Single Source - All Sinks Max Flows in Planar Digraphs

Jakub Łącki*, Yahav Nussbaum†, Piotr Sankowski‡ and Christian Wulff-Nilsen§
*Institute of Informatics, University of Warsaw
Email: j.lacki@mimuw.edu.pl
†The Blavatnik School of Computer Science, Tel Aviv University
Email: yahav.nussbaum@cs.tau.ac.il
‡Institute of Informatics, University of Warsaw
and Department of Computer and System Science, Sapienza University of Rome
Email: sank@mimuw.edu.pl
§Department of Computer Science, University of Copenhagen
Email: koolooz@diku.dk

***Abstract*—Let $G = (V, E)$ be a planar n-vertex digraph. Consider the problem of computing max st-flow values in G from a fixed source s to all sinks $t \in V \setminus \{s\}$. We show how to solve this problem in near-linear $O(n \log^3 n)$ time. Previously, nothing better was known than running a single-source single-sink max flow algorithm $n-1$ times, giving a total time bound of $O(n^2 \log n)$ with the algorithm of Borradaile and Klein.**

An important implication is that all-pairs max st-flow values in G can be computed in near-quadratic time. This is close to optimal as the output size is $\Theta(n^2)$. We give a quadratic lower bound on the number of distinct max flow values and an $\Omega(n^3)$ lower bound for the total size of all min cut-sets. This distinguishes the problem from the undirected case where the number of distinct max flow values is $O(n)$.

Previous to our result, no algorithm which could solve the all-pairs max flow values problem faster than the time of $\Theta(n^2)$ max-flow computations for every planar digraph was known.

This result is accompanied with a data structure that reports min cut-sets. For fixed s and all t, after $O(n^{1.5} \log^2 n)$ preprocessing time, it can report the set of arcs C crossing a min st-cut in $O(|C|)$ time.

***Keywords*-minimum cut; maximum flow; planar graph; all pairs**

I. Introduction

Computing max flow in a graph is a classical algorithmic problem dating back to Ford and Fulkerson [12], [13]. Given a graph $G = (V, E)$ with arc capacities, a source $s \in V$ and a sink $t \in V \setminus \{s\}$, the problem is to send as much flow as possible from s to t without violating capacity constraints and flow conservation at vertices in $V \setminus \{s, t\}$. Many polynomial-time algorithms for this problem are known. In this paper we focus on planar graphs. There exists an $O(n \log n)$ time algorithm for planar digraphs [3], [9] and an $O(n \log \log n)$ time algorithm for planar undirected graphs [18]. More recently, an $O(n \log^3 n)$ time algorithm for a more general problem with a set of sources and a set of sinks was proposed [4].

Work by J.Ł. and P.S. work was partially supported by the ERC StG project PAAl no. 259515. Work by Y.N. was partly suppoted by Israel Science Foundation grant no. 822-10 and by Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11).

It is natural to consider the problem of finding multiple max flows and min cuts, e.g., to study the connectivity of the graph. For undirected (not necessarily planar) graphs, min st-cuts for all source/sink pairs (s, t) form a laminar family and can be compactly represented by a tree structure known as a *Gomory-Hu tree* [14]. Finding a Gomory-Hu tree reduces to solving $n-1$ min st-cut problems [15]. For planar undirected graphs, this gives an $O(n^2 \log \log n)$ time bound using the algorithm in [18]. This time bound was significantly improved to $O(n \log^5 n)$ in [6]. In particular, this means that all max st-flow/min st-cut values can be computed in near-linear time.

We consider the same problem but for planar *digraphs*. In general, this problem is more challenging since the Gomory-Hu property no longer holds [2]. In fact, there might be $\Omega(n^2)$ distinct max flow values [19]. We show that this lower bound holds for planar digraphs as well. Figure 1 shows a plane n-vertex digraph with $\Theta(n^2)$ different cycles. There is a well-known duality between shortest cycles in the primal G and min st-cuts in the dual G^* of a plane graph: let f and g be distinct faces in G and let f^* and g^* be the corresponding dual vertices in G^*, respectively. Let C be a shortest clockwise cycle in G such that f is outside of C and g is inside. Then there is a min f^*g^*-cut in G^* such that the arcs crossing this cut are exactly the (dual) arcs of C. It is now easy to see that the dual of the graph in Figure 1 has $\Theta(n^2)$ distinct min cuts. Moreover, the total size of the min cut-sets in this example is $\Theta(n^3)$.

At first sight it might thus seem impossible to compute the capacities of all min cuts in $o(n^3)$ time. Indeed, repeated applications of the single-source single-sink max flow algorithm of Borradaile and Klein [3] leads to a time bound of $O(n^3 \log n)$. Arikati et al. [1] computed all of these capacities in $O(n^2 + \gamma^3 \log \gamma)$ time where γ is the minimum number of *hammocks*, a special kind of outerplanar subgraphs, into which G can be partitioned.

However, γ might be $\Theta(n)$ and the running time of this algorithm is also $O(n^3 \log n)$ in this case. Nevertheless, we are able to significantly improve this to $O(n^2 \log^3 n)$ which is optimal up to logarithmic factors. In fact, we show something stronger: for fixed source s, max st-flow values for all $n-1$ sinks $t \neq s$ can be found in a total of $O(n \log^3 n)$ time.

Our algorithm can be changed to a data structure that can report the arcs crossing the min cuts, i.e., min cut-sets. Given a fixed source s it requires $O(n^{1.5} \log^2 n)$ preprocessing time. Afterwards, it can report a min cut-set C for s and any sink $t \neq s$ in $O(|C|)$ time. Hence, the total time to compute all min cut-sets $\mathcal{C}$ in the planar digraph is $O(n^{2.5} \log^2 n + \sum_{C \in \mathcal{C}} |C|)$.

Our result shows that in planar digraphs all $O(n^2)$ max flow values can be computed in time essentially equal to the time for finding a linear number of max flows. This joins the result of Arikati et al. [1] that showed an $O(n^2)$ algorithm for the same problem for *bounded treewidth* digraphs. For general n-vertex digraphs, Hao and Orlin [16] showed how to compute $O(n)$ max flows in the time it takes for a single max flow computation and used this result to find a (global) min cut. Based on this and our result for planar digraphs, we conjecture that also in general digraphs all $O(n^2)$ max st-flow values can be computed faster than the time required for computing $O(n^2)$ max st-flows separately. The result of Hao and Orlin does not resolve this conjecture as the $O(n)$ source/sink pairs considered by their algorithm are in a sense arbitrary; in particular, their result cannot be used to efficiently find max st-flow values for fixed s and all $t \neq s$.

When all arc capacities are equal to a single unit, we get the *arc connectivity* problem. For this special case of our problem, Cheung et al. [8] showed an $O(m^\omega)$ all-pairs arc connectivity algorithm, for an m-arc digraph, where ω is the matrix multiplication exponent. This is faster than running the $O(\min\{m^{1/2}, n^{2/3}\} \cdot m)$ algorithm for st-arc connectivity of Even and Tarjan [10] for every pair of vertices when $m = O(n^{1.94})$, using the currently best known value of $\omega < 2.3727$ [24]. Cheung et al. also showed an $O(d^{\omega-2} n^{\omega/2+1})$ algorithm for the same problem for *well-separable* digraphs, which also include planar graphs, with maximum degree d.

