Flow in Planar Graphs: A Survey of Recent Results

by

S. Khuller and J. Naor

Technical Report #756
November, 1992
FLOW IN PLANAR GRAPHS: A SURVEY OF RECENT RESULTS

SAMIR KHULLER AND JOSEPH (SEFFI) NAOR

ABSTRACT: We briefly review some of the recent results on flow in planar graphs. The older results in the area are reviewed somewhat more briefly. The main interest in planar flow was regenerated when it was observed in both [41, 34] that the single source/sink edge capacity flow problem was not the "basic" problem that should be studied. This survey paper essentially reviews some of the techniques that were developed to capture more general versions of the planar flow problem. Some other interesting results related to planar flow problems are also reported.

1. INTRODUCTION

The computation of a maximum flow in a graph has been an important and well studied problem, both in the fields of Computer Science and Operations Research. Many efficient algorithms have been developed to solve this problem, see e.g., [13]. Research on flow in planar graphs is motivated by the fact that more efficient algorithms, both sequential and parallel, can be developed by exploiting the planarity of the graph. This is important, in particular for parallel algorithms, since maximum flow in general graphs was shown to be P-complete [15]. The planar flow algorithms are not only "good" because they are extremely efficient, but they are also very elegant. Planar networks also arise in practical contexts such as VLSI design and communication networks; therefore, it is of interest to find fast flow algorithms for this class of graphs.

In the popular formulation of the planar flow problem, one considers single source and sink vertices, s and t. Each edge has a capacity, and one wishes to find the max-flow from s to t. This problem has been extensively investigated by many researchers starting from the pioneering work by Ford and Fulkerson [7] who suggested an efficient way for computing the minimum cut in the special case of st-graphs (when the source and sink are on the same face). Berge and Ghouila-Houri [1] later developed an $O(n^2)$ time algorithm for computing the flow function in this case. This algorithm was later improved to an $O(n \log n)$ time algorithm by [26]. By introducing the concept of potentials, Hassin [21] gave an elegant algorithm that can be implemented in $O(n \sqrt{\log n})$ time using Frederickson's shortest path algorithm.
Technion - Computer Science Department - Technical Report CS0756.revised - 1992

SAMIR KHULLER AND JOSEPH (SEFFI) NAOR

[8]. Itai and Shiloach [26] also developed an algorithm to find a max flow in an undirected planar graph when the source and sink are not on the same face. Reif [49] showed how to find the minimum cut in this case in $O(n \log^2 n)$ time. Hassin and Johnson [22] finally completed the picture by giving an $O(n \log^2 n)$ algorithm to compute the flow function as well. Frederickson speeded up both these algorithms by an $O(\log n)$ factor by giving faster shortest path algorithms [8]. The problem of finding a minimum cut in a directed planar graph turned out to be much harder and was first solved by [29] (both sequentially and in parallel).

Miller and Naor [41] pointed out that the general maximum flow problem in planar graphs is when there are many sources and sinks. Note that one cannot reduce the multiple source-sink problem to the single source-sink version since the reduction may destroy planarity. However, we would like to design more efficient algorithms for this case: for sequential algorithms, we would like to take advantage of the planarity and improve on the performance of the best algorithms for general graphs; in parallel, it is not even known whether an NC algorithm exists for this problem. Miller and Naor [41] showed that when demands and supplies are fixed, the problem can be reduced to a "circulation problem" (with lower bounds on edge capacities), and also gave an efficient algorithm for this case. They also gave an efficient algorithm for the case where the demands and supplies are variable, but the sources and sinks belong to a bounded number of faces. If the sources and sinks belong to an arbitrary number of faces, computing a maximum flow efficiently is still open.

Khuller and Naor [34] considered the more general flow problem in which vertices as well as edges have capacity constraints. Vertex capacities may arise in various contexts such as computing vertex disjoint paths in graphs [36], and in various network situations when the vertices denote switches and have an upper bound on their capacities. For the case of general graphs this problem can be reduced to the version with only edges having capacity constraints by a simple idea of "splitting" vertices into two and forcing all the flow to pass through a "bottleneck" edge in between. In planar graphs, this reduction may destroy the planarity of the graph and thus cannot be used. (The reduction is described in Bondy and Murty ([4], page 205) from which the violation of planarity is obvious.)

Notice that in the case of general graphs, as opposed to planar graphs, the single source-sink problem with edge capacities is usually the "basic" problem, since most other formulations of the flow problem can be easily reduced to this problem. It is not clear if there is such a "basic" problem in the context of flow in planar graphs.

An application where vertex capacities play an important role is in reconfiguring VLSI/WSI (Wafer Scale Integration) arrays. Assume that the processors on a wafer are configured in the form of a grid, and due to yield problems, some are going to be faulty. Instead of treating the whole wafer as defective, the non-faulty processors can be reconfigured in the form of a grid. We assume that multiple data tracks are allowed along every grid line. It was shown in [51] that in this context, the reconfiguration problem can be abstracted combinatorially as finding a set of vertex disjoint paths from the faulty processors (the sources) to the boundary of the grid (the sink). This is a special case of a multiple source/single sink planar flow problem where all vertex capacities are equal to 1. This problem is also referred to as the escape problem in the textbook by Cormen, Leiserson and Rivest [CLR, page 628]. The algorithm given by [51] has a running time of $O(n^2 \log n)$ where $n$ is the
number of grid points. The algorithms of [34] improve over this result by an $O(\sqrt{n})$
factor. The reader is referred to [3, 6, 18, 50, 51] for more details and bibliography
of this problem and on the connection between flow problems and reconfiguration.
(The main concern of [50, 3] is the single-track model.)

The problem of computing a maximum flow is closely related to the problem
of computing a maximum matching. In fact, in a bipartite graph, computing a
maximum matching is a special case of computing a maximum flow. In the parallel
context, for arbitrary graphs, only a randomized procedure is known for computing
a maximum matching [32, 42] and also for determining whether a graph contains a
perfect matching. In contrast, in planar graphs, it can be decided deterministically
in NC [38, 54] whether a graph has a perfect matching. (Even the number of perfect
matchings can be counted in NC!) However, it is an open question whether a perfect
matching can be computed in NC. It follows from the results of [41] that a perfect
matching in planar bipartite graphs can be computed in NC.

