JOURNAL OF ALGORITHMS **29,** 82–110 (1998)
ARTICLE NO. AL980961

All-Pairs Min-Cut in Sparse Networks*

Srinivasa R. Arikati[†]

Department of Mathematical Sciences, University of Memphis, Memphis, TN 38152
E-mail: arikatis@next1.msci.memphis.edu

Shiva Chaudhuri

Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
E-mail: shiva@mpi-sb.mpg.de

and

Christos D. Zaroliagis[‡]

Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
E-mail: zaro@mpi-sb.mpg.de

Received March 22, 1996; revised January 16, 1998

Algorithms are presented for the all-pairs min-cut problem in bounded tree-width, planar, and sparse networks. The approach used is to preprocess the input n-vertex network so that afterward, the value of a min-cut between any two vertices can be efficiently computed. A tradeoff is shown between the preprocessing time and the time taken to compute min-cuts subsequently. In particular, after an $O(n \log n)$ preprocessing of a bounded tree-width network, it is possible to find the value of a min-cut between any two vertices in constant time. This implies that for such networks the all-pairs min-cut problem can be solved in time $O(n^2)$. This algorithm is used in conjunction with a graph decomposition technique of Frederickson to obtain algorithms for sparse and planar networks. The running times depend upon a topological property, γ, of the input network. The parameter γ varies between 1 and $\Theta(n)$; the algorithms perform well when $\gamma = o(n)$. The value of a min-cut can be found in time $O(n + \gamma^2 \log \gamma)$ and all-pairs min-cut can be solved in time $O(n^2 + \gamma^4 \log \gamma)$ for sparse networks. The corresponding running

* This work was partially supported by the EU ESPRIT LTR Project no. 20244 (ALCOM-IT). A preliminary version appeared in [3].

[†] Part of this work was done while the author was with the Max-Planck-Institut für Informatik, Germany.

[‡] Author to whom correspondence should be addressed.

0196-6774/98 $25.00

times for planar networks are $O(n + \gamma \log \gamma)$ and $O(n^2 + \gamma^3 \log \gamma)$, respectively. The latter bounds depend on a result of independent interest; outerplanar networks have small "mimicking" networks that are also outerplanar.

1. INTRODUCTION

Network flows are of fundamental importance in computer science, engineering, and operations research, to name a few areas. The textbook by Ahuja et al. [1] is an exhaustive reference on the subject. A central problem in network flows is that of computing an *s-t* min-cut. We are given a (directed) network, i.e., a directed graph with nonnegative capacities on its edges, and two distinguished vertices s and t. An *s-t* cut in this network is a partition of the vertices into two parts, one containing s and the other containing t. The capacity of the cut is the sum of the capacities of the edges going from the part containing s to the part containing t. An *s-t* min-cut is a cut of minimum capacity among all *s-t* cuts.

An *s-t* flow in a network is an assignment of a value, less than or equal to the capacity, to each edge such that the net flow out of each node except s and t is zero, where the net flow out of a node is the sum of flows on edges leaving the node minus the sum of flows on edges entering the node. It follows that the net flows out of s and t sum to zero. An *s-t* max-flow is a flow that maximizes the net flow out of s, which is called the value of an *s-t* max-flow. The max-flow min-cut theorem [12] states that the capacity of an *s-t* min-cut in a network is equal to the value of an *s-t* max-flow.

In this paper, we are concerned with the all-pairs min-cut problem (APMC problem, for brevity). The problem is to compute the value of an *s-t* min-cut for each pair of vertices s, t in the network. This problem has applications in statistical data security [15]. Since the value of an *s-t* min-cut can be computed by solving an *s-t* max-flow problem, the naive solution to the APMC problem solves $n(n-1)$ max-flow problems on n-vertex networks. It was shown by Gomory and Hu [17] that in undirected networks, the APMC problem can be solved by solving $n-1$ well-chosen max-flow problems. Thus, the APMC problem on an undirected network takes $O((n-1)F(n, m))$ time, where $F(n, m)$ is the time required to solve a max-flow problem on an n-vertex, m-edge network. For directed networks, the method of Gomory and Hu does not apply, and nothing better than the naive solution (taking $O(n^2 F(n, m))$ time) is known.

The time taken to compute a max-flow when nothing is known about the structure of the input network is $O(\min\{n^3/\log n, nm \log n\})$ [10, 20]. However, one can do better when the structure of the input network is

known. Recently, it was shown that the max-flow problem in the special case of undirected planar networks, where the source and the sink are on the same face, can be solved in $O(n)$ time [18]. The same time bound holds for the max-flow problem in directed or undirected *bounded tree-width* networks [16]. The tree-width is a parameter that intuitively indicates how close the structure of the network is to a tree (see Section 2.3 for a formal definition). The class of bounded tree-width networks includes (among others) outerplanar networks, series-parallel networks, and networks with bounded bandwidth or cutwidth [4, 7]. Thus giving better algorithms for this class of networks is an important step in the development of better algorithms for sparse networks, i.e., networks with $O(n)$ edges. For sparse networks, in general, the best max-flow algorithm runs in time $O(n^2 \log n)$. For the APMC problem in the undirected case, substituting the values of $F(n, m)$ yields running times of $O(n^3 \log n)$ for sparse networks and $O(n^2)$ for bounded tree-width networks. For directed networks, the corresponding running times are $O(n^4 \log n)$ and $O(n^3)$, respectively. From now on, we consider only directed networks.

The starting point of this paper is a new algorithm for the APMC problem in bounded tree-width networks that runs in $O(n^2)$ time, improving upon the previous algorithm for directed networks by a factor of n. The approach used differs from previous approaches in that, instead of computing a number of separate max-flows from scratch, we preprocess the network so that, subsequently, the value of an *s-t* min-cut (or max-flow) can be efficiently computed for any pair of vertices s and t. We show a tradeoff between the amount of preprocessing required and the time required to compute the value of an *s-t* min-cut subsequently. The tradeoff is that after $O(nI_k(n))$ preprocessing, the value of an *s-t* min-cut can be computed in $O(k)$ time, for any integer $k \geq 1$. The function $I_k(n)$, defined formally in Section 2.4, decreases rapidly as k increases; for example, $I_1(n) = \lceil \log n \rceil$ and $I_2(n) = \log^* n$. If the preprocessing is restricted to $O(n)$, then the value of an *s-t* min-cut can be computed in $O(\alpha(n))$ time (where $\alpha(n)$ is the inverse Ackermann function; see Section 2.4).

We use the algorithm for bounded tree-width networks to develop an algorithm for sparse networks; the latter algorithm is based on a decomposition of the original network into networks of bounded tree-width. Frederickson [14] showed how to decompose a sparse graph into a number of edge-disjoint outerplanar subgraphs, called hammocks. (An outerplanar graph has tree-width 2.) The number of hammocks obtained, γ, depends on the topological properties of the graph and varies between 1 and $\Theta(n)$. We give an algorithm that computes the value of an *s-t* min-cut (or max-flow) in a sparse network in time $O(n + \gamma^2 \log \gamma)$. Thus, this algorithm is always competitive with the $O(n^2 \log n)$-time algorithm [20] and

does better if $\gamma = o(n)$. This leads to an algorithm that solves the APMC problem in time $O(n^2 + \gamma^4 \log \gamma)$ on a sparse network.

The algorithms use the construction of a small network that "mimics" the flow behavior of a large network. This idea was developed in [16], where it is shown that a network G with q terminals has a mimicking network of size 2^{2^q}. In the case where G is outerplanar, we show (Section 4) that it has a mimicking outerplanar network that is a minor of G and has size $q^2 2^{q+2}$. This leads (along with the above-mentioned approach for sparse networks) to faster algorithms for planar networks. We give an algorithm that computes the value of an s-t min-cut (or max-flow) in an n-vertex planar network in $O(n + \gamma \log \gamma)$ time, which compares favorably with the $O(n \log n)$ time algorithm of [23]. We also show that the APMC problem can be solved in $O(n^2 + \gamma^3 \log \gamma)$ time.

The above algorithms output the value of a max-flow or min-cut. In the case where the actual min-cut is desired, we show how to output the edges crossing a min-cut in additional time linear in the size of the output (Section 6).

Necessary and sufficient conditions (called *external flow inequalities*) for realizable flows in multiterminal networks are derived in [16]. An important lemma in [16] shows how to combine the flow inequalities of a number of subnetworks to obtain a single set of flow inequalities for the combined network. The proof uses linear programming. We give (Section 7) a simple and direct proof of the same result that avoids linear programming and leads to a slightly faster computation of these inequalities.

The structure of the algorithms for bounded tree-width networks is derived from an algorithm used to solve shortest path queries [8]. The hammock decomposition technique has been used in shortest path problems (see, e.g., [11, 13, 14]). To our knowledge, this is the first application of this technique to a different problem.

2. PRELIMINARIES

2.1. *Flows in Multiterminal Networks*

A *network* is a directed graph $G = (V, E)$ with a nonnegative real capacity c_e associated with each edge $e \in E$. The *terminals* of G are the elements of a distinguished subset, Q, of its vertices. A *flow* in G is an assignment of a nonnegative real value $f_e \le c_e$ to each edge e such that the net flow out of each nonterminal vertex is zero, where the net flow out of a vertex is the sum of flows on edges leaving the vertex minus the sum of flows on edges entering the vertex. An *external flow* $x = (x_1, \ldots, x_{|Q|})$ is an assignment of a real value x_p to each terminal $a_p \in Q$, $1 \le p \le |Q|$. A

realizable external flow is an external flow such that there exists a flow in which the net flow out of each terminal a_p is x_p. A *cut* $(S, \bar{S})$ is a partition of the vertices of G into two subsets S and $\bar{S} = V - S$; S is called the *defining subset* of the cut. The *capacity* of the cut $(S, \bar{S})$ is the sum of capacities of edges going from vertices in S to vertices in $\bar{S}$. For a subset R of Q, an *R-separating cut* is a cut $(S, \bar{S})$ where $Q \cap S = R$. A *minimum* R-separating cut is an R-separating cut of minimum capacity.

