SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0004$01.00/0

MAXIMUM FLOW IN PLANAR NETWORKS*

ALON ITAI† AND YOSSI SHILOACH‡

Abstract. Efficient algorithms for finding maximum flow in planar networks are presented. These algorithms take advantage of the planarity and are superior to the most efficient algorithms to date. If the source and the terminal are on the same face, an algorithm of Berge is improved and its time complexity is reduced to $O(n \log n)$. In the general case, for a given $D > 0$ a flow of value D is found if one exists; otherwise, it is indicated that no such flow exists. This algorithm requires $O(n^2 \log n)$ time. If the network is undirected a minimum cut may be found in $O(n^2 \log n)$ time. All algorithms require $O(n)$ space.

Key words. algorithm, network flow, planar graph

1. Introduction.

1.1. Basics. A *directed flow network* $N = (G, s, t, c)$ is a quadruple, where:

(i) $G = (V, E)$ is a directed linear graph;

(ii) s and t are distinct vertices, the source and the terminal respectively;

(iii) $c: E \to R^+$ is the capacity function (R^+ denotes the set of nonnegative real numbers).

Henceforth, n and m denote the number of vertices and edges respectively and $u \to v$ denotes a directed edge from u to v.

A function $f: E \to R^+$ is a *flow* if it satisfies:

(a) the capacity rule: $f(e) \leqq c(e)\ \forall e \in E$;

(b) the conservation rule:

$$\mathrm{IN}\,(f, v) = \mathrm{OUT}\,(f, v) \quad \forall v \in V - \{s, t\}.$$

Where $\mathrm{IN}\,(f, v) = \sum_{\{u: u \to v \in E\}} f(u \to v)$ is the total flow entering v; and $\mathrm{OUT}\,(f, v) = \sum_{\{w: v \to w \in E\}} f(v \to w)$ is the total flow emanating from v.

The flow value $|f|$ is defined by

$$|f| = \mathrm{OUT}\,(f, s) - \mathrm{IN}\,(f, s).$$

A flow is a *maximum flow* if $|f| \geqq |f'|$ for any other flow f'.

1.2. Results. Ford and Fulkerson [6] stated and proved the Max Flow-Min Cut theorem and established the technique of augmenting paths for finding a maximum flow. Edmonds and Karp [5] provided the first polynomial algorithm ($O(nm^2)$), based on finding shortest augmenting paths. By using auxiliary graphs, Dinic [3] managed to reduce the time bound to $O(n^2m)$ (see also [4]). By the method of preflows Karzanov implemented Dinic's algorithm in $O(n^3)$ time [9]. Note that when $m = O(n)$ all these algorithms require $O(n^3)$ time [1].

A flow network $N = (G, s, t, c)$ is *planar* if G is a planar graph. (See [7, Chap. 11] for the properties of planar graphs.) In this paper we discuss the problem of finding á maximum flow in planar networks.

Section 2 deals with (s, t) planar networks (s and t are on the same face of G). Berge [2, p. 190] proposed an algorithm to find a maximum flow, a straightforward implementation of which requires $O(n^2)$ time. Here, an $O(n \log n)$ implementation is presented. It is also shown that $O(n \log n)$ is a lower bound to any implementation of

* Received by the editors February 7, 1977, and in revised form July 6, 1978.

† Department of Computer Science, Technion—Israel Institute of Technology, Haifa, Israel.

‡ Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

Berge's algorithm. (An $O(n \log n)$ algorithm to find a minimum (s, t)-cut for this case appears in [8, p. 151]; however, this algorithm does not produce the flow function itself.)

In § 3, for $D > 0$ we find a flow of value D in a directed planar network if such a flow exists, otherwise we indicate this fact. This algorithm requires $O(n^2 \log n)$ time.

In undirected graphs, let u–v denote an undirected edge between the vertices u and v. A flow network is *undirected* if the graph is symmetric, i.e. if $u \to v \in E$ then also $v \to u \in E$ and $c(u \to v) = c(v \to u)$. In this case G is considered to be undirected (each pair of directed edges $u \to v$ and $v \to u$ is replaced by the undirected edge u–v with the same capacity).

In § 4, we present an $O(n^2 \log n)$ algorithm for finding a minimum (s, t) cut in an undirected planar network. Thereby, a maximum flow in an undirected network may be found in $O(n^2 \log n)$ time.

The Appendix contains an alternative proof of the validity of Berge's algorithm.

1.3. Data structures. Throughout the paper we assume that the graph G has a fixed planar representation.

The graph is represented by incidence lists, i.e. each vertex v has a list E_v of all the edges to which v is incident (edges of the form $u \to v$ or $v \to w$).

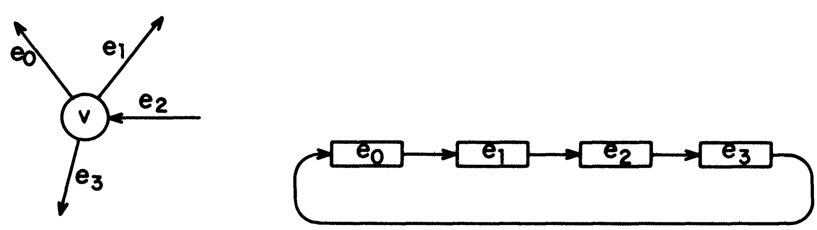


FIG. 1

The set E_v is represented by a circular list corresponding to the circular clockwise ordering of the edges around v (see Fig. 1). Each edge $e \in E_v$ has a unique successor edge $\mathrm{succ}_v(e)$ in E_v. The lists E_v are used to find successor edges. In the course of the algorithm some edges are deleted from the network. The deletion of an edge from E_v is deferred to the time it is traversed when looking for a successor edge. At this time the predecessor edge is known; consequently, singly linked lists suffice. Each edge induces a linear order on E_v as follows:

$$e_0 = e, \qquad e_i = \mathrm{succ}_v(e_{i-1}); \qquad i = 1, \cdots, |E_v| - 1.$$

2. Maximum flow algorithm on (s, t) planar networks. This sections deals with (s, t) planar networks, i.e. s and t belong to the same face, and can be connected by an edge without violating the planarity. Without loss of generality, $t \to s \in E$, (otherwise it may be added with zero capacity). We also assume that $t \to s$ is incident with the exterior face.

$P = (v_0, \cdots, v_k)$ is a directed (v_0, v_k)-*path* if $v_{i-1} \to v_i \in E$, $i = 1, \cdots, k$. A path is *simple* if all its vertices are distinct. Let $P_1 = (s = v_0, \cdots, v_k = t)$ and $P_2 = (s = u_0, \cdots, u_e = t)$ be two simple (s, t) paths. P_1 *lies above* P_2 if $v_i = u_i$, $i = 0, \cdots, r$, $u_{r+1} \neq v_{r+1}$ and $v_r \to v_{r+1}$ precedes $v_r \to u_{r+1}$ in the linear order of E_{v_r} induced by $v_{r-1} \to v_r$. (If $r = 0$ then the order on E_s is induced by $t \to s$.)

