Flow in Planar Graphs with Multiple Sources and Sinks *

(Extended Abstract)

Gary L. Miller † *Joseph Naor* ‡

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 section 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 [FF] reduced the multiple source, multiple sink problem to the single source, single sink problem by connecting the sources to a super source, the sinks to a super sink, and then computing the maximum flow from the super source to the super sink. 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 for the case of multiple sources and sinks.

We feel that this reformulation of the problem is more natural within the context of planar graphs, and has motivation in both sequential and parallel computation. Previous planar flow algorithms dealt exclusively with the single source, single sink case. Megiddo [Me1, Me2] gave an algorithm in the case of multiple sources and sinks (in a general graph) to compute optimal flows, namely to distribute the flow "fairly" among the sources and sinks. This is the only other attempt known to us that copes with multiple sources and sinks.

Maximum flow in a general network was shown to be P-complete [GSS] 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 gives an RNC algorithm when the edge capacities are represented in unary [KUW,MVV]. This emphasizes the importance of solving the problem in the case of a planar network with arbitrary capacities. In the restricted case of a single source, single sink, there do exist NC algorithms in both directed and undirected graphs [HJ, Jo].

We assume that we get as input the amount of flow (demand) at each source and sink and compute the flow function. Our algorithm runs in $O(\log^2 n)$ time using $O(n^{1.5})$ processors on an EREW PRAM. Unfortunately, when the demands are unknown, the problem still remains open. However, in the special case that the sources and sinks are all on one face (and the demands unknown), we present an algorithm that computes the maximum flow and its time complexity is $O(\log^3 n \log\log n)$ using $O(n^{1.5})$ processors. In this sense, planar graphs are different from general graphs where it was shown that knowing the value of the maximum flow, does not improve the complexity of computing the flow function [Ra]. Our results also hold for more general networks, namely when the edge capacities have both lower and upper bounds.

The novel idea in computing the flow function when the demands are known 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 the network. To compute the maximum flow when the demands are not known and all the sources and sinks are on one face, a non trivial divide-and-conquer is employed. An example where multiple sources and sinks are useful is the case of computing

*this work was supported in part by NSF grant DCR-8713489 and AT&T grant USC-5345991020.

†Current address: Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213-3890. Permanent address: Computer Science Department, University of Southern California, Los Angeles, CA 90089-2140.

‡Computer Science Department, Stanford University, Stanford, CA 94305-2140. Supported by contract ONR N00014-88-K-0166 and by a grant of Stanford's Center for Integrated Systems. Part of this work was done while the author was a postdoctoral fellow at the Computer Science Department, University of Southern California, Los Angeles, CA.

a perfect matching of a planar bipartite graph. In the standard reduction from matching to flow one part of the graph is connected to a source and the other part to a sink. This reduction in general will result in a non-planar 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 [Ka] 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 as well (see for example [Va]). Yet computing a perfect matching in a planar graph is still an open problem. This situation is interesting as it contradicts the current view of the computational difficulty of counting the number of solutions vs. constructing a solution in combinatorial problems.

Johnson [Jo] showed how to compute in parallel a 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 [Jo]. Johnson's approach was first to find the minimum cut, and then to compute the flow function. We adopt a different approach; using parametric methods, we can find the value of the maximum flow and then, the computation of the flow function follows easily. Our algorithm runs in $O(\log^3 n)$ time using $O(n^{1.5})$ processors and hence, improving upon [Jo] whose running time is $O(\log^3 n)$ using $O(n^4)$ processors or, $O(\log^2 n)$ using $O(n^6)$ processors. Our results improve also the best parallel bounds for the case of an undirected graph which were $O(\log^2 n)$ using $O(n^3)$ processors [HJ,Jo].

In light of our results, the following open problems may be considered: (i) Compute the maximum flow when the demands are not known. (ii) Compute a minimum cost circulation in a planar graph. (iii) Compute a perfect matching in a planar graph. (iv) Compute a multicommodity flow in a planar graph; even the restricted case when the sources and sinks are all on one face is open.