The sum of the net flows out of the terminals in R is called the *R-value* of a flow f and will be denoted by $|f|$. A *maximum R-flow* is a flow of maximum R-value. If $Q = \{s, t\}$, an *s-t max-flow* is a maximum $\{s\}$-flow, and its value is the $\{s\}$-value of the flow. An *s-t min-cut* is a minimum $\{s\}$-separating cut. The max-flow min-cut theorem states that the value of an *s-t* max-flow is equal to the capacity of an *s-t* min-cut.

In a network that can be decomposed into edge disjoint subnetworks, external flows in the subnetworks can be "added" to yield an external flow in the network. Let G be the edge disjoint union of G_1 and G_2. Let Q_1 and Q_2 be the terminal sets of G_1 and G_2, respectively, and let the common vertices of G_1 and G_2 be terminals in both subnetworks, that is, $V(G_1) \cap V(G_2) = Q_1 \cap Q_2$. Let $Q = Q_1 \cup Q_2$ be the terminal set of G. For external flows $x^{(1)} = \{x_v^{(1)} : v \in Q_1\}$, $x^{(2)} = \{x_v^{(2)} : v \in Q_2\}$, define their *sum*, denoted as $x^{(1)} \oplus x^{(2)}$, to be the external flow $x = \{x_v : v \in Q\}$, where $x_v = x_v^{(1)}$ if $v \in Q_1 - Q_2$, $x_v = x_v^{(2)}$ if $v \in Q_2 - Q_1$, and $x_v = x_v^{(1)} + x_v^{(2)}$ if $v \in Q_1 \cap Q_2$. Then we have

LEMMA 2.1. *Let G, G_1, and G_2 be defined as above. Then if $x^{(1)}$ and $x^{(2)}$ are realizable external flows in G_1 and G_2, respectively, then $x^{(1)} \oplus x^{(2)}$ is a realizable external flow in G, and if x is a realizable external flow in G, then there exist realizable external flows $x^{(1)}$ in G_1 and $x^{(2)}$ in G_2 such that $x = x^{(1)} \oplus x^{(2)}$.*

Proof. Let f_1 and f_2 be the flows that yield external flows $x^{(1)}$ and $x^{(2)}$ in G_1 and G_2. By taking the union of these flows in G, which is possible since the individual flows involve disjoint edge sets, we obtain a flow in G, and the resulting external flow is exactly $x^{(1)} \oplus x^{(2)}$. On the other hand, the flow corresponding to any external flow x in G induces a flow in G_1 and a flow in G_2, which yield external flows $x^{(1)}$ and $x^{(2)}$ such that $x = x^{(1)} \oplus x^{(2)}$. ∎

2.2. *Mimicking Networks*

Let G be a network with terminal set Q. A network $M(G)$ with terminal set Q' is a *mimicking network* for G if there exists a bijection between Q and Q' such that every realizable external flow in G is also realizable in $M(G)$, and vice versa.

In [16], it is shown that for any network G, there exists a mimicking network with 2^{2^q} vertices, where q is the number of terminals of G. The mimicking network in [16] is constructed by finding 2^q cuts in G, namely, a minimum R-separating cut for each $R \subseteq Q$. Those vertices of G that are on the same side of all of these cuts form equivalence classes. Induction on q shows that there can be at most 2^{2^q} equivalence classes. The network $M(G)$ is constructed by replacing each equivalence class with a single vertex. The edge between two vertices of $M(G)$ in a given direction has a capacity equal to the sum of the capacities of the edges in G between the corresponding equivalence classes, taking direction into account. For a given $R \subseteq Q$, a minimum R-separating cut (or a maximum R-flow) can be computed by the standard method of introducing a new source s^*, connected to each vertex in R with edges of infinite capacity, and a new sink t^* to which each vertex in $Q - R$ is similarly connected, and computing an s^*-t^* max-flow in the transformed network.

However, the standard method for computing minimum R-separating cuts may not preserve the structural properties of G; for example, the transformed network may not be planar, while G is planar. We give an alternative method for computing a maximum R-flow by computing a number of s-t max-flows in networks with the same structural properties as G. This will lead to efficient algorithms for planar networks in Section 5.

We first review some concepts from network flows. Let f be a flow in a network $H = (V_H, E_H)$. We may assume that if edge (i, j) exists in H, then so does (j, i), since we can always insert (j, i) with zero capacity, if it does not exist, without changing the topology of H. The *residual capacity* r_e of an edge $e = (i, j)$ is defined as $r_e = c_e - f_e + f_{e'}$, where $e' = (j, i)$. The *residual network* $H(f)$ of H for the flow f is defined as $H(f) = (V_H, E_H)$, where the capacity of edge e is r_e. An *i-j augmenting path* in the residual network $H(f)$ is a directed path from i to j consisting of edges with positive capacity. It is well known that f is an s-t max-flow in H if and only if there is no s-t augmenting path in $H(f)$ (see, e.g., Theorem 6.4 in [1]). A routine generalization yields

Fact 2.1. Let H be a network with terminal set Q and let $R \subseteq Q$. Then a flow f is a maximum R-flow iff there is no a-a' augmenting path in the residual network $H(f)$ for any $a \in R$, $a' \in Q - R$.

We wish to find a maximum R-flow in network G with terminal set Q, for some $R \subseteq Q$. Intuitively, the following procedure should work: select a vertex s of R and compute maximum flows from s to every terminal in $Q - R$. Every successive maximum flow is computed in the residual network left by the previous computation. Then, select the next vertex s' from R and do the same; the network in which the first maximum flow for s' is computed is the residual network left by the last computation

performed for s. In this manner, process each of the vertices in R. The flow obtained by adding up the individual flows is a maximum R-flow. While the above is intuitively clear, we have not found a proof in the literature. We include a proof below.

Formally, let $(s_1, t_1), (s_2, t_2), \ldots, (s_p, t_p)$ be a lexicographic ordering of the pairs in $R \times (Q - R)$. Define $G_0 = G$. For $i = 1, \ldots, p$, compute an s_i-t_i max-flow $f(i)$ in G_{i-1} and define G_i to be the residual network of G_{i-1} for flow $f(i)$. Call this procedure Lex-Max-R-Flow.

Let $f_e(i)$ be the flow through edge e in $f(i)$. Define $g_e(i) = \sum_{j=1}^{i} f_e(j)$. It is easy to verify that for each i, $\{g_e(i),\ e \in E\}$ specifies a flow $g(i)$, and G_i is the residual network of G for flow $g(i)$. Let g be the flow $g(p)$.

LEMMA 2.2. *The flow g is a maximum R-flow in G.*

Proof. We shall prove the lemma by showing that the above procedure for computing a maximum R-flow in G is equivalent to finding a minimum cost flow in a transformed network G'.

Recall that in the (classical) minimum cost flow problem, we are given a network $N = (U, A)$ with terminal set $\{s, t\}$; each edge e in N is associated (in addition to its capacity c_e) with a cost w_e per unit of flow. The cost of a flow f is $\sum_{e \in A} w_e f_e$. A flow is *minimum cost* (min-cost) if among all flows with the same value it has the minimum cost. The *min-cost flow problem* is to find an s-t max-flow of minimum cost.

Let $r = |R|$ and $q = |Q|$. Consider a new network G' constructed as follows. Introduce a new source s^* and a new sink t^* to G and edges of infinite capacity from s^* to every $s_k \in R$, $1 \le k \le r$, and from every $t_l \in Q - R$, $1 \le l \le q - r$, to t^*. Associate with all edges in G a zero cost. Let $d = \max\{r, q - r\}$. For the rest of the edges in G' associate the following costs: $w_{(s^*, s_k)} = kd$, for all $1 \le k \le r$, and $w_{(t_l, t^*)} = l$, for all $1 \le l \le q - r$.

Now, we can find a min-cost flow in G' by augmenting flow along a minimum cost path [21, Theorem 8.12], where the cost of a path is the sum of the costs of its edges. By our choice of edge costs it follows that this method for computing a min-cost flow in G' will simulate the procedure Lex-Max-R-Flow. Hence, it is easy to verify that (i) the maximum amount of flow that can be augmented along min-cost s^*-t^* paths in G' that contain the pair (s_i, t_i) is equal to $|f(i)|$; and (ii) all pairs (s_i, t_i) are processed in the same lexicographic order as in the procedure Lex-Max-R-Flow, i.e., there is no min-cost s^*-t^* augmenting path in G' containing a pair (s_j, t_j), for any $j < i$. Consequently, the computed min-cost flow in G' has value $|g|$. Furthermore, there is no s^*-t^* augmenting path in G', which implies that there is no a-a' augmenting path in G, for any $a \in R$, $a' \in Q - R$, which by Fact 2.1 proves the lemma. ∎

We have thus proved that a maximum R-flow, and hence a minimum R-separating cut, in network G can be computed by doing $O(q^2)$ max-flow computations in G, since there are $O(q^2)$ pairs in $R \times (Q - R)$. Since there are at most 2^q different R's, we have

LEMMA 2.3. *A mimicking network of size 2^{2^q} for a network G with q terminals can be computed in time $O(q^2 2^q F(G))$, where $F(G)$ is the time required to compute an s-t max-flow in G.*

Suppose we are given the mimicking networks of a number of networks. A number of pairs are specified, each pair consisting of two terminals belonging to different networks. We are asked to combine the different networks by identifying the specified pairs of terminals. Finally, we are given a subset of all of the terminals, and asked to find the mimicking network of the combined network at this new set of terminals. Note that in the combined network, the set of terminals of each subnetwork is an *attachment set* for that subnetwork, where an *attachment set* for a subnetwork is a set of vertices whose deletion disconnects the subnetwork from the rest of the network.