The "lies above" relation is a full anti-symmetric order relation on the set of all simple (s, t)-paths. Hence, it has a unique maximum the *uppermost path*. (See Fig. 2.)

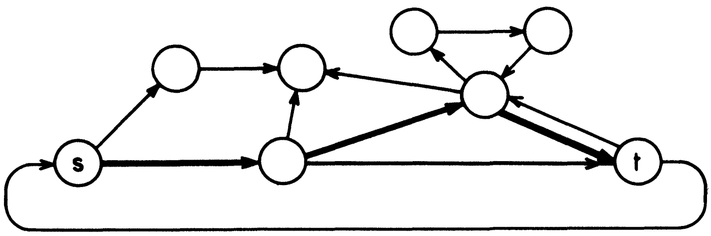


FIG. 2. *The uppermost path appears in boldface.*

2.1. Berge's algorithm. If s and t are on the exterior face, maximum flow may be found by Berge's algorithm. The algorithm starts by pushing as much flow as possible through the uppermost path. Thereby, at least one edge becomes saturated. Such an edge is deleted, and the process is repeated using the uppermost path of the resultant graph.

The algorithm uses the *residual capacities*: $\text{res}(e)=c(e)-f(e)$, where f denotes the flow found thus far by the algorithm.

Let P be an (s,t) path, an edge $e^B \in P$ is a *bottleneck* if $\text{res}(e^B)=\text{Min}_{e\in P}\,\text{res}(e)$. The bottleneck value is $\text{res}(e^B)$.

BERGE'S ALGORITHM.

1. Initialize: set $i=1$;
   start with zero flow:
   for all $e\in E$ set $f_0(e)=0$, $\text{res}(e)=c(e)$.
2. Find the uppermost path P_i^B, if none exists then stop.
3. Let e_i^B be a bottleneck of P_i^B.
4. Increase the flow by $\text{res}(e_i^B)$ units along P_i^B:

$$f_i^B(e)=\begin{cases} f_{i-1}^B(e)+\text{res}(e_i^B) & \text{if } e\in P_i^B \\ f_{i-1}^B(e) & \text{otherwise} \end{cases}$$

$$\text{res}(e)=c(e)-f_i^B(e).$$

5. Delete the bottleneck e_i^B from G.
6. Set $i=i+1$ and go to 2.

The algorithm is illustrated in Fig. 3.

A proof of the validity of Berge's algorithm can be found in [2]. See the Appendix for an alternative self-contained proof.

A straightforward implementation of Berge's algorithm (even step 4 alone) requires $O(n^2)$ time for the network of Fig. 4. (Note that all the algorithms mentioned in the introduction require $O(n^2)$ time for this network.)

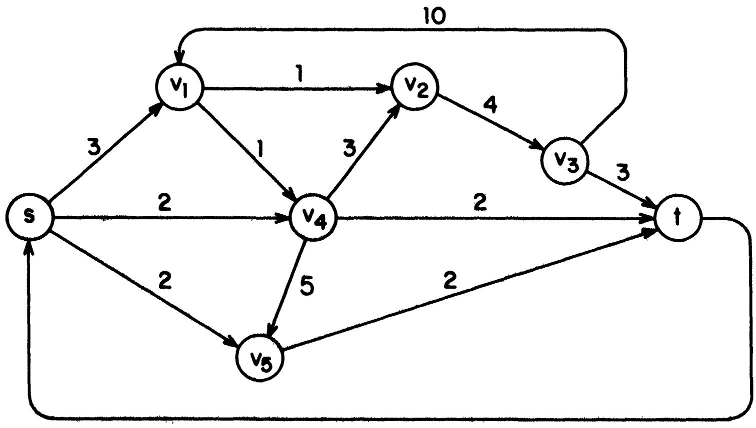
Let $I(e)$ and $L(e)$ denote the index of the first and last uppermost paths in which the edge e participates. The following lemma reveals a useful property of Berge's algorithm; its proof follows from Lemma 2.5 below.

LEMMA B. *If e participates in any uppermost path then e participates in all the paths between $P_{I(e)}^B$ and $P_{L(e)}^B$.*

COROLLARY. *Let $e\in E$ and $I(e)\leqq i\leqq L(e)$ then $f_i^B(e)=|f_i^B|-|f_{I(e)-1}^B|$.*

The proof follows immediately by induction on i using Lemma B.

2.2. The modified capacity method. We propose an $O(n\log n)$ implementation of Berge's algorithm. To this end, we use modified capacities instead of residual capacities.



The capacities are depicted above the edges:

i	The uppermost path P_i^B	Residual capacity	Bottleneck	$\lvert f_i^B \rvert$
1	(s, v_1, v_2, v_3, t)	$(3, 1, 4, 3)$	$v_1 \to v_2$	1
2	$(s, v_1, v_4, v_2, v_3, t)$	$(2, 1, 3, 3, 2)$	$v_1 \to v_4$	2
3	(s, v_4, v_2, v_3, t)	$(2, 2, 2, 1)$	$v_3 \to t$	3
4	(s, v_4, t)	$(1, 2)$	$s \to v_4$	4
5	(s, v_5, t)	$(2, 2)$	$v_5 \to t$	6

FIG. 3

Let f_i^M denote the flow after finding the ith uppermost path, then the *modified capacity* is defined by $M(e) - |f_{I(e)-1}^M| + c(e)$. Note that the modified capacity of each edge receives a value once in the algorithm and is not updated (in contrast to the residual capacity which is updated in each iteration). The flow at each iteration, is not found explicitly for each edge only its value, $|f_i^M|$, is found.

ALGORITHM M.

1. Initialize: set $|f_0^M| = 0$; $P_0^M = \varnothing$; $i = 1$.
2. Find the uppermost path P_i^M, if none exists then go to 7.
3. For $e \in P_i^M - P_{i-1}^M$, set $M(e) = c(e) + |f_{i-1}^M|$.
4. Find a bottleneck $e_i^M \in P_i$. $M(e_i^M) = \mathrm{Min}_{e \in P_i^M} M(e)$; set $|f_i^M| = M(e_i^M)$.
5. Delete e_i^M from E.
6. Set $i = i + 1$ and go to 2.