We now move to a slightly different perspective on the planar flow problem. We
study the set of integer solutions to the planar circulation problem, and characterize
an encoding for all the feasible integer circulations. It turns out that the set of
feasible circulations in planar graphs forms a distributive lattice where the meet
and join operations are defined appropriately. Other examples of problems where
the solution set has a similar structure are the stable marriage problem, and the
minimum cut problem. Picard and Queyranne [45] have shown that the set of all
minimum s-t-cuts forms a distributive lattice where the join and meet operations
are defined as intersection and union respectively. The structure of the solution set
of the stable marriage problem has been extensively investigated in the book by
Gusfield and Irving [19].

A brief outline of the paper is as follows: In Section 2 we discuss some basic flow
notation used in the rest of the paper. Section 3 is a survey of the past research
on planar flow. Section 4 describes the results of [41]. In Section 5 we describe the
lattice structure of planar flow [35]. Section 6 describes the results of [34]. Section 7
outlines some related flow problems that are NP-complete even for planar graphs
[2]. We conclude by outlining some of the major open problems in Section 8.

2. TERMINOLOGY AND PRELIMINARIES

We are going to assume that the graph $G = (V, E)$ has a fixed planar embedding.
For each edge $e \in E$, let $D(e)$ be the corresponding dual edge connecting the two
faces bordering $e$. Let $D = (F, D(E))$ be the dual graph of $G$, where $F$ is the set
of faces of $G$ and $D(E) = \{D(e) | e \in E\}$. 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 in the direction of $e$, then the
index finger points in the direction of $D(e)$ (keeping the palm face up). For a vertex
$v$, $\text{in}(v)$ refers to the arcs that are carrying incoming flow to vertex $v$. Similarly
$\text{out}(v)$ refers to those arcs that are carrying flow out of the vertex $v$.

Associate with each edge $e \in E$, a capacity $c(e) \geq 0$, and also with each vertex
$v \in V - \{S, T\}$, a capacity $c(v) \geq 0$. Let $S = s_1, \ldots, s_t$ and $T = t_1, \ldots, t_q$ be two
sets of distinguished vertices, called sources and sinks respectively. We assume that
the vertices in $S$ and $T$ have no capacities. Otherwise, suppose that vertex $s \in S$
has capacity $c(s)$; add to the graph a new distinguished vertex $s'$ adjacent only to
s, such that the capacity of the edge joining s and s' is unbounded. Remove vertex s from S and add s' to S. By performing this step for every capacitated vertex in S, and an analogous step for every capacitated vertex in T we obtain the required property.

A function $f : E \rightarrow \mathbb{Z}$ is a legal flow function if and only if:

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

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

(iii): $\forall v \in V - \{S, T\} : \sum_{e \in \text{in}(v)} f(e) \leq c(v)$.

We assume that $G$ is biconnected; otherwise, we can add edges with zero capacities appropriately to ensure that.

The cost of a dual edge is defined in the undirected case to be the capacity of the corresponding primal edge. (The dual edge is also undirected.) In the directed case, given a primal edge $e$ of capacity $c(e)$, it has two dual edges corresponding to it: one is directed according to the left hand rule and has cost $c(e)$; the other is in the converse direction and has cost 0, or in general, its cost is equal to the lower bound on the flow on edge $e$ (see Fig. 1).

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{fig1.png}
\caption{Dual graph in case of a directed graph}
\end{figure}

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 circulation is a legal flow function where condition (ii) is applied to every vertex in the graph, i.e., there are no sources and sinks.

A natural generalization of the flow problem is when edges have a lower bound different from zero on their capacity; in this case, the capacity of an edge will be denoted by $[a, b]$, where $a \leq b$.

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 directed edges $(v, w)$ and $(w, v)$ with capacities $[0, b - f]$ and $[0, f - a]$ respectively.

A spurious cycle is a directed cycle along which the flow can be reduced, without any of the edges violating the lower bounds on their capacities.

A special case of planar flow is when the source and sink are on the same face. These graphs are called $st$-graphs.
FLOW IN 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. Let \( C = c_1, \ldots, c_k \) be a 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 edge consistent if the potential difference over each edge is not larger than its capacity. Such a potential function induces a circulation in the graph. If the circulation satisfies the vertex capacities as well, the potential function is defined to be consistent. The use of a potential function as a means of computing a flow was first suggested by Hassin [21], and was later elaborated by [22] and [29]. Miller and Naor [41] use the idea of potentials to solve the problem of computing a feasible circulation as well.

The model of parallel computation used is the Exclusive-Read Exclusive-Write (EREW) Parallel Random Access Machine (PRAM). A PRAM employs synchronous processors all having access to a shared memory. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location.

3. PLANAR FLOW WITH A SINGLE SOURCE AND SINK

All the results referred to in this section deal exclusively with the single source, single sink maximum flow problem where only edges have capacities. Ford and Fulkerson [7] 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 [26]. Hassin [21] gave an elegant algorithm to compute the flow function and his algorithm can be implemented in \( O(n \sqrt{\log n}) \) time using the method of [8] for computing shortest paths in planar graphs.

Hassin’s idea can be summarized as follows: Partition the infinite face (where the source and sink are assumed to be) by a directed edge (from \( t \) to \( s \)) of infinite capacity. Let \( s^* \) denote one of the two faces generated by this partitioning as in Fig. 4 (the return edge is not shown in the picture). For each edge \( (i, j) \in E \), let \( (i', j') \in D(E) \) be the associated dual edge.

**Definition 3.0.1.** The potential \( p(f) \) of a face \( f \) (or a vertex \( f \) in the dual graph) is defined to be the length of the shortest distance from \( s^* \) to \( f \).

The flow on edge \( (i, j) \) is defined as follows: \( f(i, j) = p(j') - p(i') \). This yields an edge consistent potential function since \( p(j') \leq c(i, j) + p(i') \) (the potentials satisfy the shortest path property), and hence a valid flow function. The flow moves through the edge always keeping the face with higher potential to its right.