LEMMA 2.4. *Let $G = \bigcup_{i=1}^{m} G_i$, where the G_i's are edge-disjoint, and let G_i have attachment set C_i. Given the mimicking networks $M(G_i)$ for each G_i at terminals Q_i satisfying $C_i \subseteq Q_i$, and a set $Q' \subseteq Q = \bigcup_{i=1}^{m} Q_i$, we can compute the mimicking network $M(G)$ for G at terminals Q' in time $O(q^2 2^q \cdot (\sum_{i=1}^{m} 2^{2^{q_i}})^3)$, where $q_i = |Q_i|$ and $q = |Q|$.*

Proof. Let G' be obtained by combining the appropriate terminals of the mimicking networks $M(G_i)$. By repeated applications of Lemma 2.1, an external flow at terminals Q is realizable in G' iff it is the sum of realizable external flows in each $M(G_i)$ at Q_i. Similarly, an external flow at terminals Q is realizable in G iff it is the sum of realizable external flows in each G_i at Q_i. Since the set of realizable flows of G_i and $M(G_i)$ at terminals Q_i are the same, it follows that the sets of realizable flows of G and G' at Q are the same. Hence, G' is a mimicking network for G at terminals Q.

Now, compute the mimicking network of G' at terminals Q', using Lemma 2.3 and computing max-flows with an $O(n^3)$ algorithm (see, e.g., [1]). This mimicking network is the desired $M(G)$. The lemma follows. ∎

2.3. *Tree-width*

A *tree decomposition* of a (directed or undirected) graph $G = (V(G), E(G))$ is a pair (X, T), where $T = (V(T), E(T))$ is a tree, X is a family $\{X_i : i \in V(T)\}$ of subsets of $V(G)$ that cover $V(G)$, and the

following conditions hold:

- (*edge mapping*) $\forall (v, w) \in E(G)$, there exists an $i \in V(T)$ with $v \in X_i$ and $w \in X_i$.
- (*continuity*) $\forall i, j, k \in V(T)$, if j lies on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$, or equivalently, $\forall v \in V(G)$, the nodes $\{i \in V(T) : v \in X_i\}$ induce a connected subtree of T.

The *width* of the tree decomposition is $\max_{i \in V(T)} |X_i| - 1$. The *tree-width* of G is the minimum width over all possible tree decompositions of G. To avoid confusion, we shall use the terms "node" and "arc" to refer to the vertices and edges of T, respectively.

Bodlaender [6] gave a linear-time algorithm to compute a constant width tree decomposition of a graph with constant tree-width. In [5] a linear-time algorithm is given to convert a tree decomposition of (constant) width t into another one of tree-width $3t + 2$, in which the tree is binary. We call such a tree decomposition a *binary tree decomposition.*

Let G be an n-vertex graph of constant tree-width and let (X, T) be its tree decomposition of constant width. The edge mapping condition ensures that the endpoints of each edge in G appear together in some set $X_i \in X$, belonging to node i of T. Thus, in a sense, each edge is represented in at least one node of T. For our purposes, we need to explicitly associate each edge of G with exactly one node of T. We will, therefore, compute an *augmenting function* h: $E(G) \to V(T)$, satisfying the property that both endpoints of an edge are present in the set belonging to the node that the edge is mapped to by h. More precisely, $\forall (v, w) \in E(G)$, $\{v, w\} \subseteq X_{h(v,w)}$. Any augmenting function will suffice for our purposes. It is easy to compute one such function, by doing a traversal of T and assigning $h(v, w) = i$ for each $i \in V(T)$, if $\{v, w\} \subseteq X_i$, $(v, w) \in E(G)$ and $h(v, w)$ has not yet been assigned a value. This takes time proportional to $\sum_{i \in V(T)} |X_i|^2$, which is $O(n)$, since the tree decomposition is of constant width. The resulting tree decomposition with the values $h(v, w)$, $\forall (v, w) \in E(G)$, is called an *augmented tree decomposition.* The discussion above is summarized as the following result.

PROPOSITION 2.1. *Given an n-vertex graph G of constant tree-width t, we can compute in $O(n)$ time an augmented binary tree decomposition of G of width $O(t)$.*

2.4. *Tree Products*

For a function g let $g^{(1)}(n) = g(n)$; $g^{(i)}(n) = g(g^{(i-1)}(n))$, $i > 1$. Define $I_0(n) = \lceil n/2 \rceil$ and $I_k(n) = \min\{j \mid I_{k-1}^{(j)}(n) \leq 1\}$, $k \geq 1$. The functions $I_k(n)$

decrease rapidly as k increases; in particular, $I_1(n) = \lceil \log n \rceil$ and $I_2(n) = \log^* n$. Define $\alpha(n) = \min\{j \mid I_j(n) \leq j\}$.

The following theorem was proved in [2, 9].

THEOREM 2.1. *Let $\cdot$ be an associative operator defined on a set S, such that for $q, r \in S$, $q \cdot r$ can be computed in constant time. Let T be a tree with n nodes such that each arc is labeled with an element from S. Then (i) for each integer $k \geq 1$, after $O(nI_k(n))$ preprocessing, the composition of labels along any path in the tree can be computed in $O(k)$ time; and (ii) after $O(n)$ preprocessing, the composition of labels along any path in the tree can be computed in $O(\alpha(n))$ time.*

The above theorem will be used in the next section to efficiently compute min-cuts (or max-flows) in bounded tree-width networks by exploiting the tree-like structure of these networks.

3. BOUNDED TREE-WIDTH NETWORKS

In this section we shall show that computing min-cuts in a bounded tree-width network is as easy as computing products of arc values along paths in a tree. We show this by first defining a value for every arc of the tree decomposition of the network and an associative operator on these values. We then show that computing min-cuts reduces to computing products of these values along paths in the tree decomposition.

Let G be a network of bounded tree-width and (X, T) its augmented binary tree decomposition. For a subtree T' of T, we define the subgraph G' *spanned by* T' as follows. The vertices of G' are the vertices in the sets associated with the nodes of T', i.e., $V(G') = \bigcup_{i \in V(T')} X_i$. The edges of G' are those edges that the augmenting function maps to nodes in T', i.e., $E(G') = \{e \in E(G)\colon h(e) \in V(T')\}$. It is easy to check that node-disjoint subtrees span edge-disjoint subgraphs. (In fact, it is only to ensure this property that we introduce the augmenting function.)

Define a set $U = \{P_{ij} = (M_i, M_j, M_{ij})\colon \forall i, j \in V(T),\ i \neq j\}$, where M_i, M_j, and M_{ij} are defined as follows. For $i, j \in V(T)$ let $path(i, j)$ denote the unique path from i to j in T. Deleting the first and last arcs on this path breaks up T into three components: T_i and T_j, the ones containing i and j, respectively, and the remaining component T_{ij}. If $path(i, j)$ is an arc, then the first and last arcs on the path are the same; consequently, the component T_{ij} is empty. The nodes in T_{ij} that are adjacent to i and j are denoted n_i and n_j, respectively. Then, M_i and M_j are the mimicking networks for the subgraphs spanned by T_i and T_j at terminals X_i and X_j, respectively, and M_{ij} is the mimicking network for the subgraph spanned by T_{ij} at terminals $X_{n_i} \cup X_{n_j}$. If $T_{ij} = \emptyset$, then $M_{ij} = \emptyset$.

Let $P = (M_1, M_2, M_3)$ and $P' = (M_1', M_2', M_3')$. Then, we say that P and P' are *equal*, denoted by $P \cong P'$, iff $\forall 1 \le i \le 3$ the mimicking networks M_i and M_i' have the same terminal set and the same set of realizable external flows.

Define the following operator $\cdot$ on U. For $i, j, l, k \in V(T)$,

$$P_{ij} \cdot P_{lk} \cong \begin{cases} P_{ik} & \text{if } j = l \text{ and } path(i,k) \text{ includes node } j \\ \varnothing & \text{otherwise.} \end{cases}$$

It follows easily from the definition that $\cdot$ is associative: If a, b, c, d are nodes (appearing in that order) on a simple path in T, then $(P_{ab} \cdot P_{bc}) \cdot P_{cd} \cong P_{ac} \cdot P_{cd} \cong P_{ad}$ and $P_{ab} \cdot (P_{bc} \cdot P_{cd}) \cong P_{ab} \cdot P_{bd} \cong P_{ad}$. If a, b, c, d are not on a simple path in T, then $(P_{ab} \cdot P_{bd}) \cdot P_{cd} \cong P_{ab} \cdot (P_{bc} \cdot P_{cd}) \cong \varnothing$. In general, the product $P_{i_1 i_2} \cdot P_{i_2 i_3} \cdots \cdot P_{i_{m-1} i_m} \cong P_{i_1 i_m}$ if $i_1, \ldots, i_m$ is a path in T.

Now, we can proceed to show how an *s-t* min-cut can be computed. The main idea is as follows. Take the product of the arc values in $path(i, j)$, where $s \in X_i$ and $t \in X_j$. This product returns three mimicking networks having in total a constant number of terminals that include s and t. Combining these mimicking networks using Lemma 2.4 and retaining the appropriate set of terminals gives us a new mimicking network on which the required *s-t* min-cut can be computed using a standard algorithm.