2 Previous results in planar flow

All the results referred to in this section deal exclusively with the single source, single sink flow problem. Ford and Fulkerson [FF] 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. Itai and Shiloach [IS] showed how to compute the flow function in $O(n \log n)$ time and Hassin [Ha] gave a simple algorithm to compute the flow function when the minimum cut is known. His algorithm can be implemented in $O(n\sqrt{\log n})$ time using the method of [Fr] for computing shortest paths in a planar graph.

Itai and Shiloach [IS] 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 [Re] who gave an $O(n \log n)$ time algorithm to compute the minimum cut in an undirected planar graph. Only Hassin and Johnson [HJ] completed the picture by giving an $O(n \log^2 n)$ time algorithm to compute the maximum flow in an undirected graph by generalizing the ideas of [Ha] and [Re]. The running time of their algorithm can be improved to $O(n \log n)$ time through the methods of [Fr] for computing shortest paths in a planar graph. (An alternative algorithm for computing the minimum cut in an undirected graph was given by [JK]).

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 [JV] 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, the computational difficulty alternated between searching for the minimum cut vs. computing the flow function when the minimum cut is known.

It is easy to implement in parallel the algorithm of [HJ] for undirected graphs and its complexity is $O(\log^2 n)$ time using $O(n^3)$ processors. (The details are given in [Jo]). As for directed planar graphs, an algorithm that first computes the minimum cut, and then the flow function, was given by Johnson [Jo]. Its complexity is $O(\log^3 n)$ time using $O(n^4)$ processors or, $O(\log^2 n)$ time using $O(n^6)$ processors.

3 Terminology and preliminaries

Let $G = (V, E)$ be a planar directed graph where V is the vertex set and E is the edge set. The graph partitions the plane into connected regions called faces. For each edge $e \in E$, let $D(e)$ be the corresponding dual edge connecting the two faces bordering e. Let $\mathcal{D} = (F, D(E))$ be the dual graph of G where F is the set of faces of G and $D(E) = \{D(e) | e \in E\}$. 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.

There is a 1-1 correspondence between primal and dual edges and the direction of a primal edge e induces a direction on $D(e)$. We use a left hand rule: if the thumb points at the direction of e, then the index finger points at the direction of $D(e)$. For a vertex v, $in(v)$ ($out(v)$) denotes the incoming (outcoming) set of edges into (from) v.

A simple cycle $c_0, c_1, \ldots, c_k$ is a set of vertices such that: (i) for every $0 \leq i \leq k$, c_i and c_{i+1} $(mod\ (k+1))$ are adjacent. (ii) For every $0 \leq i, j \leq k$, $c_i \neq c_j$. In a directed cycle, for every i $(mod\ k)$, c_i is oriented towards c_{i+1}. In an undirected cycle, this property does not necessarily hold.

Associate with each edge $e \in E$ a capacity $c(e) \geq 0$ and let $S = s_1, \ldots, s_l$ and $T = t_1, \ldots, t_k$ be two sets of distinguished vertices, called sources and sinks respectively. A function $f : E \rightarrow Z$ is a legal flow function if and only if:

(i) $\forall e \in E : 0 \leq f(e) \leq c(e)$.

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

In the maximum flow problem, we are looking for a legal flow function that maximizes the amount of flow entering T (or leaving S). The amount of flow entering the sink is also called the value of the flow function. A natural generalization of the flow problem is when edges have a lower bound different from zero on their capacity; this lower bound may be either negative or positive and the capacity of an edge in that case will be denoted by $[a, b]$ where $a \leq b$. A circulation is a legal flow function where condition (ii) is applied to every vertex in the graph, i.e., there are no sinks and sources.

The flow on a dual edge $D(e)$ is always equal in value and direction to the flow on the corresponding primal edge e. Let the output flow at source s_i be denoted by $|s_i|$ and the input flow at sink t_j by $|t_j|$. These inputs and outputs are also called demands.