Cabello et al. [7] gave a simple algorithm for computing the *shortest non-contractible cycle* in a directed graph embedded in a sphere with b boundaries, which uses b minimum cut computations with a fixed source in the dual planar graph. Our result improves the time bound of this algorithm from $O(bn \log n)$ to $O(n \log^3 n)$ when $b = \omega(\log^2 n)$.

The organization of the paper is as follows. In Section II, we make some definitions and simplifying assumptions. In Section III, the overview of the algorithm for our problem is presented. Our algorithm is guided by a recursive decomposition of the input graph. It recursively computes max preflows to the outer boundaries of all subgraphs in the decomposition using previously computed max preflows to outer boundaries of the parent subgraphs. When the algorithm reaches a leaf of the recursive decomposition, a max preflow to a sink t contained in that leaf is found. The value of a max st-flow can then be identified as the amount of preflow into t. A straightforward implementation of this algorithm will not lead to a near-linear time bound. In Section IV, we show how to efficiently implement the various steps. An important tool here is a modification of a flow fixing procedure from [4]. In Section V, we show how the min st cut-sets themselves can be reported. Finally, in Section VI we give some conclusions and suggest future directions of research.

II. Preliminaries

Let $G = (V, E)$ be a simple planar digraph, where V is the vertex set and $E \subseteq V \times V$ is the set of arcs. Let $n = |V|$. Since G is planar, we have $|E| = O(n)$. We assume that for every arc $e = (u, v) \in E$ the *reversed arc* $rev(e) = (v, u)$ is also in E. This assumption can be easily satisfied for planar graphs, by using the same embedding for e and $rev(e)$. By *edge* we mean a pair of two opposite arcs, e and $rev(e)$, which share their embedding in the plane. We use the standard definition of the dual of a plane graph, such that the dual arc of a primal arc e is oriented from the right side of e to its left side. If the primal is a flow network with capacities on arcs, the dual arcs have weights equal to these capacities.

A. Flows

For graph G the capacities of the arcs are given by a *capacity function* $c : E \to \mathbb{R}^+$.

Let $s \in V$ be a source and let $t \in V \setminus \{s\}$ be a sink. We define a *flow assignment* to be a function $f : E \to \mathbb{R}$ satisfying *antisymmetry*, i.e., for all $e \in E$ we require $f(e) = -f(rev(e))$. A flow assignment is called an *st-pseudoflow* if for all arcs $e \in E$ we have $f(e) \leq c(e)$. The inflow of an st-pseudoflow f for a vertex $v \in V$ is defined as:

$$\text{inflow}_f(v) = \sum_{(u,v) \in E} f(u, v),$$

We say that st-pseudoflow f has *excess* in v when $\text{inflow}_f(v) > 0$ and in this case we refer to $\text{inflow}_f(v)$ as its *excess (value)*. Similarly, we say that f has *deficit* in v when $\text{inflow}_f(v) < 0$ and $\text{inflow}(v)$ is its *deficit (value)*. An *st-preflow* is an st-pseudoflow such that no vertex in $V \setminus \{s\}$ has deficit. An *st-flow* is an st-preflow such that no vertex in $V \setminus \{t\}$ has excess. When no confusion arises, we shall omit s and t and simply talk about pseudoflows, preflows, and flows. A *circulation* is a pseudoflow such that for every vertex $v \in V$, $\text{inflow}_f(v) = 0$. A circulation does not have a source or a sink.

The *value* of a preflow f is given as $\text{inflow}_f(t)$. Finally, a *max pseudoflow (preflow)* is a pseudoflow (preflow) such

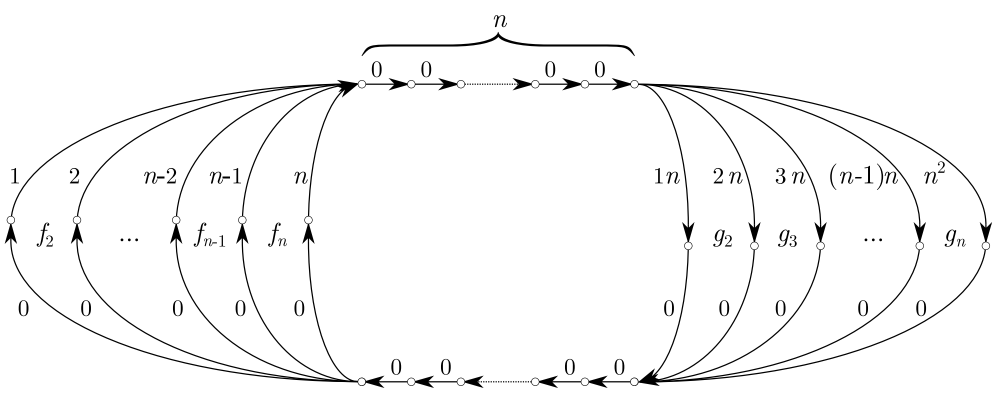

Figure 1: Example of a planar digraph on $4n$ vertices for which min $f_i^* g_j^*$-cuts in the dual are pairwise distinct, $2 \leq i, j \leq n$. There are thus $\Theta(n^2)$ min cuts and each has $\Theta(n)$ arcs for a total of $\Theta(n^3)$.

that there is no residual path that starts at a source or at a vertex with excess and ends at a sink or at a vertex with deficit. Observe that the value of a max preflow equals the value of a max flow. We can *limit* the value of a max flow by some value d, by adding a new vertex s' and an arc (s', s) of capacity d and regarding s' as a source instead of s.

B. Pieces and Recursive Decomposition

Let G be a plane graph. We assume that G is triangulated and has bounded degree. Both can be satisfied simultaneously using a standard vertex-splitting argument to get degree three and then triangulating each face such that each vertex degree in that face is increased by at most 2.

A *piece* is a subgraph of G with the same embedding as G. A vertex of P is called a *boundary vertex* of P if it is incident in G to a vertex not belonging to P. All other vertices of P are called *interior vertices* of P. We let ∂P denote the set of boundary vertices of P. A *hole* of P is a face of P which is not a face of G. The vertices of P incident to a hole H separate the whole graph G into two parts. The *interior* of H is the part that does not contain P. In certain places, we shall regard a hole as the subgraph of G contained in it. It should be clear from context what is meant.

A *decomposition* of a piece P is a set of sub-pieces $P_1, \dots, P_k$ such that the union of the vertex sets of these sub-pieces is the vertex set of P and such that every edge of P is contained in a unique sub-piece. We define every boundary vertex of P to be a boundary vertex of every sub-piece P_i that contains it. We change the standard definition of a decomposition a little and allow two sub-pieces to share edges. We include the set of edges that connect the boundary vertices of a sub-piece in this sub-piece, even if these edges belong also to another sub-piece. This does not increase the size complexity of the decomposition, and will simplify the discussion of the dual graph of the sub-piece.

