All-Pairs Min-Cut in Sparse Networks* (FSTTCS'95)

Srinivasa R. Arikati Shiva Chaudhuri Christos D. Zaroliagis

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

Abstract. Algorithms for the all-pairs min-cut problem in bounded tree-width and sparse networks are presented. The approach used is to preprocess the input network so that, afterwards, the value of a min-cut between any two vertices can be efficiently computed. A tradeoff between the preprocessing time and the time taken to compute min-cuts subsequently is shown. In particular, after $O(n \log n)$ 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 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 $O(n^2 + \gamma^4 \log \gamma)$.

1 Introduction

Network flows are of fundamental importance in computer science, engineering and operations research, to name a few areas. The textbook [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 graph (called a network) with nonnegative capacities edges, and two distinguished vertices s and t. An s-t cut in this graph 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 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 s-t max-flow. The max-flow min-cut theorem

* This work was partially supported by the EU ESPRIT Basic Research Action No. 7141 (ALCOM II).

[11] 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. 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 [15] that in undirected networks, the APMC problem can be solved by solving $n-1$ well-chosen max-flow problems. Thus the APMC problem on undirected network takes $O((n-1)F(n,m))$ time ($F(n,m)$ 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 network is $O(\min\{n^3/\log n, nm\log n\})$ [9, 16]. However, one can do better when the structure of the input network is known. Recently, it was shown that the max-flow problem in directed or undirected *bounded tree-width* networks can be solved in $O(n)$ time [14]. The tree-width is a parameter that, intuitively, indicates how close the structure of the network is to a tree (see Section 2.2 for a formal definition). The class of bounded tree-width networks includes (among others) outerplanar networks, series-parallel networks, networks with bounded bandwidth or cutwidth [3, 6]. 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 is completely different from previous approaches. Instead of computing a number of separate max-flows from scratch, our approach is to preprocess the network so that, subsequently, the value of an s-t 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 max-flow subsequently. The tradeoff is: after $O(nI_k(n))$ preprocessing, the value of an s-t max-flow can be computed in $O(k)$ time, for each integer $k \geq 1$. The function $I_k(n)$, defined formally in Section 2.3, 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 max-flow can be computed in $O(\alpha(n))$ time (where $\alpha(n)$ is the inverse-Ackermann function; see Section 2.3).

We use the algorithm for bounded tree-width networks to develop an algorithm for sparse networks, based on a decomposition of a sparse network into

networks of bounded tree-width. Frederickson [12] showed how to decompose a 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 4 vertices. (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 an s-t 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 in [16] and does better if $\gamma = o(n)$. We also show how to solve the APMC problem in time $O(n^2 + \gamma^4 \log \gamma)$ on a sparse network.

The above algorithms output the value of of a max-flow or min-cut. In case the actual min-cut is desired, we show how to output the edges crossing a min-cut in time linear in the size of the output. Specifically, for bounded tree-width networks, we show that, for each $k \geq 1$, after $O(nI_k(n))$ preprocessing, the edges crossing an s-t min-cut can be output in time $O(k + L)$, where L is the number of edges crossing the cut. After $O(n)$ preprocessing this can be done in time $O(\alpha(n) + L)$.

Necessary and sufficient conditions (called *external flow inequalities*) for realizable flows in multi-terminal networks are derived in [14]. An important lemma in that paper shows how to combine the flow inequalities of a number of subnetworks to obtain a single set of flow inequalities for the combined network. Their proof uses linear programming. We give a simple and direct proof of the same result and our proof avoids linear programming.

Our algorithms use the construction of a small network that "mimics" the flow behaviour of a large network. This idea was developed in [14]. The structure of the algorithms for bounded tree-width networks is derived from an algorithm used to solve shortest path queries [7]. The hammock decomposition technique has been used in shortest path problems (see e.g. [10, 12, 13]). To our knowledge, this paper is the first application of this technique to a different problem.

2 Preliminaries

2.1 Mimicking networks of multi-terminal 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 a distinguished subset, Q, of its vertices. A *flow* in G is an assignment of a nonnegative real value f_e not greater than c_e to each edge e such that the net flow out of each non-terminal 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 p. A *realizable external flow* is an external flow such that there exists a flow in which the net flow out of each terminal p is x_p. A *cut* is defined by a subset S of V, called its *defining subset*. The *capacity* of a cut is the sum of capacities of edges going from vertices in S to vertices in $V \backslash S$. For a subset R of Q, an *R-separating cut* is a cut with defining subset S where $Q \cap S = R$.