Itai and Shiloach [26] 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. The main idea of their algorithm is the following: find a path from the source to the sink; send the flow from the source to the sink edge-by-edge. The problem of sending the flow on each edge is an instance of an $st$-graph problem. (The tail and head of the edge are the source and sink respectively.) Notice that if too much flow is sent initially to the sink, some of it can be "returned" to the source by the same method. The running time of the algorithm is $O(n^3 \log n)$. Reif [49] gave an improved algorithm for computing the minimum cut in this case, i.e., an undirected planar graph where the source and sink are not on the same face. The running time of this algorithm was $O(n \log^2 n)$ time. Hassin and Johnson [22] completed the picture by giving an $O(n \log^2 n)$ time algorithm to compute the flow function as well, by generalizing the ideas of [21] and [49]. (The running time of their algorithm can be improved by using the methods of [8] for computing shortest paths in planar graphs.)

It is easy to obtain a parallel implementation of the algorithm given by [22] for undirected graphs and its complexity is $O(\log^2 n)$ time using $O(n^3)$ processors. (The details are given in [29].) A more efficient implementation that uses only $O(n^{1.5})$ processors can be obtained by using the methods of [46, 47, 48]. An alternative algorithm for computing the minimum cut in parallel in an undirected graph was given by [36].

The problem of computing a maximum flow in a directed planar graph (when the source and sink are not on the same face) turned out to be more difficult. The intuitive reason for this difficulty is the following. Finding the minimum cut is equivalent to finding a minimum weight cycle in the dual graph separating the source from the sink. In the undirected case, this problem can be reduced to the problem of computing a (certain) minimum weight path. This reduction cannot be applied in the directed case. This problem was eventually solved by Johnson [29] who provided both sequential and parallel algorithms for finding the minimum cut as well as the flow function. The complexity of the sequential algorithm is $O(n^{1.5} \log n)$ time (see also [31]). The parallel complexity is $O(\log^3 n)$ time using $O(n^2)$ processors, or $O(\log^2 n)$ time using $O(n^6)$ processors. Again, a more efficient implementation can be obtained by using the methods of [46, 47, 48].

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 one hand, and computing the flow function, when the minimum cut is known, on the other hand.

4. Planar Flow with Multiple Sources and Sinks

The potential method pioneered by Hassin has really paved the way for the future planar flow algorithms (which are clever elaborations of Hassin's basic potential method). A very elegant scheme for computing a flow for the multiple source/sink (when the sources and sinks have fixed supplies and demands) problem was given by Miller and Naor [41]. We proceed to outline the scheme in this section.

We then address the problem of computing the maximum flow in the case where the demands and supplies are variable, but the sources and sinks belong to a bounded number of faces. The problem when the demands and supplies belong to an arbitrary number of faces, is open. For sequential algorithms, we would like to take advantage of the planarity and improve on the performance of the best
algorithms for general graphs; in parallel, we would like to provide NC algorithm for this problem. Unfortunately, this problem is still open.

4.1. The potential method. In this section we assume that the supply at each source and the demand at each sink is known, and give an efficient algorithm that computes the flow function in this case. We denote the supply of source \( s_i \), \( 1 \leq i \leq l \), by \( |s_i| \) and the demand at sink \( t_j \), \( 1 \leq j \leq k \), by \( |t_j| \). The key idea is to compute a potential function on the faces of the planar graph such that the flow on each edge is the potential difference of the two faces that border the edge. To achieve this, we re-formulate the problem as a circulation problem with lower bounds. This is done by first computing a spanning tree \( T \) (the orientation of the edges of the graph is ignored for the purpose of computing \( T \)). Then, new edges, parallel to the edges of \( T \), are added to the graph 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. Let \( w_e \) be \( \sum_{i \in T_r} |t_i| - \sum_{i \in T_l} |s_i| \). A new edge \( e' \) is inserted to the graph to redirect the flow from the sinks back to the sources. (The orientation of the edges of the graph is ignored for the purpose of computing \( T \).) 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 there is a 1-1 correspondence between flows satisfying the supplies and demands in \( G \) and circulations in \( G' \). A circulation \( G' \) 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, \( d(u) \), is defined to be the potential function.

Sketch of Algorithm:

- **Step 1.** In the graph \( G \), compute a spanning tree \( T \). (The orientation of \( G \) is ignored in the computation of the spanning tree.)
- **Step 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 \( w_e = \sum_{i \in T_r} |t_i| - \sum_{i \in T_l} |s_i| \).
- **Step 3.** For each edge \( e \in T \); adjust its weight in \( G' \) by adding \( [w_e, w_e] \) to its weight. Let \( D' = (P', D(E')) \) be the dual graph of \( G' \).
- **Step 4.** Pick an arbitrary face in \( F' \) and compute all shortest distances from it in \( D' \).
- **Step 5.** \( \forall v \in F' : p(v) \leftarrow d(v) \)
- **Step 6.** \( \forall e \in E : f(e) \leftarrow (p(v) - p(u)) \) where \( v \) and \( u \) are the faces that border \( e \) and \( D(e) \) is oriented from \( u \) to \( v \). Now delete all the edges \( e' \) that carry "return" flow.

We have to show that the flow function that we compute is both legal and satisfies the demands and supplies. In the following it is assumed that there exists a flow function that satisfies the given demands and supplies.

**Theorem 4.1.1.** The algorithm computes a feasible flow satisfying the demands and supplies in the graph.

**Proof.** It is evident from the construction that every circulation in \( G' \) induces a flow \( f \) satisfying the demands and supplies in \( G \), and a circulation in \( G' \) exists if a feasible flow exists in \( G \). Hence it suffices to compute a circulation in \( G' \). We claim that the existence of a circulation in \( G' \) implies the existence of a consistent
potential function in $D'$, the dual graph of $G'$. Clearly, a consistent potential function $p$ exists if and only if there are no negative weight cycles in $D'$. By the correspondence between cycles and cuts in planar graphs, a negative cycle in $D'$ implies that there exists a cut in $G'$, where the lower bounds on the edges leaving one portion of the graph add up to more than the upper bounds of the edges entering that portion. This immediately violates the existence of a feasible circulation in $G'$. Since there cannot be cycles with a negative weight in $D'$, a shortest path labeling from an arbitrary face is a consistent potential function that induces a circulation. 