A *recursive decomposition* of G is obtained by first identifying a decomposition of G and then recursing on each sub-piece. The recursion stops when pieces of constant size are obtained. The recursive decomposition of G is the collection of pieces constructed over all levels of the recursion. We are interested in a special type of recursive decomposition satisfying the following: a piece P with r vertices and b boundary vertices is divided into a constant number of connected sub-pieces each containing at most $\frac{1}{2}r$ vertices, at most $\frac{1}{2}b$ boundary vertices inherited from P, and at most $O(\sqrt{r})$ additional boundary vertices. In addition, we require that each piece has only a constant number of holes. When we refer to a recursive decomposition of G in the following, we shall assume that it is of this special type. Parent/child and ancestor/descendant relationships between pieces correspond to their relationships in the decomposition tree.

Obtaining a recursive decomposition that satisfies all of the above conditions is non-trivial. The assumptions that subpieces contain at most $\frac{1}{2}r$ vertices, at most $\frac{1}{2}b$ boundary vertices inherited from P, and at most $O(\sqrt{r})$ additional boundary vertices can be satisfied by finding an r-division of P; see [18] for details. A construction of connected pieces, as we require, is given in [5]. In order to ensure that pieces are connected, we allow $O(n)$ new edges to be added while maintaining planarity and the chosen embedding of G. The arcs corresponding to these edges have zero capacity and thus do not affect max flows or min cuts.

Lemma 1: A recursive decomposition as described above can be computed in $O(n \log n)$ time.

In order to bound the running time of our algorithms we need the following lemma. The proof is based on the fact that the total sizes of the pieces and sum of the squares of the number of boundary vertices in each piece, at each level of the decomposition are both $O(n)$. The complete details are omitted due to space constraints.

Lemma 2: Let $\mathcal{P}$ be the set of pieces in a recursive decomposition of G. Then $\sum_{P \in \mathcal{P}} |P| = O(n \log n)$ and $\sum_{P \in \mathcal{P}} |\partial P|^2 = O(n \log n)$.

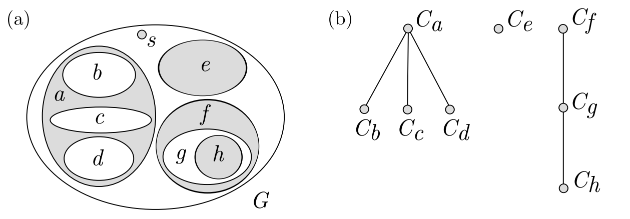

Figure 2: Figure (a) shows a recursive decomposition that satisfies out all assumptions. In Figure (b), the corresponding forest $\mathcal{F}$ is depicted.

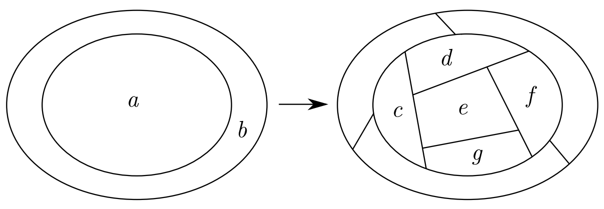

Figure 3: A single step of building the recursive decomposition. The piece is divided by finding an r-division. The outer piece of the outer boundary of a is b. The outer piece of the outer boundary of c is the part of a outside of c, which contains d, e, f, g, and the outer boundary of a.

C. Nesting of Outer Boundaries

Let $\mathcal{P}$ be the set of pieces in the recursive decomposition. We may assume that the fixed source s is not a boundary vertex of any piece of $\mathcal{P}$. This guarantees that the following definitions are unambiguous. If a hole H of P contains s in its interior, we refer to H as the *outside* of P. The subgraph of G contained in the outside of P is denoted $\text{ext}(P)$. The *outer boundary* of P is the set of boundary vertices of P contained in the outside of P. Note that for a connected piece, there is a simple cycle of the piece containing its outer boundary.

For simplicity, we also assume that each sink belongs to a piece having an outside. This is true for every vertex that belongs to a piece that s does not belong to, that is for all vertices of the graph except for a constant number of vertices which belong to the same leaf piece of the recursive decomposition as s.

These assumptions guarantee that the recursive decomposition has a rather simple structure as illustrated in Figure 2(a).

Observe that the outer boundaries nest and form a laminar family, i.e., a forest. Let the nesting of outer boundaries be represented by a forest $\mathcal{F}$. The outer boundary C is an ancestor in $\mathcal{F}$ of an outer boundary C' if the s-side of C' contains the s-side of C. For an example, see Figure 2(b).

For any $C \in \mathcal{F}$ which is not a root in $\mathcal{F}$, we define the *outer piece* of C as follows:

1) If C bounds a hole H of a piece $P \in \mathcal{P}$ and H is not the outside of P then the deepest such P in the recursive decomposition is the outer piece of C (see piece a in Figure 3).
2) Otherwise, let C' be the parent of C in $\mathcal{F}$ and let P' be the deepest piece in the recursive decomposition such that the outside of P' is bounded by C'; then there is a child piece P of P' whose outside is bounded by C (otherwise, we would be in the first case); we define the outer piece of C to be P' excluding the subgraph of P' not contained in the outside of P (see piece c in Figure 3). Note that in this case the outer piece of C is not a piece of the recursive decomposition.

To simplify our description, we shall assume in the following that all cycles $C \in \mathcal{F}$ have outer pieces of the first type. In particular, we assume that all outer pieces belong to the recursive decomposition. Dealing with outer pieces of the second type will not increase our time bound since by the following result, Lemma 2 will still hold if we include these additional outer pieces in $\mathcal{P}$:

Lemma 3: $|\mathcal{F}| = O(1)$ and each tree in $\mathcal{F}$ has constant degree.

Proof: By assumption, s is not a boundary vertex of any piece. For any outer boundary C corresponding to a root in $\mathcal{F}$, there is no other outer boundary in $\mathcal{F}$ separating C and s. Hence, every such C must lie inside a hole in the same piece P. Since P has only $O(1)$ holes, the first part of the lemma follows.

For a node of $\mathcal{F}$ to be a child of another node in $\mathcal{F}$, the outer boundaries corresponding to these nodes must share vertices with holes of the same piece. The second part of the lemma follows, again since a piece only has $O(1)$ holes and each piece is divided into a constant number of connected subpieces. ■

D. Dense Distance Dual Graphs

For a piece P, Fakcharoenphol and Rao [11] define the *dense distance graph* of P as the complete directed graph on the set of boundary vertices of P, representing shortest path distances between them in P. We need to apply this construction to the dual graph. Moreover, this definition needs to be tuned to the recursive decomposition of the graph. Let us denote the set of faces of G that lie inside a hole H as H^*. We define the set of *boundary faces* of the hole H to be a set of faces of G that are in H^* and are incident to the boundary vertices of H. The *internal dense distance dual graph* $\text{IDDG}^*(P)$ of P is the complete directed graph on the set of faces of P incident to its holes. An arc (f_1, f_2) in $\text{IDDG}^*(P)$ has weight equal to the (possibly infinite) shortest path distance from f_1 to f_2 in $G^*[P^*]$; here $G^*[P^*]$ denotes the subgraph of G^* induced by the dual vertices in P^*. Observe that due to our assumption that vertices have constant degree the number of faces incident to the holes of P is $O(|\partial P|)$.

