# Flow in Planar Graphs with Vertex Capacities [*]

*Samir Khuller* [†]
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

*Joseph Naor* [‡]
Computer Science Department
Stanford University
Stanford, CA 94305-2140

**Abstract**

Max-flow in planar graphs has always been studied with the assumption that there are capacities only on the edges. Here we consider a more general version of the problem when the vertices as well as edges have capacity constraints. In the context of general graphs considering only edge capacities is not restrictive, since one can reduce the vertex capacity problem to the edge capacity problem. However, in the case of planar graphs this reduction does not maintain *planarity* and cannot be used. We study different versions of the planar flow problem (all of which have been extensively investigated in the context of edge capacities).

[*]A preliminary version of this paper appeared in the *Proceedings of the* 1st *Integer Programming and Combinatorial Optimization Conference*, Waterloo, Canada (May '90), pp. 367-383.

[†]Currently supported by NSF grant CCR-8906949. Part of this research was done while this author was visiting the IBM Thomas J. Watson Research Center and was supported by an IBM Graduate Fellowship at Cornell University.

[‡]Supported by contract ONR N00014-88-K-0166.

## 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., [GTT]. In this paper we concentrate on flow in planar graphs. Research on planar flow is motivated by the fact that more efficient algorithms, both sequential and parallel, can be developed by exploiting the planar structure of the graph. This is important, in particular for parallel algorithms, since maximum flow in general graphs was shown to be P-complete [GSS]. 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, and hence 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 a single source vertex $s$ and a sink $t$. 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 work by Ford and Fulkerson [FF] who developed an $O(n^2)$ time algorithm for the special case of $st$-graphs (when the source and sink are on the same face). This algorithm was later improved to $O(n \log n)$ time by [IS]. By introducing the concept of potentials, Hassin [Ha] gave an elegant algorithm that runs in $O(n\sqrt{\log n})$ time using Frederickson's shortest path algorithm [Fr]. Itai and Shiloach [IS] 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. This algorithm was improved by Reif [Re] who gave an algorithm to find the value of the max-flow in $O(n \log^2 n)$ time. Hassin and Johnson [HJ] 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 a $O(\log n)$ factor, by giving faster shortest path algorithms [Fr]. The problem of finding a minimum cut in a directed planar graph turned out to be much harder and was first solved by [Jo] (both sequentially and in $NC$) who also gave algorithms to compute the flow function. Recently, Miller and Naor [MN] have pointed out that the general maximum flow problem in planar graphs is when there are many sources and sinks. They showed that when demands 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. Note that one cannot reduce the multiple source-sink problem to the single source-sink version since the reduction may destroy planarity.

In this paper we consider the version of the problem in which the vertices as well as edges have capacity constraints. Vertex capacities may arise in various contexts such as computing vertex disjoint paths in graphs [KS], 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 [BM, page 205] from which the violation of planarity is obvious.)

We show how to exploit the structure of the planar graph to develop efficient algorithms for the problem.

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 because most other formulations of the flow problem can be easily reduced to this problem. An example where this is not true is the problem of finding the maximum flow between each pair of vertices. The famous result of Gomory and Hu [GH1] does not hold for graphs with vertex capacities because the reduction from vertex capacities to edge capacities results in edges that are not symmetric. However, Granot and Hassin [GH2] showed how to extend the Gomory-Hu cut tree to graphs with vertex capacities.

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 [RBK] 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 626]. The algorithm given by [RBK] has a running time of $O(n^2 \log n)$ where $n$ is the number of grid points. Our algorithms improve over this result by an $O(\sqrt{n})$ factor. The reader is referred to [BL, CT, GG, RB, RBK] for more details and bibliography of this problem and on the connection between flow problems and reconfiguration. (The main concern of [RB, BL] is the single-track model.)

We develop algorithms for computing a minimum cut when the graph has vertex and edge capacities. When the graph has only edge capacities the minimum cut corresponds to a cycle in the dual graph that separates the source from the sink. With vertex capacities the minimum cut consists of both edges and vertices. We show that for the single source-sink case, the minimum cut corresponds to a "cycle" in the dual graph when "jumping" over faces is permitted. Our algorithms are as fast as the corresponding algorithms for computing the minimum cut with only edge capacities. For the case of $st$-graphs we obtain an $O(n\sqrt{\log n})$ algorithm for finding the minimum cut (flow value). For the case of finding an $s-t$ minimum cut in an undirected planar graph, we are able to extend the algorithm by [IS, Re], to obtain an $O(n \log n)$ algorithm for finding the value of the max-flow even when the graph has vertex capacities. To find the minimum cut in a directed planar graph is more expensive and costs $O(n^{1.5} \log n)$ [MN]. Some of our algorithms also parallelize, yielding efficient $NC$ algorithms.

For the case of $st$-graphs we obtain an $O(n \log n)$ algorithm to compute the flow function. We also give an algorithm that can be implemented in $O(n\sqrt{\log n})$ time after one call to a sorting procedure. Thus we can obtain a randomized $O(n\sqrt{\log n})$ time algorithm by using the randomized sorting algorithm of [FW]. This algorithm also parallelizes, with a running time of $O(\log^3 n)$ using $O(n^2)$ processors on an EREW PRAM. The multiple source-sink problem with given demands reduces to the circulation problem by a modified reduction of [MN]. To obtain a circulation we use the planar separator theorem to develop a divide and conquer algorithm that utilizes the $st$-graph case as a subroutine. The complexity of the algorithm is $O(n^{1.5} \log n)$.

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 deals with the problem of computing the min-cut in a planar graph, with the subsections dealing with the various cases considered. Section 4 provides the algorithms to actually compute the flow function (the flow through each edge). Section 5 concludes with some open problems.

## 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 $\mathcal{D} = (F, D(E))$ be the *dual graph* of $G$, where $F$ is the set of faces of $G$ and $D(E) = \{D(e) | e \in E\}$. 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 $v$, $in(v)$ refers to the arcs that are carrying incoming flow to vertex $v$. Similarly $out(v)$ 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 $S = s_1, \ldots, s_l$ and $T = t_1, \ldots, t_k$ 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 $f : E \rightarrow Z$ is a legal flow function if and only if:

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

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

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

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 to be the capacity of the corresponding primal edge. The capacity of an edge may sometimes also be referred to as its *cost*.

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 $a \leq b$.

Miller and Naor [MN] reduced the problem of finding a flow in a graph with multiple sources and sinks (with specified demands), to the general problem of computing a circulation in a graph. We will concentrate on the circulation problem too, since their reduction can be modified appropriately to work even when vertices have capacities.

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.

A *potential* function $p : F \to Z$ 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 2.1:** *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.

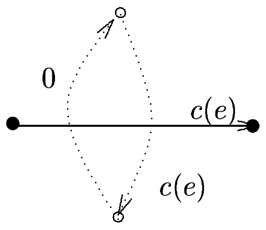
A potential function is defined to be *edge consistent* if the potential difference over each edge is not larger than its capacity. Such a potential function induces a circulation in the graph. If the circulation satisfies the vertex capacities as well, the potential function is defined to be *consistent*. The use of a potential function as a means of computing a flow was first suggested by Hassin [Ha], and was later elaborated by [HJ] and [Jo]. Miller and Naor too, use the idea of potentials to solve the problem of computing the circulation.

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. A EREW PRAM does not allow simultaneous access by more than one processor to the same memory location.

## 3. Computing the minimum cut

In this section, we show how to compute the minimum cut when vertex constraints are introduced into a graph containing a single source and sink. The problem of computing the minimum cut more efficiently (than in a general graph) when there are many sources and sinks is still open even with only edge capacities.

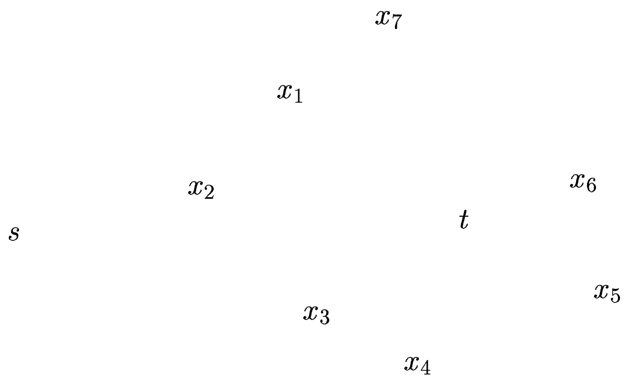
We first explain how weights are assigned to dual edges in a directed graph. Given a primal edge $e$ of capacity $c(e)$, it has two dual edges corresponding to it: one is directed according to the left hand rule and has cost $c(e)$; the other is in the converse direction and has cost 0. (See



○ Nodes of the dual graph

● Nodes of the primal graph

Figure 1: Dual graph in case of a directed graph



$C = [x_1, x_2, ..., x_7]$

$x_2, x_4, x_5, x_7$ are faces in the dual graph

figure=f1.ps $x_1, x_3, x_6$ are edges in the dual graph

Figure 2: Cycle in the Dual Graph

Fig. 1). For an undirected primal edge we introduce an undirected edge in the dual graph of cost $c(e)$.

