SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1002–1017, October 1995

FLOW IN PLANAR GRAPHS WITH MULTIPLE SOURCES AND SINKS*

GARY L. MILLER† AND JOSEPH (SEFFI) NAOR‡

Abstract. The problem of maximum flow in planar graphs has always been investigated under the assumption that there is only one source and one sink. Here we consider the case where there are many sources and sinks (single commodity) in a directed planar graph. An algorithm for the case when the demands of the sources and sinks are fixed and given in advance is presented. The algorithm can be implemented efficiently sequentially and in parallel, and its complexity is dominated by the complexity of computing all shortest paths from a single source in a planar graph. If the demands are not known, an algorithm for computing the maximum flow is presented for the case where the number of faces that contain sources and sinks is bounded by a slowly growing function. Our result places the problem of computing a perfect matching in a planar bipartite graph in NC and improves a previous parallel algorithm for the case of a single source, single sink in a planar directed (and undirected) graph, both in terms of processor bounds and its simple presentation.

Key words. planar graphs, flow, circulation

AMS subject classifications. 68R10, 05C38, 90B10, 90C35, 90C27

1. Introduction.

In the common formulation of the maximum flow problem, the maximum flow from a distinguished vertex in the graph, called the *source*, to another distinguished vertex in the graph, called the *sink*, is computed. Here we assume that the underlying network is planar; this case was extensively studied and more efficient algorithms were developed for it (see §2), yet the assumption was always that there is only one source and one sink.

In this paper we investigate the following problem: given a planar network with *many* sources and sinks, compute the maximum flow from the sources to the sinks. Ford and Fulkerson [3] reduced the multiple source, multiple sink problem to the single source, single sink problem by connecting the sources to a supersource and the sinks to a supersink, and then computing the maximum flow from the supersource to the supersink. In planar graphs, this reduction may *destroy* the planarity of the graph if the sources or sinks belong to different faces. Nevertheless, we would like to take advantage of the planarity of the graph to design more efficient algorithms, *sequential as well as parallel*, in the case of multiple sources and sinks.

We feel that the reformulation of the problem is more natural within the context of planar graphs and has motivation in both sequential and parallel computation. The only other attempt known to us that copes with multiple sources and sinks is by Megiddo [22], [23], whose algorithm computes (in a general graph) optimal flows, i.e., flows that are "fairly" distributed among the sources and sinks.

Maximum flow in a general network was shown to be P-complete [10], and hence it is widely believed not to have an efficient parallel algorithm. On the other hand, maximum flow can be reduced to maximum matching and this reduction implies an RNC algorithm when the edge capacities are represented in unary [16], [24]. This emphasizes the importance of solving the problem in the case of a planar network with arbitrary capacities. In the restricted case

*Received by the editors March 2, 1989; accepted for publication (in revised form) April 7, 1994. An extended abstract describing these results appeared in the Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 112–117.

†School of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213-3890.

‡Computer Science Department, Technion, Haifa 32000, Israel. This research was supported by the fund for the promotion of research at the Technion, Office of Naval Research contract ONR N00014-88-K-0166 and the Computer Science Department, University of Southern California, Los Angeles, California and supported in part by National Science Foundation grant DCR-8713489. Most of this work was done while the author was at the Computer Science Department, Stanford University, Stanford, California.

of a single source, single sink, there do exist NC algorithms in both directed and undirected graphs [6], [14].

The first problem we consider is the case where the amount of flow (demand) at each source and sink is given as input, and the objective is to compute a feasible flow function. We present an efficient algorithm for this problem. The sequential complexity of our algorithm is $O(n^{1.5})$ time; the parallel complexity is $O(I(n)\log^2 n)$ time using $O(n^{1.5})$ processors where $I(n)$ is the time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the concurrent read concurrent write parallel random access machine (CRCW PRAM) model and $O(\log n)$ time in the exclusive read exclusive write parallel random access machine (EREW PRAM) model. The main idea in computing the flow function in this case is redirecting the flow through a spanning tree from the sinks back to the sources. The problem then reduces to that of computing a circulation in a network with both lower and upper bounds on the capacity of the edges. Similar ideas for redirecting the flow back from the sink to the source have appeared in [5], [6], and [14] for computing the flow in the case of a single source and sink.

Note that planar graphs are different from general graphs, where it was observed that knowing the value of the maximum flow does not improve the complexity of computing the flow function [29]. In contrast to that, all the known algorithms for planar flow do take advantage of the value of the maximum flow.

We consider the special case where the sources and sinks are all on one face and the demands unknown. We present an efficient algorithm that computes the maximum flow in this case by employing a nontrivial divide-and-conquer. The sequential running time of the algorithm is $O(n\log^{1.5} n)$ time. In parallel, it can be computed using $O(n^{1.5})$ processors, where the time complexity is $O(I(n)\log^3 n)$. ($I(n)$ is the same function as previously defined.) We then show how to extend this algorithm to the case where the number of faces that contain sources and sinks is bounded by a slowly growing function.

Unfortunately, the most general problem, where the sources and sinks belong to an arbitrary number of faces and the demands are unknown, is still open; i.e., the best sequential algorithm for this problem is obtained by connecting the sources and sinks to a supersource and supersink; in parallel, there is no NC algorithm known for this problem.

An example where multiple sources and sinks are useful is the case of computing a perfect matching of a planar bipartite graph. In the standard reduction from matching to flow (see, e.g., [2]), one part of the graph is connected to a source and the other part to a sink. In general, this reduction will result in a nonplanar graph but can be utilized within our context (the demand of each source and sink is exactly one unit). This places the problem of computing a perfect matching in a planar bipartite graph in NC.

The situation in computing a perfect matching in planar graphs is very intriguing. Kasteleyn [17] had already shown how to *count* the number of perfect matchings in a planar graph, a problem that is # P-complete in general graphs, and his methods can be implemented in NC (see, e.g., [32]) as well. Yet computing a perfect matching in NC in a planar graph remains an open problem. This situation is interesting as it contradicts the current view of the computational difficulty of counting the number of solutions versus finding a solution in combinatorial problems.

Johnson [14] showed how to compute in parallel the maximum flow for the case of a single source, single sink in a directed graph. We present an algorithm for this case which improves on the number of processors and is also very *simple* in comparison with the fairly complicated algorithm of [14]. Johnson's approach [14] was the first to find the minimum cut, and then compute the flow function. Our approach is different; using parametric methods, we can find the *value* of the maximum flow; then the computation of the flow function follows easily.

Subsequent to this work, Khuller and Naor [18] considered the problem of computing a flow function in a planar graph where there are capacity constraints on both edges and vertices. Efficient algorithms for this problem are presented in [18].

