Flow in Planar Graphs with Vertex Capacities

Samir Khuller*
Joseph Naor**

TR 90-1089
January 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported by an IBM Graduate Fellowship. Part of this research was done while this author was visiting the IBM T.J. Watson Research Center.
**Supported by contract ONR N00014-88-K-0166 and by a grant of Stanford's Center for Integrated Systems.

Flow in Planar Graphs with Vertex Capacities

Samir Khuller *
Computer Science Department
Cornell University
Ithaca, NY 14853

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).

*Supported by an IBM Graduate Fellowship. Part of this research was done while this author was visiting the IBM T.J. Watson Research Center.
†Supported by contract ONR N00014-88-K-0166 and by a grant of Stanford's Center for Integrated Systems.

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. Several efficient algorithms have been developed to solve the problem [GT], [CH]. 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 planar structure of the graph. This is important, in particular for parallel algorithms because maximum flow in general graphs was shown to be P-complete [GSS]. Planar networks arise in many contexts such as VLSI design and communication networks, and 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 $O(n^2)$ algorithm for the special case of st-graphs (defined later). Subsequently, Itai and Shiloach developed an algorithm to find a max flow in an undirected planar graph [IS], which was improved by Reif [Re] who gave an algorithm to find the value of the max-flow in $O(n \log n)$ time. Hassin and Johnson [HJ] completed the picture by giving an $O(n \log^2 n)$ algorithm to compute the flow function. The problem of finding a minimum cut in a planar directed graph was 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). Note that here one cannot reduce the multiple source-sink problem to the single source-sink version since the reduction may destroy planarity. To summarize, if a planar graph is directed, then the complexity of computing the flow function is $O(n^{1.5})$ (assuming that the flow value has been given, which costs $O(n^{1.5} \log n)$ to compute).

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 and in various network situations when the vertices denote switches and have an upperbound 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. We show how to exploit the structure of the planar graph to develop efficient algorithms for the problem.

An application where vertex capacities play an important role is in reconfiguring VLSI/WSI arrays. Assume 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). The reader is referred to [RBK] and [RB] for more details and bibliography of this problem. (The main concern of [RB] is the single-track model).

We develop algorithms for computing the 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 [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 $O(n^{1.5} \log n)$ [Jo] [MN]. 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. 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 $O(n^{1.5} \log n)$.

2. Terminology and preliminaries

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 $D(e)$.

Associate with each edge $e \in E$ a capacity $c(e) \geq 0$, and also with each vertex $v \in V$ 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. 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) $\forall v \in V - \{S, T\} : \sum_{e \in in(v)} f(e) (= \sum_{e \in out(v)} f(e)) \leq c(v)$

Assume that G is biconnected, add edges with zero capacities appropriately to ensure that. Assume also 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 $s' \in S$ adjacent only to s, such that the capacity of the edge joining s and s' is unbounded. Also remove vertex s from S. By performing this step for every capacitated vertex in S and T we obtain the required property.

The cost of a dual edge is defined to be the capacity of the corresponding primal edge.

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 sinks and sources.

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

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

1. The edge $v \rightarrow w$ with flow $f(e)$ is equivalent to the edge $v \leftarrow w$ with flow $-f(e)$.
2. The edge $v \rightarrow w$ with capacity $[a, b]$ is equivalent to the edge $v \leftarrow w$ with capacity $[-b, -a]$.
3. The edge $v \rightarrow w$ with capacity $[a, b]$ is equivalent to two parallel edges: $v \rightarrow w$ of capacity b and $w \rightarrow v$ with capacity $-a$.
4. Let e_1 and e_2 be two parallel edges that are oriented in the same direction with capacities $[a_1, b_1]$ and $[a_2, b_2]$ respectively. The two edges can be replaced by one edge with capacity $[a_1 + a_2, b_1 + b_2]$ and flow $f(e_1) + f(e_2)$.

Now, the definition of vertex capacities has to be changed slightly. Since the incoming flow into a vertex is equal to its outgoing flow, we require that, $\forall v \in V-\{S,T\} : \sum_{v \in e} |f(e)| \leq 2 \cdot c(v)$.

Miller and Naor [MN] reduced the problem of finding a flow in a graph with multiple sources and sinks (with specified demands), to the 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.