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 3.1:** *We are given a planar dual graph $\mathcal{D}$ 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 $[x_1; x_2; \ldots; x_k]$ so that each $x_i$ and $x_{i+1}$ 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 $t$. We now show how to modify the dual graph so that such a cycle can be computed efficiently.

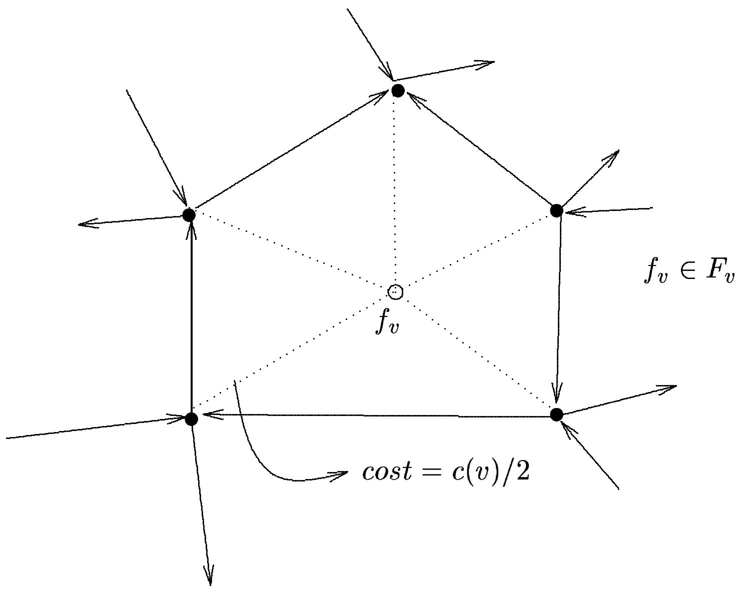


Figure 3: A face in the Dual Graph

**Reduction:** Given a planar dual graph $\mathcal{D}(F, D(E))$ with costs on the edges and faces, we compute a new planar graph $\mathcal{D}'(F', D(E)')$ as follows: Let $F' = F \cup F_V$, where each vertex in $F_V$ corresponds to a face of $\mathcal{D}$ (that corresponds to a vertex in $G$). (Essentially for each face in $\mathcal{D}$ we introduce a new vertex in $\mathcal{D}'$.) Introduce edges in $\mathcal{D}'$ from $f_v \in F_v$ to every vertex on the corresponding face. Each of these edges is given a cost of $c(v)/2$ where $c(v)$ the cost of the corresponding face $v$. These edges are undirected and can be traversed in both directions. Thus $D(E)' = D(E) \cup \{(f_v, f_i) \mid f_i \in F$ and $f_i$ is on the face corresponding to $f_v\}$ (see Fig. 3). Clearly, $\mathcal{D}'$ is planar since $f_v$ can be introduced in the corresponding face of $\mathcal{D}$. The cost of the minimum length path between two vertices incident on a face of $\mathcal{D}$ is clearly $\leq c(v)$ (by going through $f_v$).

**Lemma 3.2:** *The minimum cut in $G$ is given by the length of the shortest cycle in $\mathcal{D}'$ that separates $s$ from $t$.*

*Proof:* We show that a cycle in $\mathcal{D}'$ corresponds to a cut (with vertices and edges) in $G$. A cut that is an $s - t$ cut corresponds to a cycle that separates $s$ from $t$. Clearly, the minimum cut corresponds to such a cycle of least length.

Consider a cycle $C'$ in $\mathcal{D}'$. It is easy to see that $C'$ in $\mathcal{D}'$ corresponds to a linked cycle $C$ in $\mathcal{D}$ (of edges and faces). The vertices (edges) in $C'$ that are not vertices (edges) in $\mathcal{D}$ correspond to the faces in $\mathcal{D}$ that are in $C$. The edges of linked cycle $C$ correspond to edges of $G$ that are in the cut, and the faces of $C$ correspond to the vertices of $G$ that are in the cut. Clearly, the faces of $\mathcal{D}$ (vertices of $G$) in the interior of $C$, are separated from the faces in the exterior of $C$. Thus the linked cycle $C$ corresponds to a cut that separates $s$ from $t$. □

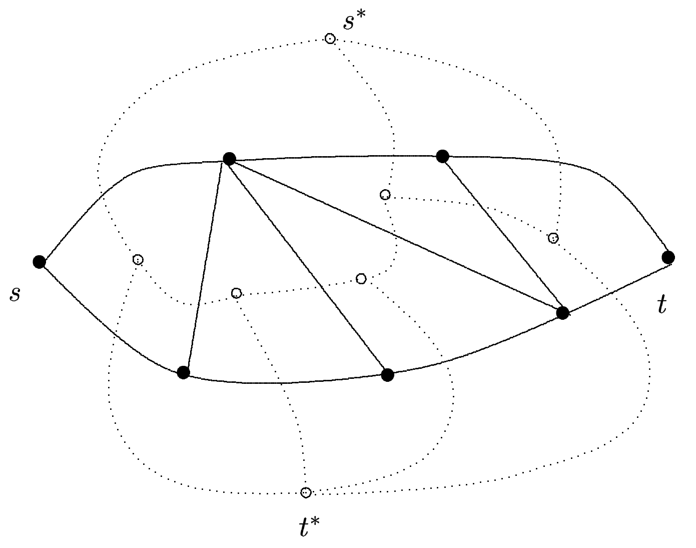
We have now reduced the problem of finding the minimum cut in a planar graph with vertex capacities to that of finding the minimum length cycle separating $s$ from $t$ in a new planar graph, $\mathcal{D}'$, that has only edge capacities. The efficiency of computing this cycle varies with respect to whether the source and sink are on the same face, or whether the graph is directed. In the following subsections we handle the different cases. In Section 3.1 we deal with the case of an $st$-graph (undirected and directed). In Section 3.2 we deal with the case of undirected graphs, when there is no restriction on the location of $s$ and $t$. It turns out that the known algorithms in the literature can handle the computation in the reduced graph $\mathcal{D}'$. In Section 3.3 we deal with the case of a directed graph. The known method for computing minimum cut [MN] has to be modified. (This is because vertex capacities cause an altered structure to the min-cut.)

## 3.1. The case of $st$-graphs

We shall begin by concentrating on the special case of $st$-graphs, when both the source and the sink are on the same face. We will assume this face to be the infinite face. Essentially we introduce two dual vertices $s^*$ and $t^*$ corresponding to the infinite face (see Fig. 4) and construct the modified graph $\mathcal{D}'$ as defined earlier.

The value of the maximum flow is given by the length of the shortest path from $s^*$ to $t^*$. This corresponds precisely to the *minimum cut* between $s$ and $t$.

