,\ \lceil q/b^i \rceil = p/b^i \rceil\}$; i.e., a pair in F belongs to M_i if $i-1$ is the most significant position where the b-ary representations of p and q differ. Note that $F = \bigcup_{i=1}^{z} M_i$. Also, let $L_i = \{(p, q\} \in M_i \mid$ among the pairs in M_i that left S in the same stage, (p, q) has largest second component$\}$.

FACT 1. *For* $1 \leqslant i \leqslant z$, $|L_i| \leqslant M$.

Proof. The pairs leaving S in the same stage constitute a chain. ∎

FACT 2. *If* $(p, q) \in M_i - L_i$ *and* (p, r) left S after (p, q), *then* $\lceil r/b^{i-1} \rceil \geqslant \lceil q/b^{i-1} \rceil + 1$.

Proof. Since $(p, q) \in M_i - L_i$, it follows that another pair (p', q') in M_i left S at the same stage as (p, q), and $q' \geqslant q$. Since the leaving pairs constitute a chain $p' \geqslant q$. By the conditions on entering pairs, $r > q'$. It follows that $\lceil r/b^{i-1} \rceil \geqslant \lceil q'/b^{i-1} \rceil > \lceil p'/b^{i-1} \rceil \geqslant \lceil q/b^{i-1} \rceil$, and $\lceil r/b^{i-1} \rceil \geqslant \lceil q/b^{i-1} \rceil + 1$. (The strict inequality because $(p', q') \in M_i$.) ▮

FACT 3. *If $(p, q_1), (p, q_2),..., (p, q_{b+1})$ are all in $M_i - L_i$ and left S in this order, then* $\lceil q_{b+1}/b^i \rceil \geqslant \lceil q_1/b^i \rceil + 1$.

Proof. Apply Fact 2 b times and divide by b.

FACT 4. $| M_i - L_i | \leqslant bN$.

Proof. Assume $(p, q_1), (p, q_2) \cdots (p, q_{b+1})$ are all in $M_i - L_i$ and left S in this order. By Fact 3 $\lceil q_{b+1}/b^i \rceil \geqslant \lceil q_1/b^i \rceil + 1$, but $\lceil q_{b+1}/b^i \rceil = \lceil p/b^i \rceil = \lceil q_1/b^i \rceil$ because (p, q_1) and (p, q_{b+1}) belong to M_i-contradiction. Therefore, there can be at most b pairs in $M_i - L_i$ that have p as a first component, and thus $| M_i - L_i | \leqslant bN$. ▮

By Fact 1, Fact 4, and the size of z, $|F| \leqslant (M + bN)(\log_b N + 1) \leqslant (2M + N) \log bN/\log b = O(M + N)\log(M + N)/\log(\lceil (M + N)/N \rceil)$. ▮

Remarks. 1. The condition on pairs leaving S can be weakened: It suffices to assume that if (p, q) and (p', q') left S in the same stage, then if $q' \geqslant q$ then $p' \geqslant q$.

2. The bound given is known to be best only for $M = \theta(N)$ and $M = \Omega(N^\alpha)$ for $\alpha > 1$ [17].

3. This bound looks different but is exactly the same as the one in [17].

4. For the purpose of proving the time bound of the algorithm the rough bound of $O((M + N)\log(M + N))$ suffices. This bound can be obtained by taking $b = 2$ in the proof above.

ACKNOWLEDGMENTS

We would like to thank Joel Seiferas, Adi Shamir, and Janos Simon for making valuable suggestions that helped to formulate the current version of the algorithm.

Note added in proof. In his Ph.D. thesis [18] Sleator found an ingenious way to implement our algorithm in time $(EV \log V)$. He modified a new data structure, called a biased 2–3 tree [19], and used this data structure instead of 2–3 trees to represent PF's. The only other change is that the order of concatenations during a stage is modified. All the PF_1's of a maximal sequence of steps 5 and 6 are first concatenated and only then this PF is concatenated to PF_0. The important property of the new data structure is that except a total of $O(E)$ operations on PF's the cost of the operations in a stage sum up telescopically (the ith cost is $a_i - a_{i-1}$). Consequently, each term in the bound for the time bound is $O(E \log V)$. The examples given here show that the $O(EV \log V)$ bound is tight.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass., 1974.
2. C. BERGE AND A. GHOUILA-HOURI, "Programming, Games and Transportation Networks," Methuen, London, 1965.
3. B. V. CHERKASKY, Algorithm of construction of maximal flow in networks with complexity of $O(V^2 \sqrt{E})$ operations, *Math. Meth. Solution Econ. Prob.* **7** (1977), 117–125 (in Russian).
4. E. A. DINIC, Algorithm for solution of a problem of maximal flow in a network with power estimation, *Soviet Math. Dokl.* **11** (1970), 1277–1280.
5. J. EDMONDS AND R. M. KARP, Theoretical improvement in algorithmic efficiency for network flow problems, *J. Assoc. Comput. Mach.* **19**, 2 (1972), 248–264.
6. L. R. FORD AND D. R. FULKERSON, "Flows in Networks," Princeton Univ. Press, Princeton, N. J., 1962.
7. L. R. FORD AND D. R. FULKERSON, Maximal flows through a network, I.E.E.E. Trans. Inform. Theory, **IT-2** (1956), 117–119.
8. L. R. FORD AND D. R. FULKERSON, Maximal flow through a network, *Canad. J. Math.* 8 (1956), 399–404.
9. Z. GALIL, A new algorithm for the maximal flow problem, *in* "Proceedings, 19th IEEE Symposium on Foundations of Computer Science, Ann-Arbor, Mich., October 1978," pp. 231–245; to appear as An $O(E^{2/3}V^{5/3})$ algorithm for the maximal flow problem, *Acta Inform.*, in press.
10. Z. GALIL AND A. NAAMAD, unpublished manuscript, August 1978.
11. Z. GALIL, "On the Theoretical Efficiency of Various Network Flow Algorithms," IBM report, RC7320, September 1978; *Theoret. Comput. Sci.*, in press.
12. A. ITAI AND Y. SHILOACH, Maximum flow in planar graphs, *SIAM Comput.* **8**, 2 (1979), 135–150.
13. A. V. KARZANOV, Determining the maximal flow in a network by the method of preflows, *Soviet Math. Dokl.* **15** (1974), 434–437.
14. V. M. MALHOTRA, M. PRAMODH KUMAR, AND S. N. MAHESHWARI, An $O(V^3)$ algorithm for finding the maximum flows in networks, *Inform. Proc. Lett.* **7**, **6** (1978), 277–278.
15. M. PATERSON, unpublished (see [17]).
16. Y. SHILOACH, "An $O(nI \log^2 I)$ Maximum-Flow Algorithm," Stanford Technical Report STAN-CS-78-702, December 1978.
17. R. E. TARJAN, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput.Mach.* 22, 2 (1975), 215–225.
18. D. D. SLEATOR, An $O(mn \log m)$ algorithm for maximum network flow, Ph.D. thesis, Stanford University, to appear.
19. S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, Biased 2–3 trees, *in* "Proceedings, 21th IEEE Symposium on Foundations of Computer Science, Syracuse, New York, October 1980," pp. 248–253.