In the special case of a network with two terminals s and t, an *s-t max-flow* is a flow that maximizes the value of the net flow out of s, which is called the value of the max-flow. An $\{s\}$-separating cut is called an *s-t 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. A direct consequence of this theorem is the following.

Corollary 1. *If f is an s-t flow and C is an $\{s\}$-separating cut such that the value of f equals the capacity of C, then f is an s-t max-flow and C is an $s-t$ min-cut.*

Let G be a network with terminal set Q. 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 [14], it is shown that for any G, there exists a mimicking network with 2^{2^q} vertices, where q is the number of terminals of G. Henceforth, when we speak of mimicking networks, we will require that they have no more than this many vertices.

The mimicking network of [14] is constructed by finding 2^q min-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 these cuts form equivalence classes. Induction on q shows that there can be at most 2^{2^q} equivalence classes. $M(G)$ is constructed by replacing each equivalence class with a single vertex. The edge between two vertices of $M(G)$ in a given direction has 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 is computed by the standard method of introducing a new source, connected to each vertex in R with infinite capacity edges, and a new sink to which each vertex in $Q \backslash R$ is similarly connected, and computing a max-flow from the source to the sink. Thus we have the following result.

Proposition 2. *A mimicking network of a network G with q terminals can be computed in $O(2^q F(G))$ time, where $F(G)$ is the time required to compute a 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 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. Using Proposition 2 and computing max-flows with an $O(n^3)$ algorithm (see e.g. [1]), we can show the the following (which is a reformulation of the result in [14]).

Lemma 3. *Let $G = G_1 \cup \ldots \cup G_m$, 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 $Q \subseteq \cup_{i=1}^{m} Q_i$, we can compute the mimicking network $M(G)$ for G at terminals Q in time $O(2^q \cdot (\sum_{i=1}^{m} 2^{2^{q_i}})^3)$, where $q_i = |Q_i|$ and $q = |Q|$.*

2.2 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 and 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.

Bodlaender [5] gave a linear-time algorithm to compute a constant width tree decomposition of a graph with constant tree-width. In [4] 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 vertex i of T. Thus, in a sense, each edge is represented in at least one vertex of T. For our applications, we need to explicitly associate each edge of G with exactly one vertex of T. We will, therefore, compute an *augmenting function* $h : E(G) \longrightarrow V(T)$, satisfying the property that both endpoints of an edge are present in the set belonging to the vertex 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 applications. 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:

Proposition 4. *Given an n-vertex graph G of constant tree-width t, in $O(n)$ time we can compute an augmented binary tree decomposition of G of width $O(t)$.*

2.3 Tree products

For a function g let $g^{(1)}(n) = g(n)$; $g^{(i)}(n) = g(g^{(i-1)}(n))$, $i > 1$. Define $I_0(n) = \lceil \frac{n}{2} \rceil$ 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, 8].

Theorem 5. *Let $\bullet$ be an associative operator defined on a set S, such that for $q, r \in S$, $q \bullet r$ can be computed in constant time. Let T be a tree with n nodes such that each edge 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.*

3 Bounded tree-width networks

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 vertices of T', i.e. $V(G') = \cup_{i \in V(T')} X_i$. The edges of G' are those edges that the augmenting function maps to vertices in T', i.e. $E(G') = \{e \in E(G) : h(e) \in V(T')\}$. It is easy to check that vertex-disjoint subtrees span edge-disjoint subgraphs. (In fact it is only to ensure this property that we introduce the augmenting function.)

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 edges on this path breaks up T into three components T_i, T_j, the ones containing i and j respectively, and the remaining component T_{ij}. If $path(i, j)$ is an edge, then the first and last edges on the path are the same; consequently, the component T_{ij} is empty. The vertices in T_{ij} that are adjacent to i and j are denoted n_i and n_j respectively.

Define a set $U = \{P_{ij} = (M_i, M_j, M_{ij}) : \forall i, j \in V(T), i \neq j\}$, where 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} is empty, then M_{ij} is empty.