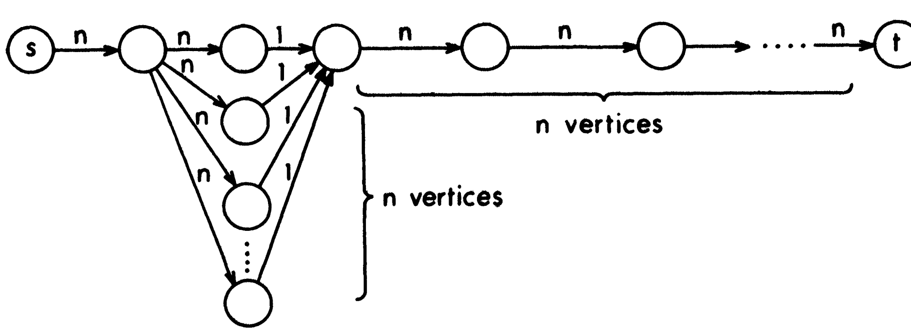
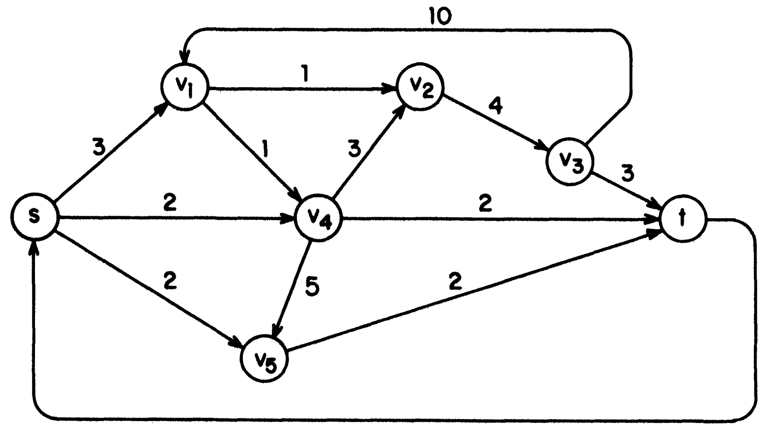


FIG. 4

7. Find the flow of each edge: set

$$f^M(e)=\begin{cases}0 & \text{if } e \text{ does not belong to any uppermost path,}\\ |f^M_{L(e)}|-|f^M_{I(e)-1}| & \text{otherwise.}\end{cases}$$

Algorithm M as applied to the network of Fig. 3 is illustrated in Fig. 5.



i	The uppermost path	Modified capacities of the path	The bottleneck	$\lvert f_i^M \rvert$
1	(s, v_1, v_2, v_3, t)	(3, 1, 4, 3)	$v_1 \to v_2$	1
2	$(s, v_1, v_4, v_2, v_3, t)$	(3, 2, 4, 4, 3)	$v_1 \to v_4$	2
3	(s, v_4, v_2, v_3, t)	(4, 4, 4, 3)	$v_3 \to t$	3
4	(s, v_4, t)	(4, 5)	$s \to v_4$	4
5	(s, v_5, t)	(6, 6)	$v_5 \to t$	6

e	$I(e)$	$L(e)$	$f(e)$
$s \to v_1$	1	2	2
$s \to v_4$	3	4	2
$s \to v_5$	5	5	2
$v_1 \to v_2$	1	1	1
$v_1 \to v_4$	2	2	1
$v_2 \to v_3$	1	3	3

e	$I(e)$	$L(e)$	$f(e)$
$v_3 \to v_1$	—	—	0
$v_3 \to t$	1	3	3
$v_4 \to v_2$	2	3	2
$v_4 \to v_5$	—	—	0
$v_4 \to t$	4	4	1
$v_5 \to t$	5	5	2
$t \to s$	—	—	0

FIG. 5

The following lemma shows that the two algorithms are equivalent.

LEMMA 2.1. *Let f_1^B be the flow found in the ith iteration of Berge's algorithm. Let $P_1^B, \cdots, P_k^B$ be the uppermost paths found in Berge's algorithm, $P_1^M, \cdots, P_l^M$ the uppermost paths found by algorithm M. If each P_i^B has a unique bottleneck e_i^B then*

i) $k = l$,

$$\left.\begin{array}{l}\text{ii) } P_i^B = P_i^M\\ \text{iii) } e_i^B = e_i^M\\ \text{iv) } f_i^B = f_i^M\end{array}\right\}\quad \text{for } i = 1, \cdots, k.$$

Proof. By induction on i. If $i = 1$ then since both P_1^B and P_1^M are the uppermost path of the same graph G, $P_1^B = P_1^M$.

At this point, for each $e \in P_1^M$, res $(e) = c(e) = M(e)$. $M(e_1^M) = \mathrm{Min}_{e \in P_1^M} M(e) = \mathrm{Min}_{e \in P_1^B}$ res $(e) =$ res (e_1^B).

Therefore, e_1^M is the unique bottleneck of P_1^B, i.e. $e_1^B = e_1^M$. Also, $|f_1^M| = M(e_1^M) =$ res $(e_1^B) = |f_1^B|$.

Suppose the lemma is valid for all $j < i$. At this stage, the graph is the same in both algorithms, since by the induction hypothesis the same bottlenecks have been deleted. Both P_i^B and P_i^M are the uppermost path of the same graph; therefore $P_i^M = P_i^B$.

For $e \in P_i^B$

$$\begin{aligned}
\text{res}\,(e) &= c(e) - f_{i-1}^B(e) \qquad \text{(from corollary to Lemma B)}\\
&= c(e) - (|f_{i-1}^B| - |f_{I(e)-1}^B|)\\
&= c(e) + |f_{I(e)-1}^M| - |f_{i-1}^B|\\
&= M(e) - |f_{i-1}^B|.
\end{aligned}$$

Since for the edges $e \in P_i^B = P_i^M$, res (e) and $M(e)$ differ only by a fixed value—$|f_{i-1}^B|$, $M(e_i^B) = \mathrm{Min}_{e \in P_1^M} M(e) = M(e_i^M)$. The equality $e_i^M = e_i^B$ follows from the hypothesis that e_i^B is the unique bottleneck of P_i^B.

Furthermore,

$$|f_1^B| = |f_{i-1}^B| + \text{res}\,(e_i^B) = |f_{i-1}^B| + (M(e_i^B) - |f_{i-1}^B|) = M(e_i^B) = M(e_i^M) = |f_i^M|.$$

Q.E.D.

If a path P_i has more than one bottleneck, Berge's algorithm does not specify which bottleneck is chosen. Therefore, for any choice of the bottlenecks in Algorithm M there is a corresponding choice in Berge's algorithm such that the sequences of paths, bottlenecks and flow values are identical in both algorithms. Since both algorithms find the same flow, and Berge's algorithms finds a maximum flow, we have:

THEOREM 2.1. *The modified capacity method* (*Algorithm* M) *finds a maximum flow.*

In order to determine the time complexity of Algorithm M, we must first specify how the uppermost paths, the bottlenecks and the indices $l(e)$ and $L(e)$ are found.

2.3. Finding uppermost paths. Let $P_{i-1} = (s = v_0, \cdots, v_r = t)$ be the $(i-1)$st uppermost path. Deleting a bottleneck $v_j \to v_{j+1}$ from P_{i-1} breaks it into two paths: P^s from s to v_j and P^t from v_{j+1} to t.

Algorithm U below constructs P_i by continuing P^s until it meets P^t (P_1 is found by connecting $P^s = (s)$ and $P^t = (t)$.) To this end, we conduct a partial depth first search from v_j until we reach a vertex of P^t.

ALGORITHM U

1. $P_i = P^s$, $v = v_j$;
2. Let $e = u \to v$ be the edge in P_i which enters v (if $v = s$ then $e = t \to s$).
3. If $E_v = \{e\}$ then (v is a deadend)
   if $v = s$ then stop (no (s, t) path exists).
   Otherwise, (backtrack) set $v = u$; delete e from G and P_i;
   go to 2. (See Fig. 6a.)
4. Let $e' = \mathrm{succ}_v(e)$. If e' enters v (e' is in the wrong direction) delete e' and go to 3. (See Fig. 6b.)
5. (In this case $e' = v \to w$.) If $w \notin P_i \cup P^t$ then include e' in P_i, set $v = w$ and go to 2. (See Fig. 6c.)