For example, in Fig. 5, the three directed primal edges shown in the figure have edge capacities of $c_1, c_2, c_4$ respectively. The primal vertex has capacity $c_3$. The min-cut consists of two forward edges and a vertex, that have capacities of $c_1, c_4$ and $c_3$ respectively. The length of

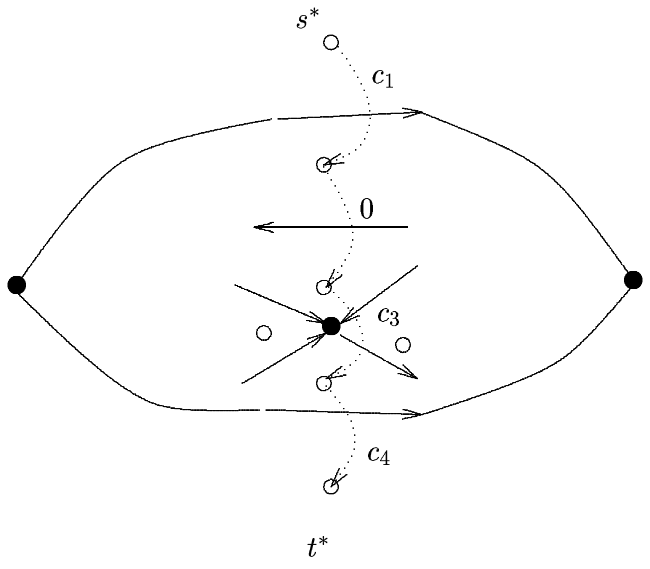


○ Nodes of the dual graph

● Nodes of the primal graph

Figure 4: The dual graph with edge/face costs for the undirected $st$-graph case

the path in the directed dual graph can easily seen to be $c_1 + c_3 + c_4$ (the edge with capacity $c_2$ contributes 0 to the length of the path since it is in the opposite direction). Running a shortest path algorithm on this graph will yield the shortest linked cycle, and that will yield the min-cut separating $s$ from $t$.



| | |
|---|---|
| ○ | Nodes of the dual graph |
| ● | Nodes of the primal graph |

Figure 5: Min-cut in the case of a directed $st$-graph (edge costs refer to costs on dual edges)

Using Frederickson's algorithm [Fr], the shortest path from $s^*$ to $t^*$ can be found in $O(n\sqrt{\log n})$ time. To implement the same algorithm in parallel we use the algorithm by [PR] for computing shortest paths in planar graphs. The algorithm runs in $O(\log^3 n)$ time and uses $O(n^{1.5})$ processors on the EREW PRAM model.

**Theorem 3.3:** *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.*

## 3.2. The single source-sink case for undirected graphs

In [Re] 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, we obtain a running time of $O(n \log n)$. We note that by using the "jumping" over faces idea (apply the algorithms of Reif and Frederickson to $\mathcal{D}'$ instead of $\mathcal{D}$), we get an $O(n \log n)$ time algorithm for computing the minimum cut in the graph even when the vertices have capacities.

## 3.3. Matrix method to compute the minimum cut in a directed graph

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 [Jo]. Recently, an elegant technique was developed by [MN] to find the minimum cut. We show that an appropriate modification of the method is able to find the minimum cut even when vertex capacities are present.

Assume that the reduced graph $\mathcal{D}'$ has been computed. The minimum cut in $G'$ corresponds to the shortest cycle in $\mathcal{D}'$ that separates $s$ from $t$. We first show how to test whether the length of the shortest cycle separating $s$ from $t$ in $\mathcal{D}'$ is greater than or equal to some value $f$. To do that, in $G'$, a directed path $P$ is added from $t$ to $s$, such that the capacity of every edge in $P$ is $[f, f]$. Intuitively, the path $P$ carries the "return flow" from $t$ to $s$; the fact that the lower bound on every edge in $P$ equals the upper bound assures us that $f$ units of flow are indeed returned to the source. The graph with the return flow is denoted by $G''$ and its dual by $\mathcal{D}''$. In the dual graph, for an edge with capacity $[f, f]$ we add an edge of cost $f$ by the usual left hand rule (rotate the edge clockwise), and add an edge of cost $-f$ by rotating the edge anti-clockwise. (Like in Fig. 5, except that we replace the 0 by $-f$.)

Let $C$ be a cycle separating $s$ from $t$ in $\mathcal{D}'$. In the dual graph $\mathcal{D}''$, the length of every such cycle $C$ is reduced by $f$ units. However, this will only happen if $P$ does not meet $C$ in a capacitated vertex; otherwise, the "jumping over faces" may ignore the negative capacity introduced by $P$ (see Fig. 6). Notice that if $c(e_1) + c(e_2) - f \geq c(v)$ then the "effect" of the edge with cost $-f$ is completely ignored, and the addition of the path $P$ will *not* reduce the capacity of each $s$-$t$ cut by $f$ (since it is shorter for the cycle to "jump" over vertex $v$).

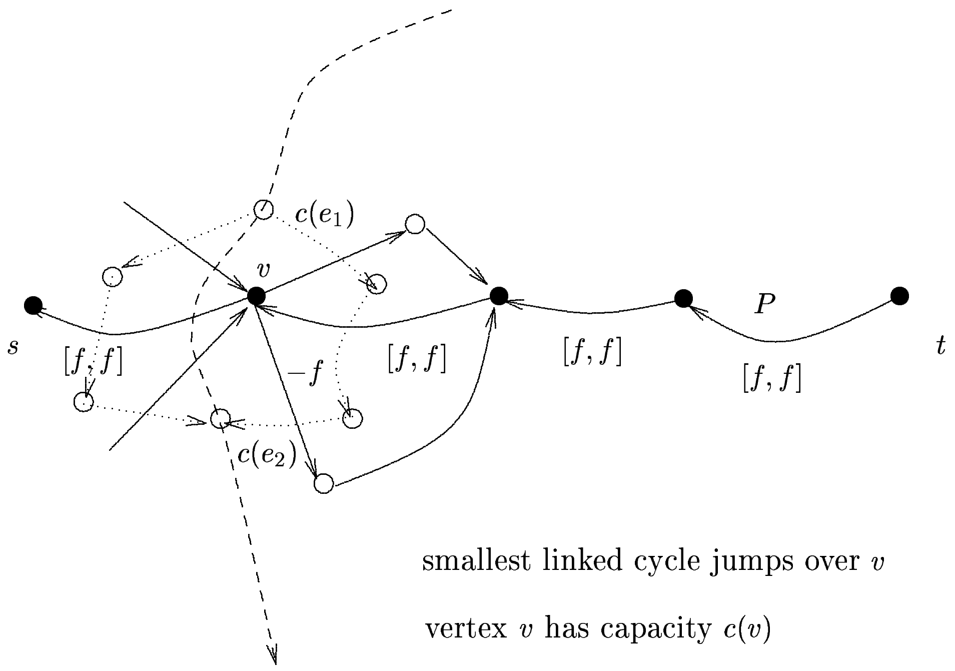
We shall not discuss here the precise details of how the path $P$ is added to the graph $G'$ without meeting capacitated vertices. In Section 4.3 we discuss the more general problem of reducing a flow problem to a circulation problem; the reader is referred to that section for details on how the return path is added.

We summarize with the following lemma.

**Lemma 3.4:** *If the dual graph $\mathcal{D}''$ contains a negative cycle, then the capacity of the minimum cut in $G'$ is strictly less than $f$.*

Our aim is to obtain the largest value of $f$ so that $\mathcal{D}''$ does not have any negative cycles. In [MN] the value of $f$ is found via a parametric search that takes at most $O(\log n)$ iterations, where at each iteration $f$ is updated, and the transitive closure is recomputed to check for negative cycles. To prove the correctness of this algorithm (Lemmas 6.1 and 6.2 in [MN]), two conditions must be met:

- All the negative edges have the same value, i.e., $-f$.



$c(e_1)$
$v$
$P$
$s$
$t$
$[f,f]$
$[f,f]$
$-f$
$[f,f]$
$[f,f]$
$c(e_2)$
smallest linked cycle jumps over $v$
vertex $v$ has capacity $c(v)$

Figure 6: Example to show what goes wrong when the return path is routed through a capacitated vertex

- All the negative edges form a directed path from $t$ to $s$.

Since these conditions are satisfied in $G''$, we can apply this parametric search. It is further shown in [MN] how to implement the parametric search using the method of nested dissection of [LRT]. We obtain:

**Theorem 3.5:** *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.*

## 4. 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 Section 4.1 we present an $O(n \log n)$ implementation of the "uppermost path" algorithm due to Ford and Fulkerson [FF] that handles vertex capacities as well. In Section 4.2 we give 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. A sequential implementation of the parallel algorithm takes $O(n\sqrt{\log n})$ time without counting the time for the step that requires sorting. (Thus we could obtain an $O(n\sqrt{\log n})$ time randomized algorithm by using the fast randomized sorting algorithm due to [FW] that runs in $O(n\sqrt{\log n})$ expected time.) We describe our algorithms for undirected graphs, the extensions to the directed graph case are straightforward using the modified dual graph for directed graphs (See Fig. 1).

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 in Section 4.3 similarly to [MN] by "piping" the flow back from the sinks to the sources, via a path that must not go through any capacitated vertices.

In Section 4.4 we present an efficient algorithm for computing a circulation when edges have lower bounds and vertices are capacitated. 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.