**Theorem 4.1.2.** The sequential running time of the algorithm is $O(n^{1.5})$; the parallel running time is $O((\log^3 n))$ in the EREW PRAM model and $O(\log n)$ time in the CRCW PRAM model, where the number of processors is $O(n^{1.5})$.

**Proof.** The most expensive part of the algorithm is Step (4) where all shortest paths from a vertex in the dual graph are computed. The sequential complexity of computing all shortest paths from a given vertex in a graph with negative edge weights is $O(n^{1.5})$ using the generalized nested dissection of [39]. To compute shortest distances in parallel, the nested dissection algorithm of [46, 47, 48] requires $O(n^{1.5})$ processors; the time complexity is $O(I(n) \log n)$ where $I(n)$ is the parallel time of computing the sum of $n$ values. $I(n)$ can be implemented in $O(\log n)$ time on 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 [43]. In parallel, Gazit and Miller [11, 12] provided a procedure for computing small separators that uses $O(n^{1+s})$ processors and $O(\log^3 n)$ time. (The running time is dominated by the procedure for computing a maximal independent set in a graph. The current best bound is $O((\log n)^2)$ time using $O(m/\log n)$ processors [14].)

An embedding of a planar graph can be computed sequentially in linear time [23], and in parallel, in $O(\log^2 n)$ time using a linear number of processors [37]. The embedding is needed for computing the dual graph. 

**4.2. Maximum flow for a bounded number of faces containing sources and sinks.** In this section we provide efficient sequential and parallel algorithms for the case when the sources and sinks belong to a fixed number of faces. Without loss of generality, we can assume that all capacities are nonnegative and that the sources and sinks alternate on every face, namely there are no two consecutive sources (or sinks). This property will be maintained during the recursive calls to the algorithm.

Let $G$ be a graph and $f$ a maximum flow. The Ford-Fulkerson cut with respect to $f$ is the set of edges between $W$, the vertices reachable by an augmenting path in $G - f$, and $V - W$. That is, the Ford-Fulkerson cut is the "first" minimum cut separating the sources from the sinks.

We first claim 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.
FLOW IN PLANAR GRAPHS

(3) Remove the edges of $C$ from the graph $G$. Compute recursively a maximum flow in each connected component.

(4) Compute a maximum flow from $R$ to $L$.

Notice that the input graph in Steps (3) and (4) is 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 implement the generic algorithm to compute a maximum flow in $G$ efficiently.

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 $R_i$, such that the set $L_i$ contains as many sources as the set $R_i$ and both sets are approximately of the same cardinality. The set $L$ is defined to be the union of the sets $L_i$ ($1 \leq i \leq k$) and the set $R$ is defined to be the union of the sets $R_i$ ($1 \leq i \leq k$). The maximum flow from $L$ to $R$ is defined to be 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.

To implement Step (2), first connect the sources in each set $L_i$ to a super source $s_i$, and the sinks in each set $R_i$ to a super sink $t_i$. This operation does not violate the planarity of the graph. Observe that the following greedy algorithm computes a maximum flow with many sources and sinks: for all pairs $i, j$, $1 \leq i, j \leq k$, compute in arbitrary order the maximum flow from $s_i$ to $t_j$; the flow is computed in the residual graph with respect to the pairs for which a maximum flow has already been computed. Computing the flow from source $s_i$ to sink $t_j$ is an instance of the problem of computing the maximum flow in a directed planar graph with a single source and single sink. Step (4) is implemented similarly. We refer the reader to [41] for the proof of this algorithm.

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

We conclude with the following theorem.

Theorem 4.2.1. 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 and in parallel) in $O(k^2)$ calls to a procedure that computes a maximum flow in the single source and single sink case.

5. THE LATTICE STRUCTURE OF PLANAR FLOW

In this section we study the structure of the set of solutions to the circulation problem for planar graphs. We first establish a one-to-one correspondence between consistent potential functions and circulations. Given a legal circulation $C$, a corresponding potential function can be constructed as in the proof of Theorem 4.1.1. (Pick an arbitrary face $r$ and compute all shortest distances from it where the edge weights in this search are the actual flows in the circulation.) It is not hard to see that this potential function induces circulation $C$. We shall henceforth view a potential function as a vector where the entries correspond to the potentials of the faces, and the potential of the root face (an arbitrary but fixed face) is always equal to zero.

Given two consistent vectors, $P_1$ and $P_2$, we say that $P_1 \geq P_2$ if for all components $i$, $P_1(i) \geq P_2(i)$. We say that circulation $C_1$ dominates $C_2$ if, for their corresponding potential vectors $P_1$ and $P_2$, $P_1 \geq P_2$. We use the term $\mathcal{P}$ to refer to
set of all the consistent potential vectors. It is easy to see that \( \mathcal{P} \) is a partial order under the dominance relation (also written as \((\mathcal{P}, \preceq)\)). We claim that the set \( \mathcal{P} \) very naturally forms a distributive lattice, where a distributive lattice is a partial order in which:

1. Each pair of elements has a greatest lower bound, or meet, denoted by \( a \wedge b \), so that \( a \wedge b \preceq a, a \wedge b \preceq b \), and there is no element \( c \) such that \( c \preceq a, c \preceq b \) and \( a \wedge b \not\preceq c \).
2. Each pair of elements has a least upper bound, or join, denoted by \( a \vee b \), so that \( a \preceq a \vee b, b \preceq a \vee b \), and there is no element \( c \) such that \( a \preceq c, b \preceq c \) and \( c \not\preceq a \vee b \).
3. The distributive laws hold, namely \( a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c) \) and \( a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \).

Given two circulations \( C_1 \) and \( C_2 \) (represented as \( P_1 \) and \( P_2 \)), we define the meet as the circulation induced by the potential vector \( P_m = \min(P_1, P_2) \). Clearly, the face at zero potential in both circulations stays at zero potential. Every face \( g \) is assigned a potential equal to \( \min(P_1(g), P_2(g)) \) where \( P_1(g) \) is the potential of \( g \) in \( C_1 \). Similarly, the join is defined as \( P_j = \max(P_1, P_2) \). The proof of the following theorem is not hard.