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

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

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-planar graphs.*

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

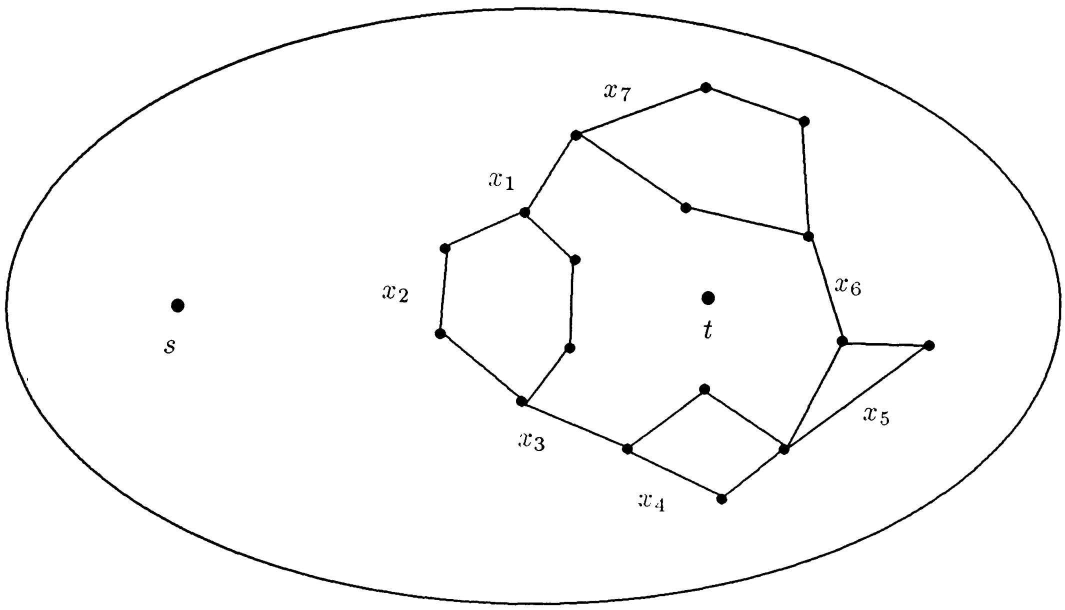
Proposition 2.1: *Let $C = c_1, \ldots, c_k$ be a cycle in the dual graph and let $f_1, \ldots, f_k$ be the potential differences over the cycle edges. Then,* $\sum_{i=1}^{k} f_i(e) = 0$.

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.

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 more efficiently (than in a general graph) the minimum cut when there are many sources and sinks is still open even when there are only edge capacities.



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

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

x_1, x_3, x_6 are edges in the dual

Figure 1: Cycle in the Dual Graph

When the graph has vertex as well as edge capacities, the minimum 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. Clearly, in the dual graph the minimum cut corresponds to a set of edges and a set of faces (that correspond to the vertices in the minimum cut). These edges and faces are "linked" together (see Fig. 1) and induce the shortest such cycle in the dual graph. We show how to modify the dual graph so that such a "linked" cycle can be computed.

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

Definition 1: *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 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. 1 for an example.) The length of a 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 cycle is defined to be the cycle with the least length.*

We show that the problem of computing a shortest cycle with jumping over faces, can be reduced to a shortest cycle problem in a planar graph with no jumping over faces.