We start by filling in the missing parts of the above idea, namely, how arc values are computed and how the product of two values is evaluated. This is done in the next two lemmas, where we show that the arc values can be efficiently computed in a linear-time preprocessing step and that the product of any two values can be computed in constant time. We begin with the latter.

Suppose we have computed P_{xy} for every x and y such that (x, y) is an arc in T. Let i, j, k be nodes of T such that j is an internal node of $path(i,k)$. Then, given P_{ij} and P_{jk}, the computation $P_{ik} \cong P_{ij} \cdot P_{jk}$ can be done in $O(1)$ time, as the following lemma shows. The main idea is to combine the mimicking networks of the subgraphs of G spanned by the tree components incident on j and retain the appropriate set of terminals.

LEMMA 3.1. *Let G be a network and let (X, T) be its augmented binary tree decomposition of constant width. Given P_{xy}, $\forall (x, y) \in E(T)$, and P_{ij}, P_{jk} for some $i, j, k \in V(T)$, $P_{ij} \cdot P_{jk}$ can be computed in constant time.*

Proof. If j is not an internal node in $path(i, k)$, there is nothing to prove (since in this case $P_{ij} \cdot P_{jk} \cong \varnothing$, by definition). Therefore, suppose that j is an internal node in $path(i, k)$; see Fig. 1. Since T is binary, j has at most one neighbor x apart from its neighbors y and z on the path from

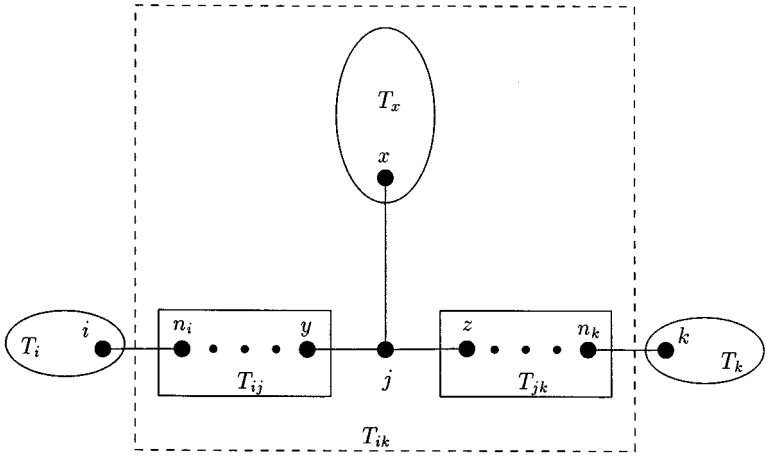

FIG. 1. Computation of P_{ij}.

i to k. Let T_x be the component of T containing x, obtained by deleting the arc (j, x). Let n_i and n_k be the neighbors of i and k in $path(i, k)$.

The value P_{ik} consists of the three mimicking networks M_i, M_k, and M_{ik}, for the subgraphs spanned by T_i, T_k, and T_{ik}, respectively. The former two are already available as part of the values P_{ij} and P_{jk}. Hence we need to compute only M_{ik}. The component T_{ik} is the union of components T_{ij}, T_{jk}, T_x, and node j, which are pairwise node-disjoint. By supposition, we have the mimicking network for the subgraph spanned by T_x, as part of the value P_{jx}. The mimicking networks for the subgraphs spanned by T_{ij} (at terminals $X_{n_i} \cup X_y$) and T_{jk} (at terminals $X_z \cup X_{n_k}$) are available in the values P_{ij} and P_{jk}. The mimicking network for the subgraph spanned by j at terminals X_j can be computed using Lemma 2.3. From the continuity property of tree decompositions, it follows that the set of terminals for each of the subgraphs is an attachment set for the subgraph and that the final set of terminals desired, namely $X_{n_i} \cup X_{n_k}$, is a subset of all of the terminals. Combining the above mimicking networks by using Lemma 2.4 yields M_{ik}. Since the total number of terminals is constant, the claimed result follows. ▮

We now show how to compute P_{ij} for each arc (i, j) in T. Root T at any node. For a node i, let S_i be the subtree rooted at i. Consider an arc (i, j) such that i is a child of j. Then P_{ij} consists of two values M_i and M_j, where M_i is the mimicking network for the subgraph spanned by S_i, with terminals X_i, and M_j is the mimicking network for the subgraph spanned by $T - S_i$, with terminals X_j. We compute P_{ij} in two phases. In the first

phase we compute M_i for each arc (i, j) with i a child of j. In the second phase, we compute M_j for each such arc.

During the first phase, suppose we are at an arc (i, j), with i a child of j. Suppose also that we have computed the mimicking networks M_l and M_r for the (at most) two arcs connecting i to its children. Then, to obtain M_i, use Lemma 2.4 to combine the mimicking networks M_l, M_r and the mimicking network for the subgraph spanned by i, retaining the terminals X_i. A postorder traversal of T with this operation performed at each arc completes the first phase.

During the second phase, suppose we are at arc (i, j), with i a child of j. Let p and c be the parent of j and the sibling of i, respectively (if they exist). Suppose we have already computed M_p, the mimicking network for the subgraph spanned by $T - S_j$. In the first phase, we have computed M_c, the mimicking network for the subgraph spanned by the subtree rooted at c. Then, use Lemma 2.4 to combine M_p, M_c and the mimicking network for the subgraph spanned by j, retaining terminals X_j. This yields M_j, the mimicking network for the subgraph spanned by $T - S_i$. A preorder traversal of T with this operation performed at each arc completes the second phase.

Each time Lemma 2.4 is invoked, it combines a constant number of networks, each with a constant number of terminals, hence taking constant time. Since the lemma is invoked twice for each arc, we have proved the following result.

LEMMA 3.2. *Let G be an n-vertex network and let (X, T) be its augmented binary tree decomposition of constant width. Then, in time $O(n)$ we can compute P_{ab} for all arcs $(a, b) \in E(T)$.*

We are now ready for our main lemma.

LEMMA 3.3. *Let G be an n-vertex network and let (X, T) be its augmented binary tree decomposition of constant width. For each integer $k \geq 1$, after $O(nI_k(n))$ preprocessing, we can find the mimicking network for G at terminals $X_i \cup X_j$ in time $O(k)$, for any $i, j \in V(T)$. Furthermore, after $O(n)$ preprocessing, we can find this mimicking network in time $O(\alpha(n))$.*

Proof. For each arc (a, b) of T, compute P_{ab} using Lemma 3.2. Use Theorem 2.1 to preprocess T, with the P_{ab} values associated with its arcs, so that queries asking for the product of P values along paths in T can be answered. A query for the product on the path from i to j returns the value $P_{ij} = (M_i, M_j, M_{ij})$. Combine these three mimicking networks using Lemma 2.4, with the desired set of terminals being $X_i \cup X_j$. This yields the mimicking network for G with these terminals. The claimed bounds follow easily by those of Theorem 2.1 and Lemma 3.1. ∎

We can now prove the main result of this section.

THEOREM 3.1. *Let G be an n-vertex network of constant tree-width. For each integer $k \geq 1$, after $O(nI_k(n))$ preprocessing, we can find the value of an s-t min-cut (or max-flow) in time $O(k)$, for each $s, t \in V(G)$. Furthermore, after $O(n)$ preprocessing, we can find the value of an s-t min-cut (or max-flow) in time $O(\alpha(n))$.*

Proof. First, compute a constant-width augmented binary tree decomposition (X, T) of G using Proposition 2.1. Preprocess G and (X, T) using Lemma 3.3.

Let $s \in X_i$ and $t \in X_j$, for some $i, j \in V(T)$. By Lemma 3.3, a single query returns the mimicking network for G at terminals $X_i \cup X_j$. Now simply compute the value of an s-t min-cut (or max-flow) in this mimicking network. Since the size of the mimicking network is constant, the entire computation after the query takes constant time, implying the time bounds in the theorem. ∎

To solve the APMC problem in a bounded tree-width network, simply apply Theorem 3.1 with $k = 2$, i.e., perform $O(n \log n)$ preprocessing so that an s-t min-cut can be computed in constant time. Thus the APMC problem can be solved by querying for s-t min-cuts, for each pair s, t in the network. This proves the following result.

COROLLARY 3.1. *The all-pairs min-cut problem can be solved for bounded tree-width networks in time $O(n^2)$.*

4. MIMICKING NETWORKS OF OUTERPLANAR NETWORKS

In Section 2.2, we described the method of [16] to compute a mimicking network with 2^{2^q} vertices for a network with q terminals. In this section we give an algorithm that finds a mimicking network of an outerplanar network. The mimicking network constructed has size $q^2 2^{q+2}$ (i.e., it is exponentially smaller than the one constructed using the general approach of [16]), and it is a *minor* of the original network (i.e., it can be obtained from the original network by contracting edges, deleting edges, and deleting isolated vertices [19, 22]). The ability to construct mimicking networks that are minors of the original outerplanar networks permits us to construct planar mimicking networks for planar networks in Section 5. In the following, when we speak of an undirected path or cycle, we are referring to a path or cycle ignoring the direction of the edges of the network. We first consider the case of biconnected networks. The general case is based on the biconnected one and is treated later.

