Nested Dissection Meets IPMs: Planar Min-Cost Flow in Nearly-Linear Time

Sally Dong * Yu Gao † Gramoz Goranci ‡ Yin Tat Lee § Richard Peng ¶
Sushant Sachdeva ‖ Guanghao Ye **

Abstract

We present a nearly-linear time algorithm for finding a minimum-cost flow in planar graphs with polynomially bounded integer costs and capacities. The previous fastest algorithm for this problem was based on interior point methods (IPMs) and worked for general sparse graphs in $O(n^{1.5}\text{poly}(\log n))$ time [Daitch-Spielman, STOC'08].

Intuitively, $\Omega(n^{1.5})$ is a natural runtime barrier for IPM based methods, since they require $\sqrt{n}$ iterations, each routing a possibly-dense electrical flow. To break this barrier, we develop a new implicit representation for flows based on generalized nested-dissection [Lipton-Rose-Tarjan, JSTOR'79] and approximate Schur complements [Kyng-Sachdeva, FOCS'16]. This implicit representation permits us to design a data structure to route an electrical flow with sparse demands in roughly $\sqrt{n}$ update time, resulting in a total running time of $O(n \cdot \text{poly}(\log n))$.

Our results immediately extend to all families of separable graphs.

1 Introduction

The minimum cost flow problem on planar graphs is a foundational problem in combinatorial optimization studied since the 1950's. It has diverse applications including network design, VLSI layout, and computer vision. The seminal paper of Ford and Fulkerson in the 1950's [19] presented an $O(n^2)$ time algorithm for the special case of max-flow on s,t-planar graphs, i.e., planar graphs with both the source and sink lying on the same face. Later, Itai and Shiloach [28] gave an nearly-linear time implementation of the Ford-Fulkerson algorithm using a priority queue. Over the decades since, a number of nearly-linear time max-flow algorithms have been developed for special graph classes, including undirected planar graphs by Reif, and Hassin-Johnson [50, 24], planar graphs by Fakcharoenphol-Rao [18], and finally bounded genus graphs by Chambers-Erickson-Nayyeri [10]. However, for the more general min-cost flow problem, there is no known result specializing on planar graphs with better guarantees than on general graphs. In this paper, we present the first nearly-linear time algorithm for min-cost flow on planar graphs:

THEOREM 1.1. (MAIN RESULT) *Let $G = (V, E)$ be a directed planar graph with n vertices. Assume that the demands $\boldsymbol{d}$, edge capacities $\boldsymbol{u}$ and costs $\boldsymbol{c}$ are all integers and bounded by M in absolute value. Then there is an algorithm that computes a minimum cost flow satisfying demand $\boldsymbol{d}$ in $O(n \log^{O(1)} n \log M)$ expected time.*

*sallyqd@uw.edu. University of Washington. Supported by a postdoctoral fellowship from NSERC (Natural Sciences and Engineering Research Council of Canada).

†ygao380@gatech.edu. Georgia Institute of Technology. Supported by NSF (National Science Foundation) award CCF-1846218.

‡gramoz.goranci@glasgow.ac.uk. University of Glasgow. Part of this work was done while the author was a postdoc at University of Toronto, and supported by Sushant Sachdeva's Discovery grant from NSERC.

§yintat@uw.edu. University of Washington. Supported by NSF awards CCF-1749609, DMS-1839116, DMS-2023166, CCF-2105772, a Microsoft Research Faculty Fellowship, a Sloan Research Fellowship, and a Packard Fellowship.

¶rpeng@cc.gatech.edu. Georgia Institute of Technology and University of Waterloo. Supported by NSF award CCF-1846218 and CCF-2106444.

‖sachdeva@cs.toronto.edu. University of Toronto. Supported by a Discovery grant awarded by NSERC.

**ghye@mit.edu. Massachusetts Institute of Technology. Supported by an MIT Presidential Fellowship. Part of this work was done while the author was a student at University of Washington.