**Theorem 5.0.2.** The partial order \((\mathcal{P}, \preceq)\) is a distributive lattice, with the meet and join defined appropriately.

It is easy to see that a lattice has a unique minimum and maximum, \( P_b \) and \( P_t \), referred to as bottom and top respectively. We now provide a simple characterization for them. The shortest path potential vector, in which the potential of a face is exactly its distance from the root face in the dual graph corresponds precisely to the top of the lattice. A vector corresponding to the bottom of the lattice can be computed as follows: the potential of face \( f \) is the length of the shortest path from \( f \) to \( r \), the root face, multiplied by -1.

It turns out that the flow functions computed by [21], [22], [29] and [41] can be interpreted as corresponding to the top element of the circulation lattice. (This is essentially a matter of notation; if we reverse the direction of the dual edges, their algorithms will be computing the bottom element in the lattice.)

The lattice representing all feasible circulations is clearly of exponential size, since there are exponentially many solutions to the circulation problem. A compact encoding of the entire lattice can be obtained by constructing a directed acyclic graph such that the predecessor-closed subsets of this partial order correspond to elements in the lattice. Although this DAG may be large, its size depends on the maximum edge-capacity—it can be represented succinctly, in polynomial size. This compact encoding of the partial order provides in turn a compact encoding of the lattice elements.

There is also a connection between the lattice and unidirectional cycles. Recall that we assumed the planar embedding was such that the infinite face is the root face. Each simple cycle divides the sphere into two nonempty disjoint sets of faces, called regions. The region containing the root face is designated the exterior region; the other region is interior. In a traversal of a directed cycle, all faces that border the cycle on its right are in the same region, the cycles right-hand region.
Definition 5.0.1. A directed cycle is clockwise if the cycles right-hand region is interior. Otherwise, the cycle is counterclockwise.

The top and bottom of the lattice can be characterized by unidirectional cycles.

Theorem 5.0.3. A circulation is clockwise maximal if and only if it corresponds to $P_1$. A circulation is counterclockwise maximal if and only if it is $P_0$.

It is tempting to believe that the dominance relation in the lattice can be stated in terms of saturating clockwise cycles. That is, if $P_1 < P_2$, then circulation $P_2$ can be obtained from $P_1$ by saturating clockwise cycles. Unfortunately, the following counterexample shows that this is not true. Let $c_1$ and $c_2$ be clockwise cycles such that $c_1$ is contained in the interior of $c_2$. We construct two circulations, $P_1$ and $P_2$, such that $P_1 < P_2$. To construct $P_1$, take $P_0$ and push one unit of flow in the cycle $c_1$ in the clockwise direction. To construct $P_2$, take $P_0$ and push one unit of flow in the cycle $c_2$ in the clockwise direction. Obviously, $P_1 < P_2$, but the only way to obtain circulation $P_2$ from $P_1$ is to push a unit of flow in the cycle $c_2$ in the clockwise direction and in the cycle $c_1$ in the counterclockwise direction.

6. Planar flow with vertex capacities

In this section we survey some of the results in [34] that address the more general problem when vertices as well as edges have capacities. As we already have mentioned, the standard trick in general graphs for eliminating vertex capacities may destroy the planarity of the graph.

Assume that a planar flow problem with vertex capacities is reduced to a circulation problem with vertex capacities. It is interesting to note that in this case, the set of feasible circulations does not form a lattice. This observation may partly account for the computational difficulty in computing flows in the case where the vertices are capacitated.

6.1. Computing the Min-cut. When the graph has vertex as well as edge capacities, a cut is not just a set of edges, but a subset $S \subseteq E \cup V$ with the property that every path from $s$ to $t$ contains an element of $S$. A minimum cut is defined to be a set $S$ of minimum capacity. In the dual graph, a cut corresponds to a set of edges and faces (that correspond to the vertices in the cut). These edges and faces can be "linked" together (see Fig. 2) and induce a "linked" cycle in the dual graph that separates the faces corresponding to $s$ and $t$.

In the dual graph we define a new shortest path computation as follows:

Definition 6.1.1. We are given a planar dual graph $D$ with a cost $c(e_i)$ on each edge $e_i$, and a cost of $c(f_j)$ on each face $f_j$ (this cost is the capacity of the corresponding primal vertex). We define a linked cycle to be a sequence of edges and faces $[z_1; z_2; \ldots; z_k]$ so that each $z_i$ and $z_{i+1}$ share a common vertex. (See Fig. 2 for an example.) The length of a linked cycle is the sum of the costs of the edges and the costs of the faces the cycle "jumps" over (to move from one edge to another). The shortest linked cycle is defined to be the linked cycle with the least length.

Under this definition, the minimum cut corresponds to the shortest "linked" cycle in the dual graph that separates $s$ from $t$. In [34] we show how to modify the dual graph so that such a cycle can be computed efficiently (the key point is that
Figure 2. Cycle in the Dual Graph

we wish to maintain planarity, to permit the use of Frederickson’s shortest path algorithms.

We have now reduced the problem of finding the minimum cut in a planar graph with vertex capacities to that of finding the minimum length cycle separating \( s \) from \( t \) in a new planar graph, \( D' \), that has only edge capacities. The efficiency of computing this cycle varies with respect to whether the source and sink are on the same face, or whether the graph is directed.

By an application of the above idea, we can show that for \( st \)-graphs we can obtain a fast algorithm for finding the min-cut. The bounds for the parallel algorithm follow from the proof of Theorem 4.1.2.

Theorem 6.1.1. We can compute the value of the max-flow in a \( st \)-graph (directed or undirected) in \( O(n \sqrt{\log n}) \) time. Moreover, we can implement this algorithm in \( O(\log^3 n) \) time using \( O(n^{1.5}) \) processors on an EREW PRAM.

In [49] it was shown that the minimum cut (or the value of the max flow) can be computed efficiently even when the vertices \( s \) and \( t \) are not on the same face in an undirected planar graph. Using Frederickson’s algorithm for shortest paths in planar graphs as a subroutine, one can obtain a running time of \( O(n \log n) \). We note that by using the “jumping” over faces idea we get an \( O(n \log n) \) time algorithm for computing the minimum cut in the graph even when the vertices have capacities.