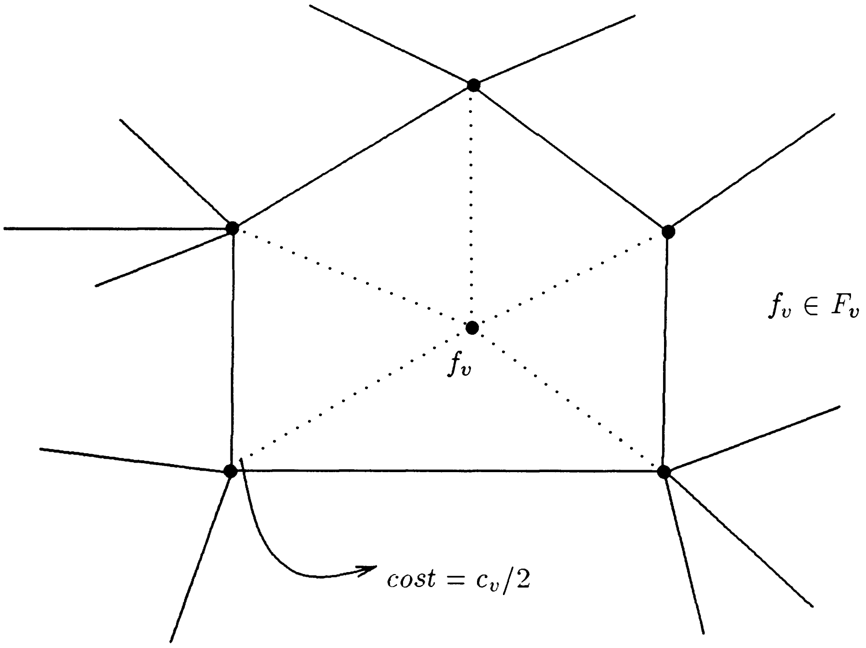


Figure 2: A face in the Dual Graph

Reduction: Given a planar dual graph $\mathcal{D}(F, D(E))$ with costs on the edges and faces, we compute a new planar dual 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 to every vertex on the corresponding face. Each of these edges is given a cost of $c_v/2$ where c_v is the capacity of the corresponding primal vertex v. Thus $D(E)' = D(E) \cup \{(f_v, f_i) \mid f_i \in F\}$ (see Fig. 2). 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). The primal graph corresponding to $\mathcal{D}'$ will be denoted by G'.

Lemma 3.1: *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 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 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 cut in a new planar graph, G', that has only edge capacities.

The efficiency of computing the minimum cut in a planar graph 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 the case of an *st*-graph (Section 3.1) and an undirected graph (Section 3.2), the known algorithms in the literature can handle the computation in the reduced graph $\mathcal{D}'$. If the graph is directed (Section 3.3), then the known method for computing minimum cut [MN] has to be modified.

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 and construct the modified dual 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^* (also called $u(t^*)$). This corresponds precisely to the *minimum cut* between s and t. Using Frederickson's algorithm [Fr], the shortest path from s^* to t^* can be found in $O(n\sqrt{\log n})$ time sequentially. To implement the same algorithm in parallel we use the algorithm by [PR] for computing shortest paths in planar graphs whose running time was improved by [GMN]. The algorithm runs in $O(\log^2 n \log\log n)$ time and uses $O(n^{1.5})$ processors on the EREW PRAM model.

Theorem 3.2: *We can compute the value of the max-flow in a directed st-planar graph in* $O(n\sqrt{\log n})$ *time. Moreover we can implement this algorithm in* $O(\log^2 n \log\log n)$ *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 we obtain a running time of $O(n \log n)$. We note that combining the "jumping" over faces idea, along with Reif's algorithm, we get an $O(n \log n)$ time algorithm for computing the minimum cut in the graph even when 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 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 cut even when vertex capacities are present. The algorithm of [Jo] can be applied directly to G'.

Assume that the reduced graph $\mathcal{D}'$ has already been computed. The idea is to first "guess" the initial value f of the flow (can be taken to be the sum of the edge capacities leaving s). Then, we put a directed path p, from t to s to "return" the flow of value f, such that every edge on this path has capacity $[f, f]$. We would like to test whether there exists a circulation in the new graph. It is important that the edges of p are put in such a manner that no edge with a lower bound (on the return path) ever goes through a vertex that has a capacity. This can be done by adding new vertices. In Section 4.2 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 inserted. Let the graph with the return path be denoted by G'' and its dual graph by $\mathcal{D}''$. Obviously, computing a circulation in G'' will induce a flow of value f in G (by deletion of the return path of flow f).

Following [MN], we have the following lemma.

Lemma 3.3: *If the dual graph* $\mathcal{D}''$ *contains a negative cycle, then there is no feasible circulation in* G''*, and we "guessed" too high a value for* f.

Proof: Let C be a cycle separating s from t in $\mathcal{D}'$. By introducing a return path p of capacity

$[f, f]$, we would like to reduce the capacity of every such cycle C by f units. But, 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. □

Our aim is to obtain the largest value of f so that $\mathcal{D}''$ does not have any negative cycles. In [MN], this value of f is found via a parametric search that takes at most $\log n$ 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., $-f$.
- 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. [GMN] show how to implement the parametric search using the method of nested dissection of [LRT]. We obtain:

Theorem 3.4: *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 $O(\log^3 n \log \log n)$ time.*

4. Computing the flow function

In this section we assume that our planar graph contains many sources and sinks where the demand of each source and sink is known. We show how to compute a flow function that will satisfy the demands, or determine that a feasible circulation does not exist. 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*. Consequently, computing the flow function becomes much more complicated than in the case where there are only edge capacities.