6. If $w \in P^t$ (the desired path has been found) include e' in P_i; delete the edges from v_{j+1} to w along P^t; add the remaining edges of P^t to P_i, and stop. (See Fig. 6d.)
7. ($w \in P_i$). Delete the edge e' and the edges P_i between w and v; Set $v = w$ and go to 2. (See Fig. 6e.)

Note that $I(e)$ and $L(e)$ can be found in Algorithm U as follows: Whenever an edge e is included in P_i (step 5 or 6) set $I(e) = i$. If an edge e is deleted in the ith iteration then set $L(e) = i - 1$; if e is not deleted $L(e)$ gets the index of the last uppermost path.

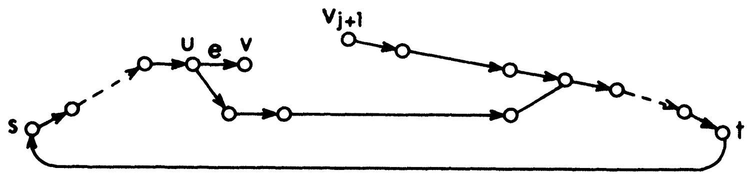


FIG. 6a

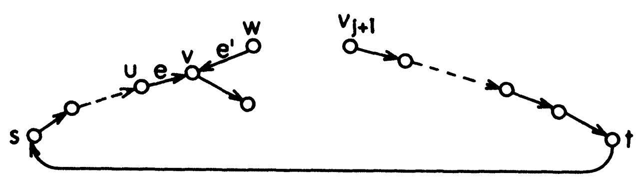


FIG. 6b

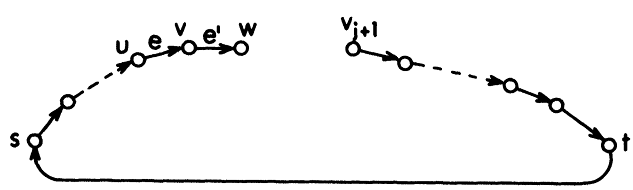


FIG. 6c

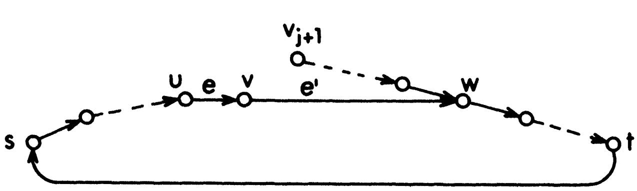


FIG. 6d

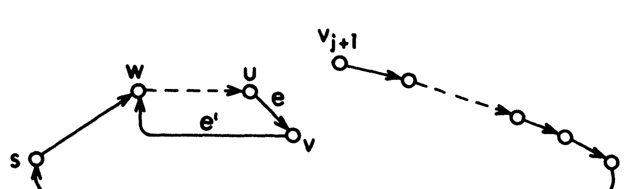


FIG. 6e

FIG. 6

2.4. A validity proof of Algorithm U. An edge e incident with the exterior face is *left-exterior* (l.e.) if it is either incident only with the exterior face, or it is incident also with another face but the exterior face is on its left hand side (see Fig. 7).

Whether an edge is l.e. depends also on the planar representation of G; we choose a particular representation in which $t \to s$ is l.e.

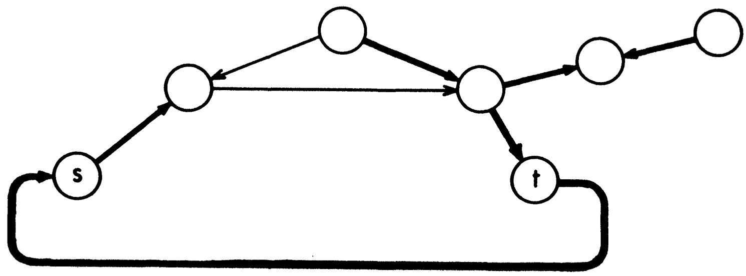


FIG. 7. *The l.e. edges appear in boldface.*

A path is l.e. if all its edges are l.e. The above definition implies the following lemma:

LEMMA 2.2. *If $u \to v$ is an l.e. edge and $v \to w = \operatorname{succ}_v(u \to v)$ then $v \to w$ is also l.e.*

The proof follows immediately from the definition of l.e.

LEMMA 2.3. *Let G_i be the graph resulting after finding the path P_i. If P^s and P^t are l.e. in G_{i-1} then P_i is l.e. in G_i.*

Proof. If $P_i = (s)$ and the edge $s \to w$ is added to P_i then $s \to w = \operatorname{succ}_s(t \to s)$ and therefore is l.e.

Assume that P_i is a nontrivial path, and $u \to v$ is its last edge. When an edge $v \to w$ is added to P_i, $v \to w = \operatorname{succ}_v(u \to v)$ and by Lemma 2.2, $v \to w$ is also l.e. The algorithm may delete edges but if an edge is l.e., then the deletion of other edges does not change this property.

The edges of P^t added to P_i (at step 6) are l.e. since P^t was l.e. in G_{i-1}. Q.E.D.

COROLLARY. *Every path P_i found by Algorithm U is l.e. in G_i.*

Proof. By induction on i. For $i = 1$, $P^s = (s)$, $P^t = (t)$ and the premise of Lemma 2.3 holds. In general, assume that P_{i-1} is l.e. Deleting the bottleneck of P_{i-1} yields $P^s, P^t \subseteq P_{i-1}$ which are also l.e. and by Lemma 2.3 P_i is also l.e. Q.E.D.

LEMMA 2.4. *If v_1, v_2, v_3 and v_4 are on the exterior face in this cyclic order then every (v_1, v_3)-path and every (v_2, v_4)-path have a common vertex.*

Proof. Assume to the contrary that P_1 and P_2 are disjoint (v_1, v_3)- and (v_2, v_4)-paths. Add a vertex v_5 in the exterior face and the edges $v_5 \to v_i$, $i = 1, \cdots, 4$. Then the resulting graph is both planar and contractible to K_5—a contradiction (see Fig. 8). Q.E.D.