The problem of finding a minimum cut (between a single source-sink pair) in the directed graph case is considerably harder than in the undirected case and was dealt with by [29]. Recently, an elegant technique was developed by [41] to find the minimum cut (this simplifies the procedure of [29]). In [34] it is shown that an
appropriate modification of the method is able to find the minimum cut even when vertex capacities are present. (This is because vertex capacities cause an altered structure to the min-cut.)

Theorem 6.1.2. The minimum cut in a directed planar graph can be found in $O(n^{1.5} \log n)$. A parallel implementation uses $O(n^{1.5})$ processors and $O(\log^4 n)$ time on the EREW PRAM.

The parallel complexity follows from the proof of Theorem 4.1.2. The additional $O(\log n)$ factor is due to the binary search for the value of the min-cut.

6.2. Computing the flow function. The main difficulty in computing the flow function with vertex capacities is that the potential function computed in the dual graph with "jumping over faces" is not consistent. As a consequence, computing the flow function becomes much more complicated than in the case where there are only edge capacities.

The first case we deal with are $st$-graph's (both undirected and directed). In [34] an $O(\log n)$ implementation of the "uppermost path" algorithm due to Ford and Fulkerson [7] is given (that handles vertex capacities as well).

We give an $O(n \log n)$ algorithm to compute a valid flow function in an undirected $st$-graph that has vertex capacities. The algorithm can be extended to the case of directed $st$-graphs quite easily by using the ideas in [26] to find the directed uppermost path in each iteration. For details on the uppermost path algorithm we refer the reader to [26] and [7] (see also [43]).

We will briefly outline the modifications to the algorithm to handle vertex capacities. The algorithm begins by pushing flow through the uppermost path from $s$ to $t$ (see Fig. 3).

The capacity of the uppermost path is defined to be the least residual capacity of either an edge or a vertex. At least one edge or vertex on the uppermost path gets saturated by pushing a flow of value equal to the capacity of the path. The saturated edge/vertex is deleted from the graph, and the process is repeated using the uppermost path in the residual graph until $s$ is disconnected from $t$.

Care needs to be taken to make the uppermost path simple each time we delete the saturated edge or vertex. The reason for this is the presence of vertex capacities in the graph. In the case of only edge capacities, pushing a flow of value equal to the capacity of the uppermost path does not violate any capacity constraint. Suppose there are vertex capacities present and the path is non-simple at a vertex that has a capacity. Pushing a flow of value $f$ on this path actually increases the incoming flow to this vertex by at least $2f$ units, which could cause a violation in the capacity constraint of this vertex. This is the main modification to the algorithm presented in [26]. The algorithm discards pieces of the graph in making the path simple at each pushing step. By the following proposition we can see that the value of the flow function computed by this modified uppermost path algorithm is the same as the value of the min-cut.

The process of making the augmenting path simple at each step does not decrease the amount of flow pushed on that augmenting path.

Theorem 6.2.1. A maximum flow function can be computed in $O(n \log n)$ time for the case of $st$-graphs, even when the vertices have capacities.
FIGURE 3. Uppermost path may be non-simple
A parallel algorithm to find the max-flow in an st-graph (directed and undirected) that works by canceling the spurious cycles in the graph is also given in [34]. (This is described in the next subsection.) A sequential implementation of the parallel algorithm takes $O(n \log n)$ time without counting the time for the step that requires sorting. (Thus we could obtain an $O(n \sqrt{\log n})$ time randomised algorithm by using the fast randomised sorting algorithm due to [9] that runs in $O(n \sqrt{\log n})$ expected time.)

If the source and sink are not on the same face, then we first find the value of the max-flow by the parametric search technique. The problem then reduces to a fixed demand problem. If there are many sources and sinks in the graph (with fixed demands), then we reduce it to the problem of computing a circulation. This is done similarly to [41] by “piping” the flow back from the sinks to the sources, via a path that must not go through any capacitated vertices.

An efficient algorithm for computing a circulation when edges have lower bounds and vertices are capacitated is also given in [34]. This tackles the case when the demand of each source and sink is known. We show how to compute a circulation that will satisfy the demands, or determine that a feasible circulation does not exist.

In the case of vertex capacities, defining the flow through an edge to be the difference of the potentials of the faces on each side as computed in $D'$ (with jumping over faces), yields a flow function that may violate vertex capacity constraints.

![Diagram](image.png)

**Figure 4. Example to show violation of vertex capacities**

Incoming flow to $v$ is 3
(exceeds the capacity)

Min-cut = 3
6.3. The parallel algorithm for st-graphs. We develop a two phase parallel algorithm to find a valid max-flow in the case of st-graphs. We only give an informal overview of the algorithm. In the first phase, we compute the potential of each face by a shortest path computation in the dual graph with \( s' \) as the source. This is done with jumping over faces permitted (which can be reduced to a shortest path computation, as was shown in Section 3). If there are no capacitated vertices then clearly this yields a valid flow function. In certain cases, it may also happen that this procedure yields a valid flow function even in the presence of vertex capacities. In general, it does not yield a valid flow function (as the earlier example showed) due to the presence of "spurious cycles".

In the second phase we show how to fix all the "unhappy vertices" (that have excess flow through them). To motivate the second phase let us see what goes "wrong" when we compute potentials via jumping over faces. Consider a vertex \( v \) that has capacity \( c_v \), and its incident faces. We assume that the incident faces have potential values \( p_0^v, p_1^v, \ldots, p_{d(v)-1}^v \) (where \( d(v) \) is the degree of \( v \)). We can assume that \( p_0^v \) is the smallest potential value and that the ordering of the faces is anticlockwise (see Fig. 5 for an example). Number the faces such that \( p_i^v \) is the potential of face \( i \). The edge incident on \( v \) between face \( i \) and \( (i + 1) \mod d(v) \) is called \( e_i \). Since the potentials were computed with "jumping over faces" we know that

\[
| p_i^v - p_j^v | \leq c_v \quad \forall i, j
\]

\[
| p_i^v - p_{i+1}^v | \leq c_{e_i} \quad \forall i.
\]

If we traverse the faces starting from face 0 in an anticlockwise direction, whenever the potential goes up it corresponds to an edge with incoming flow. The amount of incoming flow is the same as the change in potential. Correspondingly, whenever the potential goes down, it corresponds to an edge with outgoing flow. (In Fig. 6 we illustrate a vertex \( v \) with seven edges incident on it, and the corresponding potential sequence.) Clearly, each jump in the potential, either up or down, is bounded above by \( \min(c_v, c_e) \) where \( c_e \) is the capacity of the corresponding edge. As we do the traversal, the total incoming flow could easily exceed \( c_v \).