The first case we deal with is an st-graph. We present in Section 4.1 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 that works by cancelling the spurious cycles in the graph.

If the source and sink are not on the same face, or there are many sources and sinks in the graph, then the approach we take is to reduce the problem to that 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 (this may involve addition of new vertices to G).

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

4.1. The case of st-graphs revisited

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. Then $f(i,j) = u(j') - u(i')$, where $u(j')$ $(u(i'))$ is the potential of the face (shortest distance to the vertex s^* in the corresponding dual graph). It is easy to see that this yields an edge consistent potential function and hence, a valid flow function.

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. 3. 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 st-graph that has vertex capacities. For details on the uppermost path algorithm we refer the reader to [IS] and [FF] (see also [NC]). An efficient $O(n \log n)$ implementation of this algorithm was given by [IS]. We will describe the algorithm for undirected graphs, the algorithm easily extends to directed st-graphs as well.

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. 4).

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

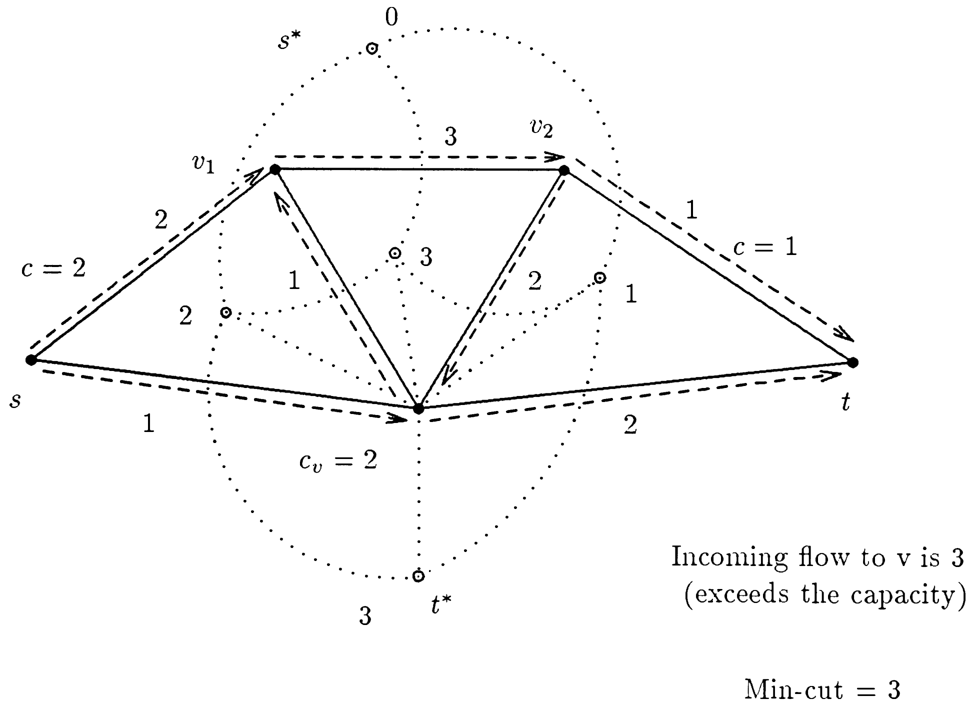


Figure 3: Example to show violation of vertex capacities

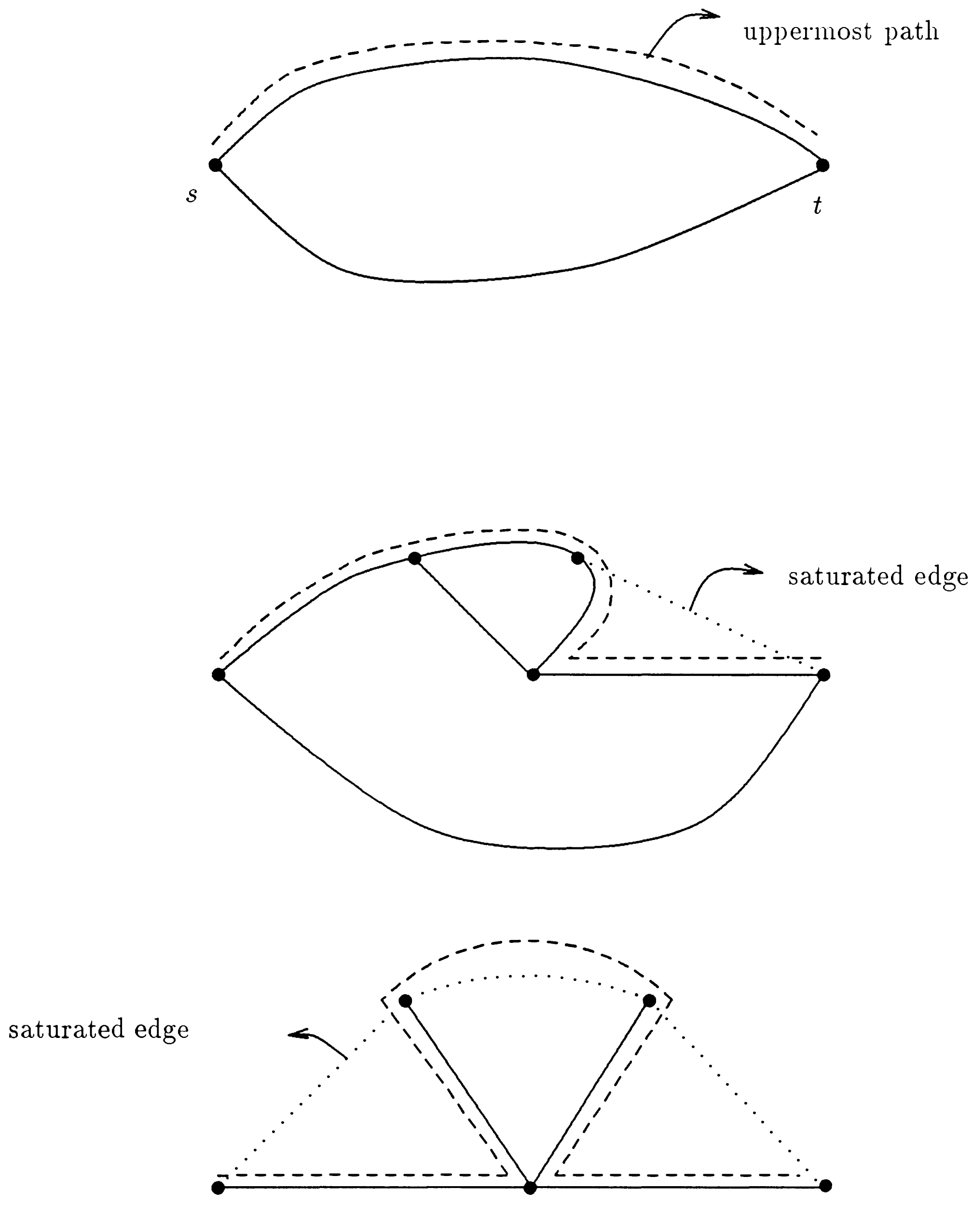


Figure 4: Uppermost path may be non-simple

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. Now pushing a flow of value f on this path actually increases the incoming flow to this vertex by atleast $2f$ units – which could cause a violation in the capacity constraint of this vertex. This is the main modification to the algorithm presented in [IS]. As a result the algorithm