We have the following equivalence rules that connect the orientation of an edge $e = (v \rightarrow w)$, the sign of its flow $f(e)$, and the sign of the lower and upper bounds on its capacity.

1. The edge $v \rightarrow w$ with flow $f(e)$ is equivalent to the edge $v \leftarrow w$ with flow $-f(e)$.

2. The edge $v \rightarrow w$ with capacity $[a, b]$ is equivalent to the edge $v \leftarrow w$ with capacity $[-b, -a]$.

3. Let e_1 and e_2 be two parallel edges that are oriented in the same direction with capacities $[a_1, b_1]$ and $[a_2, b_2]$ respectively. The two edges can be replaced by one edge with capacity $[a_1 + a_2, b_1 + b_2]$ and flow $f(e_1) + f(e_2)$.

In the paper, the capacity of an edge may sometimes be also referred to as its weight.

The residual graph is defined with respect to a given flow. Let $e = (v, w)$ be an edge with capacity $[a, b]$ and flow f. In the residual graph, e is replaced by two darts with capacities $[0, b - f]$ and $[0, a + f]$.

A special case of planar flow is when the source and sink are on the same face. These graphs are called $\{s, t\}$-planar graphs.

A potential function $p : F \rightarrow Z$ is defined on the faces of a planar graph. Let e be an edge in the graph G, and let $D(e) = (g, h)$ be its corresponding edge in the dual graph such that $D(e)$ is directed from g to h. The potential difference over e is defined to be $p(h) - p(g)$. The following proposition, proved in [Ha] and [Jo], can be easily verified.

Proposition 3.1. *Let $C = c_1, \ldots, c_k$ be an undirected cycle in the dual graph and let $f_1, \ldots, f_k$ be the potential differences over the cycle edges. Then,* $\sum_{i=1}^{k} f_i(e) = 0$.

It follows from the proposition that the sum of the potential differences over all the edges adjacent to a primal vertex is zero. A potential function is defined to be consistent if the potential difference over each edge is not larger than its capacity. Such a potential function induces a circulation in the graph. The use of a potential function as a mean of computing a flow was first suggested by Hassin [Ha].

4 Computing the flow function

In this section we assume that the demand at each source and sink is known and give an efficient algorithm that computes the flow function in this case. The key idea is to compute a potential function on the faces of the planar graph such that the flow in each edge is the potential difference of the two faces that border the edge.

To do that, we first compute in the graph a spanning tree T. In the computation of the spanning tree, the orientation of the graph is ignored. Our purpose is to add new edges to the graph, parallel to the edges of T, such that a maximum flow would be equivalent to any circulation. Intuitively, we use the spanning tree to redirect the flow from the sinks back to the sources.

An edge $e \in T$ separates the tree into two parts called right and left where T_r, the right part of the tree, is adjacent to the tail of e. Let w_e be $\sum_{s_i \in T_r} |s_i| - \sum_{t_i \in T_r} |t_i|$. A new edge e' parallel to e is inserted with capacity $[w_e, w_e]$ and it is directed from T_r to T_l. (It returns the flow from T_r to T_l). This is equivalent

to adjusting the capacity of e by the value $[w_e, w_e]$. Assigning a lower bound which is equal to the upper bound forces the flow on e' to be equal to w_e. This construction is repeated for each $e \in T$ in parallel and the new graph that results is denoted by G'.

We claim that a maximum flow in G is equivalent to a circulation in G'. This circulation is computed as follows: pick an arbitrary face in the dual of G' as a root and compute all shortest paths from it; the distance of a face u from the root, bfs(u), is defined to be the potential function. For the purpose of computing the shortest paths, an edge $v \rightarrow w$ whose capacity is $[a, b]$ is decomposed into two edges: $v \rightarrow w$ with weight b and $v \leftarrow w$ with weight $-a$.