The paper is organized as follows: In §2, we describe previous results in planar flow and in §3 we provide certain preliminaries. In §4, we show how to compute a circulation when edge capacities may have nonzero lower bounds. In §5, we present three applications: (i) computing the flow function when there are many sources and sinks and their demands are known; (ii) computing a perfect matching in a bipartite planar graph; (iii) improving previous algorithms for the case of a single source and sink. In §6, we show how to find the maximum flow in the case where all the sources and sinks are on the same face but the demands are not known. In §7 we extend these results to the case where the number of faces containing sources and sinks is bounded.

2. Previous results in planar flow. All the results referred to in this section deal *exclusively* with the single source, single sink maximum flow problem. Ford and Fulkerson [3] had already observed that a minimum cut in a planar graph is equivalent to a minimum weight cycle that separates the source from the sink in the dual graph. They gave an $O(n \log n)$ time algorithm to compute the minimum cut when the source and sink belong to the same face. Berge and Ghouila-Houri [1] suggested an $O(n^2)$ algorithm for computing the flow function, which is called the "uppermost path algorithm." This algorithm was implemented in $O(n \log n)$ time by Itai and Shiloach [12]. Hassin [5] gave an elegant algorithm for computing the flow function and his algorithm can be implemented in $O(n\sqrt{\log n})$ time using the method of [4] for computing shortest paths in planar graphs.

Itai and Shiloach [12] also gave an algorithm to compute the maximum flow in an undirected graph when the source and sink do not necessarily belong to the same face. Its running time was $O(n^2 \log n)$. This was improved by Reif [30] who gave an $O(n \log^2 n)$ time algorithm for computing the minimum cut in an undirected planar graph. Only Hassin and Johnson [6] completed the picture by giving an $O(n \log^2 n)$ time algorithm for computing the maximum flow in an undirected graph by generalizing the ideas of [5] and [30]. (The running time of their algorithm can be improved to $O(n \log n)$ through the methods of [4] for computing shortest paths in a planar graph.)

Computing the maximum flow in planar directed graphs is more difficult as it is not clear how to reduce the problem of computing a minimum weight cycle to that of computing a minimum weight path. Johnson and Venkatesan [15] gave an $O(n^{1.5} \log n)$ time algorithm to compute both a minimum cut and a maximal flow.

In the course of the evolution of efficient algorithms for planar flow, an interesting phenomenon occurred. The computational difficulty alternated between searching for the minimum cut on the one hand, and computing the flow function when the minimum cut is known on the other hand.

It is easy to implement the algorithm of [6] for undirected graphs in parallel, and its complexity is $O(\log^2 n)$ time using $O(n^3)$ processors. (The details are given in [14].) It should be mentioned that an alternative algorithm for computing the minimum cut in parallel in an undirected graph was given in [13].

As for directed planar graphs, an algorithm that first computes the minimum cut, and then the flow function, was given by Johnson [14]. Its complexity is $O(\log^3 n)$ time using $O(n^4)$ processors or $O(\log^2 n)$ time using $O(n^6)$ processors. The processor bounds of the algorithms of [6], [14] can be improved through the methods of [26]–[28].

3. Terminology and preliminaries. Throughout the paper let $G = (V, E)$ be an embedded planar graph where V is the *vertex set* and E is the *arc set*. An *edge* consists of two

arcs, an arc and its reflection. Let $R(e)$ be the permutation which takes an arc e to its reflection. The graphs considered may have multiple edges but every arc will have a distinct reflection.

A graph is said to be embedded in the plane if it is intuitively "drawn" in the plane with no crossing edges, where an edge and its reflection are drawn on top of each other. This definition is not very algorithmic, hence we assume that the graph is given by one of the many combinatorial definitions of planar embedding; see [25], for example. An embedding of a planar graph can be computed sequentially in linear time [11] and in parallel in $O(\log^2 n)$ time using a linear number of processors [19]. An embedding is needed to compute the dual graph. In all of our algorithms, computing an embedding will never dominate the cost of the algorithm.

An embedded graph G partitions the plane into connected regions called *faces*. Let $G^* = (F, E^*)$ be the *dual graph* of G, where F is the set of faces of G, and E^* is the set of dual arcs. There is a one-to-one correspondence between E and E^* as follows: for each arc $e \in E$, let e^* be the corresponding *dual arc* connecting the right face bordering e with the left face bordering on e.

We use a left-hand rule: if the thumb points in the direction of e, then the index finger points in the direction of e^*. The dual G^* is also known as the *clockwise dual* of G. For a vertex v, in(v) (out(v)) denotes the incoming (outgoing) set of arcs into (from) v. For an arc e, tail(e) (head(e)) denotes the vertex at the tail (head) of e.

The dual graph is planar too, but may contain self loops and multiple edges. We sometimes refer to the graph G as the *primal* graph.

A *cycle* C is an ordered set of arcs $e_0, e_1, \ldots, e_k$ such that for every $0 \le i \le k$, head(e_i) = tail(e_{i+1}) (mod $k+1$). The cycle C is simple if the vertices between the arcs are distinct. Thus all cycles are *directed*.

3.1. Flow in planar graphs. In this section we formally define the planar flow problem. We generalize the problem in two ways. First, we allow multiple sources and sinks. Second, we introduce vertices with fixed flow demands. This is the definition we shall use throughout the rest of this paper.

DEFINITION 3.1. *A flow graph with sources, sinks, capacities, and demands is the following five-tuple* (G, S, T, c, d) *such that*

- $G = (V, E)$ *is a graph, where* V *is a set of vertices and* E *is a set of arcs;*
- *the sources and sinks with variable demands are* $S \subseteq V$ *and* $T \subseteq V$*, respectively, where the sets* S *and* T *are disjoint;*
- *the map* $c : E \to \Re$ *is the edge capacity (for all* $e \in E$*:* $c(e) = -c(R(e))$*);*
- *the map* $d : V - \{S, T\} \to \Re$ *is the demand at nonsource and nonsink vertices, i.e., vertices for which the demand is fixed.*

Observe, that $c(e)$ and $d(v)$ may be negative. Vertices for which $d(v) > 0$ are called sources with fixed demands, and vertices for which $d(v) < 0$ are called sinks with fixed demands.

DEFINITION 3.2. *A function* $f : E \to \Re$ *is a flow function if*

(i) $\forall e \in E : f(e) = -f(R(e))$;

(ii) $\forall v \in V - \{S, T\} : \sum_{\text{tail}(e)=v} f(e) = d(v)$.

DEFINITION 3.3. *The function* f *is a legal (or feasible) flow function if, in addition,* $\forall e \in E$, $f(e) \le c(e)$*, where* $c(e)$ *denotes the capacity of edge* e.

Given a flow function f, we define for all $v \in V$, $f(v) = \sum_{\text{tail}(e)=v} f(e)$.