## 4.1. Computing the flow function in $st$-graphs

Hassin [Ha] showed that it is possible to compute a maximum flow function (for the edge capacity case) as follows: for each edge $(i, j) \in E$, let $(i', j') \in D(E)$ be the associated dual edge.

**Definition 4.1:** *The potential $u(f)$ 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: $f(i, j) = u(j') - u(i')$. This yields an edge consistent potential function since $u(j') \leq c(i, j) + u(i')$ (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.

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 $\mathcal{D}'$ (with jumping over faces), yields a flow function that may violate vertex capacity constraints. We illustrate the inconsistency of the potential function by an example in Fig. 7. All the capacities of the edges that are not shown are considered to be high. The min-cut is due to the edge incident at $t$ of capacity 1, and the vertex $v$ of capacity 2. The rest of the potentials are as shown on the faces of the graph. Note that $v$ is the vertex that has excess flow through it. The cancellation of the "spurious cycle" of length three $(v - v_1 - v_2 - v)$, of unit flow, rooted at $v$ has the effect of producing a valid flow function, without changing the value of the max-flow.

We now 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 [IS] to find the directed uppermost path in each

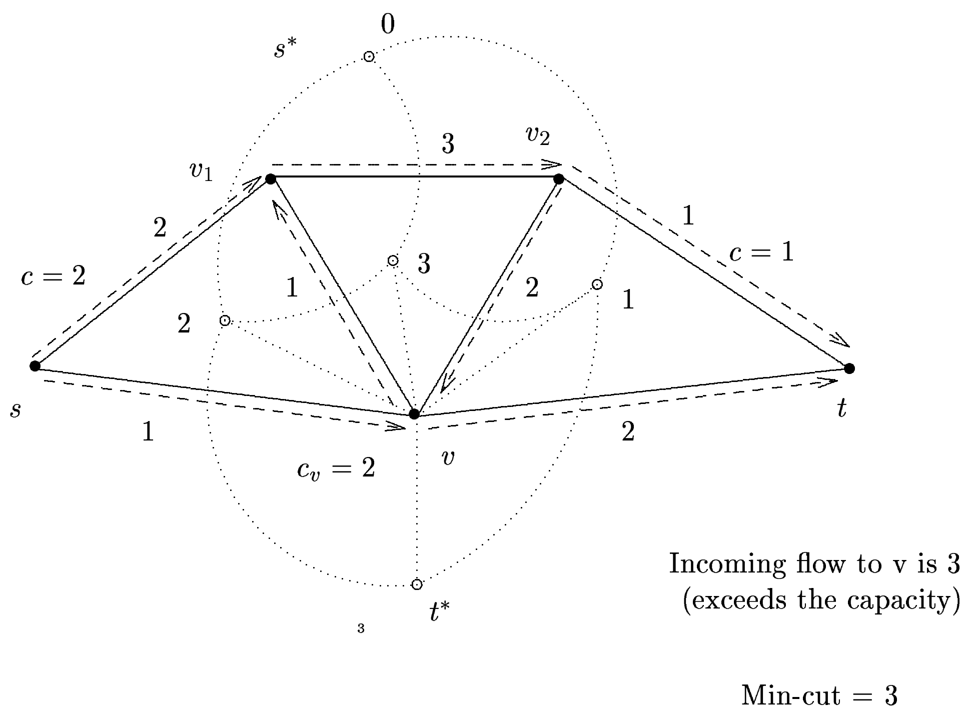


Figure 7: Example to show violation of vertex capacities

iteration. For details on the uppermost path algorithm we refer the reader to [IS] and [FF] (see also [NC]).

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. 8).

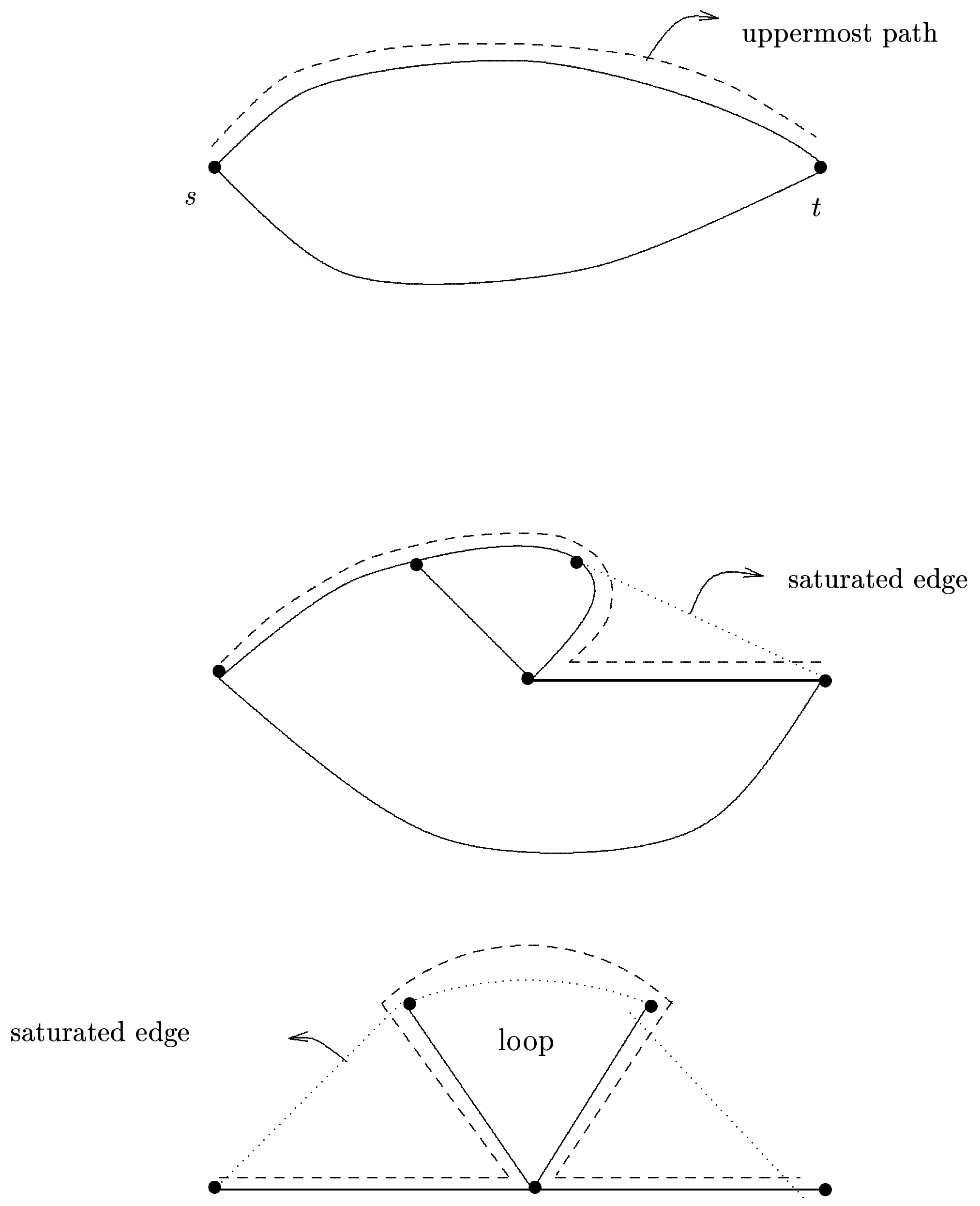


Figure 8: Uppermost path may be non-simple

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 [IS]. 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.

**Proposition 4.2:** *The process of making the augmenting path simple at each step does not decrease the amount of flow pushed on that augmenting path.*

**Implementation:** The capacities of the edges and vertices are kept in a heap. At each step we choose the minimum from the heap, which defines the amount of flow to be pushed at that step. After deleting the appropriate edge or vertex we need to update the uppermost path to make it simple. If the bottleneck was due to an edge, this is done by simply traversing the edges on the face adjacent to the edge (the face other than the external face) and introducing them into the heap. During this traversal, we can detect if any of the vertices already belong to the uppermost path and are causing a non-simplicity. If this condition is detected the entire sub-chain causing the non-simplicity is discarded and the corresponding values from the heap are deleted. As can be seen in Fig. 8, not just a sub-chain but the entire portion of the graph that lies in the "loop" needs to be deleted. However, note that this is done only once for each edge/vertex (when they are introduced into the uppermost path); once they are deleted, they remain deleted, and are never encountered again. If the bottleneck on the path is due to a vertex $v$, to recompute the new uppermost path we need to traverse the faces adjacent to $v$ (clockwise from the forward edge of the path incident on $v$). During the traversal we ensure that the new path is simple and discard all the edges and vertices that belong to the segment of the path causing the non-simplicity.