4.1. *The Biconnected Case*

Let G be a biconnected outerplanar network with terminal set Q. Then, G has an undirected Hamiltonian cycle. Throughout, we work with a fixed embedding of G, and the boundary of this embedding is the Hamiltonian cycle. Let $1, 2, \ldots, n$ be the numbering of vertices of G in clockwise order along the boundary of this embedding. Let $[i, j]$ denote the interval of vertices in clockwise order along the boundary from vertex i to vertex j, i.e., $[i, j]$ denotes the set $\{i, i+1, \ldots, j\}$ of vertices, if $i \leq j$, and it denotes $\{i, i+1, \ldots, n, 1, \ldots, j\}$, if $i > j$. A *chain* is the set of vertices determined by some interval $[i, j]$.

The main idea of our approach is the following. We show that every minimum R-separating cut, $R \subseteq Q$, divides the vertices of G into at most $2q - 2$ chains. Since there are 2^q such possible subsets R, we have a total of at most $(2q - 2)2^q$ chains in G. The vertex-sets resulting by taking the intersection of all of these chains determine the equivalence classes (and the size) of the mimicking network; outerplanarity is preserved by contracting edges whose endpoints belong to the same equivalence class and replacing multiple edges by single edges. We start with some basic definitions and results.

Any coloring of the vertices of G with green and red colors defines a cut, namely, the cut separating the green vertices from the red ones. For a subset $R \subseteq Q$ of terminals, let $(S, \bar{S})$ be a minimum R-separating cut. We color the vertices of S green and those of $\bar{S}$ red. A *green unit* is defined to be a maximal chain of green vertices, and a *red unit* is defined analogously. Define the *support* of a green unit to be a green terminal such that some (and therefore every) vertex in the unit has an undirected path, consisting only of green vertices, to this terminal. Similarly, define the support of a red unit. We say a green unit is *unsupported* if no vertex in the unit has an undirected path, consisting only of green vertices, to a green terminal. Define an unsupported red unit analogously. A collection of unsupported units is *connected* if there is an undirected path, not including a vertex from any supported unit, between any two units of the collection.

PROPOSITION 4.1. *The cut obtained by changing the color of any maximal monochromatic connected collection of unsupported units is also a minimum R-separating cut.*

Proof. Assume that the color of the connected collection is green. By the maximality of the collection, there is no edge from the collection to any other unsupported green unit, and because the units are unsupported, there is no edge to any supported green unit. Hence, the capacity of the cut obtained by changing the color of the collection to red is not more

than the capacity of the minimum R-separating cut $(S, \bar{S})$. Interchanging the roles of red and green yields the proposition. ∎

PROPOSITION 4.2. *In any minimum R-separating cut in G in which there are no unsupported units, the number of units is at most* $2q - 2$, *where q is the number of terminals*.

Proof. Construct an undirected graph H from the undirected version of G, by contracting each edge between two vertices belonging to the same unit, and replacing multiple edges in the resulting graph by single edges. These operations preserve the outerplanarity of the graph. Each unit of G corresponds to a vertex in H, and the colors of the units induce a coloring of the vertices of H. The vertices of H corresponding to the units of G that contains a terminal are called *special*. The outerplanar embedding of G naturally induces an embedding of H, and we work with this embedding. The following properties of H are easily verified:

(i) H is outerplanar.

(ii) The outer face of H is a Hamiltonian cycle, and the colors of successive vertices on this cycle alternate.

(iii) There are at most q special vertices, and at least one special vertex of each color.

(iv) Every vertex of H has a path, consisting only of vertices of the same color, to a special vertex of the same color.

We claim that any graph with properties (i)–(iv) has at most $2q - 2$ vertices, for $q \geq 2$. Consider a counterexample to the claim with the minimum value of q. Since the counterexample has at least $2q - 1 > q$ vertices, there is a nonspecial vertex. Without loss of generality, assume that there is a red nonspecial vertex. Property (iv) implies that there is a nonspecial red vertex that has an edge to a special red vertex. Property (ii) implies that the path between these two vertices along the Hamiltonian cycle in either direction includes a green vertex. Contradicting this edge splits the Hamiltonian cycle of (ii) into two smaller cycles $\mathscr{C}_1$ and $\mathscr{C}_2$, which share exactly one red vertex. Designate this vertex as special. Consider the two subgraphs induced by the vertices on the two cycles. Each of them contains a green vertex and hence must contain a special green vertex. If not, (i) implies that the corresponding green vertices in H violate (iv). It is now easily verified that both of the subgraphs satisfy (i)–(iv) for some smaller values of q.

Let p_i (resp. t_i) denotes the number of vertices (resp. special vertices) in $\mathscr{C}_i$, $i = 1, 2$. Then, $p_1 + p_2 \geq 2q - 1$ and $t_1 + t_2 \leq q + 1$ (the common vertex is counted twice in both sums). Since H is a counterexample with the minimum value of q, we have that $p_1 \leq 2t_1 - 2$ and $p_2 \leq 2t_2 - 2$.

But $2q - 1 \leq p_1 + p_2 \leq 2(t_1 + t_2) - 4 \leq 2(q + 1) - 4 = 2q - 2$, a contradiction. Consequently, either $p_1 > 2t_1 - 2$ or $p_2 > 2t_2 - 2$, i.e., one of the two subgraphs is a counterexample with a smaller value of q, contradicting the minimality of q. Thus the claim holds.

The proposition follows, since the number of units in G is the same as the number of vertices in H. ▮

We now give an algorithm that finds a minimum R-separating cut satisfying the hypothesis of Proposition 4.2. We first find a minimum R-separating cut using our algorithm given in Section 2.2 and color the units induced by this cut. Then, for each terminal, we find the units that it supports, using a standard graph traversal algorithm. Consider a maximal contiguous (along the Hamiltonian cycle) group of unsupported units, and assume that one of the (supported) units bordering it is green. Mark each of the units in the group, and mark every unsupported unit in each maximal connected collection of unsupported units that includes a unit from the group. Color all of the marked units green, inducing a new R-separating cut. By Proposition 4.1, this is also a minimum R-separating cut. The green units become larger by absorbing the neighboring new green units, and all of the marked units are now supported (by the terminal that supports the bordering green unit). Perform an analogous operation if the bordering units are red. Continue this process until no unsupported units remain.

Maximal contiguous groups can be identified by a walk around the boundary of the embedding, and units can be marked by a standard graph traversal. Note that an edge is traversed once, by exactly one traversal. Thus the total time for all traversals is linear. The time taken by the algorithm is dominated by the q graph traversals done from the q terminals, and the time taken to find a minimum R-separating cut, which is $O(q^2 n)$, where n is the number of vertices in G. We can now prove the following:

LEMMA 4.1. *For any n-vertex biconnected outerplanar network G with terminal set Q, there is a mimicking network $M(G)$ of G at terminals Q such that $M(G)$ is outerplanar and has at most $q2^{q+1}$ vertices, where $q = |Q|$. $M(G)$ can be constructed in $O(q^2 2^q n)$ time. The undirected version of $M(G)$ is a minor of the undirected version of G.*

Proof. Recall the procedure described in Section 2.2 to construct a mimicking network. It finds a minimum R-separating cut for each $R \subseteq Q$ and then replaces each equivalence class of vertices that have not been separated by any cut by a single vertex. When we find R-separating cuts by the algorithm above, each cut divides the vertices into at most $2q - 2$ chains, by Proposition 4.2. This can be viewed as marking at most $2q - 2$

edges on the boundary of the embedding (the edges that delimit the chains). Doing this for each of the 2^q possible subsets R corresponds to marking at most $(2q - 2)2^q$ edges on the boundary of the embedding. The equivalence classes of vertices not separated by any cut are exactly the maximal groups of vertices without any marked edge between two vertices in the same group. Since at most $(2q - 2)2^q$ edges have been marked, there are at most this many equivalence classes.

The mimicking network is constructed by contracting the edges between every two vertices belonging to the same equivalence class, and replacing multiple edges by a single edge of capacity equal to the sum of the capacities of the edges it replaces. As before, outerplanarity is preserved. The running time of the algorithm follows by Lemma 2.3 and Theorem 3.1 (an outerplanar network has tree-width 2). ▮

4.2. *The General Case*

We now consider the case of general outerplanar networks. Recall that a *biconnected component* of a graph is a maximal induced subgraph with the property that deleting any vertex from the subgraph does not disconnect it. It follows that two biconnected components have at most one vertex in common, called an *articulation vertex*.

There are two easy approaches to dealing with a nonbiconnected outerplanar network. As we shall see, however, both of them are inadequate for our purposes; namely, in Section 5 we want to construct mimicking networks for planar and sparse networks using the hammock decomposition [14], which decomposes a planar or sparse graph into a number of edge-disjoint outerplanar subgraphs (called *hammocks*), each of which is connected with the rest of the graph via at most four vertices (called *attachment vertices*). For this reason, we want to construct mimicking networks for the hammocks and then combine them using Lemma 2.4 to obtain a mimicking network for a planar or sparse network. Our goal is to find a uniform way to handle outerplanar networks regardless of whether they are hammocks or not.

The first (easy) approach is to add an appropriate number of edges with zero capacity to eliminate the articulation vertices. This approach may fail for a planar network G in the case where an articulation vertex v of a hammock H is also an attachment vertex of H (i.e., it is incident to edges not belonging to H). Now, insertion of additional edges may destroy the planarity of G. This problem can be handled by the second approach, which is based on the observation that a nonbiconnected outerplanar graph can be obtained from a biconnected one by contracting edges. This means that we can replace an articulation vertex v shared by k biconnected components ($k \geq 2$) with k vertices $v_0, v_1, \ldots, v_{k-1}$ and connect

them (in a ring-like fashion) with edges $(v_{i \bmod k}, v_{(i+1) \bmod k})$ of infinite capacity, $0 \leq i \leq k-1$. By appropriately dividing the edges incident to v among the new vertices $v_0, v_1, \ldots, v_{k-1}$, we can preserve both the outerplanarity of H and the planarity of G. Moreover, if v is a terminal, then we can designate one of $v_0, v_1, \ldots, v_{k-1}$ as a terminal, and hence the total number of terminals remains the same. This approach, however, may fail in the case where v is an attachment vertex of H, since it may increase the number of attachment vertices from 4 to $4k$, a quantity that may not be bounded by a constant. And this rules out the application of Lemma 2.4, because either the terminal set Q_i of a subnetwork is nonconstant or its attachment set C_i is no longer a subset of Q_i.