Min-cost flow	Time Bound	Reference
Strongly polytime	$O(m^2 \log n + mn \log n)$	[49]
Weakly polytime	$\widetilde{O}((m + n^{3/2}) \log M)$	[59]
Pseudo polytime + unit-capacity	$m^{\frac{4}{3}+o(1)} \log M$	[5]
Planar graph	$\widetilde{O}(n \log M)$	**This paper**
Unit-capacity planar graph	$O(n^{4/3} \log M)$	[33]
Graphs with treewidth τ	$\widetilde{O}(n\tau^2 \log M)$	[16]
Outerplanar graph	$O(n \log^2 n)$	[32]
Unidirectional, bidirectional cycle	$O(n)$, $O(n \log n)$	[56]

Table 1: Fastest known exact algorithms for the min-cost flow problem, ordered by the generality of the result. Here, n is the number of vertices, m is the number of edges, M is the maximum of edge capacity and cost value.

Our algorithm is fairly general and uses the planarity assumption minimally. It builds on a combination of interior point methods (IPMs), approximate Schur complements, and nested-dissection, with the latter being the only component that exploits planarity. Specifically, we require that for any subgraph of G with k vertices, we can find an $O(\sqrt{k})$-sized balanced vertex separator in nearly-linear time. As a result, the algorithm naturally generalizes to all graphs with small separators. Given a class $\mathcal{C}$ of graphs closed under taking subgraphs, we say it is n^α-*separable* if there are constants $0 < c < 1$ and $b > 0$ such that every graph in $\mathcal{C}$ with n vertices and m edges has a balanced vertex separator with at most bn^α vertices, and both components obtained after removing the separator vertices have at most cm edges. Then, our algorithm generalizes as follows:

THEOREM 1.2. *Let $\mathcal{C}$ be a n^α-separable graph class such that we can compute a balanced separator for any graph in $\mathcal{C}$ with m edges in $s(m)$ time for some concave function s. Given a graph $G \in \mathcal{C}$ with n vertices and m edges, and integer demands $\boldsymbol{d}$, edge capacities $\boldsymbol{u}$ and costs $\boldsymbol{c}$, all bounded by M in absolute value, there is an algorithm that computes a minimum cost flow on G satisfying demand $\boldsymbol{d}$ in $O(m^{1/2+\alpha} \log^{O(1)} m \log M + s(m) \log^{O(1)} m)$ expected time.*

Beyond the study of structured graphs, we believe our paper is of broader interest. The study of efficient optimization algorithms on geometrically structured graphs is a topic at the intersection of computational geometry, graph theory, combinatorial optimization, and scientific computing, that has had a profound impact on each of these areas. Connections between planarity testing and 3-vertex connectivity motivated the study of depth-first search algorithms [55], and using geometric structures to find faster solvers for structured linear systems provided foundations of Laplacian algorithms as well as combinatorial scientific computing [43, 23]. Several surprising insights from our nearly-linear time algorithm are:

1. We are able to design a data structure for maintaining a feasible primal-dual (flow/slack) solution that allows sublinear time updates – requiring $\widetilde{O}(\sqrt{nK})$ time [1] for a batch update consisting of updating the flow value of K edges. This ends up not being a bottleneck for the overall performance because the interior point method only takes roughly $\sqrt{n}$ iterations and makes K-sparse updates roughly $\sqrt{n/K}$ times, resulting in a total running time of $\widetilde{O}(n)$.

2. We show that the subspace constraints on the feasible primal-dual solutions can be maintained implicitly under dynamic updates to the solutions. This circumvents the need to track the infeasibility of primal solutions (flows), which was required in previous works.

We hope our result provides both a host of new tools for devising algorithms for separable graphs, as well as insights on how to further improve such algorithms for general graphs.

1.1 Previous Work The min-cost flow problem is well studied in both structured graphs and general graphs. Table 1 summarizes the current best algorithms for different settings.