**Theorem 4.3:** *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.*

*Proof:* An $O(n^2)$ implementation of the algorithm is trivial. There are at most $O(n)$ augmenting paths (uppermost paths) because at each augmentation step we delete an edge or a vertex. Even though the entire description was for undirected graphs, the algorithm can be easily modified to work for the case of directed graphs (see [IS]). Finding the bottleneck edge and updating the flow through each edge can easily be done in $O(n)$ time. Using a simple trick developed in

[IS], we can keep the capacities in a heap that makes it easy to find the bottleneck edge/vertex. Moreover, when introducing exposed elements into the heap we adjust their capacities in a manner that avoids the need to update the residual capacities of the edges/vertices already in the heap. Each element is deleted at most once and thus we get a running time of $O(n \log n)$. We leave the details to the interested reader. □

As mentioned earlier, essentially the same algorithm works for the case of directed graphs. The main modification is that we need to obtain the uppermost directed path in each iteration, and to keep it simple in each step.

## 4.2. 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 first give an informal overview of the algorithm and then proceed to outline it in detail. 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. 9 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 $(i+1) \pmod{d(v)}$ 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. 10 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 $c_v$.

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"

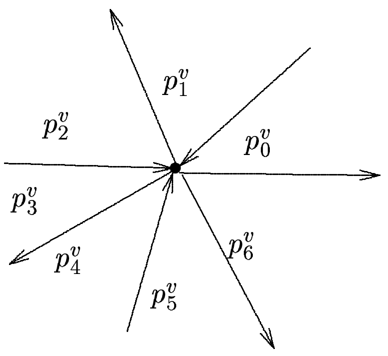


Figure 9: A capacitated vertex and the potentials of its incident faces

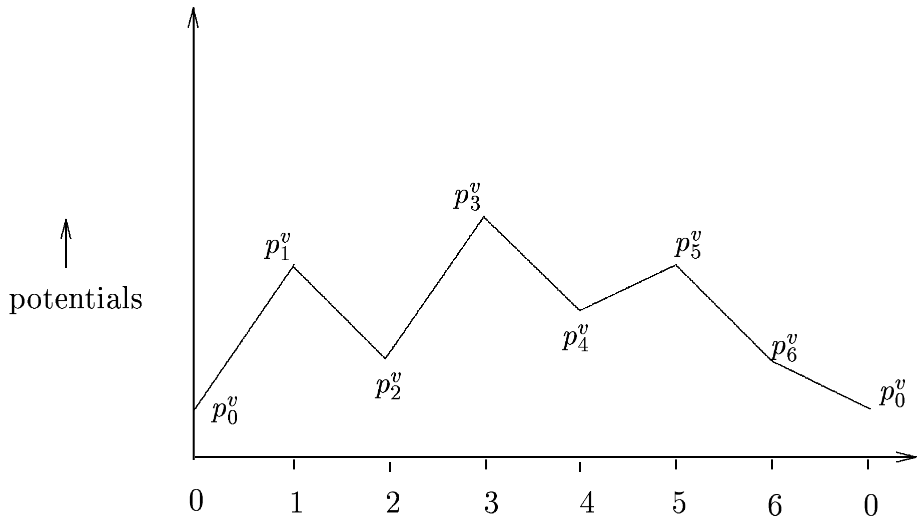


Figure 10: Potentials of faces incident on $v$

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 outline the algorithm in more detail. We first describe a sequential implementation of the algorithm and then show how it can be implemented in parallel.

**Notation:** By $f_v$ we denote the face corresponding to dual vertex $v$. Assume that all the potentials have been sorted in non-decreasing order $p_1, p_2, \ldots, p_{|F|}$, where $\mid F \mid$ is the number of faces. ($p_1$ is the potential of $s^*$ and is 0.)

**Definition 4.4:** *Define* $T_D^i = \{j \mid j \leq i\}$. $T_D^i$ *consists of the* $i$ *vertices in* $T_D$ *that have the smallest potential values.*

**Definition 4.5:** *Define* $R^i = \bigcup\{f_j \mid j \leq i\}$. $R^i$ *is the union of the set of faces corresponding to vertices in* $T_D^i$.

We refer to the boundary of $R^i$ as $\delta R^i$ (this can easily be computed from the embedding of $T_D$).

**Algorithm for the $st$-graph case:**

*Step 1.* Add a directed edge of infinite capacity from $t$ to $s$. This has the effect of "splitting" the infinite face into two faces (see Fig. 11). The dual vertex corresponding to the infinite face is $s^*$, and the new face (vertex in the dual graph) formed by the addition of the return edge is called $t^*$.

*Step 2.* Construct the dual graph (with edge costs and face costs) and compute the shortest path tree $T_D$ in the dual graph. (This can be done by reducing the problem to a regular shortest path problem as shown in Section 3.) When there are face costs, the "parent" of a vertex $v$ in the shortest path tree is not necessarily adjacent to $v$. (But must share a common face with $v$.)

*Step 3.* There are $f-1$ phases; in the $i^{th}$ phase do:
Identify the non-simplicities in $\delta R^i$ that were not present in $\delta R^{i-1}$. (In Fig. 11 the shaded region is $R^i$. $R^{i-1}$ is $R^i - i$.) If the addition of $i$ to $R^{i-1}$ created a non-simplicity then continue, else the phase ends.
If $i$ is not marked "done" then define $loop_i$ to be the finite region enclosed by the non-simple portion of $\delta R^i$ (see Fig. 11). Also Redefine the potential of all the faces in $loop_i$ to be $p_i$. Mark all the dual vertices in $loop_i$ as "done".

*Step 4.* Define the flow on each edge to be the difference of the potentials of the adjacent faces. As before, the convention is that the flow moves in the direction with the smaller potential on the left.

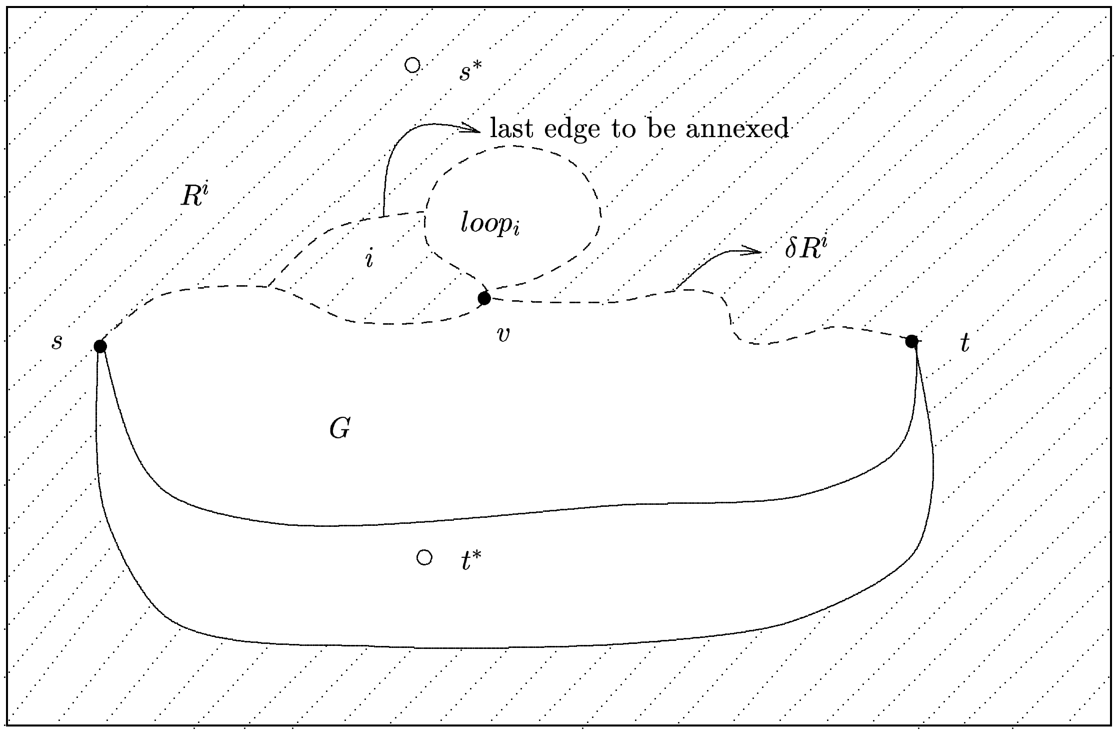


Figure 11: Region $R^i$ and a "loop"

We briefly comment on the implementation of Step 3, and then show why the algorithm works.