The *dense distance dual graph* of H, denoted $\text{DDG}^*(H)$, is the complete directed graph on the set of boundary faces of H. Each arc (f_1, f_2) of $\mathrm{DDG}^*(H)$ has weight equal to the shortest path distance in $G^*[H^*]$ from face f_1 to face f_2. The union of $\mathrm{DDG}^*(H)$ over all holes of P is denoted by $\mathrm{DDG}^*(P)$. Lemma 2 and our assumption that vertices have constant degree imply that the total size of $\mathrm{DDG}^*(P)$ over all pieces P in a recursive decomposition of G is $O(n \log n)$.

Fakcharoenphol and Rao showed how to compute a dense distance graph of a piece P containing r vertices in $O(r \log^3 r)$ time [11]. This time bound can be improved to $O(r \log r)$ using the algorithm of Klein [23]. However, we do not use his algorithm here since it cannot be executed on dense distance graphs. Here, the advantage of Fakcharoenphol and Rao's result is the faster implementation of Dijkstra which we refer to as *FR-Dijkstra*. This algorithm can find a shortest path tree in a graph composed of dense distance graphs and arcs from the (original) planar graph in $O(b \log^2 b)$ time, where b is the sum of the total number of boundary vertices, counted with multiplicity, and the total number of arcs. This result allows us to prove the following lemma.

Lemma 4: The graphs $\mathrm{DDG}^*(P)$ for all pieces $P \in \mathcal{P}$ can be computed in $O(n \log^3 n)$ time.

Proof: Our algorithm proceeds bottom-up on each tree $T \in \mathcal{F}$. For each $C \in T$, it considers the hole H inside C and computes $\mathrm{DDG}^*(H)$ as follows. Let P_C be a piece with outer boundary C. We compute the dense distance dual graph of P_C in $O(|P_C| \log^2 |P_C|)$ time. Next, we build a graph D^* composed of $\mathrm{IDDG}^*(P_C)$ and dense dual distance graphs $\mathrm{DDG}^*(H')$ for all holes $H' \neq \mathrm{ext}(P_C)$ of P_C. In order to compute $\mathrm{DDG}^*(H)$ we need to run FR-Dijkstra $O(|C|) = O(|\partial P_C|)$ times on D^*. Each run takes $O(|\partial P_C| \log^2 |\partial P_C|)$ time, so in total we need $O(|\partial P_C|^2 \log^2 |\partial P_C|)$ time. Applying Lemma 2 for the whole recursion we get $O(n \log^3 n)$ total time. Finally, to obtain $\mathrm{DDG}^*(P)$ for all P we simply sum up $\mathrm{DDG}^*(H')$ for all H' in P. ■

The original implementation of FR-Dijkstra by Fakcharoenphol and Rao [11] does not support *reduced lengths* defined by a *potential function* (see Section IV-A). We use an extension of this implementation by Kaplan et al. [22] which allows reduced lengths.

III. Single Source - All Sinks Max Flow

For a fixed source s, we wish to find the value of a max st-flow in G for every sink $t \in V \setminus \{s\}$. In this section we describe an algorithm which is based on computing the value of a max flow from s to every outer boundary in $\mathcal{F}$ in a recursive fashion. From this we can easily find the values we are looking for. In Section IV we describe an efficient implementation of the algorithm.

A. Max Preflow in a Separated Graph

First, we consider a more specific problem, which enables us to implement the recursive step of our algorithm. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be non-empty subgraphs of G, where $V_1 \cup V_2 = V$ and $E_1 \cap E_2$ is a simple cycle. Let C be the *separator* $V_1 \cap V_2$ in G. Assume that $s \in V_1 \setminus C$ and $t \in V_2 \setminus C$. This assumption can be made without loss of generality. Algorithm MaxPreflowInSeparatedGraph finds a max st-preflow in G by considering G_1 and G_2 separately:

Algorithm MaxPreflowInSeparatedGraph:

1) Find a max sC-preflow in G_1.
2) Find a max Ct-flow in G_2. Denote the value of this flow by d_2.
3) Add the two flow assignments of G_1 and G_2 to form a pseudoflow in G.
4) Send flow among vertices of C until there is no residual path from a vertex $u \in C$ with excess to a vertex $v \in C$ with deficit; update the residual network accordingly.
5) If there are vertices in C with remaining deficit, let $-\ell$ be the sum of deficits over all vertices of C with a deficit; reset the flow on G, and rerun the algorithm a limit of $d_2 - \ell$ on the Ct-flow pushed in step 2.

The above algorithm is based on the *flow partition* scheme of Johnson and Venkatesan [21], which was also used by [4]. Our implementation is a little different from the original scheme in its last step. First, we do not return excess flow from C to s; we are not required to do so, since we are looking for a preflow. Second, we do not return deficit flow from C to t. Instead we recompute a Ct-flow with value limited to $d - \ell$. This computation is equivalent, but as we show below, we can implement it faster.

Notice that since edges of $E_1 \cap E_2$ are incident only to vertices of C, we may assume that the max sC-preflow in G_1 and the max Ct-flow in G_2, from the two first steps of the algorithm, do not assign flow to any of these edges. Therefore at step 3 we indeed obtain a pseudoflow that does not violate the arc capacities.

We shall refer to an algorithm implementing step 4 as a *flow fixing procedure* (along C in the residual network).

B. Max Preflow in Pieces

The main algorithm of this section, which we call Algorithm MaxPreflowToBoundaries, uses Algorithm MaxPreflowInSeparatedGraph in recursion guided by the nesting of pieces. The problem that Algorithm MaxPreflowToBoundaries solves is to identify, for each piece P, a max preflow from s to the outer boundary of P. This max preflow is entirely outside of P.

Instead of considering every piece, it is enough to consider only outer boundaries, since if two pieces share the same outer boundary, we do not need to compute a max preflow from s to this outer boundary twice. Our algorithm recurses on each tree $T \in \mathcal{F}$. Let C be the outer boundary corresponding to the root node of T. For the base of the recursion, we compute a max preflow from s to C outside of C. This is done by embedding a super-sink t' inside C, connecting every vertex of C to t' by infinite capacity arcs, applying a max st'-flow algorithm for planar graphs, and removing t'.

In the general recursive step, consider an outer boundary C', let C be its parent in T, and assume we have computed a max preflow from s to C outside of C. We add a super-sink t' inside C' and infinite capacity arcs as before, and our problem becomes finding a max preflow from s to t'. (Note that C and C' might share some vertices, in this case we do not connect the shared vertices to t', to avoid flow of infinite value.) We do so using Algorithm MaxPreflowInSeparatedGraph. We can regard C as a separator in the graph, where G_1 is the s-side (outside) of C and G_2 is the t-side (where C' is). We already know the max sC-preflow required by step 1 of Algorithm MaxPreflowInSeparatedGraph due to the recursive structure of our algorithm. It remains to execute the other steps of the algorithm. In the rest of this section and in the following section we focus on the implementation of these steps.