discards pieces of the graph in making the path simple at each pushing step. By the following proposition we can show 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 4.1: *The process of making the augmenting path simple at each step does not decrease the amount of flow pushed on that augmenting path.*

The capacities of the edges and vertices are kept in two separate heaps. At each step we choose the minimum from each 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 subchain causing the non-simplicity is discarded and the corresponding values from the heap are deleted. (See Fig. 4). However, note that this is done only once for each edge/vertex (when they are introduced into the uppermost path), after they are deleted they stay 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). Again 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 4.2: *A maximum flow function can be computed in $O(n \log n)$ time for the case of directed st-planar graphs, even when the vertices have capacities.*

Proof: An $O(n^2)$ implementation of the algorithm is trivial. Since there are at most $O(n)$ augmenting paths (uppermost paths) because at each augmentation step we delete an edge or a vertex. 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. □

4.2. The parallel algorithm

We develop a two phase parallel algorithm to find a valid max-flow in the case of st-planar graphs. In the first phase we compute potentials by a shortest path computation from s^* (with jumping over faces permitted). 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. The uppermost path algorithm really corresponds to growing a shortest path tree from s^*, where the augmenting path at each step directly corresponds to the "fringe" of the dual tree at various stages of a Dijkstra shortest path computation. When the fringe of the tree is non-simple then the uppermost path is 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.