[1] Throughout the paper, we use $\widetilde{O}$ to omit polylogarithmic factors in n.

Min-cost flow / max-flow on general graphs. Here, we focus on recent exact max-flow and min-cost flow algorithms. For an earlier history, we refer the reader to the monographs [37, 3]. For the approximate max-flow problem, we refer the reader to the recent papers [11, 51, 35, 52, 53, 6].

To understand the recent progress, we view the maximum flow problem as finding a unit s,t-flow with minimum ℓ_∞-norm, and the shortest path problem as finding an unit s,t-flow with minimum ℓ_1-norm. Prior to 2008, almost all max-flow algorithms reduced this ℓ_∞ problem to a sequence of ℓ_1 problem (shortest path) since the latter can be solved efficiently. This changed with the celebrated work of Spielman and Teng, which shows how to find electrical flows (ℓ_2-minimizing unit s,t-flow) in nearly-linear $\widetilde{O}(m)$ time [54]. Since the ℓ_2-norm is closer to ℓ_∞ than ℓ_1, this gives a more powerful primitive for the maximum flow problem. In 2008, Daitch and Spielman demonstrated that one could apply Interior Point Methods (IPMs) to reduce computing min-cost flow to roughly $\sqrt{m}$ electrical flow computations and obtained an algorithm with a running time of $\widetilde{O}(m^{3/2}\log M)$ [14]. This follows from the fact that IPMs take $\widetilde{O}(\sqrt{m})$ iterations and each iteration requires solving an electrical flow problem, which can be now be solved in $\widetilde{O}(m)$ time due to the work of Spielman and Teng. Since then, several algorithms have utilized electrical flows and other stronger primitives for solving max-flow and min-cost flow problems.

For graphs with unit capacities, Mądry gave a $\widetilde{O}(m^{10/7})$-time max-flow algorithm, the first that broke the 3/2-exponent barrier [45]. It was later improved and generalized to $m^{4/3+o(1)}\log M$ [5] for the min-cost flow problem. Kathuria et al. [34] gives a similar run-time of $m^{4/3+o(1)}U^{1/3}$ where U is the max capacity. The runtime improvement comes from decreasing the number of iterations of IPM to $\widetilde{O}(m^{1/3})$ via a more powerful primitive of $\ell_2+\ell_p$ minimizing flows [40].

For general capacities, the runtime has recently been improved to $\widetilde{O}((m+n^{3/2})\log M)$ [59] for min-cost flow and $\widetilde{O}(m^{\frac{3}{2}-\frac{1}{328}}\log M)$ [20] for max-flow. Both algorithms focus on decreasing the per-iteration cost of IPMs by dynamically maintaining electrical flows.

Max-flow on planar graphs. The planar max-flow problem has an equally long history. We refer the reader to the thesis [7] for a detailed exposition. In the seminar work of Ford and Fulkerson [19] that introduced the max-flow min-cut theorem, they also gave a max-flow algorithm for s,t-planar graphs (planar graphs where the source and sink lie on the same face). This algorithm iteratively sends flow along the top-most augmenting path. Itai and Shiloach showed how to implement each step in $O(\log n)$ time, thus giving an $O(n\log n)$ time algorithm for s,t-planar graphs [28]. For s,t-planar graphs, the current best run-time is $O(n)$ by Henzinger, Klein, Rao, and Subramanian [25].

For undirected planar graphs, Reif first gave an $O(n\log^2 n)$ time algorithm for finding the max-flow value [50]. Hassin and Johnson then showed how to compute the flow in the same run-time [24]. The current best run-time is $O(n\log\log n)$ by Italiano, Nussbaum, Sankowski, and Wulff-Nilsen [29].

For general planar graphs, Weihe gave the first $O(n\log n)$ time algorithm, assuming the graph satisfies certain connectivity conditions [61]. Later, Borradaile and Klein gave an $O(n\log n)$ time algorithm for any planar graph [8].