Sketch of Algorithm (I):

1. In the graph G, compute a spanning tree T. (The orientation of G is ignored in the computation of the spanning tree).
2. For each edge $e \in T$, compute its return flow: it is equal to the flux between the two parts of the tree which is $w_e = \sum_{s_i \in T_r} |s_i| - \sum_{t_i \in T_r} |t_i|$.
3. For each edge $e \in T$: adjust its weight in G' by adding $[w_e, w_e]$ to its weight. Let $\mathcal{D}' = (\mathcal{F}', \mathcal{D}(\mathcal{E}'))$ be the dual graph of G'.
4. Pick an arbitrary face in F' and compute all shortest distances from it in $\mathcal{D}'$.
5. $\forall v \in F' : p(v) \leftarrow bfs(v)$
6. $\forall e \in E : f(e) \leftarrow (p(v) - p(u)) - w_e$ where v and u are the faces that border e and $D(e)$ is oriented from u to v. ($w_e = 0$ if $e \notin T$).

4.1 Proof of correctness

We have to show that the flow function that we compute is legal and maximal. The following lemmata which imply Theorem 4.4, are based on the assumption that there exists a flow function satisfying the demands given as input to the algorithm.

Lemma 4.1. *Every circulation in G' induces a maximal flow f in G.*

When the shortest paths are computed, define the weight of a directed cycle to be the sum of the weights of its edges.

Lemma 4.2. *In $\mathcal{D}'$, the dual graph of G', a consistent potential function p exists if and only if there are no negative weight cycles.*

Lemma 4.3. *A circulation in a planar graph implies the existence of a consistent potential function on its faces.*

Theorem 4.4. *The algorithm computes a maximal legal flow in the graph. Its sequential running time is $O(n^{1.5})$; its parallel running time is $O(\log^2 n \log\log n)$ using $O(n^{1.5})$ processors in the CREW PRAM model.*

5 Maximum flow on the disk

In this section we describe an algorithm for computing a maximum flow when the sources and sinks are all on the same face and the demands are not known. Without loss of generality, one can assume that the sources and sinks are on the outer face, and that they alternate, namely there are no two consecutive sources or sinks. These two properties will be maintened during the recursive calls to the algorithm.

The Ford-Fulkerson minimum cut with respect to a given maximum flow is the set of edges inbetween W, the vertices reachable in the residual graph from the sources, and $V - W$. Suppose the vertices of the outer face are separated into two consecutive sets L and R; the maximum flow f from L to R is defined to be the flow that maximizes the input of the sinks in R.

The main idea of the algorithm is the following. Divide the vertices of the outer face into two sets (as aforementioned) and compute the maximum flow from L to R. The Ford-Fulkerson minimum 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, there are no augmenting paths. Assume that the edges of the Ford Fulkerson minimum cut are removed from the graph. A connected component that does not contain either sources or sinks is called an island. The next two lemmas explain how the above decomposition into regions is achieved.

Lemma 5.1. *After the maximum flow from L to R is computed, all the islands can be discarded.*

Lemma 5.2. *The dual of the Ford-Fulkerson minimum cut with respect to f is a set of disjoint, non nesting paths connecting pairs of faces adjacent to the outer face. (Assuming the islands were discarded).*

Let V_s denote the sources and sinks contained in a set of vertices V. Let $\mathcal{C}$ denote the connected components of G after the edges of the Ford-Fulkerson minimum cut are deleted.

Corollary 5.3. *A connected component $c \in \mathcal{C}$ cannot contain both a source from L and a sink from R.*

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

Sketch of algorithm (II)

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 the maximum flow from L to R and the residual graph G' induced by this flow. Discard all the islands from the graph.
3. Delete the edges of the Ford-Fulkerson minimum cut and compute $\mathcal{C}$, the connected components of G'; recursively, compute the maximum flow in each $c \in \mathcal{C}$.
4. Add together all the flows computed.
5. Compute the maximum flow from R to L.