For each outer boundary C corresponding to a leaf node of T, Algorithm MaxPreflowToBoundaries computes a max preflow in the outside of C from s to C. Note that the inside G_C of C has constant size. To find a max preflow from s to each sink t in G_C, we can treat t as a degenerate outer boundary C' and extend the above algorithm with an additional recursion level. From this, we can obtain the value of a max st-preflow for each sink $t \neq s$ in G, as required.

C. Pushing Flow Through Holes

Let C, C', and $G' = G_1 + G_2$ be as defined in the description of algorithm MaxPreflowToBoundaries. Let P be the outer piece of C'. Note that C is the outer boundary of P. In step 2 of Algorithm MaxPreflowInSeparatedGraph we push flow from C to C' in G_2. This procedure requires at least time proportional to the size of G_2. Because of the possibility of holes in P, G_2 might be much bigger than P. We cannot afford to spend that much time in this step. Rather, we would like to bound the amount of work here by the size of P, so that we can use Lemma 2 to bound the total time of our algorithm.

Let $H_1, \ldots, H_k$ denote the holes of P bounded by neither C nor C' and let $C_1, \ldots, C_k$ be the boundaries of these holes. We observe that $G_2 = P + H_1 + \cdots + H_k$. In this subsection, we show how to decompose the computation of the max flow found in step 2 of Algorithm MaxPreflowInSeparatedGraph into smaller steps according to this partition of G_2. In Section IV, we will give an efficient implementation of each of these steps.

The single hole case: We first consider the case where $k = 1$. Let $H = H_1$ and let $C_H = C_1$ be the boundary of H. We give here an algorithm PushFlowThroughHoles(P, C, C', H) which finds a max flow inside $P + H$ from C to C'. The algorithm is as follows (each step uses the residual network for the flow found in earlier steps):

Algorithm PushFlowThroughHoles(P, C, C', H):

1) Compute a max flow in P from C to C'.
2) Compute a max flow in P from C to C_H. Denote the value of this flow by d_2.
3) Compute a max flow in P from C_H to C'. Denote the value of this flow by d_3.
4) Run a flow fixing procedure along C_H in $P + H$.
5) Let ℓ_2 be the sum of excesses over vertices of C_H with an excess, and let $-\ell_3$ be the sum of deficits over all vertices of C_H with a deficit; reset the flow to zero in $P + H$, and rerun the previous steps with a limits of $d_2 - \ell_2$ and $d_3 - \ell_3$ on the flow pushed in steps 2 and 3, respectively.

Lemma 5: Algorithm PushFlowThroughHoles(P, C, C', H) finds a max flow from C to C' in $P + H$.

Proof: We will show that after (the first execution of) step 4, we have a pseudoflow with no residual paths in $P + H$ from C to C', from C to deficit vertices on C_H, from excess vertices on C_H to C', or from excess vertices on C_H to deficit vertices on C_H. This suffices to show that at termination we have a max flow from C to C' in $P + H$, since step 5 in fact only returns excess to s and deficit to t, as in the flow partition scheme of [21].

Let us identify a cut with the set of arcs crossing it. After step 1, there is a saturated cut K_1 in P separating C from C'. Thus, any residual paths from C to C' in $P + H$ must cross H. After step 2, such residual paths cannot exist. Note that K_1 stays saturated after Step 2. We now have a new saturated cut K_2 in P with C on one side and C' and H on the other side. After Step 3, K_2 stays saturated (since C' and H are on the same side of K_2) and we get another saturated cut K_3 in P separating H and C'.

Saturated cuts K_2 and K_3 partition $P + H$ into three subsets, X, Y, and Z. Set X contains C, Y contains H, and Z contains C'. Consider an augmenting path implementation of the flow fixing procedure in step 4. Each augmenting path has both its endpoints on H so it crosses neither K_2 nor K_3 (as it would have to cross K_2 or K_3 in both directions). Hence, K_2 and K_3 stay saturated after step 4 so there is no residual path in $P + H$ from C to C', from C to H, or from H to C'. The flow fixing procedure ensures that there are no residual paths in $P + H$ from excess vertices on C_H to deficit vertices on C_H. This shows the desired. ■

Generalizing PushFlowThroughHoles *to k holes:* Now let us generalize Algorithm PushFlowThroughHoles(P, C, C, C', H) to arbitrary k. Instead of a single hole H, it gets a set $\{H_1, \ldots, H_k\}$ of holes as its fourth parameter. For $k \geq 2$, we can regard $P + H_1 + \ldots + H_{k-1}$ as a piece P' with one hole H_k in addition to the two bounded by C and C'. Hence the call PushFlowThroughHoles$(P + H_1 + \ldots + H_{k-1}, C, C', H_k)$ solves the problem. Plugging in these parameters in the procedure for the single hole case, we get the following recursive algorithm:

Algorithm PUSHFLOWTHROUGHHOLES($P + H_1 + \ldots + H_{k-1}, C, C', H_k$):

1) Execute PUSHFLOWTHROUGHHOLES(P, C, C', $\{H_1, \ldots, H_{k-1}\}$) (finds max flow in P' from C to C').
2) Execute PUSHFLOWTHROUGHHOLES(P, C, C_k, $\{H_1, \ldots, H_{k-1}\}$). Denote the value of this flow by d_2 (finds max flow in P' from C to C_k).
3) Execute PUSHFLOWTHROUGHHOLES(P, C_k, C', $\{H_1, \ldots, H_{k-1}\}$). Denote the value of this flow by d_3 (finds max flow in P' from C_k to C').
4) Run a flow fixing procedure along C_k in $P + H_1 + \ldots + H_k$ (i.e., in $P' + H_k$).
5) Let ℓ_2 be the sum of excesses over vertices of C_k with an excess, and let $-\ell_3$ be the sum of deficits over all vertices of C_k with a deficit; reset the flow to zero in $P' = P + H_1 + \ldots + H_{k-1}$, and rerun the previous steps with a limits of $d_2 - \ell_2$ and $d_3 - \ell_3$ on the flow pushed in steps 2 and 3, respectively.

Lemma 6: A call to PUSHFLOWTHROUGHHOLES(P, C, C', $\{H_1, \ldots, H_k\}$) computes a max flow from C to C' in $P + H_1 + \ldots + H_k$. The total number max flow computations in P between boundaries of holes and the total number of calls to a flow fixing procedure is $O(1)$.

Proof: Correctness follows from Lemma 5 and a trivial induction on the recursive procedure. The number of max flow computations in P and the number of calls to the flow fixing procedure are both exponential in k. Since $k = O(1)$, the second part of the lemma follows. ■

IV. AN EFFICIENT IMPLEMENTATION

In this section, we will give an efficient implementation of Algorithm MAXPREFLOWTOBOUNDARIES. Recall that it applies Algorithm MAXPREFLOWINSEPARATEDGRAPH to obtain a max preflow from s to an outer boundary C' outside C', given a preflow from s to C outside C, where C is the parent of C' in $\mathcal{F}$. We will show that by maintaining flows implicitly, such a call to Algorithm MAXPREFLOWINSEPARATEDGRAPH can be implemented to run in $O((|P| + |\partial P|^2) \log^2 |P|)$ time, where P is the outer piece of C'. Combined with Lemma 2, this will give us the required $O(n \log^3 n)$ time bound for Algorithm MAXPREFLOWTOBOUNDARIES. An important tool that we use is the fast implementation of the flow fixing procedure by Borradaile et al. [4] which we go through briefly in the following.