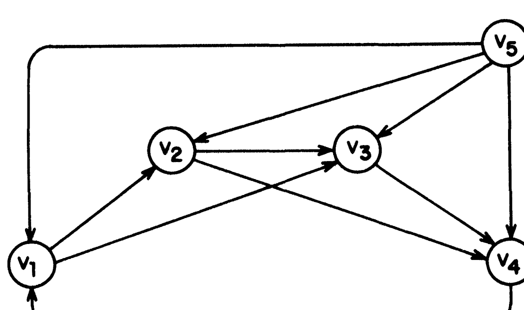


FIG. 8

LEMMA 2.5. *Eevery edge deleted by Algorithm* U *cannot participate in any subsequent uppermost path.*

Proof. Edges are deleted in four places:

i) (step 3). The vertex v is a deadend and no (s, t)-path can pass through v; therefore, $e = u \to v$ is useless (Fig. 6a).

ii) (step 4). Let $e' = w \to v$. Since $e \in P_i$, is an l.e. edge, (corollary to Lemma 2.3). $e' = \text{succ}_v(e)$ and therefore e' is incident with the exterior face. Since $t \in P_{i-1}$, s, v, w and t are on the exterior face in this cyclic order. Thus, by Lemma 2.4, every directed (s, t) path which uses $w \to v$ must cross itself. This property is not changed when edges are deleted. Therefore, any subsequent (s, t) path containing $v \to w$ is not simple and is not uppermost (Fig. 6b).

iii) (step 6). If edges are deleted in this step then $v_{j+1} \neq w$ and w is incident with three l.e. edges. Consequently, w is an articulation point separating the deleted edges from the vertices s and t, and any (s, t) path which uses any of the deleted edges is not simple (Fig. 6d).

iv) (step 7). Since the edge $v \to w$ is an l.e. edge then w is an articulation point and there is no simple (s, t)-path through any vertex x which belongs to the directed cycle closed by $v \to w$, (Figure 6e). Q.E.D.

The above lemmas yield:

THEOREM 2.2. *If there exists an (s, t)-path then Algorithm* U *finds the uppermost path.*

Proof. If there exists an (s, t)-path there exists an uppermost path. By Lemma 2.5 after deleting edges there still exists an (s, t)-path. In this case the algorithm terminates in step 6 and a path is returned. By the corollary to Lemma 2.3 this path is l.e. It is easy to see that any l.e. (s, t)-path is uppermost. Therefore, the path is the uppermost path of the resultant graph. Since by Lemma 2.5 only useless edges are deleted, this path is also the uppermost path of the initial graph. Q.E.D.

At this point we wish to make a few observations. Algorithm U finds the uppermost paths and can be used both in Berge's Algorithm and Algorithm M. The validity of Berge's Algorithm does not depend upon the method by which the uppermost paths are found. However, since by a proper choice of bottlenecks every method yields the same sequence of uppermost paths, Algorithm U may be used to prove properties of Berge's Algorithm, in particular Lemma B above.

Proof of Lemma B. It suffices to prove that if $e \in P_i$, $e \notin P_{i+1}$, then $e \notin P_j$ for $j > i$. If e is the bottleneck of P_i then it is deleted by Berge's Algorithm and cannot participate in any subsequent uppermost path. Otherwise, e is deleted by Algorithm U, and by Lemma 2.5 cannot participate in any subsequent uppermost path. Consequently, $e \notin P_j$ for $j > i$. Q.E.D.

Note that Lemma B is a property of Berge's Algorithm, not of Algorithm U. Therefore, it may be used to show the equivalence of Berge's Algorithm and Algorithm M.

2.5. Efficient implementation of Steps 5–7 of Algorithm U. To obtain an $O(n \log n)$ algorithm, Steps 5–7 must be implemented efficiently.

Step 5. In this step we should identify the new vertices (those vertices which have not appeared in P_i or any previous uppermost path). To this end, on initialization (step 1 of Algorithm M) we mark vertices s and t as **old** and all other vertices **new**. Step 5 should be:

5. If w is **new**, then: include e' in P_i,
mark w **old**, set $v = w$ and go to 2.

The paths P_i and P^t are represented as follows:
Every vertex belongs to at most one of the paths P_i or P^t. Every vertex x has one pointer field. If $x \in P_i$ then the pointer points to its predecessor in P_i; if $x \in P^t$ then it points to its predecessor in P^t.

Steps 6, 7. Here we should determine whether an **old** vertex w is in P^t or P_i. This is done by backtracking along the back pointers. If $w \in P^t$ then the backtracking from w stops when we encounter v_{j+1} and the backtracking from v stops when s is met. If $w \in P_i$ then when backtracking from w, s is encountered and when backtracking from v, w is encountered. If the backtracking is done from v and w in parallel and stopped when the first terminating condition is met, the number of edges processed is at most twice the number of edges deleted in Steps 6 and 7.

LEMMA 2.6. *The number of edge traversals in Algorithm* M (*insertions to an uppermost path, deletions from the graph and backtracking*) *is proportional to the number of edges.*

Proof. Each edge may be inserted and deleted at most once. An edge is traversed at insertion or deletion, and at backtracking. From the previous discussion, the total number of edge traversals caused by backtracking is at most twice the number of deletions, and thus it is also linear. Q.E.D.

2.6. The complexity of Algorithm M. In order to find a bottleneck efficiently, we use a priority queue. A priority queue [10] is a data structure to which we may insert or delete an element in $O(\log q)$ time (q is the number of elements in the queue), and find the minimum in constant time. We keep the modified capacities of the edges of the current P_i and P^t, in the same priority queue. Edges are inserted to the priority queue, when added to P_i in Steps 5 and 6 of Algorithm U. Whenever an edge of the graph is deleted, it is deleted also from the priority queue (provided it was there). Each edge is inserted and deleted at most once. Therefore, there may be at most m edges on the queue, and the entire deletion and insertion time is $O(m \log m) = O(n \log n)$. By Lemma 2.6 this bound also dominates the execution of the entire algorithm. Consider the graph of Fig. 9. The c_i's are the bottlenecks. In any implementation of Berge's Algorithm they are found in an increasing order. Therefore, Berge's Algorithm may be used to sort $\{c_1, \cdots, c_n\}$. Hence, Berge's Algorithm (in any implementation which uses comparisons to find the bottleneck) requires at least $O(n \log n)$ time.

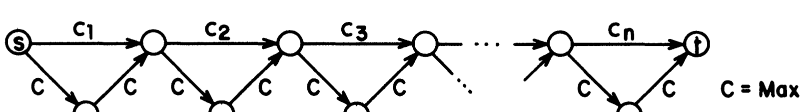


FIG. 9

3. Finding a flow in a general planar network.

3.1. Preliminaries. Let N be a general planar network (i.e. s and t are not necessarily on the same face) and let $D \in R^+$. We wish to find a flow f of value D in N. Algorithm G, described below, finds f if it exists, otherwise, the algorithm terminates indicating that there is no such flow. The algorithm requires at most $O(n^2 \log n)$ time. The Max Flow-Min Cut theorem [6] implies that such a flow exists iff $D \leqq C$—the value of a minimum cut. However, we did not find an $O(n^2 \log n)$ algorithm to determine C in a general directed planar network. In § 4 we present an $O(n^2 \log n)$ algorithm to find a minimum cut in an undirected planar network.

A function $f: E \to R^+$ is a *pseudo-flow* if it satisfies the conservation rule. Since the capacity rule is not necessarily satisfied, a pseudo-flow is not necessarily a flow. An edge

e is *over-flowed* (with respect to a pseudo-flow f) if $f(e) < c(e)$.If $e = u \to v$, then $\bar{e}$ denotes the edge $v \to u$. We make use of two conventions concerning the edges $e, \bar{e}$:

i) If $e \in E$ then also $\bar{e} \in E$ ($\bar{e}$ may be added with zero capacity).

ii) If a flow (pseudo-flow) passes through e, no flow passes through $\bar{e}$ (i.e. if $f(e) > 0$ then $f(\bar{e}) = 0$).