Let the region R^1 be initially defined to be $f(s^*)$. (Where $f(v)$ denotes the face corresponding to dual vertex v.) We are going to assume that the infinite face has been divided into two faces (separating s^* from t^*) by the addition of a return edge from t to s of infinite capacity.

In general, R is the set of faces that have been "reached" by the dual shortest path tree. Each time an edge on the uppermost path is saturated, it corresponds precisely to growing the region R by annexing a new face (the next "closest" face from s^*). Each uppermost (augmenting) path corresponds directly to a fringe of the dual tree at a snapshot of the algorithm. We would like to 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.

In the first phase we computed the dual tree T_D. In the second phase we consider various sub-tree's as follows:

$$T_D^i = \{v \mid p(v) \leq p_i\}$$

Assuming all the potentials have been sorted in non-decreasing order $p_1, p_2, \ldots, p_f$, where f is

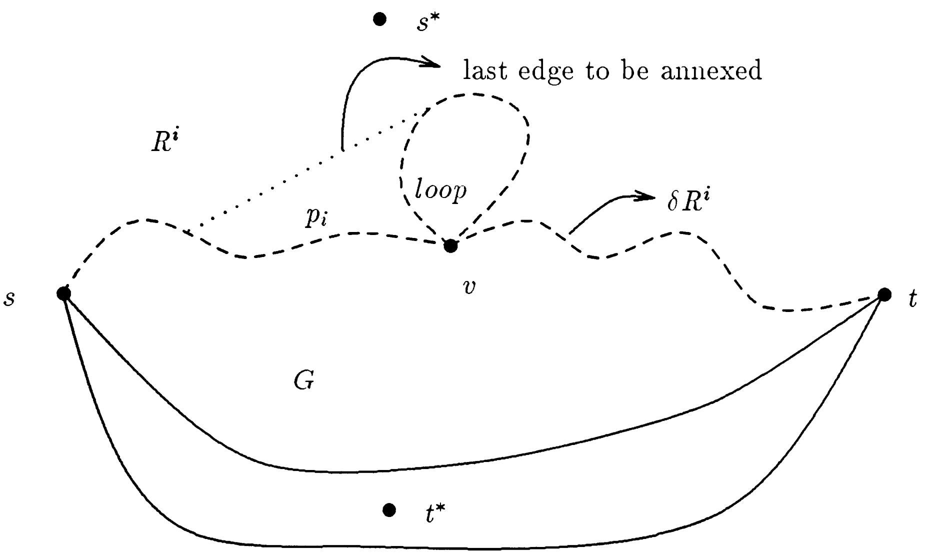


Figure 5: Region R^i, the dual tree T_D^i and a "loop"

the number of faces.

We will also assume that the embedding of each T_D^i is also known (this can be obtained from the embedding of G and its dual). We define R^i to be the union of the set of faces in T_D^i, and δR^i to be the boundary of R^i. Let the fringe of T_D^i be δR^i.

We are going to lower the potentials of certain faces, and then define the flow to be the difference of potentials of the adjacent faces. As before, the convention is that the flow moves in the direction with the smaller potential on the left.

In parallel, we need to identify the non-simplicities in δR^i. In Fig. 5 we show the formation of a non-simplicity at a capacitated vertex and how to get rid of it. Suppose that the non-simplicity is formed in T_D^i, and is not present in T_D^{i-1}, then we define the "loop" to be the finite region enclosed by the non-simple portion of δR^i (see Fig. 5).

We now redefine the potential of all the faces in the "loop" to be p_i. This has the effect of making the flow on *all* the internal edges of the loop to be zero. This ofcourse has the effect of making all the unhappy vertices in the interior of the loop to be happy (and we instantly inform the processors that are trying to fix these unhappy vertices that the vertices are now fixed).