In the maximum flow problem, we are looking for a legal flow that maximizes the *total* amount of flow entering T (or leaving S). The amount of flow entering the sinks is also called the *value* of the flow. A *circulation graph* is a flow graph with no sinks or sources, i.e., $S = T = \emptyset$ and $d(v) = 0$ for all vertices. A *circulation function* is a flow function with

respect to a circulation graph. A flow graph with *fixed demands* is a flow graph with no sinks or sources that have variable demand, i.e., $S = T = \emptyset$.

Flow graphs and flow functions can be added and subtracted as follows: given a flow graph G, flow functions f_1 and f_2, and a real number α, the sum $f = \alpha f_1 \pm f_2$ is the flow where, for every edge e, $f(e) = \alpha f_1(e) \pm f_2(e)$. We define the addition of a flow graph and a flow function to obtain a new flow graph. Let (G, c, d) be a flow graph with capacity c and demands d, and let f be a flow function defined on G. The flow graph $G + f$ will be the triple $(G, c + f, d + f)$. The demand function $d + f$ is only defined for vertices v for which d is defined and $(d + f)(v) = d(v) + f(v)$. The *residual graph* with respect to a given flow f is formally defined as $G - f$. Observe that the residual graph with respect to a legal flow function cannot have negative capacities.

An *augmenting path* in a flow graph G is a path that starts at a source, ends at a sink, and uses only arcs with strictly positive capacity. We do not distinguish between an augmenting path as a set of arcs or, alternatively, as a flow of one unit on the arcs in the augmenting path. We say this more formally. A path P is *edge disjoint* if the arcs in P are distinct and no arc and its reflection both belong to P. Let P be an edge disjoint path from a source to a sink. The *unit flow* on P is defined as follows:

$$P(e) = \begin{cases} 1 & \text{if } e \in P, \\ -1 & \text{if } e \in R(P), \\ 0 & \text{otherwise.} \end{cases}$$

A *potential graph* is a graph where each arc is assigned a weight $w \in \Re$. A *potential function* on a potential graph $G = (V, E)$ is any real valued function p defined on the vertices. The function p is called *consistent* if the potential difference over each edge is not larger than the edge weight, i.e., $\forall e \in E, \quad w(e) \geq p(\text{head}(e)) - p(\text{tail}(e))$.

If $G = (V, E)$ is an embedded planar flow graph then its *potential dual graph* is $G^* = (F, E^*)$ such that $w(e*) = c(e)$. Given a consistent potential function p defined on G^*, we obtain a flow function f by setting $f(e) = p(\text{head}(e^*)) - p(\text{tail}(e^*))$. Clearly, f satisfies condition (i) of Definition 3.2. To see that it also satisfies condition (ii) in the case where all the demands are zero, we use the following easy yet fundamental proposition proved in [5] and [14].

PROPOSITION 3.1. *Let $C = e_1^*, \ldots, e_k^*$ be a cycle in the dual graph. Then the flow f satisfies the following property: $f(e_1) + \cdots + f(e_k) = 0$.*

Let the cycle C in the dual graph correspond to the set of primal edges adjacent to a primal vertex. Then it follows that f satisfies condition (ii) of Definition 3.2. Hence, any potential function induces a circulation in a planar flow graph. Furthermore, if the potential function is consistent then the flow function is legal.

The use of a potential function as a mean of computing a flow (and circulation) was first suggested by Hassin [5] and was later elaborated by [6] and [14], but not stated in terms of circulations.

An important procedure that will be used in all of our algorithms is computing all shortest paths from a given vertex in a planar graph which may contain negative edge weights. The sequential complexity of this procedure is $O(n^{1.5})$ using the generalized nested dissection method of [20]. To compute in parallel shortest distances, the parallel nested dissection implementation of [26]–[28] requires $O(n^{1.5})$ processors; the time complexity is $O(I(n) \log^2 n)$, where $I(n)$ is the parallel time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the CRCW PRAM model and $O(\log n)$ time in the EREW PRAM model.

To implement the method of nested dissection we need to compute small separators in planar graphs. A small separator can be computed sequentially in linear time; see [25]. In parallel, Gazit and Miller [7], [8] provided a procedure for computing small separators, where

the complexity is $O(n^{1+\epsilon})$ processors and the running time is dominated by the running time of a procedure for computing a maximal independent set in a graph (not necessarily planar). The current best bound is $O(\log^3 n)$ time using a linear number of processors [9].

4. Computing circulations. In this section we show how to compute a circulation in a planar graph. It is clear that the difficult case occurs when some of the edges have negative capacity. Otherwise, we can set the flow on each edge to be zero and obtain a legal circulation. An edge which has negative capacity may be viewed as having a lower bound on the flow through it; for example, an edge which has a lower bound of a and an upper bound of b on the capacity can be replaced by two edges of opposite direction, which have capacities b and $-a$. We provide a precise characterization of planar circulation problems that have feasible solutions.

As stated in the previous section, the key idea is to compute a consistent potential function on the faces of the planar graph and define the flow in each edge as the potential difference of the two faces that border the edge. We show that finding such a potential function reduces to computing the shortest paths from a single source. Throughout this section, let G denote a planar circulation graph with capacities $c(e)$ and a dual G^* which has weights $w(e)$.

A *shortest path* numbering for a graph (from some source vertex x) is an assignment of real values to the vertices such that the value of a vertex y equals the minimum weight path from x to y. We shall simply call it an SP numbering. It is well known that every strongly connected graph has an SP numbering if and only if it has no negative weight cycles.

LEMMA 4.1. *Let G be a connected, embedded circulation graph. Then, the following conditions are equivalent:*

1. *G^* has no negative weight cycles.*
2. *G^* has an SP numbering from every vertex (face in G).*
3. *G has a legal circulation.*

Proof. By the comment above, if G^* does not contain negative weight cycles, then G^* has an SP numbering from every vertex. To see that the existence of an SP numbering also implies the existence of a circulation, let the potentials assigned to the faces be equal to their SP numbering from some arbitrary vertex. Let p denote the potential function and e^* be a dual edge directed from face F to face F'. Since p is an SP numbering, it follows that

$$p(F') - p(F) \leq w(e^*).$$

Therefore, p is a consistent potential function and G has a legal circulation by Proposition 3.1.

To see that if G has a legal circulation, then G^* has no negative weight cycles, assume the contrary: G^* contains a negative weight cycle $e_1^*, \ldots, e_k^*$ yet G has a legal circulation f. Note that a cycle in the dual graph G^* separates the plane into two regions. Hence the incoming flow into it has to be equal to the outgoing flow, i.e., $f(e_1) + \cdots + f(e_k) = 0$, but $f(e_1) + \cdots + f(e_k) \leq c(e_1) + \cdots + c(e_k) < 0$, resulting in a contradiction. □

This lemma gives a precise characterization for the existence of a feasible circulation in a planar graph: the dual graph cannot contain negative weight cycles.

To summarize, the potential function is computed as follows: Choose an arbitrary vertex in the dual graph and compute the shortest path from it to all other vertices. The length of the shortest path is defined as the potential of the vertex. The flow on each edge is defined as the potential difference of the two faces bordering it.