A. The Flow Fixing Procedure of Borradaile et al.

Recall that a flow fixing procedure gets as input a cycle separator C for a graph $G_1 + G_2$ and a pseudoflow for $G_1 + G_2$ where all excess and deficit vertices are on C. The procedure updates the flow network by sending as much flow as possible from excess to deficit vertices. The flow fixing procedure in [4] takes advantage of the fact that all such vertices are on a single simple cycle in the plane, namely C. It processes the vertices of C one by one in cyclic order. At each step, if the current vertex v_i has excess, as much flow as possible is sent from v_i to the unprocessed vertices. This is implemented by a call to Hassin's algorithm [17]. This algorithm computes a single-source single-sink max flow in a plane digraph where the source and sink are on the same face. It does this by adding an infinite capacity arc e_∞ from the sink to the source and then computes shortest path distances in the dual from the face to the right of e_∞. These distances define a potential function and it can be shown that this function induces a max flow in the primal graph.

In the flow fixing procedure, we use Hassin's algorithm to compute a max flow from v_i to its successor v_{i+1} on C in a slightly modified version of $G_1 + G_2$, where arcs of C connecting unprocessed vertices have their capacities increased to infinity, thereby essentially identifying all these vertices with v_{i+1}. Conversely, if v_i has deficit, a similar call to Hassin's algorithm sends as much flow as possible from the unprocessed vertices to v_i. The total number of calls to Hassin's algorithm is thus $O(|C|)$. At termination, it can be shown that there is no residual path from any excess vertex to any deficit vertex.

During the course of the algorithm, the current flow is represented as a sum of a flow assignment on arcs of C and a circulation in $G_1 + G_2$ where the circulation is defined by a potential function ϕ on the dual vertices. Every computation uses a residual network for the flow found in all previous steps, so the computed face potential is accumulated in ϕ after every step. Hassin's algorithm finds a max flow by computing a shortest path tree in the dual. To describe this part, let us define X^* to be the set of faces incident to vertices of C, these are the dual vertices from which we start the shortest path computations. Since the vertices of $G_1 + G_2$ have constant degree, $|X^*| = O(|C|)$. Moreover, let G^*_{-C} be the graph obtained from $(G_1 + G_2)^*$ by removing the dual arcs of C and their reverses. Note that this splits G^*_{-C} into two disconnected parts: the inside and outside of C.

Following Hassin's algorithm, we compute shortest path distances from each vertex of X^* in $(G_1 + G_2)^*$ with respect to a weight function on the arcs induced by the arc lengths $l(u, v)$ of $(G_1 + G_2)^*$ and the current potential function ϕ. More precisely, the *reduced length l_ϕ with respect to ϕ* is used, where $l_\phi(u, v) = l(u, v) + \phi(u) - \phi(v)$ [20]. A crucial observation to make in order to get an efficient implementation is that the only reduced distances needed in G^*_{-C} are those starting and ending in vertices of X^* and these can be obtained from the the X^*-to-X^* distances in G^*_{-C} and the restriction of ϕ to X^*. To see this, consider a path $Q = v_1 v_2 \ldots v_k$ in G^*_{-C} with $v_1, v_k \in X^*$. Then by a telescoping sums argument, the reduced length $l_\phi(Q)$ of Q

is $\sum_{i=1}^{k-1} l(v_i v_{i+1}) - \phi(v_{i+1}) + \phi(v_i) = l(Q) + \phi(v_1) - \phi(v_k)$, showing the desired.

In order to compute shortest path distances in $(G_1+G_2)^*$, FR-Dijkstra is applied, where G^*_{-C} is represented by a matrix of X^*-to-X^* distances and the dual arcs of C^* are given explicitly. The algorithm only computes distances to vertices belonging to X^*. As we saw above, these values suffice both to update the flow on C and to update the X^*-to-X^* reduced lengths in G^*_{-C}. Hence, it suffices for the algorithm to maintain only the *restriction* of the face potential to vertices of X^*.

B. *Efficient Implementation of Algorithm* MAXPREFLOWINSEPARATEDGRAPH

We are now ready to describe our implementation of Algorithm MAXPREFLOWINSEPARATEDGRAPH. Let P, C, and C' be defined as in the beginning of the section and assume that a max preflow from s to C has been computed. Denote by H_C and $H_{C'}$ the outside of C and C', respectively. Motivated by the above, we shall represent the residual network contained in $\text{ext}(C)$ implicitly as $\text{DDG}^*(H_C)$ together with the flow values on arcs of C. Note that the vertex set of $\text{DDG}^*(H_C)$ is the set F_C of boundary faces contained in H_C.

The goal is to find in $O((|P| + |\partial P|^2)\log^2 |P|)$ time a max preflow from s to C' in $\text{ext}(C')$, implicitly represented in $\text{DDG}^*(H_{C'})$ and explicitly represented in C'. Let us first consider the simple case where the only holes of P are H_C and the hole bounded by C'. In this case, Algorithm MAXPREFLOWINSEPARATEDGRAPH should send a max flow from C to C' in P and then run the flow fixing procedure on C in $\text{ext}(C')$.

The first part can be done in $O(|P|\log|P|)$ time by a single call to the algorithm of Borradaile and Klein [3] in P (after adding a super-source and a super-sink, again we do not connect the shared vertices of C and C', if any, to the super-sink). Denote by $\hat{F}_C$ the set of faces incident to C and belonging to P. From the flow computed in P, we obtain $\text{IDDG}^*(P)$ restricted to vertex set $\hat{F}_C$ representing the updated residual network in P. With Klein's algorithm [23], this takes $O((|P| + |\partial P|^2)\log|P|)$ time since the number of faces in P is $O(|P|)$ and since the number of faces in $\hat{F}_C$ is bounded by the size of ∂P, implying that the number of pairs in $\hat{F}_C$ is $O(|\partial P|^2)$.

We now have dense distance dual graphs on F_C and $\hat{F}_C$ which together with the flow on C represent the updated pseudoflow and we can apply the flow fixing procedure of Borradaile et al. to C in $\text{ext}(C')$. This procedure outputs the updated network as the sum of a flow assignment f_C along C and a (partially represented) circulation ρ in $\text{ext}(C')$. The circulation is represented by a potential function ϕ restricted to $F_C + \hat{F}_C$. In order to extend the new flow to P we use the same trick as in [4] of replacing ρ by another circulation ρ'. This amounts to replacing ϕ by a new potential function ϕ' representing ρ'. We need to ensure that $f_C + \rho'$ is *feasible*, i.e., that it does not violate capacity constraints. As shown in [4], this can be done with a single Dijkstra computation in the dual, where arc weights are the residual capacities. In our case, the dual is the subgraph of G^* contained in the outside of C'. We apply FR-Dijkstra in the union of $G^*[P^*]$, $\text{DDG}^*(H_C)$, and the edges of C. This allows us to extend the flow to P in time $O((|P| + |\partial P|^2)\log^2|P|)$.