Lemma 4.3: *Setting the potentials of all the faces in the loop 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 both the adjacent 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). □

This has the effect of "deleting" the loop and making the augmenting path simple. 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 cancelled in the first dual tree that they occur (i.e., a loop is cancelled in T_D^i, if it is not contained in a loop in T_D^j for $j < i$). After all the potentials have been adjusted in this manner we can show the following property:

Lemma 4.4: *For a capacitated vertex v, if we consider the sequence of potentials of faces incident on it (in order) starting from the smallest potential, there is only one distinct local maxima.*

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 faces (the proof is similar for more than two such faces). Let the potentials of the two faces be p_{v_1} and p_{v_2}. The faces in-between these two faces (from both sides) all have lower potential. Clearly, in some dual tree there will be a non-simplicity formed in the boundary that includes either p_{v_1} or p_{v_2} in the loop, which would have caused a lowering of one of p_{v_1} or p_{v_2}. □

Using this property it is easy to see that all the capacitated vertices do not have any excess flow through them. (Since the maximum difference of potentials of any two faces incident to v is bounded by c_v.) We can now bound the incoming and outgoing flows by c_v.

By using the fast matrix multiplication of [PR] and its improvement by [GMN] we obtain:

Theorem 4.5: *A max-flow in an st-graph can be found in $O(\log^2 n \log\log n)$ time using $O(n^2)$ processors.*

Proof: We use the algorithm by [GMN] 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_D^i. □

Alternate Scheme:

Note that each dual subtree T_D^i 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 augmeting paths it participates in. 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.

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 insure that the demands of the sources and sinks are satisfied.

We first compute a spanning tree T in G. (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, called tail and head. T_{tail} is the subtree adjacent to the tail of e and T_{head} adjacent to the head of e. Let $r(e)$ be $\sum_{v \in T_{tail}} d(v) = -\sum_{v \in T_{head}} d(v)$. 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 (with the same direction) whose capacity is $[r(e), r(e)]$.

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

Let us now elaborate on how the return flow edges are inserted without being adjacent to capacitated vertices. 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 $v \to w$ 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. 6).

Let u be the vertex that succeeds w on the path and let u^* and v^* be faces adjacent to u and v respectively. Let $e_1^*, \ldots, e_k^*$ be a dual path 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 split each edge e_i into two edges, e_i' and e_i'', by adding a vertex a_i in the

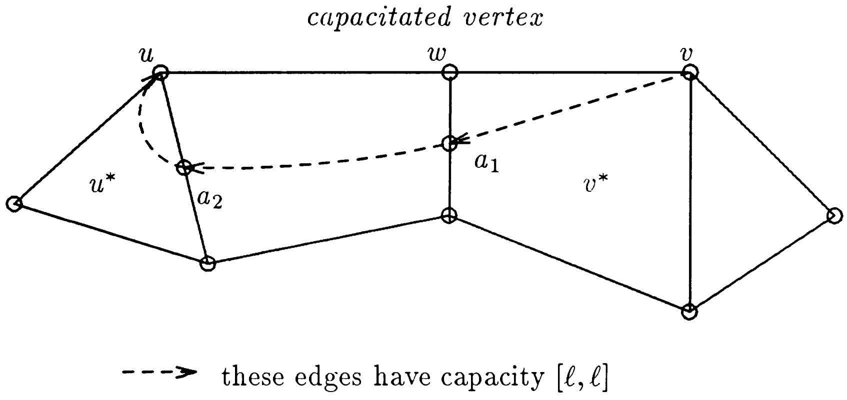


Figure 6: Embedding the edge $v \to w$

middle. 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 and 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 two, by conservation of flow the flow on e_i' and e_i'' is the same, 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.

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. 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. The vertices on this cycle correspond to a set of faces in the primal graph, such that the faces corresponding to the adjacent vertices on the cycle share a common edge (edges in E_C). This cycle corresponds to a set of faces in the primal graph that 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. 7). Call the new graph G'. This cycle is directed clockwise, and its edges have zero capacity.