The cost of computing a legal circulation is dominated by the cost of computing one SP numbering. The sequential and parallel complexity of computing an SP numbering are given in §3.1. Thus, we have the following theorem.

THEOREM 4.1. *A legal circulation can be computed in a planar graph in $O(n^{1.5})$ time sequentially. In parallel it can be computed using $O(n^{1.5})$ processors; the time complexity is $O(I(n)\log^2 n)$, where $I(n)$ is the parallel time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the CRCW PRAM model and $O(\log n)$ time in the EREW PRAM model.*

5. Applications of the circulation algorithm. In this section we present three applications of the circulation algorithm. In §5.1 we show how to compute a feasible flow function in a flow graph without variable sources and sinks, i.e., when all demands are fixed and known in advance. In §5.2 we show that a perfect matching in a planar bipartite graph can be computed in NC. In §5.3 we present a simple algorithm for computing the maximum flow for the case of a single source and sink with variable demand.

5.1. Computing the flow function for fixed demands. In this section we assume that our input is a planar graph with many sources and sinks, where the demand at each source and sink is known in advance. We present an efficient algorithm that determines whether a feasible solution, i.e., a solution that satisfies the demands, exists, and if so computes it. Suppose that a demand function d is defined on the vertices (see §3.1) such that sources have positive demands and sinks have negative demands. The rest of the vertices have zero demand. We may assume that the sum of the demands is zero since there is no feasible solution otherwise.

The main idea of the algorithm is that computing a flow function with fixed demands can be reduced to computing a circulation. This reduction is achieved via a tree T that spans the sources and sinks. We add new edges to the graph parallel to the edges of T, resulting in a new graph G'.

Recall that in the algorithms of [6] and [14], a similar reduction is achieved by returning the flow from the sink to the source via a simple path. This idea is generalized here and the spanning tree T is used to redirect the flow from the sinks back to the sources. Any circulation computed in G' will induce a flow satisfying the demands in G.

SKETCH OF ALGORITHM (I).
Input: a planar flow graph (G, c, d) where the demand function is defined for all vertices of G.
Output: a legal flow function f that satisfies the demands.

1. Find any flow function f' for G.
2. Construct the residual flow graph $G' = G - f'$ where G' is a circulation graph.
3. Compute a circulation f'' in G'.
4. Return the flow $f = f' + f''$.

We now elaborate on the implementation of each step. To do step 1, we first compute a spanning tree T in G, where an edge in T consists of both an arc and its reflection. We now describe how the spanning tree is used to redirect the flow from the sinks back to the sources.

For all arcs that are not in T, we set f' to zero. An arc $e \in T$ separates the tree into two subtrees, called tail and head. T_{tail} is the subtree adjacent to the tail of e and T_{head} is adjacent to the head of e. Let $f'(e)$ be defined as $\sum_{v \in T_{\text{tail}}} d(v)$. The last term is also equal to $-\sum_{v \in T_{\text{head}}} d(v)$ since T is a spanning tree and the sum of the demands on all vertices in the graph is zero. To see that $f'(v) = d(v)$, let E' be the set of arcs whose head is v. Note that if an arc $e \in E'$ is not in T, then $f'(e) = 0$.

$$f'(v) = \sum_{e \in E'} f'(e) = \sum_{w \in V - \{v\}} -d(w) = d(v).$$

We need only show that $f' + f''$ is a legal flow which meets the demands, given that f'' is a circulation. The flow $f' + f''$ meets the demand since

$$(f' + f'')(v) = f'(v) + f''(v) = f'(v) + 0 = d(v).$$

To see that it is legal,

$$(f' + f'')(e) = f'(e) + f''(e) \leq f'(e) + (c - f')(e) = c(e).$$

The most expensive part of the algorithm is step 3, where all shortest paths from a vertex in the dual graph are computed. The sequential and parallel complexity of computing all shortest paths from a given vertex in a graph with negative edge weights is discussed in §3.1. Thus, we have the following theorem.

THEOREM 5.1. *A flow function with fixed demands can be computed in a planar graph in $O(n^{1.5})$ time sequentially. In parallel, it can be computed using $O(n^{1.5})$ processors; the time complexity is $O(I(n)\log^2 n)$, where $I(n)$ is the parallel time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the CRCW PRAM model and in $O(\log n)$ time in the EREW PRAM model.*

5.2. Finding a perfect matching. In this section we show how to compute a perfect matching in a planar bipartite graph $G = (A, B, E)$, where A and B are the two parts of the vertex set. In the standard reduction from matching to flow (see, e.g., [2]), E is directed from A to B; a source s is connected to all the vertices of A, and a sink t is connected to all the vertices of B. All the edges in the reduced graph have unit capacity; the saturated edges in a maximum flow constitute a maximum matching in G. Obviously, this reduction may, in general, destroy the planarity of the graph.

To compute the perfect matching efficiently, each vertex in A becomes a source and each vertex in B becomes a sink. The demand at each source and sink is exactly one unit. The edges are oriented as before from A to B with unit capacity.

The sequential complexity of our algorithm is $O(n^{1.5})$ time and it matches the best sequential bound for computing a maximum matching in a planar graph [21]. In parallel, our result places in NC the problem of computing a perfect matching in a planar bipartite graph.

5.3. Planar maximum flow with a single source and sink. In this section we show how to improve the algorithm of [14] in the case of a single source, single sink in a directed planar graph. We present a simple algorithm for this problem and improve the processor bound with respect to [14]. We also handle the case where the capacities are possibly negative and the case where some of the vertices may have fixed demands by returning flow as in §5.1, Algorithm (I).

The approach taken in [14] is first to find the minimum cut and then compute the flow function. We proceed differently; to apply Algorithm (I), we need to compute the *value* of the minimum cut, denoted by α. This will be done by a parametric search method. Once the value of the minimum cut is known, the flow function can be computed by Algorithm (I).

Recall that in Algorithm (I) the flow is returned from the sinks to the sources via a spanning tree. Note that in the case of a single source and sink, the spanning tree is a simple path, denoted by P, from t to s. The difficulty is that we do not know how much flow to return from t to s. We first guess an initial value α, which is greater than or equal to the value of the maximum flow from s to t; e.g., initially, we set α equal to the sum of the capacities of the edges leaving the source. Throughout the rest of this section, we let f_α denote the flow, which is α for arcs in P, $-\alpha$ for arcs in $R(P)$, and zero otherwise. Let $G' = G - f_\alpha$.

The characterization of feasible circulations in Lemma 4.1 implies that we should compute the maximum α (if it exists at all) such that the dual graph of G' does not contain negative

weight cycles. If our initial guess was too large, then in the dual graph of G', where the shortest paths are computed, there must be a negative weight cycle. (Otherwise, a circulation that would correspond to a flow whose value is α can be computed.) It should be noted that there may be negative cycles in the dual graph that exist independently of the value we set for α. This can only happen if there is no feasible flow in G even if α is set to zero. If this case this is detected we halt, returning with no flow. We can henceforth assume that there exists a legal flow function for G for some positive value of α.