Let f_1, f_2 be pseudo-flows; the pseudo-flows $f_1 + f_2$ and $f_1 - f_2$ are defined by:

$$(f_1 \pm f_2)(e) = \text{Max}\,\{0, f_1(e) - f_1(\bar{e}) \pm (f_2(e) - f_2(\bar{e}))\}.$$

Therefore, if for example, $f_1(e) = 3$, $f_2(\bar{e}) = 5$, then

$$(f_1 + f_2)(e) = 0, \qquad (f_1 - f_2)(e) = 8,$$
$$(f_1 + f_2)(\bar{e}) = 2, \qquad (f_1 - f_2)(\bar{e}) = 0.$$

3.2. General planar flow algorithm. Algorithm G starts with an initial pseudo-flow the value of which is equal to D.

At each stage we pick an over-flowed edge $x \to y$ and construct a new pseudo-flow of the same value. The new pseudo-flow satisfies the capacity rule for the edges which satisfied it before, as well as for the edge $x \to y$.

ALGORITHM G.

1. Find a shortest (s, t)-path, P.
2. Let f be the pseudo-flow obtained by pushing D units of flow through P.
3. Choose an over-flowed edge $e_0 = x \to y$. If none exists stop—f is a legal flow of value D.
4. Let $N' = (G', x, y, c)$ where $G = (V, E')$, $E' = E - \{e_0, \bar{e}_0\}$

   and $$c'(e) = \begin{cases} 0 & \text{if} \quad f(e) > c(e), \\ c(e) - f(e) & \text{if} \quad c(e) \geqq f(e) > 0, \\ c(e) + f(\bar{e}) & \text{otherwise } (f(e) = 0). \end{cases}$$

   Find a flow f' in N' such that $|f'| = f(e_0) - c(e_0)$. If none exists then stop, there exists no flow of value D in N.
5. Set $f'(\bar{e}_0) = |f'|$; $f = f + f'$; go to 3.

3.3. The validity and complexity of Algorithm G. In this section we prove the following theorem:

THEOREM 3.1. *Let N be a general planar network and $D \in R^+$.*

i) *If there exists a flow of value D in N then Algorithm* G *finds one.*

ii) *If there exists no such flow then Algorithm* G *terminates indicating this fact* (*at step* 4).

iii) *Algorithm* G *requires at most $O(n^2 \log n)$ time.*

First, we show that the algorithm always terminates.

LEMMA 3.1. *Let p denote the number of edges of the path P* (*found in Step* 1), *then the number of iterations of Algorithm* G *is bounded by p.*

Proof. From the definition of c' it follows that if an edge e satisfied the capacity rule for f, then after updating f in Step 5 the rule is still satisfied, i.e.

$$\text{if } f(e) \leqq c(e) \quad \text{then} \quad (f + f')(e) \leqq c(e).$$

Moreover, after the execution of Step 5, the edge e_0 also satisfies the capacity rule ($f(e_0) \leqq c(e_0)$). Consequently, after each iteration the number of over-flowed edges strictly decreases. Since there are at most p such edges, the number of iterations is bounded by p. Q.E.D.

The proof of the theorem depends on the following lemma.

LEMMA 3.2. *If $D \leqq C$ then in Step 4 there exists a flow f' in N' of value*: $|f'| = f(e_0) - c(e_0)$.

Proof. Let f^D be a flow of value D in N. Define $f^* = f^D - f$. $f^*_{|E'}$ (f^* restricted to E') is a flow in N': Since $|f| = |f^D| = D$, f^* satisfies the conservation rule at s and t as well as for all the other vertices. The capacity rule is satisfied because of the definition of c'.

Since f^* satisfies the conservation rule at x, the value of $f^*_{|E'}$ is:

$$\begin{aligned}|f^*_{|E'}| &= f^*(\bar{e}) - f^*(e) = (f^D(\bar{e}) - f(\bar{e})) - (f^D(e) - f(e)) \\ &= f^D(\bar{e}) + f(e) - f^D(e) \geqq f(e) - f^D(e) \geqq f(e) - c(e) > 0.\end{aligned}$$

Since N' has a flow, the value of which is at least $f(e) - c(e)$, it also has a flow f' of value $f(e) - c(e)$. Q.E.D.

Proof of Theorem 3.1.

i) If $D \leqq C$ then there exists a flow of value D in N. By Lemma 3.2 the algorithm terminates at Step 3, when no over-flowed edges exist, i.e. the final f is a flow. Since throughout the algorithm the value of f is not changed, at termination, flow of value D is found.

ii) If $D > C$ then the algorithm cannot terminate in Step 3. Since by Lemma 3.1 the algorithm is finite, it terminates in Step 4, indicating that no flow of value D exists.

iii) We bound the execution time of each step.
Step 1. requires $O(m) = O(n)$ time;
Step 2. $O(p) \leqq O(n)$ time;
Step 3–5. are executed at most p times. On each iteration, Step 3 requires at most $O(1)$ time.

In Step 4 a flow f' of value $f(e) - c(e)$ is required. To find f', N' is augmented by the vertex x_s and the edge $x_s \to x$ of capacity $f(e) - c(e)$.

Let $f^{\max}$ be a maximum flow from x_s to y. If $|f^{\max}| = f(e) - c(e)$ then the desired flow is $f^{\max}$ restricted to E'. Otherwise, $|f^{\max}| < f(e) < c(e)$, there exists no flow f', and the algorithm immediately terminates.

In N', x and y are on the same face. Hence, there exists a planar representation of the augmented network, in which x_s and y are also on the same face. Therefore, we may use Algorithm M to find $f^{\max}$ in $O(n \log n)$ time. Consequently, Step 4 requires $O(n^2 \log n)$ time.

Hence, the complexity of Algorithm G is $O(pn \log n) \leqq O(n^2 \log n)$. Q.E.D.

Note that in some cases a shorter initial path can be found by adding edges of zero capacity.

4. Finding a minimum (s, t) cut in an undirected planar network. In this section we present an $O(n^2 \log n)$ algorithm for finding a minimum (s, t)-cut in an undirected planar network.

Henceforth, we assume that G is triconnected. Otherwise, the graph may be triangulated in linear time using zero capacity edges. (Every triangulated planar graph with more than three vertices is triconnected.) The value of a minimum (s, t)-cut obviously does not change by this process. The minimum cut of the original graph consists of the original edges which participate in a minimum cut of the new graph.