**Figure 5. A capacitated vertex and the potentials of its incident faces**

We now show a correspondence between the uppermost path algorithm and the shortest path algorithm. This is important for understanding how the potentials can be adjusted to cancel the relevant spurious cycles. The uppermost path algorithm...
really corresponds to growing a shortest path tree $T_D$ from $s^*$. The augmenting path at each step corresponds to the “fringe” of the faces corresponding to vertices in tree $T_D$ at various stages of a Dijkstra shortest path computation. When the fringe is non-simple, the uppermost path is also non-simple and needs to be made simple.

The flow function computed by assigning potentials, directly corresponds to an uppermost path algorithm without making the path simple at each step – this is precisely what causes excess flow to go through capacitated vertices.

In the second phase we will try and cancel all the “spurious cycles” that cause capacitated vertices to be unhappy. The idea is to consider various snapshots of the dual tree. Examining the snapshots of the dual tree encode the various stages of an uppermost path algorithm. The non-simplicities are easy to detect and the potentials can be adjusted to cancel some spurious cycles (at least enough cycles so as to satisfy the capacity constraints of all vertices).

We now get the following theorem.

**Theorem 6.3.1.** A max-flow in an st-graph (directed and undirected) can be found in $O(\log^3 n)$ time on an EREW PRAM using $O(n^2)$ processors.

6.4. Reduction from flow to circulations. A flow problem is transformed to a circulation problem with lower bounds on the edges. This will be done by adding new edges that will return the flow from the sinks back to the sources (as in Section 4.1). These edges will have lower bounds so as to ensure that the demands of the sources and sinks are satisfied. This reduction works for both undirected and directed graphs, but generates a directed graph. In the new graph we will compute a circulation and obtain a legal flow that satisfies the demands by removing the newly added edges.

6.5. Computing circulations. In this section we show how to use the planar separator theorem [40] to obtain a solution for the circulation problem when the
graph contains edge capacities (upper and lower) as well as vertex capacities. We will assume that the graph is triangulated. This approach is similar to the algorithm developed by [31].

An overview of the algorithm:

**Step 1.** Find a separating cycle $C$ of size $O(\sqrt{n})$. Let the *interior* and *exterior* of $G$ be denoted by $G_I$ and $G_E$.

**Step 2.** Recursively find a circulation in $G_I + C$ and $G_E + C$.

**Step 3.** Merge the circulations computed in Step 2, to obtain a circulation in $G$.

In [34] the following lemma is proven.

**Lemma 6.5.1.** There exists a feasible solution in $G_I + C$ and $G_E + C$.

To combine the two circulations, we need to redirect flow on the edges of the cycle $C$. This is done by making $O(\sqrt{n})$ calls to the $st$-graph subroutine (see details in [34]), and this dominates the complexity of the algorithm.

We can prove the following:

**Theorem 6.5.2.** The complexity of computing a circulation in a planar graph with vertex capacities is $O(n^{1.5} \log n)$.

### 7. NP-completeness results

There are several flow problems that are known to be NP-complete for general graphs. We show that these are NP-complete when we restrict our attention to planar graphs as well. Usually, it is unclear a priori if the restriction of planarity makes the problem computationally easier or not. There are at least two problems that come to mind that are NP-complete for general graphs, but solvable in polynomial time for planar graphs. The first is the problem of checking if a graph $G$ contains a bipartite subgraph with at least $K$ edges. The second is the problem of finding a max-cut in a weighted graph. Both these problems were shown to be solvable in polynomial time for planar graphs by Hadlock [20], even though for general graphs they are NP-complete [10].

On the other hand, many NP-complete problems, e.g., Hamiltonian path, 3-coloring, remain NP-complete even when restricted to planar graphs.

Consider the following flow problems. In [2] it was shown that the first two problems remain NP-complete even if we restrict our attention to planar graphs.

1. **Min-edge cost flow:** Given a directed graph $G(V,E)$ and specified sets of vertices $S = \{s_1, \ldots, s_k\}$ and $T = \{t_1, \ldots, t_l\}$, capacity $c(e) \geq 0$, and price $p(e)$ for each $e \in E$, demands $R_1, \ldots, R_l$ (for each sink) and bound $B$. Is there a flow function $f$, such that
   - For each $e \in E$, $f(e) \leq c(e)$.
   - For each vertex $v \in V - (S \cup T)$, the incoming flow is equal to the outgoing flow.
   - The net flow into $t_i$ is at least $R_i$.
   - If $E' = \{e \mid f(e) \neq 0\}$, then $\sum_{e \in E'} p(e) \leq B$.

2. **Integral flow with multipliers:** Given a directed graph $G(V,E)$, specified sets of vertices $S = \{s_1, \ldots, s_k\}$ and $T = \{t_1, \ldots, t_l\}$, multiplier $h(v) \in Z^+$ for each $v \in V - (S,T)$, capacity $c(e)$ for each $e \in E$, demand $R_i$ for sink $t_i$. Is there an integral flow function $f$, such that
   - For each $e \in E$, $f(e) \leq c(e)$.
FLOW IN PLANAR GRAPHS

- For each \( v \in V - \{S, T\} \), \( \sum_{(u,v) \in E} h(v).f((u,v)) = \sum_{(v,u) \in E} f((v,u)) \).
- The net flow into sink \( t_i \) is at least \( R_i \).

3. A circulation in an undirected graph: Given an undirected graph \( G(V,E) \) where each edge \( e \in E \) has both a lower bound \( l(e) \) and an upper bound \( u(e) \) on its capacity, compute a feasible circulation \( f \), i.e., a circulation in which
- for every edge \( e \), \( l(e) \leq f(e) \leq u(e) \).
- flow conservation is maintained in each vertex.