The multiple-source multiple-sink version of max-flow is considered much harder, even on planar graphs. The first result of $O(n^{1.5})$ time was by Miller and Naor when sources and sinks are all on same face [47]. This was then improved to $O(n\log^3 n)$ in [9].

For generalizations of planar graphs, Chambers, Ericskon and Nayyeri gave the first nearly linear-time algorithm for max-flow on graphs embedded on bounded-genus surfaces [10]. Miller and Peng gave an $\widetilde{O}(n^{6/5})$-time algorithm for approximating undirected max-flow [48] for the class of $O(\sqrt{n})$-separable graphs, although this is superseded by the previously mentioned works for general graphs [51, 35].

Min-cost flow on planar graphs. Imai and Iwano gave a $O(n^{1.594}\log M)$-time algorithm for min-cost flow [27] for the more general class of $O(\sqrt{n})$-separable graphs. To the best of our knowledge, there is little else known about min-cost flow on general planar graphs. In the special case of unit capacities, [4, 42] gives an $O(n^{6/5}\log M)$-time algorithm for min-cost perfect matching in bipartite planar graphs, and Karczmarz and Sankowski gives a $O(n^{4/3}\log M)$-time algorithm for min-cost flow. Currently, bounded treewidth graphs is the only graph family we know that admits min-cost flow algorithms that run in nearly-linear time [16].

1.2 Challenges Here, we discuss some of the challenges in developing faster algorithms for the planar min-cost flow problem from a convex optimization perspective. For a discussion on challenges in designing combinatorial

algorithms, we refer the reader to [36]. Prior to our result, the fastest min-cost flow algorithm for planar graphs is based on Interior point methods (IPMs) and takes $\widetilde{O}(n^{3/2}\log M)$ time [14]. Intuitively, $\Omega(n^{3/2})$ is a natural runtime barrier for IPM-based methods, since they require $\Omega(\sqrt{n})$ iterations, each routing a possibly-dense electrical flow.

Challenges in improving the number of iterations. The $\Omega(\sqrt{n})$ term comes from the fact that IPM uses the electrical flow problem (ℓ_2-type problem) to approximate the shortest path problem (ℓ_1-type problem). This $\Omega(\sqrt{n})$ term is analogous to the flow decomposition barrier: in the worst case, we need $\Omega(n)$ shortest paths (ℓ_1-type problem) to solve the max-flow problem (ℓ_∞-type problem). Since ℓ_2 and ℓ_∞ problems differ a lot when there are $s-t$ paths with drastically different lengths, difficult instances for electrical flow-based max-flow methods are often serial-parallel (see Figure 3 in [11] for an example). Therefore, planarity does not help to improve the $\sqrt{n}$ term. Although more general $\ell_2+\ell_p$ primitives have been developed [40, 2, 1], exploiting their power in designing current algorithms for exact max-flow problem has been limited to perturbing the IPM trajectory, and such a perturbation only works when the residual flow value is large. In all previous works tweaking IPMs for breaking the 3/2-exponent barrier [45, 46, 13, 34, 5], an augmenting path algorithm is used to send the remaining flow at the end. Due to the residual flow restriction, all these results assume unit-capacities on edges, and it seems unlikely that planarity can be utilized to design an algorithm for polynomially-large capacities with fewer than $\sqrt{n}$ IPM iterations.