Since G is triconnected, it has a unique dual $G^d = (X, A)$, [11, Chap. 3]. G^d is also triconnected. Let F and Φ denote the set of faces of G and G^d respectively. There exists a 1–1 correspondence between the elements of $V \leftrightarrow \Phi$, $E \leftrightarrow A$ and $F \leftrightarrow X$ (see Fig. 10). Let $\alpha^d \in E$ denote the dual of $\alpha \in A$. The length of an edge $\alpha \in A$ is defined by:

$$l(\alpha) = c(\alpha^d).$$

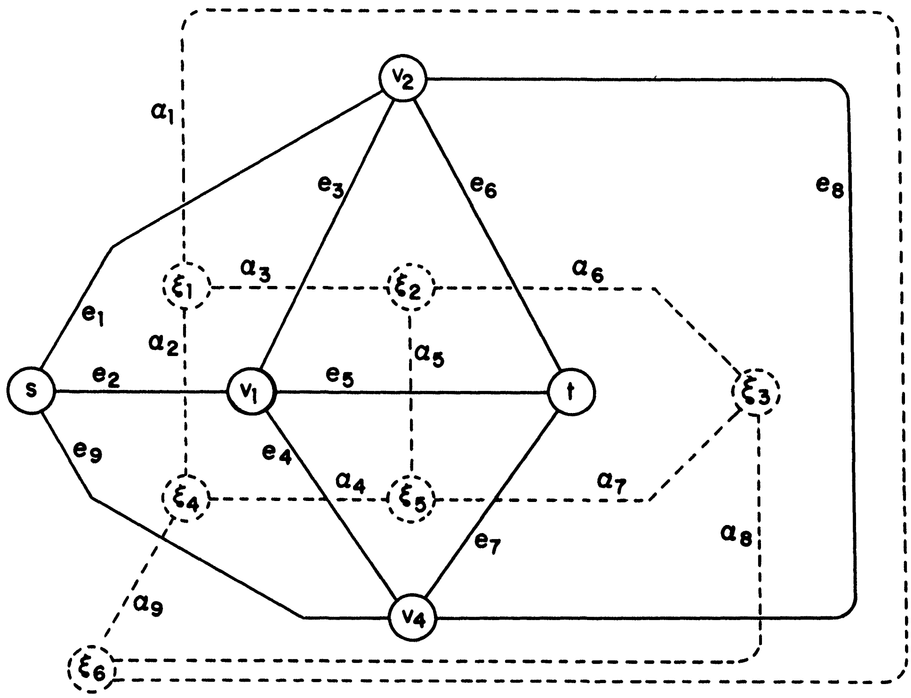


FIG. 10

Let φ_s and φ_t denote the faces in G^d which correspond to s and t respectively. Henceforth, we assume that φ_s is the exterior face of G^d. The following lemma is intuitive; however its formal proof is tedious, and therefore, omitted.

LEMMA 4.1. *If C is a minimum (s, t) cut then $C^d = \{\alpha \mid \alpha^d \in C\}$ is a cycle of minimum length enclosing φ_t.*

Let $\xi^s \in \varphi_s$, $\xi^t \in \varphi_t$ and let $\Pi = (\xi^s = \xi_1, \cdots, \xi_k = \xi^t)$ be a shortest (ξ^s, ξ^t)-path in G^d. Let $\alpha_i = \xi_{i-1} - \xi_i$ for $i = 2, \cdots, k$.

Let A_Π denote the set of all edges of G^d which have exactly one endpoint on Π. An edge $\xi - \xi_i \in A$ is Π-*left* if it precedes α_{i+1} in the linear order around ξ_i induced by α_i. (See § 1.3.) The edge $\xi - \xi_i$ is Π-*right* if it succeeds, α_{i+1} in this order. Two vertices ξ_0, ξ_{k+1} and two edges $\alpha_0 = \xi_0 - \xi_1$ and $\alpha_{k+1} = \xi_k - \xi_{k+1}$ are added to G^d (see Fig. 11) to make this definition meaningful also for the edges which are incident with $\xi^s = \xi_1$ and $\xi_k = \xi^t$.

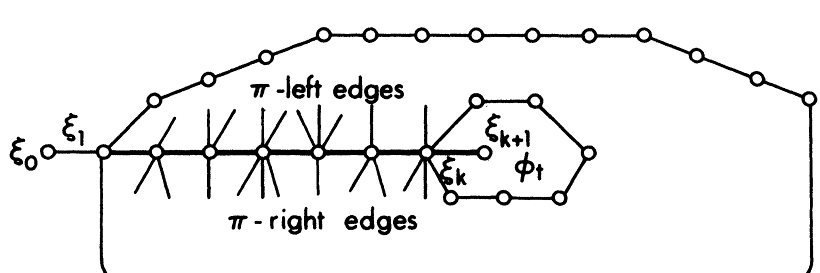


FIG. 11

Note that since G^d is triconnected no edge is both Π-left and Π-right. A *ξ_i-cycle* is a simple cycle which uses exactly one Π-left and one Π-right edge and its Π-left edge is incident with ξ_i, (see Fig. 12).

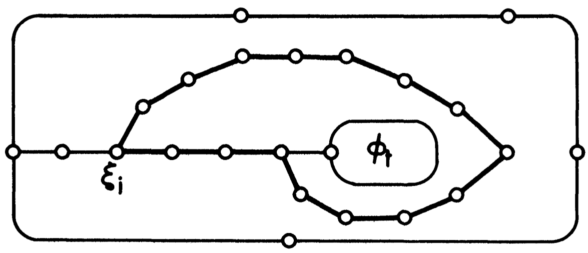


FIG. 12

It is easy to see that every ξ_i-cycle $(i=1,\cdots,k)$ encloses φ_t.

LEMMA 4.2. *Let C be a shortest cycle enclosing φ_t. Then there exists a ξ_i-cycle of the same length.*

Proof. The proof follows immediately from the fact that Π is a shortest (ξ^s, ξ^t) path and therefore a subpath of Π between ξ_i and any ξ_j is a shortest (ξ_i, ξ_j) path. Moreover, every cycle enclosing φ_t must intersect Π. Q.E.D.

(This argument does not work in directed graphs.)

The previous lemma implies that in order to find a minimum cycle enclosing φ_t we may find for each $i=1,\cdots,k$ a minimum ξ_i cycle. The shortest of these k cycles is a minimum cycle enclosing φ_t. In order to find minimum ξ_i-cycle we use the following construction.

Let $\vec{G}^d$ be the directed graph obtained from G^d in the following manner: Every Π-left edge $\xi_i - \eta$ is directed from ξ_i to η. Every Π-right edge $\xi_i - \eta$ is directed from η to ξ_i. All the other edges $\xi - \eta$ are replaced by two edges $\xi \to \eta$ and $\eta \to \xi$.

LEMMA 4.3. *Let $\xi_i \in \Pi$. If ξ_i is a shortest simple nontrivial directed path from ξ_i to itself in $\vec{G}^d$, then the corresponding undirected edges in G^d form a shortest ξ_i-cycle.*

Proof. It can be easily verified from the definition of $\vec{G}^d$ that if a directed path from ξ_i to itself uses more than one Π-left edge or more than one Π-right edge, then it crosses itself and therefore it is not a shortest ξ_i-cycle. Q.E.D.

Finding a minimum ξ_i-path for a given i is therefore equivalent to finding a shortest nontrivial (ξ_i, ξ_i)-path in $\vec{G}^d$. This can be done in $O(m \log n) = O(n \log n)$ time and therefore the entire algorithm requires at most $O(n^2 \log n)$ time.