For all the above reasons, we have chosen to follow a different approach, which is described in the rest of this section. The main idea is to divide the given outerplanar network G into appropriate groups of biconnected components such that a group is either a single biconnected component containing at least one terminal (''singleton''), or a sequence of biconnected components containing no terminals and such that every articulation vertex is shared by at most two biconnected components (''pipe''). We then find mimicking networks for these groups and join them at the corresponding articulation vertices to get a mimicking network for G.

We start by discussing some structural properties of nonbiconnected graphs. It is well known that the biconnected components of a graph have a ''tree'' structure, in the sense that any simple path between two fixed vertices must pass through the same set of articulation vertices in the same order. Select any biconnected component and call it the *root*. Define the *children* of the root to be those components that share an articulation vertex with the root, and define the *parent* of these components to be the root. Inductively, define the children of any component B that has a parent to be those components that share an articulation vertex with B but not with B's parent (if a component shares an articulation vertex with both B and B's parent, then all three components share the same articulation vertex). Construct a graph with one vertex for each biconnected component and an edge between each vertex and its parent. This graph will be a tree, which we call the *tree of biconnected components*. A *leaf component* is a biconnected component corresponding to a leaf in this tree. The *degree* of a component is the degree of the vertex corresponding to it in the tree. As for tree decomposition, we shall use ''nodes'' (resp. ''arcs'') to refer to the vertices (resp. edges) of the tree of biconnected components.

THEOREM 4.1. *For any n-vertex outerplanar network G with terminal set Q, there is a mimicking network $M(G)$ of G at terminals Q such that $M(G)$ is*

outerplanar and has at most $q^2 2^{q+2}$ *vertices, where* $q = |Q|$. *Moreover,* $M(G)$ *can be constructed in* $O(q^2 2^q n)$ *time. The undirected version of* $M(G)$ *is a minor of the undirected version of* G.

Proof. We assume G is connected; if not, we simply work with each of the connected components of G separately. For reasons of clarity of notation, we will refer to the terminals of G as *sockets*. In the following, when we speak of the biconnected components of G, we are referring to the biconnected components ignoring the direction of the edges. When we speak of flows, however, we take the direction of edges into account.

We transform G into a new graph G' as follows. Consider the tree of biconnected components of G. Consider a leaf component that contains no sockets, except for its articulation vertex. We contract all edges of this leaf component (i.e., we delete the leaf component), and its articulation vertex denotes the contracted component. We repeat this process in the remaining graph until every leaf component in the tree of biconnected components contains a socket. The resulting graph is the graph G'. We claim that a mimicking network for G' is also a mimicking network for G.

Let G'' be the graph obtained from G by removing one such leaf component B with articulation vertex v. To prove that a mimicking network for G'' is also a mimicking network for G, it suffices to show that for any subset R of the sockets, the minimum R-separating cuts in G and G'' have the same capacity, or, equivalently, the maximum R-flows in G and G'' have the same value. This is immediate since $B - v$ has no sockets, which implies that the net flow into $B - v$ is always zero. The claim is thus proved.

Partition the nodes of the tree of biconnected components of G' into groups as follows. (When we refer to a node containing a socket, we mean that the biconnected component corresponding to it contains a socket.) First assign each socket to exactly one of the nodes containing it (the reason for this is to assign sockets that are articulation vertices to one of the components that share it). Now, place each node containing a socket into a group by itself. Place in a group by itself each node of degree at least 3 that is not yet in any group. Finally, each maximal connected set of nodes that are not yet in any group is put together in a single group. This last type of group is called a *pipe*. Thus the nodes of the tree of biconnected components of G' are partitioned into two types of groups, namely, singleton groups and pipes. It is easy to check that if components $B_1, \ldots, B_p$ correspond to the nodes in a pipe, one can label the left and right articulation vertices of component B_i with l_i and r_i such that $r_i = l_{i+1}$ for $1 \leq i < p$. Articulation vertices l_1 and r_p are called the *end vertices* of the pipe. The only vertices in these components that could be sockets are the end vertices.

The mimicking network for G' is obtained by constructing, for each group, the mimicking network of the corresponding biconnected component, and then joining the mimicking networks at the corresponding articulation vertices.

The mimicking network of a singleton group is computed by invoking Lemma 4.1 with terminals as the articulation vertices and sockets contained in the group.

The mimicking network of a pipe H is computed as follows. The terminals are the end vertices, where the articulation vertices of the components $B_1, \ldots, B_p$ of H are labeled as before. Fix an embedding for each component B_i, and label the vertices of this component in clockwise order along the boundary of the embedding, starting with the left articulation vertex l_i. For $i = 1, \ldots, p - 1$, define pred(i) to be the predecessor vertex of the right articulation vertex r_i in component B_i. For $i = 2, \ldots, p$, define succ(i) to be the successor vertex of the left articulation vertex l_i in component B_i. Now, construct a biconnected outerplanar network H^* from the pipe H by introducing new edges (pred(i), succ($i + 1$)) of zero capacity. Figure 2 illustrates an example for a pipe with four components. (The embeddings of some components B_i in H may have to be *flipped* to get an outerplanar embedding of H^*, i.e., interchange the embedding of the vertices in the chain $[l_i, r_i]$ with the embedding of the vertices in the chain $[r_i, l_i]$, except for l_i and r_i.) Using Lemma 4.1 compute the mimick-

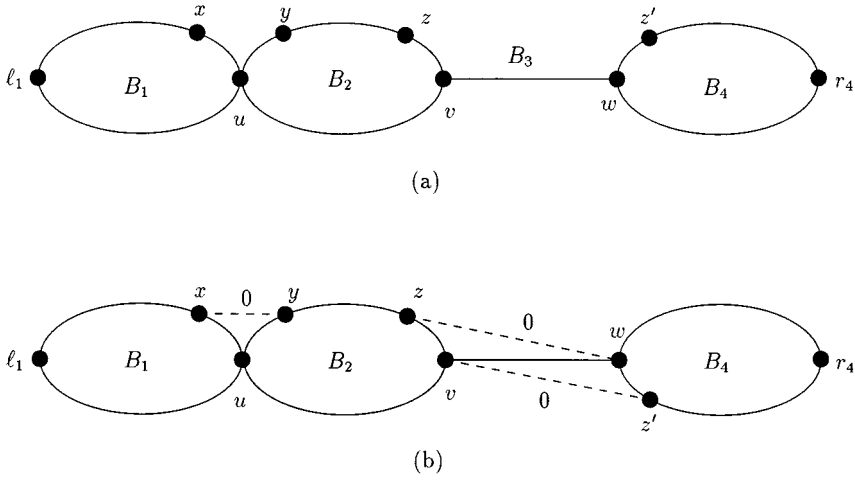

FIG. 2. (a) A pipe H consisting of four biconnected components. We have that $u = r_1 = l_1$, x = pred(1), y = succ(2), $v = r_2 = l_3$ = pred(3), z = pred(2), $w = r_3 = l_4$ = succ(3), and z' = succ(4). (b) The new biconnected network H^*. The embedding of B_4 had to be flipped.

ing network of H^* at terminals l_1 and r_p. This mimicking network has at most four vertices (Lemma 4.1 implies, when $q = 2$, a bound of 16 on the number of vertices; but it is clear from the proof of the lemma that the correct bound is $2(2q - 2) = 4$, when $q = 2$). Transform this mimicking network into a mimicking network of the pipe H as follows. If, for all i, either (a) pred(i) and succ($i + 1$) belong to different equivalence classes, or (b) pred(i) and succ($i + 1$) belong to the same equivalence class and vertex r_i also belongs to this class, then we are done by taking the mimicking network of H^* as the mimicking network of H. Otherwise, consider an equivalence class, corresponding to a vertex of this mimicking network, such that both pred(i) and succ($i + 1$) belong to this class and vertex r_i does not. Split the equivalence class into two classes, one containing pred(i) and all of the vertices of the class that belong to components $B_1, \ldots, B_i$, and the other containing succ($i + 1$) and the remaining vertices of the class. The capacity of an edge joining a new class with some other class is equal to the sum of the capacities of the edges in H between these two equivalence classes. We perform, the splitting operation for all such equivalence classes, and the resulting network is a mimicking network for the pipe H at the end vertices. The correctness of this procedure follows from the following observation, whose proof is immediate by the definition of equivalence classes. Splitting an equivalence class, corresponding to a vertex of a mimicking network of any network, still results in a mimicking network. Consequently, the mimicking network of H thus constructed has at most eight vertices.

The mimicking network of G', which is obtained by joining the mimicking networks of singleton groups and pipes, is also a mimicking network $M(G)$ of G, as proved earlier. The network $M(G)$ is outerplanar, since each equivalence class created in its construction is a connected subgraph of G. Constructing the tree of biconnected components and forming the groups can be done in linear time. Observing that the sum of the number of vertices in all components is $O(n)$, we have the claimed time bound for the construction.