We now elaborate on the steps of the algorithm. It is clear that in step 2, there exists a maximum flow from L to R in which the input of the sinks in L and the output of the sources in R are set to zero. Hence, all the sources in L can be connected to a new vertex, a super source, and all the sinks in R can be connected to a new vertex, a super sink. 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 [FF,IS,Ha].

In step 3, the maximum flow in each $c \in \mathcal{C}$ is recursively computed. Suppose for example that a connected component c contains sources from L; then all the vertices in c that are adjacent to the Ford-Fulkerson minimum cut (denoted U) become sinks. Let $U = u_1, \ldots, u_m$ and let the input of u_i be g_i. To avoid step 3 from augmenting the flow entering the vertices in U, we connect them to a new sink w. For each u_i, the capacity of the edge connecting it to w is $[g_i, g_i]$ (the orientation is from u_i to w). The sinks in the set U do not function anymore as sinks and become ordinary vertices.

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 may be only two cases: sinks from L with sources and sinks from R, or sources from R with sinks and sources from L. When a maximum flow inside c is computed, then in the first case, we connect all the sinks that belong to L to a super sink whereas in the second case, we connect all the sources that belong to R to a super source.

In step 4, the flows computed in step 3 are disjoint and hence can be added together.

In the final step, we compute the maximum flow from R to L, similarly to step 2.

In the following lemma, assume for the sake of the proof that all edges are of unit capacity. If A is an augmenting path from some source s to some sink t, then the edges in A are ordered according to their distance from s.

Lemma 5.4. *Assume that Steps 1-4 of Algorithm (II) have been performed. Obtaining the flow in Step 5 can be described by a set S of augmenting paths with the following properties: (i) For each edge e, the augmenting paths either always increase the flow on an edge, or always decrease it. (ii) Each augmenting path crosses the Ford Fulkerson minimum cut at most once.*

Theorem 5.5. *Algorithm (II) computes a maximum flow. Its running time is $O(\log^3 n \log\log n)$ using $O(n^{1.5})$ processors.*

6 Applications

The first application is computing 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 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 capacity $[0, 1]$ and 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 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 [LT]. In parallel, our result places the problem of computing a perfect matching in a planar bipartite graph in NC.

The second application is the improvement of the algorithm of [Jo] in the case of a single source, single sink in a directed planar graph. We present a simple algorithm for this problem and also improve the processor bound with respect to [Jo]. The approach taken in [Jo] is first to find the minimum cut, and then to compute the flow function. We employ a different approach. The obvious difficulty in applying Algorithm (I) is that we do not know in advance the value of the maximum flow (or the capacity of the minimum cut) in the graph. To overcome that, we guess an initial value f which is the sum of the capacities of the edges leaving the source. The spanning tree that redirects the flow from the sink back to the source is now a path

q from s to t and we construct G' and $\mathcal{D}'$ according to steps 1-3 of Algorithm (I).

If our initial guess was too large, then in the dual graph of G' where the shortest paths are computed, there must be negative weight cycles. (Otherwise, a circulation can be computed and it would correspond to a flow whose value is f). Hence, our aim is to find the largest value of f such that the dual graph of G' will not contain negative weight cycles. We can assume without loss of generality that a flow of value zero exists. The next lemma is not hard.

Lemma 6.1. *Assume that the dual graph $\mathcal{D}'$ was constructed according to steps 1-3 in Algorithm (I) and let W_f be the weight of any simple cycle C in $\mathcal{D}'$ when the initial guess is f. Then $W_f \geq W_0 - f$.*

The computation of the shortest paths proceeds 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 appeared in the diagonal of A^{2^k} and let l be the most negative entry of the diagonal in that iteration. We claim that by updating f by l and starting the computation of the shortest paths from the beginning, A^{2^k} will not have negative entries anymore in its diagonal. Hence, at most $\log n$ computations of the shortest paths algorithm suffice to compute the maximum flow function. Once the flow function is known, computing the minimum cut reduces to a reachability computation in the residual graph. The next lemma establishes the claim.