If there are no deficit vertices on C, we have the desired s-to-C' preflow in $\text{ext}(C')$, represented explicitly in P and implicitly outside C. Otherwise, we rerun the above steps with a limit on the amount of flow pushed in P as described in step 5 of Algorithm MAXPREFLOWINSEPARATEDGRAPH.

Next, we obtain the updated $\text{IDDG}^*(P)$ restricted to vertex set $\hat{F}_C + F_{C'}$ (again using Klein's algorithm). Finally, to obtain the desired $\text{DDG}^*(H_{C'})$, we run FR-Dijkstra on the union of $\text{IDDG}^*(P)$, $\text{DDG}^*(H_C)$, and the dual edges of C. Total time for all these steps is $O((|P|+|\partial P|^2)\log^2|P|)$.

We assumed above that P had only two holes, namely H_C and the hole bounded by C'. We now show how to handle the general case with k additional holes $H_1, \ldots, H_k$ bounded by cycles $C_1, \ldots, C_k$, respectively. We will execute step 2 of Algorithm MAXPREFLOWINSEPARATEDGRAPH using an efficient implementation of Algorithm PUSHFLOWTHROUGHHOLES$(P, C, C', \{H_1, \ldots, H_k\})$. It follows from the description of this algorithm and from Lemma 6, that we need to support a sequence of $O(1)$ operations of the following two types:

1) Find a max flow in P from a cycle C_i to a cycle C_j (the one-hole case of Algorithm PUSHFLOWTHROUGHHOLES),
2) Run the flow fixing procedure along C_i in $P + H_1 + \ldots + H_i$, for some i.

During the course of Algorithm PUSHFLOWTHROUGHHOLES, we will represent the current pseudoflow in a way similar to the one above. The pseudoflow will be explicitly represented in P and implicitly for the arcs belonging to $H_i \backslash C_i$, for $i = 1, \ldots, k$. Before the call to Algorithm PUSHFLOWTHROUGHHOLES, all non-zero flow is contained in $\text{ext}(C)$ so we can represent the initial residual network in each H_i implicitly as $\text{DDG}^*(H_i)$. By Lemma 4, each of these dense distance dual graphs can be precomputed within the desired time bound of our overall algorithm.

Let us extend the definition of sets F_C and $\hat{F}_C$ to holes $H_1, \ldots, H_k$. Denote by F_i the set of boundary faces contained in H_i and denote by $\hat{F}_i$ the set of faces incident to C_i and contained in P. Note that the dual vertices of $\text{DDG}^*(H_i)$ are the faces of F_i. As flow is accumulated during the course of Algorithm PUSHFLOWTHROUGHHOLES, the residual network inside each H_i needs to be updated. Since the flow fixing procedure is applied to holes of P, it suffices to maintain a potential function ϕ on $F_1 + \ldots + F_k + \hat{F}_1 + \ldots + \hat{F}_k$.

Now let us describe how to execute the second type of operation above, running the flow fixing procedure along C_i in $P + H_1 + \ldots + H_i$, for some i. First we obtain $\mathrm{IDDG}^*(P)$ restricted to $\hat{F}_1 + \ldots + \hat{F}_i$ representing the current residual network in P. Again, we obtain this graph in $O((|P|+|\partial P|^2) \log |P|)$ time with Klein's algorithm [23]. We then apply FR-Dijkstra to a graph defined by the union of $\mathrm{IDDG}^*(P)$, $\mathrm{DDG}^*(H_1), \ldots, \mathrm{DDG}^*(H_i)$, and the edges of $C_1, \ldots, C_i$ weighted by their residual capacities. This gives in $O(|\partial P|^2 \log^2 |P|)$ time a dense distance dual graph implicitly representing the residual network in $P + H_1 + \ldots + H_i$. The flow fixing procedure is given this graph and it identifies a potential function ϕ on $F_i + \hat{F}_i$ defining the circulation ρ found as well as a flow assignment f_{C_i} on C_i. This step also takes $O(|\partial P|^2 \log^2 |P|)$ time.

As in the two holes case, we will compute a new potential function ϕ' representing a circulation ρ' such that $f_{C_i} + \rho'$ is feasible. As the circulation can push flow through P and any of the holes $H_1, \ldots, H_i$, we need to extend ϕ' to the faces of P as well as the sets $F_1 + \ldots + F_i$ of boundary faces. We can do this with a single FR-Dijkstra computation in the union of P, $\mathrm{DDG}^*(H_1), \ldots, \mathrm{DDG}^*(H_i)$, and the edges of $C_1, \ldots, C_i$. Finally we update the explicit flow in P with respect to the circulation ρ'.

We have shown how to support type 2 operations in $O((|P| + |\partial P|^2) \log^2 |P|)$ time. Type 1 operations can be executed in $O(|P| \log |P|)$ time with a single call to the algorithm of Borradaile and Klein since we maintain flow explicitly in P. By Lemma 6, there are only $O(1)$ type 1 and type 2 operations in total so Algorithm PUSHFLOWTHROUGHHOLES and hence step 2 of Algorithm MAXPREFLOWINSEPARATEDGRAPH runs in $O((|P| + |\partial P|^2) \log^2 |P|)$ time. From the description above, it follows that the remaining steps of Algorithm MAXPREFLOWINSEPARATEDGRAPH can be executed within the same time bound.

We can now conclude this section with the main result of the paper.

Theorem 1: Given a planar n-vertex digraph $G = (V, E)$ and given a fixed source s in G, max st-flow values from s to each sink $t \in V \setminus \{s\}$ can be computed in a total of $O(n \log^3 n)$ time.

V. REPORTING MIN CUT SETS

The same ideas we used in Section III and Section IV can also lead to an efficient algorithm for reporting minimum st-cut sets. Due to space constraint, the complete details are left for the full version of this paper. We use the duality between cut-sets in a planar graph and cycles in the dual planar graph. Let C be an st-cut set, that is a set of arcs whose removal separates s from t. Then, the dual arcs of the arcs of C form a cycle which separates s from t and goes clockwise around t. In particular, if f is a max st-flow, then a clockwise cycle M (in the dual graph) which separates s from t and whose length is equal to the value of f is a dual of a min-cut. In this case we call M *a shortest st-separating cycle*. If we consider the dual of the residual graph (the lengths are given by residual capacities with respect to f), then the length of M is 0.

We begin by decomposing the graph into $O(n^{1/2})$ pieces of size $O(n^{1/2})$, each having $O(n^{1/4})$ boundary vertices and a constant number of holes. This can be done in $O(n \log n)$ time with an algorithm by Italiano et al. [18]. Consider a source s and a sink t, let C be the outer boundary of the piece containing t. Our algorithm for this problem finds a max st-flow in a way similar to our algorithm from the previous section, again using the flow partition scheme of [21]. First we find a max sC-flow, then we find a max Ct-flow, next we run the flow fixing procedure, and last we return any excess to s and any deficit to t. However, we do not compute the st-flow itself, but an implicit representation of the flow, as in Section IV.