It remains to bound the size of $M(G)$. Let l be the number of leaves of the tree of biconnected components of the graph G'. Then, the number of nodes of degree at least 3 is at most $l - 2$. Consequently, the number of singleton groups formed is at most $q + (l - 2)$, where the first term is the contribution of nodes containing sockets and the second term of nodes of degree at least three. It is easy to argue that the number of pipes is at most $2l - 3$. Since each leaf contains a distinct socket, the number l of leaves is at most the number q of sockets. Thus the number of singleton groups formed is at most $2q - 2$, and the number of pipes is at most $2q - 3$.

The number of articulation vertices of any component is bounded by its degree, which is bounded by $q - i$, where i is the number of sockets it

contains, since all arcs leaving a node must lead to leaves containing distinct sockets. Thus the number of terminals in the mimicking network of any group is at most q. The number of vertices in the mimicking network of a singleton group is at most $q2^{q+1}$, by Lemma 4.1, and the number of vertices in the mimicking network of a pipe is at most eight. Hence, the total number of vertices in the mimicking network is at most $(2q-2)q2^{q+1} + (2q-3)\cdot 8 \leq q^2 2^{q+2}$. This completes the proof of the theorem. ▮

5. SPARSE AND PLANAR NETWORKS

Our algorithms for sparse and planar networks are based on the so-called *hammock decomposition*. Frederickson [14] shows how to decompose a sparse graph G into γ outerplanar subgraphs, called *hammocks*, each of which is connected to the rest of the graph via at most four vertices, called *attachment vertices*. The parameter γ is $O(g+p)$, where g is the genus of G and p is the minimum number of faces that cover all vertices of G, over all possible cellular embeddings into an orientable surface of genus g. Note that $g+p$ is the minimum possible number of hammocks in such a decomposition. It is known that γ can vary between 1 and $\Theta(n)$. The algorithm in [14] runs in linear time and does not require an embedding to be provided with the input. In this section, we give algorithms whose running times depend on γ, and which perform well when $\gamma = o(n)$. The idea is to decompose the given sparse (or planar) network G into hammocks, construct mimicking networks for the hammocks and then combine them (by retaining the appropriate terminals) by using Lemma 2.4 to obtain a mimicking network for G. Then, by using standard algorithms, we can compute min-cuts (or max-flows) in G.

Let G be a sparse network that is decomposed into hammocks $H_1, \ldots, H_\gamma$. Let A_r be the set of (at most four) attachment vertices of H_r, $1 \leq r \leq \gamma$. We now show how to preprocess G so that s-t min-cuts (or max-flows) can be efficiently found. Let $s \in V(H_i)$ and $t \in V(H_j)$. Define G_{ij} to be the network obtained by replacing each hammock H_k, $k \notin \{i, j\}$, by its (constant size) mimicking network at terminals A_k. The terminals of G_{ij} are $A_i \cup A_j$. Note that G_{ij} has $O(\gamma)$ vertices and edges. Construct G_{ij} and find the mimicking network for G_{ij} at terminals $A_i \cup A_j$. Find the mimicking network for H_i at terminals $\{s\} \cup A_i$ and for H_j at terminals $\{t\} \cup A_j$. (If $i = j$, then find the mimicking network for H_i at terminals $\{s, t\} \cup A_i$.) Combining these networks yields a mimicking network for G at terminals $\{s, t\} \cup A_i \cup A_j$. Now the value of an s-t min-cut (or max-flow) can be found by using a standard algorithm. Note that the mimicking

network is of constant size. The correctness of the approach follows by Lemma 2.4.

To estimate the time complexity, finding mimicking networks for the hammocks at the appropriate terminals by using Theorem 4.1 takes $O(n)$ time (over all hammocks). Now, constructing G_{ij} takes $O(\gamma)$ time, and finding its mimicking network takes $O(\gamma^2 \log \gamma)$ time, when we apply Lemma 2.3 with a max-flow algorithm for which $F(G) = O(nm \log n)$ on an n-vertex, m-edge network G (see, e.g., [1]). The remaining computation takes constant time. We summarize the above discussion:

THEOREM 5.1. *The value of an s-t min-cut (or max-flow) in an n-vertex sparse network G can be computed in time $O(n + \gamma^2 \log \gamma)$, where γ is the number of hammocks of G.*

If G is a planar network, the use of Theorem 4.1 in the above procedure ensures that G_{ij} is a minor of G, and is therefore planar. Now the time required to compute the mimicking network for G_{ij} is $O(\gamma \log \gamma)$, by applying Lemma 2.3 with the max-flow algorithm in [23] (for which $F(G) = O(n \log n)$ for an n-vertex planar network G). It follows that

THEOREM 5.2. *The value of an s-t min-cut (or max-flow) in an n-vertex planar network G can be computed in time $O(n + \gamma \log \gamma)$, where γ is the number of hammocks of G.*

To solve the APMC problem, preprocess the H_r's, $1 \le r \le \gamma$, using $O(|H_r| \cdot \log |H_r|)$ time, so that for any $s, t \in V(H_r)$ the mimicking network for H_r at terminals $\{s, t\} \cup A_r$ can be found in constant time. For each $i, j \in \{1, 2, \ldots, \gamma\}$, construct G_{ij} and find its mimicking network. Now for each $s, t \in V(G)$, such that $s \in V(H_i)$ and $t \in V(H_j)$, find the mimicking network for H_i at terminals $\{s\} \cup A_i$ and for H_j at terminals $\{t\} \cup A_j$. (If $i = j$, then find the mimicking network for H_i at terminals $\{s, t\} \cup A_i$.) Combine these mimicking networks with the mimicking network for G_{ij} and find the value of an s-t min-cut, as before. Once the H_r's have been preprocessed and the mimicking networks for the G_{ij}'s found, computing an s-t min-cut takes constant time for each pair s, t. Hence, the following result has been established.

THEOREM 5.3. *The all-pairs min-cut problem for an n-vertex planar (resp. sparse) network G can be solved in $O(n^2 + \gamma^3 \log \gamma)$ (resp. $O(n^2 + \gamma^4 \log \gamma)$) time, where γ is the number of hammocks of G.*

Remark. For the case of sparse networks, we do not necessarily need Theorem 4.1 to compute mimicking networks for the hammocks. Instead, we can make use of the fact that outerplanar networks have tree-width 2, and find mimicking networks for the hammocks at the desired terminals by adding the attachment vertices of a hammock to every set X_i of the

augmented tree decomposition (X, T) associated with the hammock and by using Lemmas 3.2 and 3.3. This approach is described in [3].

6. OUTPUTTING THE EDGES CROSSING AN *s-t* MIN-CUT

In this section we outline an extension of the methods in Sections 2.2, 3, 4, and 5 that allows us to output the edges crossing an *s-t* min-cut in time linear in the number of edges in the cut.

The essential feature is the computation of supplementary information when a mimicking network is computed. Let G be a network and let $M(G)$ be its mimicking network, as computed by the method described in Section 2.2, or, if G is outerplanar, by the method given in Section 4. In both constructions, each vertex of $M(G)$ represents a subset of the vertices of G, and each edge (u, v) of $M(G)$ represents a subset of the edges of G, namely, the edges between the subsets of vertices of G represented by u and v. During the construction of $M(G)$, for each edge e of $M(G)$ we compute a value $trace(e)$, which is a list of the edges of G that e represents. It is easily verified that distinct edges of $M(G)$ represent disjoint subsets of edges of G.

For every mimicking network we compute, we will also compute the trace information associated with their edges. For edges of the input network, the trace value of an edge is simply the edge itself. For reasons of efficiency, which will become clear later, we have one special condition: if an edge e of $M(G)$ represents a single edge e' of G, then $trace(e)$ is defined to be the same as $trace(e')$. In other words, instead of being a singleton list containing e, $trace(e)$ is the same list as $trace(e')$. This condition ensures that except for edges of the original input network, the trace value of each edge is a list with at least two elements. Regarding the elements in the trace value of an edge as the children of the edge, we have that each edge e is the root of a tree defined by the trace values, whose leaves are edges of the input network. We call this tree the *trace subtree* of e. It is not hard to see that the leaves of the trace subtree are exactly those edges of the input network that e represents. Furthermore, the condition above ensures that every nonleaf vertex in the trace subtree has at least two children.

Consider the method used in Section 3 to compute an *s-t* min-cut in a network G of bounded tree-width. Then, as in the proof of Theorem 3.1, we compute a mimicking network $M(G)$ of constant size, whose terminals include s and t, for the input network G. We compute an *s-t* min-cut in $M(G)$, which corresponds to an *s-t* min-cut in G in the natural way. Each

edge crossing the cut in $M(G)$ represents a subset of edges crossing the cut in G, i.e., the leaves of the trace subtree of the edge. Any standard tree traversal algorithm will output the leaves of the trace subtree in time linear in the size of the tree, which is linear in the number of leaves, since each nonleaf vertex has at least two children. Doing this for each edge crossing the cut in $M(G)$ outputs in linear time all of the edges crossing the cut in G. This yields the following result.