**Implementation of Step 3:** If we have a list containing the vertices that belong to $\delta R^{i-1}$, then it is easy to walk around the vertices of face $p_i$ and obtain the list for $\delta R^i$. When doing this walk we can also detect if we traverse any vertices that are already on $\delta R^{i-1}$. This will detect non-simplicities in the boundary. Once this is detected, it is easy to traverse the nodes in $loop_i$ and mark them as "done". (A vertex is set to "done" once we are sure that its potential does not need to be changed again.)

In general, $R^i$ is the set of faces that have been "reached" by the dual shortest path tree at the start of phase $i$. Each time an edge on the uppermost path is saturated, it corresponds precisely to growing the region $R^i$ by annexing a new face (the next "closest" face from $s^*$). Each uppermost path corresponds directly to a fringe of the dual tree $T_D^i$ at the snapshot of the algorithm in phase $i$. We identify the various augmenting paths and make them simple at each step by completely discarding the "loops" as was done in the uppermost path algorithm.

**Correctness of the Algorithm:** We now prove that the algorithm obtains a valid flow function.

**Lemma 4.6:** *Setting the potentials of all the faces in $loop_i$ in $T_D^i$ to be $p_i$, preserves edge capacities.*

*Proof:* The flow on all the internal edges of the loop becomes zero since the potentials of all faces is $p_i$. Notice that the flow on the edges of the boundary of the loop is only decreased; since the higher potentials (of faces in the loop) are lowered, this keeps the flow to be less than the capacity of the corresponding edge. □

Reducing the potential has the effect of "deleting" the loop and making the augmenting path simple. It also has the effect of making all the unhappy vertices in the interior of the loop happy.

In this manner we can identify all spurious cycles and cancel them, obtaining a flow function that does not violate any capacitated vertices.

The loops are canceled in the first dual tree that they occur (i.e., a loop is canceled in $T_D^i$, if it is not contained in a loop in $T_D^j$ for $j < i$).

**Definition 4.7:** *In a circular sequence a local maximum is a contiguous set of identical values (from position $i$ to $j$), that are greater than the values at positions $i-1$ and $j+1$. We refer to the contiguous set of values forming a local maximum as a locally maximal chain.*

**Example:** Consider a sequence of values $5, 5, 4, 4, 6, 6, 6, 3, 5, 5$. In this sequence there are two local maximum, one is the consecutive chain of $6'$s at positions $5, 6$ and $7$. The other is the consecutive chain of $5'$s at positions $9, 10, 1, 2$. These are also the locally maximal chains.

After all the potentials have been adjusted by the above algorithm (Step 3) we can show the following property:

**Lemma 4.8:** *For a capacitated vertex $v$, if we consider the sequence of potentials of faces incident on it (in anticlockwise order) starting from the smallest potential, there is only one local maximum.*

*Proof:* If $v$ belongs to the interior of some loop, then the property is trivially true as all the potentials of the faces incident on it are the same. Suppose that there are two locally maximal chains (the proof is similar for more than two such chains). Let the potentials of the two chains be $p_{v_1}$ and $p_{v_2}$. The faces in-between these two faces (from both sides) all have lower potential. Let us assume that $p_{v_1} \leq p_{v_2}$. Suppose that in the potential sequence $p_{v_1}$ has position $i$. If we consider the dual tree $T_D^{i-1}$ then there will be a non-simplicity formed in the boundary. This is because the faces corresponding to the two maximal chains are not in $R^{i-1}$, and are separated by faces that belong to $R^{i-1}$. (The faces in between all have lower potential than $p_{v_1}$.) Either, $p_{v_1}$, or $p_{v_2}$, is in the loop formed due to the non-simplicity, which would have caused a lowering of one of $p_{v_1}$ or $p_{v_2}$ by Step 3 of the algorithm, thus giving a contradiction. □

**Theorem 4.9:** *The algorithm described above obtains a valid flow function, satisfying: (i) edge capacity constraints; (ii) vertex capacity constraints; (iii) flow conservation constraints.*

*Proof:* The first phase produces a flow function that satisfies all the constraints in (i) and (iii). Since the potential of $t^*$ does not change in the second phase we still have a max-flow. The second phase only reduces flow on edges as was shown in Lemma 4.6 so the edge capacity constraints are not violated. The fact that the incoming flow to a vertex is the same as the outgoing flow follows from Proposition 2.1. This proof does not require that the potential function be computed by a shortest path computation, but works for *any* potential values. We now show that all the vertex capacity constraints are met. If the vertex is an interior vertex on any "loop" that is identified in the second phase, then the flow through the vertex is zero and it trivially satisfies its capacity constraint.

Consider a potential sequence of a vertex $v$, $p^v_0, \ldots, p^v_{d(v)-1}$. Let $p^v_y$ be the local maxima (by Lemma 4.8 it is unique). By definition $p^v_0$ is the lowest potential in the sequence. If we traverse the faces anticlockwise, the potentials are all non-decreasing until $p^v_y$ and then non-increasing until $p^v_0$. This follows from the fact that the local maxima is distinct. Since the maximum difference of potentials of any two faces incident to $v$ is bounded by $c(v)$, it is easy to see that all the capacitated vertices do not have any excess flow through them. We can now bound the incoming and outgoing flows by $c(v)$. □

**Parallel Implementation:** Step 1 and 2 are easy to execute by doing a shortest path computation in parallel [PR, MN]. Once we have the tree $T_D$, using $O(n^2)$ processors we can compute all the subtrees $T^i_D$ (by allocating $n$ processors for each $i$). After we compute the subtrees the only remaining step is to obtain the non-simple sections of these subtrees. Since we have $O(n)$ processors for each subtree $T^i_D$, this is easy to compute. Each face computes the smallest $j$ such that it occurs in a loop in $T^j_D$, and adjusts its potential to be $p_j$. Now define the flow on an edge to be the difference of the potentials of its adjacent faces.

We now get the following theorem.

**Theorem 4.10:** *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.*

*Proof:* We use the algorithm by [PR] to compute the shortest path tree in parallel (tree rooted at $s^*$). The second phase is easy to implement in $O(\log n)$ time using $O(n)$ processors for each dual tree $T^i_D$. □

**Alternate Scheme:**

Note that each dual subtree $T^i_D$ constructed corresponds to an augmenting path (via the "fringe" of the tree). Each edge of the graph belongs to some set of augmenting paths. To obtain a legal flow function, we simply have to cancel flow on all the non-simple portions of the augmenting paths (by the amount of flow that was pushed on that augmenting path). For each edge $e$, we can compute the set of augmenting paths it participates in. This can be done by examining all the fringes of the faces of nodes in the snapshots of the dual tree $T_D$. We then compute the set of augmenting paths for which $e$ is on the non-simple part of the boundary and reduce the flow by an amount equal to the capacity of all these augmenting paths. This method would also yield a parallel algorithm of the same complexity.

This algorithm can be used to compute $k$ vertex-disjoint $s$-$t$ paths in a $st$-planar graph in $NC$, for *arbitrary* $k$. The parallel algorithm of [KS] works in $NC$ for only fixed values of $k$ (but for general graphs).

### 4.3. Reduction from flow to circulations

We now show how to transform a flow problem 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. 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.

We first compute a spanning tree $T$ in $G$ (on the sources and sinks). (We treat $G$ as an undirected graph for the purpose of computing $T$.) Notice that if there is only one source and sink, then $T$ is a path. Let us denote the demand of a vertex $v$ by $d(v)$ and assume that if $v$ is a source, then $d(v) < 0$, whereas if $v$ is a sink, then $d(v) > 0$.

An edge $e \in T$ separates the tree into two subtrees $T_1$ and $T_2$. Let $r(e)$ be $\sum_{v \in T_1} d(v) = -\sum_{v \in T_2} d(v)$. The previous equality follows from the fact that $T$ is a spanning tree and the sum of the demands is zero. We will add an edge parallel to $e$ whose capacity is $[\mid r(e) \mid, \mid r(e) \mid]$. The direction of the edge depends on the sign of $r(e)$. If $r(e)$ is positive it points from $T_1$ to $T_2$, otherwise it points from $T_2$ to $T_1$.