Lemma 6.2. *If the value of the flow is updated from f to $f + l$, then A^{2^k} has no negative entries.*

To improve processor bounds, the same technique can be applied with the method of parallel nested dissection of [PR].

Theorem 6.3. *Computing the maximum flow in a directed (or undirected) planar graph with a single source, single sink can be done in $O(\log^3 n \log\log n)$ time using $O(n^{1.5})$ processors. The sequential running time is $O(n^{1.5} \log n)$.*

Notice that the sequential running time matches the result of [JV].

References

[FF] L. R. Ford and D. R. Fulkerson, *Maximal flow through a network,* Canadian Journal of Mathematics, Vol. 8, pp. 399-404 (1956).

[Fr] G. N. Fredrickson, *Fast algorithms for shortest paths in planar graphs, with applications,* Siam J. on Computing, Vol.16, pp. 1004-1022 (1987).

[Ha] R. Hassin, *Maximum flows in (s,t) planar networks,* Information Processing letters, Vol. 13, page 107 (1981).

[HJ] R. Hassin and D. B. Johnson, *An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks,* Siam J. on Computing, Vol. 14, pp. 612-624 (1985).

[GSS] L. Goldschlager, R. Shaw and J. Staples, *The maximum flow problem is log space complete for P,* Theoretical Computer Science, Vol. 21, pp. 105-111 (1982).

[IS] A. Itai and Y. Shiloach, *Maximum flow in planar networks,* Siam J. on Computing, Vol. 8, pp. 135-150 (1979).

[Jo] D. B. Johnson, *Parallel algorithms for minimum cuts and maximum flows in planar networks,* Journal of the ACM, Vol. 34, (4), pp. 950-967 (1987).

[JK] L. Janiga and V. Koubek, *A note on finding cuts in directed planar networks by parallel computation,* Information Processing Letters, Vol. 21, pp. 75-78 (1985).

[JV] 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,* Proceedings of the 20th Annual Allerton Conference on Communication, Control and Computing, University of Illinois, Urbana-Champaign, Ill., pp. 898-905 (1982).

[Ka] P. W. Kasteleyn, Graph theory and crystal physics, *Graph Theory and Theoretical Physics,* Ed.: F. Harary, Academic Press, New York, pp. 43-110 (1967).

[KUW] R. M. Karp, E. Upfal and A. Wigderson, *Constructing a perfect matching is in random NC,* Combinatorica, Vol. 6, pp. 35-48 (1986).

[LT] R. J. Lipton and R. E. Tarjan, *Applications of a planar separator theorem,* Siam J. on Computing, Vol. 9, pp. 615-627 (1980).

[Me1] N. Megiddo, *Optimal flows in networks with multiple sources and sinks,* Mathematical Programming, Vol. 7, pp. 97-107 (1974).

[Me2] N. Megiddo, *A good algorithm for lexicographically optimal flows in multi-terminal terminals,* Bulletin of the AMS, Vol. 83, pp. 407-409 (1977).

[MVV] K. Mulmuley, U. V. Vazirani and V. V. Vazirani, *Matching is as easy as matrix inversion,* Combinatorica, Vol. 7, pp. 105-113 (1987).

[PR] V. Pan and J.H. Reif, *Fast and efficient parallel solution of linear systems,* to appear, Siam J. on Computing.

[Ra] V. Ramachandran, *Flow value, minimum cuts and maximum flows,* An unpublished manuscript.

[Re] J. H. Reif, *Minimum $s-t$ cut of a planar undirected network in $O(n \log^2 n)$ time,* Siam J. on Computing, Vol. 12, No. 1, pp. 71-81 (1983).

[Va] V. V. Vazirani, *NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems,* Scandinavian Workshop on Algorithm Theory (SWAT), Helmstad, Sweden (1988).