Define the following operator $\bullet$ on U. For $i, j, k \in V(T)$, $P_{ij} \bullet P_{jk} = P_{ik}$, if the tree path from i to k includes the vertex j, and $P_{ij} \bullet P_{jk} = \emptyset$ otherwise. It follows easily from the definition that $\bullet$ is associative: If i, j, k, l are vertices (appearing in that order) on a simple path in T, then $(P_{ij} \bullet P_{jk}) \bullet P_{kl} = P_{ik} \bullet P_{kl} = P_{il}$ and $P_{ij} \bullet (P_{jk} \bullet P_{kl}) = P_{ij} \bullet P_{jl} = P_{il}$. If i, j, k, l are not on a simple path in T, then $(P_{ij} \bullet P_{jk}) \bullet P_{kl} = P_{ij} \bullet (P_{jk} \bullet P_{kl}) = \emptyset$. In general, the product $P_{i_1 i_2} \bullet P_{i_2 i_3} \cdots \bullet P_{i_{m-1} i_m} = P_{i_1 i_m}$ iff $i_1, \ldots, i_m$ is a path in T.

Suppose we have computed P_{ab} for every a and b such that (a, b) is an edge in T. Then the operator $\bullet$ can be implemented by combining networks as follows.

Suppose the path from i to k passes through vertex j and we wish to compute $P_{ik} = P_{ij} \bullet P_{jk}$, given P_{ij} and P_{jk}. Then, since T is binary, j has at most one neighbour x apart from its neighbours on the path from i to k. Let T_x be the component of T containing x, obtained by deleting edge (j, x). Let n_i and n_k be the neighbours of i and k on the path from i to k. (See Figure 1.)

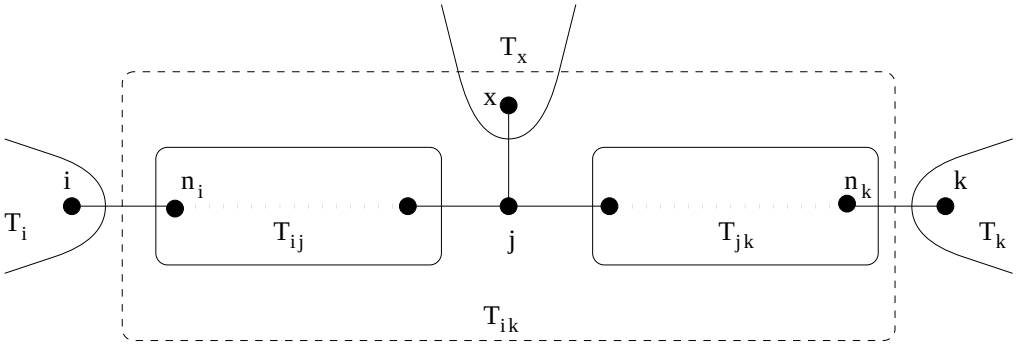

Fig. 1. Computation of P_{ij}.

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 vertex j, which are pairwise vertex-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} and T_{jk} are available in the values P_{ij} and P_{jk}. The mimicking network for the subgraph spanned by j can be computed using Proposition 2. ¿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 the terminals. Combining the above mimicking networks using Lemma 3 yields M_{ik}. Since the total number of terminals is constant, we have the following result.

Lemma 6. *Let G be a network and let (X, T) be its augmented binary tree decomposition of constant width. Given $P_{ab}, \forall (a, b) \in E(T)$, and P_{ij}, P_{jk} for some $i, j, k \in V(T)$, $P_{ij} \bullet P_{jk}$ can be computed in constant time.*

We now show how to compute P_{ij} for each edge (i, j) in T. Root T at any vertex. For a vertex i, let S_i be the subtree rooted at i. Consider an edge (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 $T \backslash S_i$, with terminals X_j. We compute P_{ij} in two phases. In the first phase we compute M_i for each edge (i, j) with i a child of j. In the second phase, we compute M_j for each such edge.

During the first phase, suppose we are at an edge (i, j), with i a child of j. Suppose also that we have computed the mimicking network M_l and M_r for the (at most) two edges connecting i to its children. Then, to obtain M_i, use Lemma 3 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 edge completes the first phase.

During the second phase, suppose we are at edge (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 $T \backslash S_j$. 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 3 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 $T \backslash S_i$. A preorder traversal of T with this operation performed at each edge completes the second phase.

Each time Lemma 3 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 edge, we have proved the following result.

Lemma 7. *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 edges $(a, b) \in E(T)$.*

We can now prove the main result of this section.