Let C be a simple cycle in the dual graph G^*. The *net crossings* of the path P by the cycle C is defined as being equal to the number of times C crosses P from right to left minus the number of crossings from left to right. (Right and left are defined with respect to the partition of the plane by the path P.)

We observe the following straightforward fact.

LEMMA 5.1. *The net crossings of P by C is either 0 or ± 1.*

Proof. Assume the cycle C crosses the path P more than once. We prove that two consecutive crossings on P alternate between left-right crossings and right-left ones. Suppose, to the contrary, that edges $e, e' \in C$ are two consecutive crossings that are oriented in the same direction, where e' is to the right of e with respect to their orientation. The cycle C partitions the plane into two regions, interior and exterior, and without loss of generality let the interior be the region left of the cycle C with respect to its orientation. Now, however, there are two points in the plane, one point to the right of e (in the exterior) and the other to the left of e' (in the interior), that can be connected without crossing the cycle C. Hence, there can be at most one crossing that is not "canceled" and the correctness of the lemma follows. □

The following definitions hold for both sequential and parallel algorithms. An algorithm that computes shortest paths in a graph is called *oblivious* if any decision on which paths in the graph to compare is independent of the weights of the edges. In particular, for a fixed (unweighted) graph, an oblivious algorithm will always compare the same paths for any assignment of weights to the edges. An algorithm that computes shortest paths in a graph is called *additive* if, for any nonsimple cycle C, its weight is computed in a time step subsequent to the time steps in which the weight of each of its components that form simple cycles is computed.

We first describe a generic algorithm for finding the maximum feasible α under the assumption that an (sequential or parallel) algorithm $\mathcal{A}$ for computing shortest paths in a graph that is both oblivious and additive is available.

GENERIC ALGORITHM.
Input: a planar flow graph (G, c, d) where the demand function is a variable for precisely one source and one sink; an oblivious and additive algorithm $\mathcal{A}$ for computing shortest paths in G.
Output: the maximum legal flow.

1. Find any flow f' for G; Set $G \leftarrow G - f'$. (Flow f' satisfies the fixed demands in the graph.)
2. Find a simple path P from s to t and construct flow f_α for some large constant α.
3. Set $G_\alpha \leftarrow G - f_\alpha$.
4. Test whether G_α^* has negative cycles. Run Algorithm $\mathcal{A}$ on G_α^*:
   (a) Let τ be the first step in which a negative cycle is detected (in Algorithm $\mathcal{A}$) and let l be the weight of the most negative cycle detected at Step τ.
   (b) Set $\alpha \leftarrow \alpha + l$; restart Algorithm $\mathcal{A}$, i.e., goto Step 3.

THEOREM 5.2. *Assume that the running time of Algorithm $\mathcal{A}$ is $O(T)$. Then, the generic algorithm terminates after at most $O(T^2)$ steps and computes the maximum feasible value of α.*

Proof. Let τ denote the first step of Algorithm $\mathcal{A}$ in which a negative cycle is detected. We will prove that after updating the value of α and restarting Algorithm $\mathcal{A}$, a negative cycle can be detected only in time steps subsequent to τ. Hence, the generic algorithm terminates after at most $\sum_{\tau=1}^{T} \tau$ steps, which is at most $O(T^2)$ steps.

Let $\mathcal{C}$ denote the set of cycles in which Algorithm $\mathcal{A}$ has computed their weight up to Step τ. We claim that after updating the value of α, the weight of all cycles in the set $\mathcal{C}$ must be nonnegative. Since Algorithm $\mathcal{A}$ is oblivious, it follows that in any subsequent iteration, a negative cycle cannot be detected before Step $\tau + 1$.

To prove the claim, suppose to the contrary that in time step $\tau' \leq \tau$, a negative weight cycle $C \in \mathcal{C}$ is discovered in $G^*_{\alpha+l}$. (Assume that τ' is the first step in which a negative cycle is detected.) We first prove that C must be a simple cycle. If C is not a simple cycle, then it can be decomposed into a set of simple cycles. By the additivity property of Algorithm $\mathcal{A}$, this set of simple cycles must belong to $\mathcal{C}$. Since the weight of C is the sum of the weights of the simple cycles in its decomposition, some of the simple cycles must have negative weight. Again, by the additivity property, the weight of these simple cycles will be computed in a time step τ'', where $\tau'' < \tau'$. Therefore, we can assume that C is a simple cycle.

Recall that for every negative weight simple cycle in $\mathcal{C}$, the net number of crossings of P must be 0 or ± 1. Therefore, the weight of cycle C in G^*_α cannot be less than l, implying that all the simple cycles in $G^*_{\alpha+l}$ belonging to $\mathcal{C}$ cannot be of negative weight. In step 4(b) the value of α can only decrease, and hence the weight of the cycles in $\mathcal{C}$ will remain nonnegative.

It remains to prove that the algorithm computes the maximum α. Let $\alpha_{\max}$ denote the final value of α computed by the algorithm and let C be the most negative cycle detected by the algorithm in the last step in which a negative cycle was detected. It is easy to see that in any graph $G_{\alpha'}$ (where $\alpha' > \alpha_{\max}$) the weight of cycle C must be negative. □

5.3.1. Implementing the generic algorithm. The most efficient way of implementing the generic algorithm would be by using the nested dissection method of [20] and its implementation in parallel by Pan and Reif [26]–[28]. We provide a high level description of it.

The basic idea underlying the method of nested dissection is partitioning the graph by a family of separators, where each separator partitions the graph into equal size components (up to constant factors). Each separator is a cycle, and and we refer to the two components of the graph generated by the cycle as being "inside" and "outside" the separator. Hence we can think of the separators as forming a binary tree as follows: the root of the tree is the graph G; the descendants of each vertex in the tree are the two components generated by the separator. The shortest distances in the graph are computed bottom up in the tree.

Pan and Reif obtained their best time bounds [28] by streamlining the computation. Intuitively, this can be thought of as if vertices in a certain level in the tree of separators begin computing their transitive closure before the vertices in the levels below them have finished computing their transitive closure.

For us, the important feature of shortest path algorithms based on nested dissection is that they are oblivious and additive.

THEOREM 5.3. *The maximum flow in a directed planar graph with a single source, single sink can be computed in $O(\log^4 n)$ time using $O(n^{1.5})$ processors in the CRCW model.*

Proof. The proof follows from the discussion. □

The sequential running time was shown in [15] to be $O(n^{1.5} \log n)$.