THEOREM 6.1. *Let G be an n-vertex network of constant tree-width. For each integer $k \geq 1$, after $O(nI_k(n))$ preprocessing, we can output the edges crossing an s-t min-cut in time $O(k + L)$, where L is the number of edges crossing the cut. Furthermore, after $O(n)$ preprocessing, we can output the edges crossing an s-t min-cut in time $O(\alpha(n) + L)$.*

Consider the method used in Section 5 to compute the value of an s-t min-cut in a planar or sparse network. The final step in the method consists of finding a min-cut in a mimicking network of constant size. From this, the edges that cross the min-cut in the mimicking network can easily be found. Now, as above, the trace information associated with each of these edges can be output in time linear in the number of edges crossing the min-cut in the original network. Thus, we have

THEOREM 6.2. *Let G be an n-vertex sparse or planar network. Let T be the time taken to compute an s-t min-cut in G by the appropriate algorithm in Section 5. Then, the edges crossing the cut can be output in time $O(T + L)$, where L is the number of edges crossing the cut.*

7. CHARACTERIZATION OF FLOWS IN MULTITERMINAL NETWORKS

In [16] necessary and sufficient conditions are derived for an external flow to be realizable:

LEMMA 7.1 ([16]). *An external flow $(x_1, \ldots, x_q)$ is realizable in a network G with terminals $Q = \{a_1, \ldots, a_q\}$, iff (i) $\sum_{a_p \in Q} x_p = 0$ and (ii) $\sum_{a_r \in R} x_r \leq b_R$, $\forall R \subseteq Q$, where b_R is the minimum capacity of an R-separating cut.*

Thus the realizable external flows of a network with q terminals can be characterized by the above system of 2^q linear inequalities, where each inequality is represented by the pair (R, b_R). A system of inequalities for a network G, of the form as in Lemma 7.1, is called the *external flow inequalities* of G *at terminals* Q. The external flow inequalities can be obtained by computing the capacities of minimum R-separating cuts in G, for every $R \subseteq Q$.

Suppose we wish to combine several networks by identifying terminals, in a manner similar to that in Lemma 2.4. In [16] the following lemma is proved, by combining the external flow inequalities of the given networks using linear programming methods. We give a simpler proof avoiding linear programming. We note that the proof in [16] results in an algorithm with running time exponential in the square of the total number of terminals, whereas our proof results in a time that is exponential in the total number of terminals.

LEMMA 7.2. *Let $G = \bigcup_{i=1}^{m} G_i$, where the G_i's are edge-disjoint, and let C_i be the attachment set of G_i. Assume that C_i is a subset of the terminals Q_i in G_i, for all i. Given the external flow inequalities for each G_i at terminals Q_i, and a set $Q' \subseteq Q = \bigcup_{i=1}^{m} Q_i$ of terminals, we can compute the external flow inequalities for G at terminals Q' in time $O(q2^q)$, where $q_i = |Q_i|$ and $q = q_1 + \cdots + q_m$.*

Proof. By repeated applications of Lemma 2.1, each realizable external flow in G, at terminals Q, is the sum of realizable external flows in each G_i at terminals Q_i. Let $R \subseteq Q$, and define $R_i = Q_i \cap R$. Let the realizable external flow, x, that maximizes $\sum_{r \in R} x_r$ be the sum of external flows $x^{(i)}$, for each i. Then, in G_i, the flow $x^{(i)}$ must maximize $\sum_{r \in R_i} x_r^{(i)}$, and from the external flow inequalities of G_i, the value of $x^{(i)}$ is b_{R_i}. Hence, we have $\sum_{r \in R} x_r = \sum_{i=1}^{m} \sum_{r \in R_i} x_r^{(i)} = \sum_{i=1}^{m} b_{R_i}$.

Now, given the external flow inequalities for G_i at terminals Q_i, the algorithm used to compute the external flow inequalities of G at terminals Q is simple. For each $R \subseteq Q$, compute $R_i = Q_i \cap R$. Find the m inequalities of the form $\sum_{r \in R_i} x_r \leq b_{R_i}$, in the flow inequalities of the G_i's, and create an inequality $\sum_{r \in R} x_r \leq \sum_{i=1}^{m} b_{R_i}$ for G. This yields the external flow inequalities for G at terminals Q. The entire computation can be done using standard methods in time $O(q2^q)$. (The above argument is different from the argument in [16], and results in better running time. The rest of the proof is similar to what is done in [16], and is included for completeness.)

To find the external flow inequalities of G at terminals $Q' \subseteq Q$, we have to drop some terminals—this corresponds to setting all variables x_i, where $a_i \in Q - Q'$, to zero, in the inequalities for G at terminals Q. To see this, observe that the set of all realizable flows in G with terminals Q' is precisely that subset of all realizable flows in G with terminals Q in which the net flow out of any terminal in $Q - Q'$ is zero. Set the variables corresponding to vertices in $Q - Q'$ to zero. The resulting collection of inequalities describes the realizable external flows in G at terminals Q'. We only have to remove the redundant inequalities. Consider a fixed $R \subseteq Q'$. In the collection of inequalities, there will be an inequality of the form $\sum_{a_r \in R} x_r \leq \cdots$ for each set $P \subseteq Q$, satisfying $P \cap Q' = R$. From

each such set of inequalities we retain only one inequality with the minimum right-hand side, since all of the others are redundant. Doing this for every $R \subseteq Q'$ yields the desired set of inequalities. Once again, using standard methods, this computation can be done in time $O(q2^q)$. ▮

8. CLOSING REMARKS

We presented efficient algorithms for the all-pairs min-cut problem on bounded tree-width, planar, and sparse networks. The constants in the running time of the algorithms are not small, since they depend on the size of the mimicking networks. For example, in the algorithm for networks of tree-width t, the constant is $2^{2^{O(t)}}$. Designing practical algorithms for the APMC problem on sparse networks remains an important open question.

ACKNOWLEDGMENTS

We are indebted to an anonymous referee for carefully reading the paper and making many valuable comments and suggestions that improved the overall presentation and simplified the proof of Lemma 2.2.

REFERENCES

1. R. Ahuja, T. Magnanti, and J. Orlin, "Network Flows," Prentice-Hall, 1993.
2. N. Alon and B. Schieber, "Optimal Preprocessing for Answering On-line Product Queries," Technical Report 71/87, Tel-Aviv University, 1987.
3. S. Arikati, S. Chaudhuri, and C. Zaroliagis, All-pairs min-cut in sparse networks, *in* "Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)," LNCS 1026, pp. 363–376, Springer-Verlag, Berlin/New York, 1995.
4. S. Arnborg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey, *BIT* **25** (1985), 2–23.
5. H. Bodlaender, NC-algorithms for graphs with small treewidth, *in* "Proceedings of the 14th Workshop on Graph-Theoretic Concepts in Computer Science (WG'88)," LNCS 344, pp. 1–10, Springer-Verlag, Berlin/New York, 1989.
6. H. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, *in* "Proceedings of the 25th ACM Symposium on Theory of Computing (STOC'93)," pp. 226–234, Assoc. Comput. Mach., New York, 1993.
7. H. Bodlaender, A tourist guide through treewidth, *Acta Cybernet*. **11**(1–2) (1993), 1–21.
8. S. Chaudhuri and C. Zaroliagis, Shortest path queries in digraphs of small treewidth, *in* "Proceedings of the 22nd International Colloquium on Automata, Languages and Programming (ICALP'95)," LNCS 944, pp. 244–255, Springer-Verlag, Berlin/New York, 1995.
9. B. Chazelle, Computing on a free tree via complexity-preserving mappings, *Algorithmica* **2** (1987), 337–361.

10. J. Cheriyan, T. Hagerup, and K. Mehlhorn, Can a maximum flow be computed on $o(nm)$ time? *in* "Proceedings of the 17th Colloquium on Automata, Languages and Programming (ICALP'90)," LNCS 443, pp. 235–248, Springer-Verlag, Berlin/New York, 1990.
11. H. Djidjev, G. Pantziou, and C. Zaroliagis, On-line and dynamic algorithms for shortest path problems, *in* "Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science (STACS'95)," LNCS 900, pp. 193–204, Springer-Verlag, Berlin/New York, 1995.
12. L. R. Ford and D. R. Fulkerson, Maximal flow through a network, *Canad. J. Math.* **8** (1956), 399–404.
13. G. N. Frederickson, Searching among intervals and compact routing tables, *in* "Proceedings of the 20th Colloquium on Automata, Languages and Programming (ICALP'93)," LNCS 700, pp. 28–39, Springer-Verlag, Berlin/New York, 1993.
14. G. N. Frederickson, Using cellular graph embeddings in solving all pairs shortest path problems, *J. Algorithms* **19** (1995), 45–85.
15. D. Gusfield, A graph theoretic approach to statistical data security, *SIAM J. Comput.* **17** (1988), 552–571.
16. T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde, Characterizations of k-terminal flow networks and computing network flows in partial k-trees, *in* "Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms (SODA'95)," pp. 641–649, Assoc. Comput. Mach., New York, and SIAM, Philadelphia, 1995.
17. R. E. Gomory and T. C. Hu, Multi-terminal network flows, *J. SIAM* **9** (1961), 551–570.
18. P. Klein, S. Rao, M. Rauch, and S. Subramanian, Faster shortest-path algorithms for planar graphs, *in* "Proceedings of the 26th ACM Symposium on Theory of Computing (STOC'94)," pp. 27–37, Assoc. Comput Mach., New York, 1994.
19. N. Robertson and P. D. Seymour, Graph minors—a survey, *in* "Surveys in Combinatorics," pp. 153–171, Cambridge Univ. Press, Cambridge, 1985.
20. D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comput. Syst. Sci.* **26** (1983), 362–391.
21. R. E. Tarjan, "Data Structures and Network Algorithms," SIAM, Philadelphia, 1983.
22. K. Wagner, Über eine Eigenschaft der ebenen Komplexe, *Math. Ann.* **114** (1937), 570–590.
23. K. Weihe, Maximum (s, t)-flows in planar networks in $O(|V|\log|V|)$ time, *in* "Proceedings of the 35th IEEE Symposium on Foundations of Computer Science (FOCS'94," pp. 178–189, IEEE Comput. Soc., Los Alamitos, CA, 1994.