Lemma 8. *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 $i, j \in V(T)$. Further, after $O(n)$ preprocessing, we can find these mimicking networks in time $O(\alpha(n))$.*

Proof. For each edge (a, b) of T, compute P_{ab} using Lemma 7. Use Theorem 5 to preprocess T, with the P_{ab} values associated with its edges, 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 3, with the desired set of terminals being $X_i \cup X_j$. This yields the mimicking network for G with these terminals.

Theorem 9. *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 max-flow in time $O(k)$, for each $s, t \in V(G)$. Further, after $O(n)$ preprocessing, we can find the value of an s-t max-flow in time $O(\alpha(n))$.*

Proof. First, compute a constant-width augmented binary tree decomposition (X, T) of G using Proposition 4. Preprocess G and (X, T) using Lemma 8.

Let $s \in X_i$ and $t \in X_j$, for some $i, j \in V(T)$. 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 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.

In order to solve the APMC problem in a bounded tree-width network, simply apply Theorem 9 with $k = 2$, i.e. perform $O(n \log n)$ preprocessing so that an s-t max-flow can be computed in constant time. Thus the APMC problem can be solved by querying for s-t max-flows, for each pair s, t in the network. This proves the following result.

Corollary 10. *The all-pairs min-cut problem can be solved for bounded tree-width networks in time $O(n^2)$.*

4 Sparse networks

Frederickson [12] 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 4 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 [12] 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 $\gamma = o(n)$.

Let G be a sparse graph which is decomposed into hammocks $H_1, \ldots, H_\gamma$. Let A_i be the set of (at most 4) attachment vertices of H_i. We now show how to preprocess G so that max-flows can be efficiently found. Preprocess each hammock H_i as follows. First, find an augmented binary tree decomposition (X', T) of H_i, of constant width (outerplanar graphs have tree-width 2). Replace each set of $X'_j \in X'$ by $X_j = X'_j \cup A_i$, i.e. add the attachment vertices to each set. Let X be the collection of sets so obtained. Then (X, T) is also an augmented binary tree decomposition of H_i of constant width. We will work with this new tree decomposition. Use Lemma 7 to preprocess H_i in $O(|H_i|)$ time, so that for each edge $(a, b) \in T$, the mimicking network for H_i at terminals $X_a \cup X_b$ can be found using a single query.

Now, (i) the mimicking network for H_i at terminals A_i can be found in constant time, and (ii) for any $s, t \in V(H_i)$ the mimicking network for H_i at terminals $\{s, t\} \cup A_i$ can be found in time $O(\alpha(n))$. The first claim follows from the fact that the values P_{ab}, for each edge $(a, b) \in T$, are computed during preprocessing. $P_{ab} = (M_a, M_b, \emptyset)$, where M_a and M_b are the mimicking networks for the subgraphs of H_i spanned by the two components of T obtained by deleting

edge (a,b). Recall that $A_i \subseteq X_a$ and $A_i \subseteq X_b$. Combining M_a and M_b and retaining terminals A_i yields the desired mimicking networks. The second claim follows by selecting $c,d \in V(T)$ such that $s \in X_c, t \in X_d$, applying Lemma 8 and retaining the desired terminals.

We can now find the value of an s-t max-flow as follows. 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 and deleting H_i and H_j except their attachment vertices. The terminals of G_{ij} are $A_i \cup A_j$. Note that G_{ij} has $O(\gamma)$ vertices and edges and can be constructed in time $O(\gamma)$, since each mimicking network can be found in constant time. Construct G_{ij} and find the mimicking network for G_{ij} at terminals $A_i \cup A_j$ using Proposition 2. Find the mimicking network for H_i at terminals $\{s\} \cup A_i$ and H_j at terminals $\{t\} \cup A_j$ in time $O(\alpha(n))$, as described above. (If $i = j$, then find the mimicking network for H_i at terminals $\{s,t\} \cup A_i$.) Combining these networks yields the mimicking network for G at terminals $\{s,t\} \cup A_i \cup A_j$. Now the value of s-t max-flow can be found using the method described in the proof of Theorem 9. To estimate the time complexity, once the hammocks have been preprocessed and the mimicking networks for G_{ij} found, the remaining computation takes constant time. Preprocessing the hammocks takes $O(n)$ time and finding the mimicking network for G_{ij} takes $O(\gamma^2 \log \gamma)$ time, when we apply Proposition 2 with a max-flow algorithm (see e.g. [1]) for which $F(G) = O(nm \log n)$ on an n-vertex, m-edge network G. We summarize the above discussion:

Theorem 11. *The value of an s-t 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.*

To solve the APMC problem, preprocess the H_i's using $O(|H_i| \cdot \log |H_i|)$ time so that the mimicking network for H_i at the appropriate terminals (as in (ii) above) 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 s-t max-flow, as before. Once the H_i's have been preprocessed and the mimicking networks for the G_{ij}'s found, computing an s-t max flow takes constant time for each pair s,t. Hence, the following result is proved.