Challenges in improving the cost per iteration. Recently, there has been much progress on utilizing data structures for designing faster IPM algorithms for general linear programs and flow problems on general graphs. For general linear programs, robust interior point methods have been developed recently with running times that essentially match the matrix multiplication cost [12, 57, 60, 26, 58]. This version of IPM ensures that the ℓ_2 problem solved changes in a sparse manner from iteration to iteration. When used to design graph algorithms, the i-th iteration of a robust IPM involves computing an electrical flow on some graph G_i. The edge support remains unchanged between iterations, though the edge weights change. Further, if K_i is the number of edge weight changes between G_i and G_{i+1}, then robust IPMs guarantee that

$$\sum_i \sqrt{K_i} = \widetilde{O}(\sqrt{m}\log M).$$

Roughly, this says that, on average, each edge weight changes only poly-log many times throughout the algorithm. Unfortunately, any sparsity bound is not enough to achieve nearly-linear time. Unlike the shortest path problem, changing *any* edge in a connected graph will result in the electrical flow changing on essentially *every* edge. Therefore, it is very difficult to implement (roubst) IPMs in sublinear time per iteration, even if the subproblem barely changes every iteration. On moderately dense graphs with $m=\Omega(n^{1.5})$, this issue can be avoided by first approximating the graph by sparse graphs and solving the electrical flow on the sparse graphs. This leads to $\widetilde{O}(n) \ll \widetilde{O}(m)$ time cost per step [60]. However, on sparse graphs, significant obstacles remain. Recently, there has been a major breakthrough in this direction by using random walks to approximate the electrical flow [20]. Unfortunately, this still requires $m^{1-\frac{1}{328}}$ time per iteration.

Finally, we note that [16] gives an $\widetilde{O}(n\tau^2\log M)$-time algorithm for linear programs with τ treewidth. Their algorithm maintains the solution using an implicit representation. This implicit representation involves a $\tau\times\tau$ matrix that records the interaction between every variable within the separator. Each step of the algorithm updates this matrix once and it is not the bottleneck for the $\widetilde{O}(n\tau^2\log M)$-time budget. However, for planar graphs, this $\tau\times\tau$ matrix is a dense graph on $\sqrt{n}$ vertices given by the Schur complement on the separator. Hence, updating this using their method requires $\Omega(n)$ per step.

Our paper follows the approach in [16] and shows that this dense graph can be sparsified. This is however subtle. Each step of the IPM makes a global update via the implicit representation, hence checking whether the flow is feasible takes at least linear time. Therefore, we need to ensure each step is exactly feasible despite the approximation. If we are unable to do that, the algorithm will need to fix the flow by augmenting paths at the end like [34, 5], resulting in super-linear time and polynomial dependence on capacities, rather than logarithmic.

1.3 Our Approaches

In this section, we introduce our approach and explain how we overcome the difficulties we mentioned. The minimum-cost flow problem can be reformulated into a linear program in the following

primal-dual form:

$$(\text{Primal}) = \min_{\mathbf{B}^\top \boldsymbol{f}=\mathbf{0},\, \boldsymbol{l}\le\boldsymbol{f}\le\boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{f} \quad \text{and} \quad (\text{Dual}) = \min_{\mathbf{B}\boldsymbol{y}+\boldsymbol{s}=\boldsymbol{c}} \sum_i \min(\boldsymbol{l}_i \boldsymbol{s}_i, \boldsymbol{u}_i \boldsymbol{s}_i),$$

where $\mathbf{B} \in \mathbb{R}^{m\times n}$ is an edge-vertex incidence matrix of the graph, $\boldsymbol{f}$ is the flow and $\boldsymbol{s}$ is the slack (or adjusted cost vector). The primal is the minimum-cost circulation problem and the dual is a variant of the minimum-cut problem. Our algorithm for min-cost flow is composed of two new data structures (Section 2.4) and a novel application of IPM (Section 2.1). The IPM method reduces solving a linear program to applying a sequence of $\widetilde{O}(\sqrt{m}\log M)$ projections and the data structures implement the primal and dual projection steps roughly in $\widetilde{O}(\sqrt{m})$ amortized time.