In the new graph we will compute a circulation and obtain a legal flow that satisfies the demands by removing the newly added tree edges.

It is crucial for the algorithm in Section 3.3 that return flow edges are inserted without being adjacent to capacitated vertices. Let us now elaborate on how this is done. Assume that $T$ is a simple path. (The procedure easily generalizes to trees). $T$ is inserted edge-by-edge; assume that we are currently inserting the edge $v \rightarrow w$ with return flow $[l, l]$, where $w$ is capacitated and $v$ is not. This condition is initially satisfied as the sources and sinks are not capacitated (see Fig. 12).

Let $u$ be the vertex that succeeds $w$ on the path and let $u^*$ and $v^*$ be any faces adjacent to $u$ and $v$ respectively. Let $e_1^*, \ldots, e_k^*$ be a path in the dual graph from $v^*$ to $u^*$ and let $e_1, \ldots, e_k$ be the corresponding primal edges. The idea is that the path will arrive at $u$ through the edges $e_1, \ldots, e_k$ instead of $w$. To do that, we create a "crossing" by adding a vertex $a_i$. The path from $v$ to $u$ will go through the vertices $a_1, \ldots, a_k$ by adding new edges with capacity $[l, l]$. This is repeated for each edge on $T$. Notice that the last vertex on $T$ is not capacitated. In this manner we are able to "bypass" all the capacitated vertices to return the flow from the sinks to the sources.

Since the degree of each vertex $a_i$ is exactly four, with the flow on the return edges being a fixed value $l$, by conservation of flow the flow on the old edges is not affected, both in value and direction. Hence, when the return edges and the vertices $a_i$ are removed, we still have a legal flow.

This procedure increases the size of the graph by $O(n)$ vertices, since there are $O(n)$ edges of the spanning tree that need to be embedded.

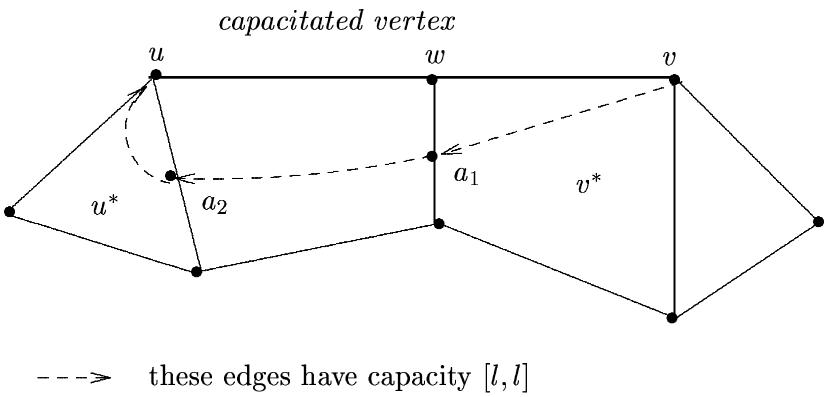


Figure 12: Embedding the edge $v \to w$

## 4.4. Computing Circulations

In this section we show how to use the planar separator theorem [Mi] 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 [JV].

**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 $G$.

Let us now elaborate on each step. In the first step, we compute a separating cycle of size $O(\sqrt{n})$. However, we require a separating cycle that has no capacitated vertices on it. To obtain such a cycle, we first find a separating cycle in the dual graph $\mathcal{D}$. The vertices on this cycle correspond to a set of faces in the primal graph, such that faces corresponding to adjacent vertices on the cycle share a common edge (edges in $E_C$). This cycle can be decomposed into two cycles $C_1$ and $C_2$ and a set of edges $E_C$ between them. We introduce a new vertex in the middle of each edge in $E_C$, and connect a cycle $C$ through the new vertices by adding new edges with zero capacity. (See Fig. 13). Call the new graph $G'$. This cycle is directed clockwise, and its edges have zero capacity.

**Lemma 4.11:** *The cycle $C$ is a separator of size $O(\sqrt{n})$ such that the number of vertices in $G_I$ and $G_E$ is bounded by $cn$ ($c < 1$).*

*Proof:* In the dual graph the size of the separating cycle is $O(\sqrt{f})$ where $f$ is the number of faces in $G$ (and the number of vertices in the dual). We know that $f = 2n - 4$ (if the graph

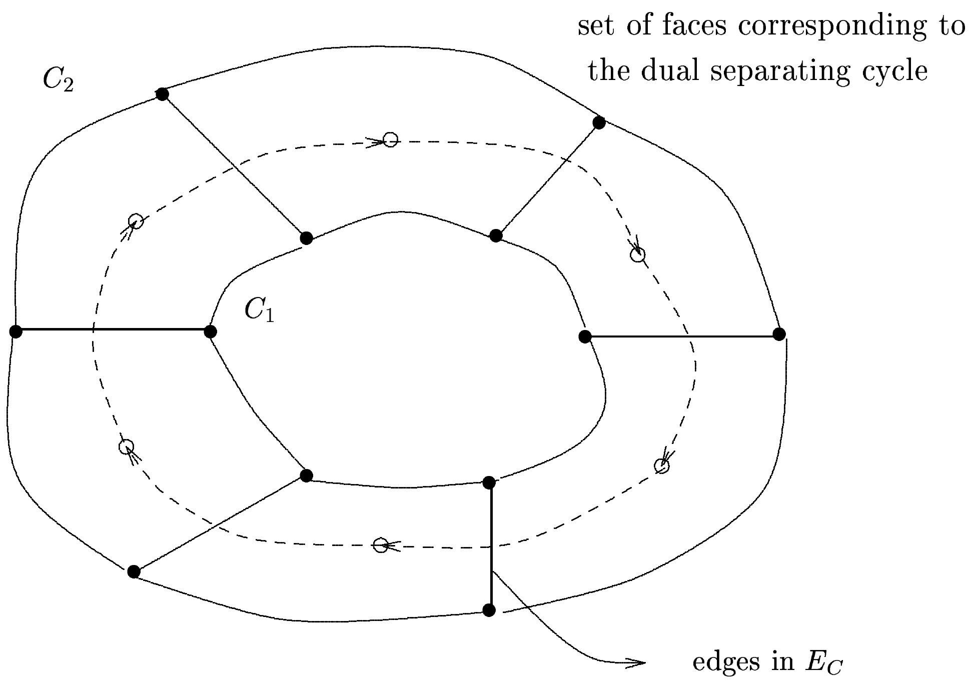


Figure 13: Finding an uncapacitated separating cycle

is triangulated by adding extra zero capacity edges). Hence the separating cycle $C$ has size $O(\sqrt{n})$.

The number of dual vertices in each component of $\mathcal{D}$ after deletion of the cycle is upper bounded by $\frac{2}{3}f$. Thus the number of faces in each component of $G$ after deletion of $C$, is upper bounded by $\frac{2}{3}f$. The number of vertices is about $\frac{1}{2}f + O(1)$. Thus we establish that the number of vertices in each component of $G$ is $\frac{2}{3}n + O(\sqrt{n})$ (due to addition of new vertices to form the cycle). □

**Lemma 4.12:** *Any circulation in $G'$ induces a circulation in $G$.*

*Proof:* Since all the extra edges added in $G'$ have zero capacity, a circulation in $G'$ directly yields a circulation in $G$ by dropping the edges that have zero capacity (and thus no flow). We can also easily drop the vertices that were added to create the separating cycle. □

In the second step, we compute a circulation recursively in $G_I + C$ and $G_E + C$. Since edges have lower bounds, we have to ensure that a feasible solution exists in the graph restricted to $G_I + C$ and $G_E + C$. This is done by giving each edge on $C$ an infinite capacity during the recursive calls.