In other words, can the edges be oriented such that a feasible circulation exists? This problem was shown by Itai [24] to be NP-complete. Following the results in [41] (see Section 4.1), this problem can be recast in planar graphs as follows. Consider an undirected planar graph, where each edge is associated with two numbers \( l(e) \) and \( u(e) \). Replace each edge \( e \) by two directed edges, oppositely oriented, with weights \( -l(e) \) and \( u(e) \), such that there are no negative weight cycles in the graph.

Suppose that \( l(e) = 1 \), \( u(e) = k - 1 \) and the graph is bridgeless. Tutte [53] observed that a feasible circulation in this case exists if and only if the dual graph is \( k \)-colorable. (Using our terminology, this follows by taking the potentials on the faces modulo \( k \).) Tutte used this correspondence between circulations and colorings to obtain an equivalent formulation of the four-color problem; he called circulations of this form in undirected graphs nowhere-zero flow. Jaeger [27, 28] showed that every bridgeless graph, (not necessarily planar), has a nowhere-zero flow where \( k = 8 \). This was subsequently shown by Seymour [52] to hold for the case \( k = 6 \) as well and it is open whether it holds for the case \( k = 5 \). (Note that it is NP-complete to decide if a planar graph has a nowhere-zero flow where \( k = 4 \).)

8. Conclusions and open problems

There are still many interesting open questions that are related to flow in planar graphs. Here we mention a few of them.

Computing a maximum flow with many sources and sinks.

We have presented efficient algorithms for the case where the sources and sinks belong to a small number of faces, and for the case where the demands and supplies are given as part of the input. However, this problem, in its most general setting where the sources and sinks belong to an arbitrary number of faces, is still open. 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. In the parallel context, maximum flow in a general network was shown to be P-complete [15] 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 [32, 42]. Since even in the unary capacity case we do not have a deterministic parallel algorithm, this emphasizes the importance of solving the problem in the case of a planar network. Notice that for the restricted case of
a single source, single sink, there exist NC algorithms [22, 29].

Computing a maximum matching in NC in a planar graph.

We mention this problem here since matchings and flows are closely related problems. The situation with computing a perfect matching in planar graphs is very intriguing. Kasteleyn [33] 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., [38, 54]) 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. It follows from the results in Section 4.1 that a perfect matching can be computed in a planar bipartite graph (see also [41]).

Computing a minimum cost circulation in a planar graph.

The minimum cost circulation problem is that of obtaining a circulation of minimum cost in a network whose edges have both capacities and costs per unit of flow. The problem is equivalent to the transshipment problem and has wide applicability to a variety of optimization problems.

The current best algorithm for computing a minimum cost circulation in a general graph is Orlin's algorithm [44]. His algorithm implies an $O(n^2 \log^{1.5} n)$ algorithm for minimum cost circulation in planar graphs. (Frederickson's shortest path algorithm [8] is used). A natural open question is whether the special properties of planar graphs can be exploited to obtain a faster sequential algorithm, or an NC parallel algorithm. (Notice that minimum cost circulation is \#P-complete since maximum flow is a special case of it).

The only algorithm that exclusively deals with the minimum cost circulation in planar graphs is by Imai and Iwano [25] who suggested both weakly polynomial algorithms and parallel algorithms (not NC) based on interior point methods and planar separators. Let $\gamma$ denote the maximum absolute value of the cost and capacity in the graph. The running time of the sequential algorithm is $O(n^{1.594} \sqrt{n \log(n \gamma)})$ and the parallel time is $O((n \log^3 n \log(n \gamma))$ using $O(n^{1.094})$ processors.

For planar graphs, the cost function can be interpreted as a modular function on the lattice where the cost of a face is defined to be the sum of the costs of its edges traversed in the clockwise direction. Minimizing a modular function over a lattice is a well known problem in operations research and can be solved as follows: compute the minimum cut (max flow) in a network obtained from the directed acyclic graph in which there is a 1-1 correspondence between lattice elements and closed subsets. However, this approach does not directly yield a new polynomial-time min-cost circulation algorithm, since the directed acyclic graph is too large.

We conclude by noting that the minimum cost circulation problem in planar graphs can be cast as a linear program where the variables are the potentials on the face.

The case of vertex capacities
FLOW IN PLANAR GRAPHS

We have shown a simple reduction for computing the minimum cut in a graph with capacitated vertices to a graph that has only edge capacities. However, this reduction holds only if there is one source and sink. If there are many sources and sinks, then it is not true that the minimum cut is equal to a collection of cycles of minimum capacity that separates the sources from the sinks in $D'$, i.e., with "jumping over faces". The reason is that two cycles in this collection are not necessarily "independent" (if they share a common capacitated vertex). We conjecture that if there are many sources and sinks, then a simple reduction of the above form does not exist.

It seems that the major difficulty with vertex capacities is in computing the flow function. Suppose that we want to compute the flow function via a potential function in a similar way to [41]. As already pointed out, even if we use "jumping over faces" for computing the potential function, we do not necessarily get a legal circulation. (See Fig. 4). To obtain a legal circulation, a set of spurious cycles has to be identified and canceled. Can these cycles be efficiently identified? If the graph contains only one source and sink, then the spurious cycles have a more special structure. In every spurious cycle, the flow on an edge needs only to be decreased and never increased. Can the spurious cycles in this case be efficiently identified? In the case of undirected graphs with a single source and sink, our algorithm is slower than that of [22]. We conjecture that the special structure of the spurious cycles will enable them to be canceled easily.

In the case of $st$ graphs, the cycles have a special structure that is exploited by the parallel algorithm. We conjecture that a deterministic $O(n\sqrt{\log n})$ algorithm exists to compute the flow function for $st$-graphs that works by canceling these spurious cycles.

Another natural open problem is how to compute the flow function in parallel. We can do that only for $st$-graphs. Can that be done for more general classes of planar graphs? How difficult is it to compute a circulation in parallel (with vertex capacities)?

REFERENCES

2. D. Boneh, S. Khuller and J. Naor, Flow problems that are NP-complete for planar graphs, manuscript.