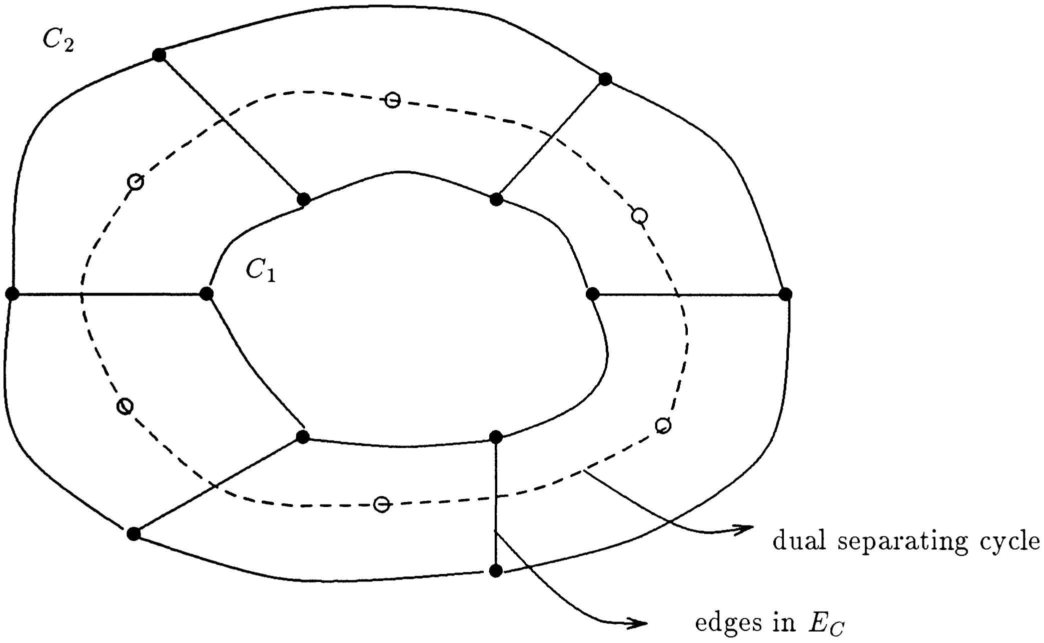


Figure 7: Finding an uncapacitated separating cycle

Lemma 4.6: *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 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 upperbounded by $\frac{2}{3}f$. Thus the number of faces in each component of G after deletion of C, is upperbounded 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 4.7: *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. This is done by giving each edge on C an infinite capacity.

Lemma 4.8: *There exists a feasible solution in $G_I + C$ and $G_E + C$.*

Proof: Let us assume that there is a feasible circulation $\mathcal{C}$ in G. We show that in $G_I + C$ there is also a feasible circulation after the capacities of the vertices have been changed to infinity. (The proof for $G_E + C$ is similar.) Consider the restriction of $\mathcal{C}$ 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 $\mathcal{C}$ 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 is violated. Notice, that vertex capacities are not violated due to the merging step.

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) in the residual graph. Notice that in the residual graph there are no lower bounds on edges and that in this sub-problem v and w are on the same face. A solution to this problem can be obtained by using a modification of the algorithm presented earlier for st-graphs. (In pushing flow in the residual graph, vertex capacities are "active" for only a subset of the edges, since when we push flow on an edge it is actually either increasing or decreasing the flow through the vertices incident on it.) 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 f in the original graph, and let f' be the current circulation, in which the flow cannot be redirected from v to w. Then, $f - f'$ is a collection of residual cycles such that augmenting them in f' will redirect the flow from v to w.

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

Theorem 4.9: *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

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

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).

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

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

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

This gives us the value of 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 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 anymore 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 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 get a legal circulation necessarily. (See Fig. 3). To obtain a legal circulation, a set of spurious cycles has to be identified and cancelled. 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 cancelled easily.

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

Acknowledgements

We would like to thank Donald Johnson, Gary Miller and Vijay Vazirani for fruitful discussions.

References

[CH] J.Cheriyan and T.Hagerup, *A randomized $O(nm + n^2(\log n)^4)$ maximum flow algorithm,* 30th Annual Symposium on Foundations of Computer Science (1989).

[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).

[GT] A.V.Goldberg and R.E.Tarjan, *A new approach to the maximum flow problem,* Journal of the ACM, vol 35, 4, pp. 921-940 (1988).

[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).

[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).

[GMN] H. Gazit, G. L. Miller and J. Naor, *A fast parametric search method in planar graphs*, manuscript.

[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).

[LRT] R. J. Lipton, D. J. Rose and R. E. Tarjan, *Generalized nested dissection*, Siam J. 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 (1986), pp 265-279.

[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 (1989), pp. 112-117.

[NC] Nishizeki and 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,* to appear, Proceedings of the 1^{st} Symposium on Discrete Algorithms (1990).

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