**Lemma 4.13:** *There exists a feasible solution in $G_I + C$ and $G_E + C$.*

*Proof:* Let us assume that there is a feasible circulation $Z$ in $G$. We show that in $G_I + C$ there is also a feasible circulation after the capacities of the edges have been changed to infinity. (The proof for $G_E + C$ is similar.) Consider the restriction of $Z$ to $G_I$. Clearly all the vertices strictly in the interior of $C$ satisfy the flow conservation requirement (as they are not affected in any way). The only vertices that are affected are the ones on the boundary of the cycle $C$ (the new uncapacitated vertices). Some of them suddenly become deficient in the flow they receive (due to deletion of the edges in the exterior of $C$) and some have excess flow. Since $Z$ was a legal circulation, the sum of the excesses equals the sum of the deficiencies. The edges of $C$ are now used to redistribute all the excesses to the deficient vertices. Hence the graph $G_I + C$ has a legal circulation. □

In the third step, we merge the two solutions obtained previously. The merged flow satisfies the flow conservation constraint for each vertex, but the capacity of the edges of the separator are violated, since their "real" capacity is zero. We now discuss how this is fixed. However, notice that vertex capacities are not violated due to the merging step because the vertices of the cycle $C$ are not capacitated.

Let $e = (v \to w)$ be an edge of $C$ with flow $f_e$. Since it has zero capacity, the flow from $v$ to $w$ needs to be redirected, i.e., $v$ becomes a source that needs to send $f_e$ units of flow to the sink $w$, but not via the edge $e$. Notice that the source $v$ and sink $w$ are on the same face. The redirection is performed in the residual graph (with respect to the current circulation). The edges in the residual graph have no lower bounds. Suppose an edge $e$ has capacity $[a, b]$ in $G$ and flow $f$ in the current solution. In the residual graph there will be two edges, directed in opposite directions, with capacities $b - f$ and $f - a$. In pushing flow in the residual graph, vertex capacities are "active" for only a subset of the vertices, since when we push flow on an edge it is actually either increasing or decreasing the flow through the vertices incident on it. (For example, pushing flow through an edge in the residual graph may be decreasing the actual flow that was pushed in an earlier step; in this case we don't want the vertex capacity to be "active".) The vertex capacity applies only to the situation when the flow through the vertex is increasing. It is easy to see that a simple modification to the algorithm presented in the earlier section will work. This procedure is repeated for each edge on the separator. Observe that if the vertices of $C$ were *capacitated* then it is not possible to simply merge the solutions.

If at any step there is no way of redirecting the flow in the residual graph from $v$ to $w$, then the algorithm halts and claims that there is no circulation in the graph. The correctness of this claim can be easily seen from the following argument. Suppose that there is a circulation $C$ in the original graph, and let $C'$ be the current circulation, in which the flow cannot be redirected from $v$ to $w$. Then, $C - C'$ is a collection of residual cycles such that augmenting them in $C'$ will redirect the flow from $v$ to $w$.

Since the size of the separator is $O(\sqrt{n})$ we obtain:

**Theorem 4.14:** *The complexity of computing a circulation in a planar graph with vertex capacities is $O(n^{1.5} \log n)$.*

*Proof:* The algorithm makes recursive calls to two graphs that are bounded in size by $\frac{2}{3}n + O(\sqrt{n})$ (where $n$ is the number of vertices in the graph). The cost of merging the solutions is $O(n^{1.5}\log n)$ as we make $O(\sqrt{n})$ calls to a procedure that pushes a given amount of flow in an $st$-planar graph. Each call to the algorithm costs us $O(n\log n)$ time (using the algorithm presented in the previous section).

Thus letting the running time be $T(n)$; we have

$$T(n) \leq c_1 \text{ for } n \leq n_0$$

$$T(n) \leq T(n_1) + T(n_2) + c_2 n^{1.5}\log n$$

Note that, $c_1, c_2, n_0$ are constants.

$$n_1, n_2 \leq \frac{2}{3}n + O(\sqrt{n})$$

Hence, we know that

$$n_1, n_2 \geq \frac{1}{3}n + O(\sqrt{n})$$

A similar recurrence is obtained in [JV], but since the proof of the recurrence is not given, we outline it for completeness.

We show that $T(n) = Cn^{1.5}\log n$, for $n \geq n'$. The LHS is

$$(C - c_2)n^{1.5}\log n \leq Cn_1^{1.5}\log n_1 + Cn_2^{1.5}\log n_2$$

We prove this by showing that the LHS is smaller than a lower bound for the RHS (by replacing $\frac{n}{3}$ for $n_1$ and $n_2$).

$$(C - c_2)n^{1.5}\log n \leq 2C(\frac{n}{3})^{1.5}\log\frac{n}{3}$$

$$(C - c_2)\log n \leq \frac{2C}{3\sqrt{3}}(\log n - \ln - \log 3)$$

$$\frac{2C}{3\sqrt{3}}\log 3 \leq (\frac{2C}{3\sqrt{3}} + c_2 - C)\log n$$

This gives us the value of $n'$ (which is a constant). The equation holds for all $n > n'$. □

Notice that our algorithm for computing a circulation is slower than that of [MN] (for edge capacities) by only a logarithmic factor. For the case of computing a max-flow the complexity of our algorithm (Section 3.2 and 4.4) is the same as that of computing a min-cut (the value of the max-flow).

## 5. Conclusions and open problems

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 $\mathcal{D}'$, 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 [MN]. 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. 7). 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 [HJ]. 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) ?

## Acknowledgments

We would like to thank Hal Gabow, Donald Johnson, Gary Miller and Vijay Vazirani for fruitful discussions. We would also like to thank the referees for providing extremely valuable comments on an earlier draft of the paper.

## References

[BL] Y. Birk and J. B. Lotspiech, *A fast algorithm for connecting grid points to the boundary with nonintersecting straight lines*, Proceedings of the $2^{nd}$ Symposium on Discrete Algorithms, pp. 465-474 (1991).

[BM] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*, North-Holland (1977).

[CLR] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to algorithms*, The MIT Press (1990).

[CT] B. Codenetti and R. Tamassia, *A network flow approach to reconfiguration of VLSI arrays*, IEEE Transactions on Computers, Vol. 40, No. 1, pp. 118-121 (1991).

[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. Frederickson, *Fast algorithms for shortest paths in planar graphs, with applications,* SIAM Journal on Computing, Vol.16, pp. 1004-1022 (1987).

[FW] M. L. Fredman and D. E. Willard, *Blasting through the information theoretic barrier with fusion trees*, Proceedings of $22^{nd}$ Annual Symposium on Theory of Computing, pp. 1-7 (1990).

[GG] J. W. Greene and A. El-Gamal, *Configuration of VLSI arrays in the presence of defects*, Journal of the ACM, Vol. 31, No. 4, pp. 694-717 (1984).

[GH1] R. E. Gomory and T. C. Hu, *Multi-terminal network flows*, SIAM Journal on Applied Math, Vol. 9, pp. 551-570 (1961).

[GH2] F. Granot and R. Hassin, *Multi-terminal maximum flows in node-capacitated networks*, Discrete Applied Math, Vol. 13, pp. 157-163 (1986).

[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).

[GTT] A.V.Goldberg, E. Tardos and R.E.Tarjan, Network flow algorithms, in *Paths, Flows and VLSI-Design*, Springer Verlag, pp. 101-164, (1990).

[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 Journal on Computing, Vol. 14, pp. 612-624 (1985).

[IS] A. Itai and Y. Shiloach, *Maximum flow in planar networks,* SIAM Journal 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).

[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).

[KS] S. Khuller and B. Schieber, *Efficient parallel algorithms for testing $k$-connectivity and finding disjoint $s$-$t$ paths in graphs,* SIAM Journal on Computing, Vol. 20, No. 2, pp. 352-375 (1991).

[LRT] R. J. Lipton, D. J. Rose and R. E. Tarjan, *Generalized nested dissection*, SIAM Journal on Numerical Analysis, Vol. 16, pp. 346-358 (1979).

[Mi] G.L.Miller, *Finding small simple separators for 2-connected planar graphs*, Journal of Computer and System Sciences, 32, pp. 265-279 (1986).

[MN] G. L. Miller and J. Naor, *Flow in planar graphs with multiple sources and sinks*, Proceedings of the 30th Annual Symposium on Foundations of Computer Science, pp. 112-117 (1989).

[NC] T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Annals of Discrete Math (32), North Holland Mathematical Studies.

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

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

[RB] V. P. Roychowdhury and J. Bruck, *On finding non-intersecting paths in a grid and its application in reconfiguration of VLSI/WSI arrays,* Proceedings of the $1^{st}$ Symposium on Discrete Algorithms, pp. 454-464 (1990).

[RBK] V. P. Roychowdhury, J. Bruck, and T. Kailath, *Efficient Algorithms for Reconfiguration in VLSI/WSI Arrays*, IEEE Trans. on Computers, C-39:4 (Special Issue on Fault Tolerant Computing), pp. 480-489 (1990).