Another oblivious and additive algorithm for computing shortest paths is via matrix multiplication and doubling-up. The computation of the shortest paths proceeds in this case by successive squaring of the adjacency matrix A of G^* until we get A^n. Let k be the first iteration in which a negative entry appears in the diagonal of A^{2^k} and let l be the most negative entry of the diagonal in that iteration. We update α by l, i.e., setting $G' \leftarrow G - f_{\alpha+l}$, and start the

computation of the shortest paths from the beginning. It follows from the proof of Theorem 5.2 that A^{2^t} will not have negative entries any more in its diagonal. Hence, at most $\log n$ computations of the shortest paths algorithm suffice to compute the value of the minimum cut.

6. Maximum flow on the disk. In this section we describe an algorithm for computing a maximum flow for the case where all the sources and sinks with variable demands lie on the same face. Without loss of generality, one can assume that the sources and sinks are on the outer face and that they alternate, i.e., there are no two consecutive sources or sinks. These two properties will be maintained during the recursive calls to the algorithm. We discuss two cases. In the first case we will assume that G has no negative capacities and no vertices with nonzero demands, i.e., the zero flow is a legal flow. In the second case, we allow negative capacities and nonzero demands and we show how to reduce this case to the first case.

6.1. Maximum flow on the disk with positive capacities and zero demands. Let G be a flow graph and f be a maximum flow on G. Among all minimum cuts separating the sources from the sinks, we are interested in the "first" minimum cut, defined as follows: Let W be the set of vertices that are reachable from the sources in $G - f$. The *Ford–Fulkerson cut* with respect to f is the set of edges between W and $V - W$. Suppose the sources and sinks of the outer face are separated into two consecutive sets L and R; the maximum flow f from L to R is defined as the flow that maximizes the flow from the sources in L to the sinks in R. In particular, we can set the demand of each sink in L and each source in R to zero.

The main idea of the algorithm is the following: Divide the vertices of the outer face into two (aforementioned) sets and compute the maximum flow from L to R. The Ford–Fulkerson cut associated with this flow decomposes the disk into regions and in each region the maximum flow is computed recursively. In the last step of the algorithm, the maximum flow is computed from R to L. We prove that when the algorithm terminates, the flow cannot be augmented.

Assume that after the flow is computed from L to R, the edges of the Ford–Fulkerson cut are removed from the graph in the recursive calls.

Let V_s denote the sources and sinks belonging to a set of vertices V. Let C denote the set of connected components of $G - f$ after the edges of the Ford–Fulkerson cut are deleted.

We are now ready to present an outline of the algorithm for computing the maximum flow.

SKETCH OF ALGORITHM (II).
Input: a planar flow graph (G, S, T, c, d) where the demand function is zero for all vertices of G, the capacities are all positive, and the sources and sinks are on the outer face.
Output: the maximum flow function f in G.
If G has at most one source or one sink then return the zero flow.
Else:

1. Divide the sources and sinks into two consecutive sets, L and R, such that $|L_s| = |R_s|$ and L contains at least as many sources as R.
2. Compute a maximum flow f_{LR} from L to R. Compute the residual graph $G' = G - f_{LR}$.
3. Delete the edges of the Ford–Fulkerson cut from G' and compute C, the connected components of G'; recursively, compute the maximum flow for each component $c \in C$. The bottom of the recursion is when a component c contains a unique source and no sinks, or vice versa. Let $f_{L\&R}$ be the sum of the flows computed for each component.
4. Compute the residual graph $G'' = G - f_{LR} - f_{L\&R}$. Compute the maximum flow f_{RL} from R to L in G''.
5. Return the flow $f_{LR} + f_{L\&R} + f_{RL}$.

We now elaborate on the steps of the algorithm. In step 2 connect all the sources in L to a new vertex, a supersource, and all the sinks in R to a new vertex, a supersink. This can be done without destroying the invariant that all the sources and sinks of G lie on the outer face, since we have set the demand of the sinks in L and the sources in R to zero. The problem then reduces to computing a maximum flow in an $\{s, t\}$ planar graph, a graph in which both the source and the sink are on the same face [3], [12], [5]. Observe that all recursive calls are for graphs with nonnegative capacities and zero demands.

In step 3 the maximum flow in each $c \in C$ is recursively computed. The capacities of the edges in c are the residual capacities with respect to the flow computed in step 2. We now recursively compute the maximum flow inside c. If a connected component contains vertices from both L_s and R_s, then there can be only two cases: (i) sinks from L with sources and sinks from R; (ii) sources from R with sinks and sources from L. In the recursive call, in the first case we connect all the sinks that belong to L to a supersink, and in the second case we connect all the sources that belong to R to a supersource. This is done to ensure that the number of sources and sinks decreases by a constant factor in each recursive call. Observe that the flows computed in each component are disjoint and hence the sum is a legal flow.

In step 4 we compute the maximum flow from R to L similar to step 2.

We now prove the correctness of the algorithm.

We first need a technical fact about writing a flow as a sum of augmenting paths which are viewed as flows. Let G be a flow graph with sources and sinks, nonnegative capacities, and zero demands. Furthermore, let f be a legal flow for G. A sum of augmenting paths $\alpha_1 f_1 + \cdots + \alpha_k f_k = f$ is a *positive decomposition* of f if

(i) $\alpha_i > 0$ for $1 \le i \le k$;

(ii) the arcs e and $R(e)$ cannot both belong to any of the paths $f_1, \ldots, f_k$;

(iii) each path starts at a source and ends at a sink.

LEMMA 6.1. *Let G and f be as above. Then there exists another legal flow f' on G which has a positive decomposition into augmenting paths and agrees with f on the sinks and sources of G.*

Proof. We pick a source s in G such that $f(s) > 0$. Starting from s and ending at some sink, we pick a path P of arcs such that the flow in f on each arc is positive. Let α_1 be the minimum flow on any arc in P. Set $f = f - \alpha_1 f_1$ and observe that f is still a legal flow. We continue in this greedy fashion until all sinks have zero flow. Let $f_1, \ldots, f_k$ be the constructed augmenting paths. At this point we set $f' = \alpha_1 f_1 + \cdots + \alpha_k f_k$ and discard the remaining flow in f. □

THEOREM 6.1. *Algorithm* (II) *correctly computes a maximum flow.*

Proof. The proof is by induction on the number of sinks and sources in G. If G has only one source or sink, then its flow must be zero since the demand at all other vertices is assumed to be zero. Since Algorithm (II) returns zero in this case, we may assume inductively that at the end of step 4, a maximum flow was computed in each connected component $c \in C$. That is, there is no augmenting path contained in c for the flow graph $G - f_{LR} - f_{L\&R}$. In the time analysis we shall bound more closely the number of sinks and sources in each recursive call, but it is clear that each call has strictly fewer sources and sinks.