Theorem 12. *The all-pairs min-cut problem for an n-vertex sparse network G can be solved in $O(n^2 + \gamma^4 \log \gamma)$ time, where γ is the number of hammocks of G..*

5 Computing s-t min-cut in bounded tree-width networks

In this section we outline an extension of the methods in Sections 2.1 and 3 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 in Section 2.1. In this construction, 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 computed in Section 3 we will also compute the trace information associated with their edges. For edges of the input graph, 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 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 graph, 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 graph. 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 of the input graph that e represents. Further, the condition above ensures that every non-leaf vertex in the trace subtree has at least two children.

We now perform preprocessing as described in Lemma 8, computing trace information for each mimicking network constructed in the process. Suppose we now wish to compute an s-t min-cut for some s, t. Then, as in the proof of Theorem 9, we compute a mimicking network $M(G)$ of constant size, whose terminals include s and t, for the input graph 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 non-leaf vertex has at least two children. Doing this for each edge crossing the cut in $M(G)$ outputs in linear time all the edges crossing the cut in G. This yields the following result.

Theorem 13. *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. Further, after $O(n)$ preprocessing, we can output the edges crossing an s-t min-cut in time $O(\alpha(n) + L)$.*

6 Characterization of flows in multi-terminal networks

In [14] necessary and sufficient conditions are derived for an external flow to be realizable:

Lemma 14. *An external flow $(x_1, \ldots, x_{|Q|})$ is realizable in a network G with terminals Q iff (i) $\sum_{p \in Q} x_p = 0$ and (ii) $\sum_{r \in R} x_r \leq b_R$, $\forall R \subseteq Q$, 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 14, is called the *external flow inequalities* of G *at terminals* Q.

Given a network $G = (V, E)$ with terminals Q, it is possible to compute its external flow inequalities by following the method described in Section 2.1 for computing the capacities of minimum R-separating cuts in G, for every $R \subseteq Q$, in time $O(2^q F(G))$, where $q = |Q|$.

Using linear programming methods, it is shown in [14] how to combine the external flow inequalities of a number of networks. Here we give a simpler proof of this result; our proof is based on the max-flow min-cut theorem (see Corollary 1) and avoids linear programming. We note that the proof in [14] 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 twice the total number of terminals.

Let G be the union of two edge-disjoint networks G_1 and G_2. Observe that the attachment sets for both networks are the same. Assume that the attachment set is a subset of the terminals of both networks. Given the external flow inequalities of these two networks, we wish to find the external flow inequalities of G at a subset of all the terminals. We solve this problem in two steps. In the first step, we find the external flow inequalities at all the terminals, and in the second step we drop some terminals.

We now find the capacity of a minimum R-separating cut, where R is a subset of all the terminals. Let S and T be intersections of R with the terminals of G_1 and G_2, respectively. Further, let X and Y be the defining subsets of a minimum S-separating cut in G_1 and a minimum T-separating cut in G_2. Since the attachment set is a subset of the terminals of both networks, observe that the intersections of the attachment set with X and Y are equal. Hence $X \cup Y$ is a defining subset of an R-separating cut in G. Moreover, the capacity of this cut is the sum of the capacities of the cuts defined by X and Y, because G_1 and G_2 are edge-disjoint. On the other hand, by adding the corresponding max-flows in G_1 and G_2, we obtain a flow from R to the rest of the terminals in G and the value of this flow is equal to the sum of the values of the two former flows. We have thus proved (by Corollary 1) that $X \cup Y$ is a defining subset of a minimum R-separating cut and that the capacity of this cut is equal to the sum of the above two capacities. The latter two capacities are already available in the given

external flow inequalities. Since this computation is done for each subset of all the terminals, the time taken in this step is exponential in the total number of terminals.