Robust IPM. We first explain the IPM methods briefly. To minimize $\boldsymbol{c}^\top \boldsymbol{f}$, each step of the IPM method moves the flow vector $\boldsymbol{f}$ to the direction of $-\boldsymbol{c}$. However, such $\boldsymbol{f}$ may exceed the maximum or minimum capacities. IPM incorporates these capacity constraints by routing flows slower when they are approaching their capacity bounds. This is achieved by controlling the edge weights $\mathbf{W}$ and direction $\boldsymbol{v}$ in each projection step. Both $\mathbf{W}$ and $\boldsymbol{v}$ are roughly chosen from some explicit entry-wise formula of $\boldsymbol{f}$ and $\boldsymbol{s}$, namely, $\mathbf{W}_{ii} = \psi_1(\boldsymbol{f}_i, \boldsymbol{s}_i)$ and $\boldsymbol{v}_i = \psi_2(\boldsymbol{f}_i, \boldsymbol{s}_i)$. Hence, the main bottleneck is to implement the projection step (computing $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$). For the min-cost flow problem, this projection step corresponds to an electrical flow computation.

Recently, it has been observed that there is a lot of freedom in choosing the weight $\mathbf{W}$ and the direction $\boldsymbol{v}$ (See for example [12]). Instead of computing them exactly, we maintain some entry-wise approximation $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ of $\boldsymbol{f}, \boldsymbol{s}$ and use them to compute $\mathbf{W}$ and $\boldsymbol{v}$. By updating $\overline{\boldsymbol{f}}_i, \overline{\boldsymbol{s}}_i$ only when $\boldsymbol{f}_i, \boldsymbol{s}_i$ changed significantly, we can ensure $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ has mostly sparse updates. Since $\mathbf{W}$ and $\boldsymbol{v}$ are given by some entry-wise formula of $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$, this ensures that $\mathbf{W}, \boldsymbol{v}$ change sparsely and in turn allows us to maintain the corresponding projection $\mathbf{P}_{\boldsymbol{w}}$ via low-rank updates.

We refer to IPMs that use approximate $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$ as robust IPMs. In this paper, we apply the version given in [16] in a black-box manner. In Section 2.1, we state the IPM we use. The key challenge is implementing each step in roughly $\widetilde{O}(\sqrt{m})$ time.

Separators and Nested Dissection. Our data structures rely on the separability property of the input graph, which dates back to the nested dissection algorithms for solving planar linear systems [43, 21]. By recursively partitioning the graph into edge-disjoint subgraphs (regions) using balanced vertex separators, we can construct a hierarchical decomposition of a planar graph G which is called the separator tree [18]. This is a binary search tree over the edges in G. Each node in the separator tree represents a region in G. In planar graphs, for a region H with $|H|$ vertices, a $O(\sqrt{|H|})$-vertex separator suffices to partition it into two balanced sub-regions which are represented by the two children of H in the separator tree. The two subregions partition the edges in H and share only vertices in the separator. We call the set of vertices in a region H that appear in the separators of its ancestors the *boundary* of H. Any two regions can only share vertices on their boundaries unless one of them is an ancestor of the other.

Nested dissection algorithms [43, 21] essentially replaces each region by a graph involving only its boundary vertices, in a bottom-up manner. For linear systems, solving the dense $\sqrt{n}\times\sqrt{n}$ submatrix corresponding to the top level vertex separator leads to a runtime of $n^{\omega/2}$ where ω is the matrix multiplication exponent. For other problems as shortest path, this primitive involving dense graphs can be further accelerated using additional properties of distance matrices [18].

Technique 1: Approximate Nested Dissection and Lazy Propagation Our representation of the Laplacian inverse, and in turn the projection operators, hinges upon a sparsified version of the nested dissection representation. That is, instead of a dense inverse involving all pairs of boundary vertices, we maintain a sparse approximation. This sparsified nested dissection has been used in the approximate undirected planar flow algorithm from [48]. However, that work pre-dated (and in some sense motivated) subsequent works on nearly-linear time approximations of Schur complements on general graphs [39, 41, 38]. Re-incorporating these sparsified algorithms gives run-time dependencies that are nearly-linear, instead of quadratic, in separator sizes, with an overall error that is acceptable to the robust IPM framework.