To prove the theorem, we have to show that there are no augmenting paths from any source s to any sink t in $G - f_{LR} - f_{L\&R} - f_{RL}$. By Lemma 6.1 we can replace the flow f_{RL} with another legal flow f'_{RL} such that it can be written as the positive sum $\alpha_1 f_1 + \cdots + \alpha_k f_k$. It will suffice to show that there are no augmenting paths with respect to the flow $G - f_{LR} - f_{L\&R} - f'_{RL}$. Note that all the paths f_i $(1 \le i \le k)$ are augmenting paths with respect to the flow $G - f_{LR} - f_{L\&R}$. Furthermore, for any edge e belonging to the Ford–Fulkerson cut, only $R(e)$ may belong to an augmenting path f_i. Hence an augmenting path f_i can only leave a connected component but not enter a connected component.

Now suppose that A is an augmenting path from a source s to a sink t in $G - f_{LR} - f_{L\&R} - f'_{RL}$. There are four cases, depending on whether s is in L or R and t is in L and R, which we denote by LL, LR, RR, and RL. We show that the existence of A results in a contradiction in each case.

We know that A cannot be a type RL since step 4 computed a maximum flow from right to left.

Suppose that A is of type LL. We claim that one of the arcs of A must have zero residual capacity in $G - f_{LR} - f_{L\&R}$. If A contains an arc of the Ford–Fulkerson cut, the claim is clearly true. If not, then A is contained in one of the connected components c from step 3. Since step 3 returns a maximum flow for c (by the induction hypothesis), the path A cannot be augmenting for $G - f_{LR} - f_{L\&R}$ and, therefore, the arc must exist. Let e be the first such arc on A. Since e has residual capacity in $G - f_{LR} - f_{L\&R} - f'_{RL}$, it must be the case that one of the augmenting paths f_i from the positive decomposition of f'_{RL} contains the arc $R(e)$. Consider the following path A' that consists of the arcs of A up to but not including e plus the arcs in f_i following but not including $R(e)$. By the observation above, the arcs in f_i following $R(e)$ must belong to c. Thus, the path A' is an augmenting path for $G - f_{LR} - f_{L\&R}$, resulting in a contradiction. Therefore, type LL augmenting paths do not exist.

We handle the last two cases, LR and RR, together. As in case LL, some arc on A must have zero capacity in $G - f_{LR} - f_{L\&R}$. In this case let e be the last arc on A with zero capacity. Furthermore, let f_i be an augmenting path containing $R(e)$. We construct an augmenting path A' from arcs on f_i before $R(e)$ and arcs on A that follow e. It follows that A' is an augmenting path for $G - f_{LR} - f_{L\&R}$, again a contradiction. □

THEOREM 6.2. *The running time of Algorithm* (II) *is* $O(n \log^{1.5} n)$ *sequentially. In parallel, it can be computed using* $O(n^{1.5})$ *processors; the time complexity is* $O(I(n) \log^3 n)$, *where* $I(n)$ *is the parallel time of computing the sum of* n *values.* $I(n)$ *can be implemented in* $O(1)$ *time in the CRCW PRAM model and in* $O(\log n)$ *time in the EREW PRAM model.*

Proof. In steps 2 and 5 we compute the maximum flow in an $\{s, t\}$-planar graph. This can be done by Hassin's algorithm [5], and its time complexity is $O(n\sqrt{\log n})$ sequentially [5]. In parallel, Hassin's algorithm can be implemented in $O(I(n) \log^2 n)$ time and $O(n^{1.5})$ processors by using the shortest path procedure outlined in §3.1. Since all the recursive calls at a given level of the recursion are on vertex disjoint subgraphs, we will only need $O(n^{1.5})$ processors for the full algorithm.

Observe that if G has $2k$ sinks and sources, then each connected component will have at most $k + 1$ sinks and sources for k odd and k sinks and sources for n even. It follows that the number of alternations of sources and sinks in each connected component of C is reduced by a constant fraction. Note that in step 3, at most one source or one sink is added instead of the edges of the Ford–Fulkerson cut.

Let $T_n(a)$ and $P_n(a)$ denote the time and number of processors, respectively, and let n and a denote the number of vertices and alternations, respectively. For the parallel running time we get the following recursive formula:

$$T_n(a) \leq T_n(a/2 + 1) + O(I(n) \log^2 n),$$

and hence the running time of the algorithm is $O(I(n) \log^3 n)$. As already mentioned, the number of processors then needed is $O(n^{1.5})$.

For the sequential running time we get the following recursive formula:

$$T_n(a) \leq 2T_n(a/2 + 1) + O(n\sqrt{\log n}),$$

and hence the sequential running time of the algorithm is $O(n \log^{1.5} n)$. □

6.2. Maximum flow on disk with negative capacities. In this section we show how to reduce the maximum flow problem on a disk to the special case needed in §6.1. There are two conditions that must be met in order to apply Algorithm (II):

- Vertices must have zero demand.
- Edges must have nonnegative capacities.

The reduction is quite straightforward and consists of the following steps: First, choose any flow f that meets the demands. Then, find a maximum legal flow f' in $G - f$ and return the flow $f + f'$.

The first step is implemented very similarly to algorithm (I): the flow is returned from the sinks to the sources via a spanning tree. Now, all vertices which are not variable sources or sinks have demand equal to zero. However, we may still have negative capacities. To reduce the case of negative capacities to the case of nonnegative capacities we need only find any flow. We introduce a supersource-sink vertex s and connect every source and sink to s. We also set the demand at s and at every source and sink to zero. This gives a planar flow graph G, which is in fact a circulation graph. Thus, the second step above will actually consist of two substeps: first, find any flow f'' in $G + f$ using the reduction to the circulation problem; then, find a maximum flow f' in $G + f - f''$ using Algorithm (II).

7. Maximum flow for a bounded number of faces containing sources and sinks. There are several extensions of our work. As previously mentioned, efficiently computing (sequentially and in parallel) a maximum flow in a planar graph with many sources and sinks with variable demands is still open. However, we observe that we can provide efficient sequential and parallel algorithms for the case where the sources and sinks belong to a fixed or slowly growing number of faces. As in §6, we discuss the special case when all capacities are nonnegative and all demands are zero.

First observe that the following greedy algorithm computes a maximum flow with many sources and sinks. Suppose that G is any flow graph with sources $s_1, \ldots, s_k$ and sinks $t_1, \ldots, t_l$. We claim that the following algorithm finds a maximum flow.

SKETCH OF ALGORITHM (III).
Input: a planar flow graph (G, S, T, c, d) where the demand function is 0 for all vertices and all capacities are positive.
Output: a maximum flow function f.

1. Set $f = 0$
2. **For** $1 \le i \le k$ and $1 \le j \le l$ **do**
   (a) Set the demand at all s_u and all t_v for $u \ne i$ and $v \ne j$ to zero and find a maximum flow f_{ij} from s_i to t_j in $G - f$.
   (b) **Set** $f = f + f_{ij}$.