5. Conclusions. We have presented an $O(n \log n)$ algorithm to find a maximum flow in an (s, t) planar network. The algorithm was programmed and compared on (s, t) planar networks with Berge's and Dinic's algorithms. On networks which exhibit Dinic's $O(n^3)$ behavior, the special purpose algorithms (Berge's and ours) were superior.

The tests were also conducted on random data. Since it was unclear how random (s, t) planar graphs can be algorithmically constructed, the algorithm was tested on several (s, t) planar graph with random capacities. For these networks the results were less clear cut. The performance of our algorithm and Dinic's were about the same; however there were differences on different networks. Berge's algorithm was superior to both.

This behavior is explained by two observations:

i) The number of augmenting paths found by Dinic's algorithm was much less than the upper bound.

ii) The priority queue involves considerable overhead.

In the general case the value of a maximum flow is equal to that of the minimum cut. We have presented an $O(n^2 \log n)$ algorithm to find the minimum cut in an undirected planar network. Using this algorithm and Algorithm G a maximum flow in an undirected planar network may be found in $O(n^2 \log n)$ time.

We have not found an $O(n^2 \log n)$ method to find the value of the minimum cut for the directed case. However, since Algorithm G indicates whether D is less than or equal to the value of the maximum flow, if the capacities are integers it may be used to find the maximum flow. However, this method requires $\log \sum_{e \in E} c(e)$ iterations of Algorithm G, and hence its complexity is a function of the size of the capacities, as well as the number of vertices. Nevertheless, if the capacities are all small integers the method is superior to the existing algorithms.

Appendix. A validity proof of Berge's algorithm. Let f be a flow in $N = (G, s, t, c)$, $G = (V, E)$; then the graph G_f is defined by:

$$G_f = (V, E_f), \qquad E_f = \{e \colon e \in E \text{ and } f(e) > 0\}.$$

Let $P = (s = v_0, \cdots, v_k = t)$ be the uppermost path of G, and $e_h = v_h \to v_{h+1}$ for $h = 0, \cdots, k-1$. Let f be a maximum flow such that

$$\text{(A.1)} \qquad \sum_{h=1}^{k} f(e_h) \geqq \sum_{h=1}^{k} f'(e_h) \quad \text{for any maximum flow } f'.$$

LEMMA A.1. *Let e^B be the bottleneck of P then $f(e_h) \geqq c(e^B)$, $(h = 0, \cdots, k-1)$.*

Proof. Assume to the contrary that r is the first index such that $f(e_r) < c(e^B)$. Then

$$\text{(A.2)} \qquad f(e_h) < c(e^B) \quad \text{for } h = r, r+1, \cdots, k-1.$$

We prove (A.2) by induction on h. By hypothesis it is true for $h = r$. Assume it holds for $h = r, r+1, \cdots, j-1$.

If $f(e_j) \geqq c(e^B)$ then $f(e_j) > f(e_r)$ and therefore there exists an (s, v_j) path P_1 in G_f, which does not pass through e_r. Since OUT $(f, v_r) > f(e_r)$ there exists a (v_r, t) path P_2 in G_f, which does not pass through e_r. By Lemma 2.4, P_1 crosses P_2; let x be their common vertex (see Fig. 13).

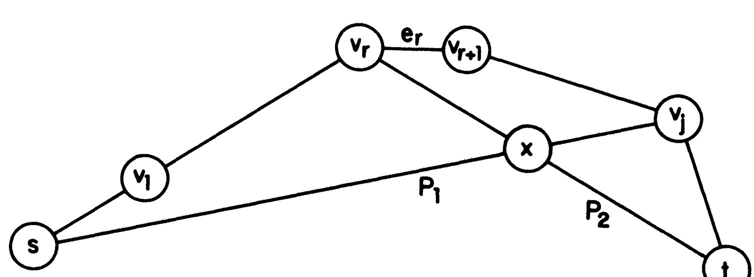


FIG. 13.

Let P_3 be the path in G_f constructed from the subpath of P_2 from v_r to x and the subpath of P_1 from x to v_j. P_3 is a (v_r, v_j) path in G_f. Let P' denote the subpath of P from v_r to v_j. The edge e_r belongs to P' but not to P_3, therefore, $P' \neq P_3$. By the induction hypothesis the edges of P' are not saturated. Thus, we may divert flow from P_3 to P'. The resultant flow f' violates (A.1), this completing the proof of (A.2).

To complete the proof of the lemma, let P_2 be a (v_r, t)-path in G_f; then by diverting flow from P_2 to P, (A.1) is violated. Q.E.D.

THEOREM A.1. *Berge's algorithm finds a maximum flow.*

Proof. By induction on the number of edges:

i) The claim is obvious if the network contains only one edge.

ii) For $m>1$ edges, let e^B be the bottleneck of P, define the flow network $\bar{N}=(G, s, t, \bar{c})$ as follows:

$$\bar{c}(e)=\begin{cases} c(e) & \text{if } e\in E-P,\\ c(e)-c(e^B) & \text{if } e\in P.\end{cases}$$

Let

$$\bar{f}(e)=\begin{cases} f(e) & \text{if } e\in E-P,\\ f(e)-c(e^B) & \text{if } e\in P.\end{cases}$$

By Lemma A.1 $\bar{f}(e)\geqq 0\ \forall e\in P$, and therefore $\bar{f}$ is a legal flow. Obviously, $\bar{f}$ is a maximum flow in $\bar{N}$ and $|\bar{f}|=|f|-c(e^B)$.

In Berge's algorithm we push $c(e^B)$ units of flow through P and then apply the same process on the resultant network—$\bar{N}$ which has at least one edge (e^B) less than N. By the induction hypothesis—the algorithm, applied to $\bar{N}$ finds maximum flow of value $|f|-c(e^B)$.

Consequently, the algorithm applied to N finds a flow of value $(|f|-c(e^B))+c(e^B)=|f|$. That is, Berge's algorithm finds a maximum flow. Q.E.D.

Note added in proof. It was brought to our attention by Professor T. C. Hu that what we call "Berge's Algorithm" was originated by L. R. Ford and D. R. Fulkerson in their paper *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

REFERENCES

[1] A. E. BARATZ, *Construction and analysis of network flow problem which forces Karzanov algorithm to $O(N^3)$ running time*, MIT Laboratory for Computer Science Report MIT/LCS/TM-83, Mass. Inst. of Tech., Cambridge, 1977.

[2] C. BERGE AND A. GHOUILA-HOURI, *Programming, Games and Transportation Networks*, Methuen, Agincourt, Ontario.

[3] E. A. DINIC, *Algorithm for solution of a problem of maximal flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[4] S. EVEN AND R. TARJAN, *Network flow and testing graph connectivity*, this Journal, 4 (1975), pp. 507–518.

[5] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[6] C. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[7] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[8] T. C. HU, *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

[9] A. V. KARZANOV, *Determining the maximal flow in a network by the method of the preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[10] D. E. KNUTH, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA, 1973.

[11] O. ORE, *The Four Color Problem*, Academic Press, New York, 1967.