In the second step, we wish to drop some terminals and find the capacity of a minimum P-separating cut, where P is a subset of the remaining terminals. A subset R of all the terminals is said to be consistent with P if it contains every terminal in P and doesn't contain any other remaining terminal. For any such R, observe that every R-separating cut is also a P-separating cut and hence the capacity of a minimum P-separating cut is at most the capacity of a minimum R-separating cut. On the other hand, the intersection of all the terminals with the defining subset of a minimum P-separating cut is consistent with P. We have thus proved that the capacity of a minimum P-separating cut is equal to the minimum capacity of all R-separating cuts, where R is consistent with P. In the first step, we have computed the latter capacities. Thus we can compute the former capacity in time exponential in the total number of terminals, by considering all possible consistent subsets. Since this computation is done for all subsets of the remaining terminals, the time taken in this step is at most an exponential in twice the total number of terminals.

The proof of the following lemma now comes by a simple induction on m.

Lemma 15. *Let $G = G_1 \cup \ldots \cup G_m$, 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 $Q' \subseteq \cup_{i=1}^{m} Q_i$ of terminals, we can compute the external flow inequalities for G at terminals Q' in time $O(2^{2(q_1+\cdots+q_m)})$, where $q_i = |Q_i|$.*

7 Remarks

We presented efficient algorithms for the all-pairs min-cut problem on bounded tree-width networks and sparse networks. The constants in the running time of the algorithms are not small, however. For example, in the algorithm for networks of tree-width t, the constant is $2^{2^{O(t)}}$. An open problem is to design more practical algorithms for these problems.

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", Tech. Rep. No. 71/87, Tel-Aviv University, 1987.
3. S. Arnborg, "Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability - A Survey", *BIT*, 25, pp.2-23, 1985.
4. H. Bodlaender, "NC-algorithms for Graphs with Small Treewidth", *Proc. 14th WG'88*, LNCS 344, Springer-Verlag, pp.1-10, 1989.
5. H. Bodlaender, "A Linear Time Algorithm for Finding Tree-decompositions of Small Treewidth", *Proc. 25th ACM STOC*, pp.226-234, 1993.

6. H. Bodlaender, "A Tourist Guide through Treewidth", *Acta Cybernetica*, Vol.11, No.1-2, pp.1-21, 1993.
7. S. Chaudhuri and C. Zaroliagis, "Shortest Path Queries in Digraphs of Small Treewidth", *Proc. 22nd Int. Col. on Automata, Languages and Prog. (ICALP'95)*, LNCS 944, Springer-Verlag, pp. 244–255.
8. B. Chazelle, "Computing on a Free Tree via Complexity-Preserving Mappings", *Algorithmica*, 2, pp.337-361, 1987.
9. J. Cheriyan, T. Hagerup and K. Mehlhorn, "Can a maximum flow be computed on $o(nm)$ time?", *Proc. 17th Intl. Col. on Automata, Languages and Prog. (ICALP'90)*, LNCS 443, Sringer-Verlag, pp.235-248, 1990.
10. H. Djidjev, G. Pantziou and C. Zaroliagis, "On-line and Dynamic Algorithms for Shortest Path Problems", *Proc. 12th Symp. on Theor. Aspects of Comp. Sc. (STACS'95)*, LNCS 900, Springer-Verlag, pp.193-204, 1995.
11. L.R. Ford and D.R. Fulkerson, "Maximal flow through a network", *Canadian Journal of Mathematics*, 8, pp.399-404, 1956.
12. G.N. Frederickson, "Using Cellular Graph Embeddings in Solving All Pairs Shortest Path Problems", *Proc. 30th Annual IEEE Symp. on FOCS*, 1989, pp.448-453.
13. G.N. Frederickson, "Searching among Intervals and Compact Routing Tables", *Proc. 20th Int. Col. on Automata, Languages and Prog. (ICALP'93)*, LNCS 700, Springer-Verlag, pp.28-39, 1993.
14. T. Hagerup, J. Katajainen, N. Nishimura and P. Ragde, "Characterizations of k-Terminal Flow Networks and Computing Network Flows in Partial k-Trees", *Proc. 6th ACM-SIAM Symp. on Discr. Alg. (SODA'95)*, pp.641-649, 1995.
15. R.E. Gomory and T.C. Hu, "Multi-terminal network flows", *Journal of SIAM*, 9, pp.551-570, 1961.
16. D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees", *Journal of Computer and System Sciences*, 26, pp.362-391, 1983.
17. K. Weihe, "Maximum (s,t)-Flows in Planar Networks in $O(|V|\log|V|)$ Time", *Proc. 35th Annual IEEE Symp. on FOCS*, 1994, pp.178-189.

This article was processed using the LaTeX macro package with LLNCS style