   Return f

LEMMA 7.1. *Algorithm* (III) *computes a maximum flow.*

We next observe that the proof of correctness of Algorithm (II) is actually independent of the following: (i) the fact that the sources and sinks are all on one face; (ii) the planarity of the underlying graph. This implies that the following generic algorithm computes a maximum flow in a graph G (not necessarily planar) with many sources and sinks:

1. Partition the sources and sinks into two disjoint sets L and R.
2. Compute the maximum flow from L to R, i.e., from the sources in L to the sinks in R. Let C denote the Ford–Fulkerson minimum cut with respect to the maximum flow.
3. Remove the edges of C from the graph G. Recursively compute a maximum flow in each connected component (in the residual graph).
4. Compute a maximum flow from R to L (in the residual graph).

Let G be a planar graph with variable sources and sinks that lie on at most k faces of G, denoted by $F_1, \ldots F_k$. We now show how to combine Algorithm (III) with the generic algorithm to efficiently compute a maximum flow in G.

Step 1 in the generic algorithm is implemented as follows: the sources and sinks on face F_i $(1 \leq i \leq k)$ are partitioned into two sets, L_i and R_i, in the same way that the sources and sinks on the disk are partitioned in Algorithm (II). The set L is defined as the union of the sets L_i $(1 \leq i \leq k)$ and the set R is defined as the union of the sets R_i $(1 \leq i \leq k)$.

To implement step 2, first connect the sources in each set L_i to a supersource s_i, and the sinks in each set R_i to supersink t_i. This operation does not violate the planarity of the graph. The maximum flow from L to R is computed by Algorithm (III). Computing the flow from source s_i to sink t_j in Algorithm (III) is an instance of the problem of computing the maximum flow in a directed planar graph with one source and one sink. Step 4 is implemented similarly.

Note that in step 3, the number of alternations of sources and sinks is reduced by a constant factor for each face F_i $(1 \leq i \leq k)$.

Hence we get recursive formulas for the running time which are similar to those obtained in the proof of Theorem 6.2. For the parallel running time we get

$$T_n(a) \leq T_n(a/2+1) + O(k^2 I^2(n) \log^4 n),$$

and hence the running time of the algorithm is $O(k^2 log^5 n)$ in the CRCW model. The number of processors needed is $O(n^{1.5})$.

For the sequential running time we get

$$T_n(a) \leq 2T_n(a/2+1) + O(k^2 n^{1.5} \log n),$$

and hence the sequential running time of the algorithm is $O(k^2 n^{1.5} \log^2 n)$.

We conclude with the following theorem.

THEOREM 7.1. *If G is a planar flow graph with variable sources and sinks that lie on at most k faces of G, then a maximum flow for G can be computed sequentially in $O(k^2 n^{1.5} \log^2 n)$ time. In parallel, the running time is $O(k^2 \log^5 n)$ using $O(n^{1.5})$ processors in the CRCW model.*

REFERENCES

[1] C. BERGE AND A. GHOUILA-HOURI, *Programming, Games and Transportation Networks*, John Wiley, New York, 1965.

[2] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, Maryland, 1979.

[3] L. R. FORD AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

[4] G. N. FREDERICKSON, *Fast algorithms for shortest paths in planar graphs with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[5] R. HASSIN, *Maximum flows in (s, t) planar networks*, Inform. Process. Lett., 13 (1981), p. 107.

[6] R. HASSIN AND D. B. JOHNSON, *An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks*, SIAM J. Comput., 14 (1985), pp. 612–624.

[7] H. GAZIT AND G. L. MILLER, *A parallel algorithm for finding a separator in planar graphs*, Proc. 28th Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 238–248.

[8] ———, *A deterministic parallel algorithm for finding a separator in planar graphs*, manuscript.

[9] M. GOLDBERG AND T. SPENCER, *Constructing a maximal independent set in parallel*, SIAM J. Discrete Math., 2 (1989), pp. 322–328.

[10] L. GOLDSCHLAGER, R. SHAW, AND J. STAPLES, *The maximum flow problem is log space complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[11] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[12] A. ITAI AND Y. SHILOACH, *Maximum flow in planar networks*, SIAM J. Comput., 8 (1979), pp. 135–150.

[13] L. JANIGA AND V. KOUBEK, *A note on finding cuts in directed planar networks by parallel computation*, Inform. Process. Lett., 21 (1985), pp. 75–78.

[14] D. B. JOHNSON, *Parallel algorithms for minimum cuts and maximum flows in planar networks*, J. Assoc. Comput. Mach., 34 (1987), pp. 950–967.

[15] D. B. JOHNSON AND S. VENKATESAN, *Using divide and conquer to find flows in directed planar networks in $O(n^{1.5}\log n)$ time*, Proc. 20th Allerton Conference on Communication, Control and Computing, University of Illinois, Urbana-Champaign, Urbana, IL, 1982, pp. 898–905.

[16] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[17] P. W. KASTELEYN, *Graph theory and crystal physics*, in Graph Theory and Theoretical Physics, F. Harary, ed., Academic Press, New York, 1967, pp. 43–110.

[18] S. KHULLER AND J. NAOR, *Flow in planar graphs with vertex capacities*, Algorithmica, 11 (1994), pp. 200–225.

[19] P. N. KLEIN AND J. H. REIF, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., 37 (1988), pp. 190–246.

[20] R. J. LIPTON, D. J. ROSE, AND R. E. TARJAN, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[21] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, SIAM J. Comput., 9 (1980), pp. 615–627.

[22] N. MEGIDDO, *Optimal flows in networks with multiple sources and sinks*, Math. Programming, 7 (1974), pp. 97–107.

[23] ———, *A good algorithm for lexicographically optimal flows in multi-terminal networks*, Bull. Amer. Math. Soc., 83 (1977), pp. 407–409.

[24] K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[25] T. NISHIZEKI AND N. CHIBA, *Planar graphs: Theory and algorithms*, in Ann. Discrete Math., Vol. 32, North–Holland Mathematical Studies, 1988.

[26] V. PAN AND J. H. REIF, *Fast and efficient parallel solution of sparse linear systems*, Tech. report 88–19, Computer Science Department, State University of New York at Albany, Albany, New York, 1988.

[27] ———, *Fast and efficient solution of path algebra problems*, J. Comput. System Sci., 38 (1980), pp. 494–510.

[28] ———, *The parallel computation of minimum cost paths in graphs by stream contraction*, Inform. Process. Lett., 40 (1991), pp. 79–83.

[29] V. RAMACHANDRAN, *Flow value, minimum cuts and maximum flows*, unpublished manuscript.

[30] J. H. REIF, *Minimum $s-t$ cut of a planar undirected network in $O(n\log^2 n)$ time*, SIAM J. Comput., 12 (1983), pp. 71–81.

[31] Y. SHILOACH AND U. VISHKIN, *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[32] V. V. VAZIRANI, *NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems*, Inform. and Comput., 80 (1989), pp. 152–164.