By maintaining objects with size nearly equal to the separator in each node of the separator tree, we can support updating an single edge or a batch of edges in the graph efficiently. Our data structures for maintaining the intermediate vector $\boldsymbol{z}$, the flow and slack vector and for maintaining Schur complements all ultilize this idea. For example, to maintain Schur complement of a region H onto its boundary (which is required in implementating the step), we maintain (1) Schur complements of its children onto their boundaries recursively and (2) Schur

complement of the children's boundaries onto the boundary of H. Thus, to update an edge, the path in the decomposition tree from the leaf node H containing the edge to the root is visited. To update multiple edges in batch, each node in the union of the tree paths are visited. The runtime depends nearly linear to the total number of boundary vertices on all nodes (regions) in the union. For K edges being updated, the runtime is bounded by $\widetilde{O}(\sqrt{mK})$. The edge weight $\mathbf{W}$ and direction $\boldsymbol{v}$ are maintained similarly. Each step of our IPM algorithm takes $\widetilde{O}(\sqrt{mK_i})$ where K_i is the number of coordinates changed in $\mathbf{W}$ and $\boldsymbol{v}$ in the i-th step. Such a recursive approximate Schur complement structure was used in [22], where the authors achieved a running time of $\widetilde{O}(\sqrt{m}K_i)$.

Technique 2: Batching the changes. It is known that over t iterations of an IPM, the number of coordinate changes (by more than a constant factor) in $\mathbf{W}$ or $\boldsymbol{v}$ are bounded by $O(t^2)$. This directly gives $\sum_{i=1}^{\widetilde{O}(\sqrt{m})} K_i = m$ and thus a total run-time of $\sqrt{m}\left(\sum_{i=1}^{\widetilde{O}(\sqrt{m})}\sqrt{K_i}\right) = \widetilde{O}(m^{1.25})$. In order to obtain a nearly-linear run-time, the robust IPM carefully batches the updates in different steps. In the i-th step, if the change in an edge variable has exceeded some fixed threshold compared to its value in the $i - 2^l$-th step for some $l \leq \ell_i$, we adjust its approximation. (Here, ℓ_i is the number of trailing zeros in the binary representation of i, i.e. 2^{ℓ_i} is the largest power of 2 that divides i.) This ensures that K_i, the number of coordinate changes at step i, is bounded by $\widetilde{O}(2^{2\ell_i})$. Since each value of ℓ_i arises once every 2^{ℓ_i} steps, we can prove that the sum of square roots of the number of changes over all steps is bounded by $\widetilde{O}(m)$, i.e., $\sum_{i=1}^{\widetilde{O}(\sqrt{m})}\sqrt{K_i} = \widetilde{O}(\sqrt{m})$. Combined with the claim in the previous paragraph, this gives an $\widetilde{O}(m)$ overall runtime.

Technique 3: Maintaining feasibility via two projections. A major difficulty is maintaining a flow vector $\boldsymbol{f}$ that satisfies the demands exactly and a slack vector $\boldsymbol{s}$ that can be expressed as $\boldsymbol{s} = \boldsymbol{c} - \mathbf{B}\boldsymbol{y}$. If we simply project approximately in each step, the flow we send is not exactly a circulation. Traditionally, this can be fixed by computing the excess demand each step and sending flow to fix this demand. Since our edge capacities can be polynomially large, this step can take $\Omega(m)$ time. To overcome this feasibility problem, we note that different projection operators $\mathbf{P}_{\boldsymbol{w}}$ can be used in IPMs for $\boldsymbol{f}$ and $\boldsymbol{s}$ as long as each projection is close to the true projection and that the step satisfies $\mathbf{B}^\top \Delta\boldsymbol{f} = \mathbf{0}$ and $\mathbf{B}\Delta\boldsymbol{y} + \Delta\boldsymbol{s} = \mathbf{0}$ for some $\Delta\boldsymbol{y}$.