Unlike Section IV, here we need to update the dense distance dual graph, which stores the implicit representation of the flow, following the operation which returns excess flow from C to s. The following observation is the key for this step. Fix two faces x, y incident to C. Then, for every shortest st-separating cycle M, which contains a subpath Q that is embedded outside C and goes from x to y in the dual graph, the path Q is one of two possible paths: either the shortest x to y path that goes clockwise around s, or the shortest x to y path that goes counterclockwise around s, with respect to the original capacity of the graph (regardless of the residual flow).

After we have the implicit max st-flow representation, we retrieve a min st-cut set using a lemma which states that a dual cycle M with length 0, with respect to the residual capacity, such that M has flow entering it, is a shortest st-separating cycle. We consider only arcs of the dense distance graph of the implicit representation of the flow which have length 0, and look for a cycle that consists of such arcs, such that at least one of these arcs corresponds to a dual path that has flow crossing it.

Altogether, we get the following result:

Theorem 2: There exists a data structure which, given a planar n-vertex digraph $G = (V, E)$ and a fixed source s in G, can answer queries about min st-cut sets for a given t. The data structure requires $O(n^{1.5} \log^2 n)$ time for preprocessing and the min st-cut set M is computed in $O(|M|)$ time. The data structure requires $O(n^{1.5})$ space.

VI. CONCLUDING REMARKS

We gave an $O(n \log^3 n)$ time algorithm for the problem of finding max st-flow values for a fixed source s and all sinks $t \in V \setminus \{s\}$ in an n-vertex planar digraph $G = (V, E)$. The previous best known solution was to perform $n - 1$ executions of a single-source single-sink max flow algorithm, which gave an $O(n^2 \log n)$ time bound with the algorithm of Borradaile and Klein [3].

An immediate corollary of our result is a near-quadratic time algorithm for finding max st-flow values for all source/sink pairs (s, t). We showed that the number of distinct max st-flow values is quadratic in the worst case. Hence, our algorithm is optimal up to logarithmic factors.

Prior to this result, computing all max-flow values in a directed planar graph required up to $\Theta(n^2)$ max-flow computations. We conjecture that a similar improvement is possible for general directed graphs. Another interesting direction to pursue is to prove or disprove the existence of a data structure similar to the oracle in [6], which can answer queries for max st-flow values in constant time after $o(n^2)$ preprocessing. A related question is whether it is possible to improve the time needed to find max st-flow values for k given input pairs (s, t)? Also, is it possible to remove the log-factors, i.e., compute all max-flow values in optimal $O(n^2)$ time? Finally, the time needed to compute all cut-sets $\mathcal{C}$ in the planar digraph with our algorithm is $O(n^{2.5} \log^2 n + \sum_{C \in \mathcal{C}} |C|)$. Can this running time be improved to $O(n^{2.5-\epsilon} + \sum_{C \in \mathcal{C}} |C|)$?

References

[1] S. R. Arikati, S. Chaudhuri, and C. D. Zaroliagis, "All-pairs min-cut in sparse networks," *J. Algorithms*, vol. 29, pp. 82–110, 1998.

[2] A. A. Benczúr, "Counterexamples for directed and node capacitated cut-trees," *SIAM J. Comput.*, vol. 24, no. 3, pp. 505–510, 1995.

[3] G. Borradaile and P. Klein, "An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph," *J. ACM*, vol. 56, no. 2, pp. 1–30, 2009.

[4] G. Borradaile, P. N. Klein, S. Mozes, Y. Nussbaum, and C. Wulff-Nilsen, "Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time," in *Proc. 52nd IEEE Annu. Symp. Found. Comput. Sci. (FOCS)*, 2011, pp. 170–179.

[5] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen, "Min st-cut oracle for planar graphs with near-linear preprocessing time," submitted to a journal. Announced at [6].

[6] ——, "Min st-cut oracle for planar graphs with near-linear preprocessing time," in *Proc. 51st IEEE Annu. Symp. Found. Comput. Sci. (FOCS)*, 2010, pp. 601–610.

[7] S. Cabello, É. Colin de Verdière, and F. Lazarus, "Finding shortest non-trivial cycles in directed graphs on surfaces," in *Proc. 26th Annu. Symp. on Comput. Geom. (SoCG)*, 2010, pp. 156–165.

[8] H. Y. Cheung, L. C. Lau, and K. M. Leung, "Graph connectivities, network coding, and expander graphs," in *Proc. 52nd IEEE Annu. Symp. Found. Comput. Sci. (FOCS)*, 2011, pp. 190–199.

[9] J. Erickson, "Maximum flows and parametric shortest paths in planar graphs," in *Proc. 21st Annu. ACM-SIAM Symp. Discret. Algorithms (SODA)*, 2010, pp. 794–804.

[10] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM J. Comput.*, vol. 4, no. 4, pp. 507–518, 1975.

[11] J. Fakcharoenphol and S. Rao, "Planar graphs, negative weight edges, shortest paths, and near linear time," *J. Comput. Syst. Sci.*, vol. 72, no. 5, pp. 868–889, 2006.

[12] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Can. J. Math.*, pp. 399–404, 1956.

[13] ——, *Flows in Network*. Princeton Univ. Press, 1962.

[14] R. E. Gomory and T. C. Hu, "Multi-terminal network flows," *J. SIAM*, vol. 9, no. 4, pp. 551–570, 1961.

[15] D. Gusfield, "Very simple methods for all pairs network flow analysis," *SIAM J. Comput.*, vol. 19, no. 1, pp. 143–155, 1990.

[16] J. Hao and J. B. Orlin, "A faster algorithm for finding the minimum cut in a directed graph," *J. Algorithms*, vol. 17, pp. 424–446, 1994.

[17] R. Hassin, "Maximum flow in (s, t) planar networks," *Inf. Process. Lett.*, vol. 13, no. 3, p. 107, 1981.

[18] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen, "Improved algorithms for min cut and max flow in undirected planar graphs," in *Proc. 43rd Annu. ACM Symp. Theor. Comput. (STOC)*, 2011, pp. 313–322.

[19] F. Jekinke, "On the maximum number of different entries in the terminal capacity matrix of oriented communication nets," *IEEE Trans. Circuit Theor.*, vol. 10, no. 2, pp. 307–308, 1963.

[20] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, no. 1, pp. 1–13, 1977.

[21] D. B. Johnson and S. M. Venkatesan, "Partition of planar flow networks," in *Proc. 24th IEEE Annu. Symp. Found. Comput. Sci. (FOCS)*, 1983, pp. 259–264.

[22] H. Kaplan, S. Mozes, Y. Nussbaum, and M. Sharir, "Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications," in *Proc. 23th Annu. ACM-SIAM Symp. Discret. Algorithms (SODA)*, 2012, pp. 338–355.

[23] P. N. Klein, "Multiple-source shortest paths in planar graphs," in *Proc. 16th Annu. ACM-SIAM Symp. Discret. Algorithms (SODA)*, 2005, pp. 146–155.

[24] V. V. Williams, "Multiplying matrices faster than Coppersmith-Winograd," in *Proc. 44th Annu. ACM Symp. Theor. Comput. (STOC)*, 2012, pp. 887–898.