This two-operator scheme is essential to our improvement since one can prove that any projection that gives feasible steps for $\boldsymbol{f}$ and $\boldsymbol{s}$ simultaneously must be the exact electrical projection, which takes linear time to compute.

Technique 4: Detecting large changes by maintaining a random sketch on the separator tree. To detect the set of edges where the flow (or slack) value has changed by more than the threshold, we would like to query the sum of changes of squares of flow/slack values in a subgraph. (By supporting such queries efficiently, we can detect the set of edges recursively on the separator tree.) However, maintaining this sum directly is challenging since a single update may cause the flow value to change on each edge. We instead maintain a random sketch of the flow vector for each node. Roughly speaking, the flow vector in any region H can be written as $\overline{\mathbf{M}}_H\boldsymbol{z}$ where $\overline{\mathbf{M}}_H$ is defined recursively as $\overline{\mathbf{M}}_H = (\overline{\mathbf{M}}_{D_1} + \overline{\mathbf{M}}_{D_2})\mathbf{M}_H$, for D_1 and D_2 being the two children of H in the separator tree. This forest structure allows us to maintain $\boldsymbol{\Phi}\overline{\mathbf{M}}\boldsymbol{z}$ so that each change to $\mathbf{M}_H$ only affects its ancestors and any K-sparse update costs $\widetilde{O}(\sqrt{mK})$ time.

2 Main Theorems and Proof Outline

In this section, we give formal statements of the main theorems proved in the paper, along with an outline of the proof for our main result. The details of the proofs are deferred to the full version of the paper.

The main components of this paper are: the IPM from [16] (Section 2.1); the data structure to maintain a collection of Schur complements via nested dissection of the graph (Section 2.2); the sketching-based data structure to maintain the approximations $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$ needed in the IPM (Section 2.3), and finally the data structures to maintain the solutions $\boldsymbol{f}, \boldsymbol{s}$ and their corresponding projection matrices (Section 2.4).

2.1 Robust Interior Point Method In this subsection, we explain the robust interior point method developed in [16], which is a refinement of the methods in [12, 57]. Although there are many other robust interior point

methods, we simply refer to this method as RIPM. Consider a linear program of the form[2]

$$\min_{\boldsymbol{f}\in\mathcal{F}} \boldsymbol{c}^\top \boldsymbol{f} \quad \text{where} \quad \mathcal{F} = \{\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{b},\ \boldsymbol{l} \le \boldsymbol{f} \le \boldsymbol{u}\} \tag{2.1}$$

for some matrix $\mathbf{B} \in \mathbb{R}^{m\times n}$. As with many other IPMs, RIPM follows the central path $\boldsymbol{f}(t)$ from an interior point ($t \gg 0$) to the optimal solution ($t = 0$):

$$\boldsymbol{f}(t) \stackrel{\text{def}}{=} \arg\min_{\boldsymbol{f}\in\mathcal{F}} \boldsymbol{c}^\top \boldsymbol{f} - t\phi(\boldsymbol{f}) \quad \text{where } \phi(\boldsymbol{f}) \stackrel{\text{def}}{=} -\sum_i \log(\boldsymbol{f}_i - \boldsymbol{l}_i) - \sum_i \log(\boldsymbol{u}_i - \boldsymbol{f}_i),$$

where the term ϕ controls how close the flow $\boldsymbol{f}_i$ can be to the capacity constraints $\boldsymbol{u}_i$ and $\boldsymbol{l}_i$. Following the central path exactly is expensive. Instead, RIPM maintains feasible primal and dual solution $(\boldsymbol{f}, \boldsymbol{s}) \in \mathcal{F} \times \mathcal{S}$, where $\mathcal{S}$ is the dual space given by $\mathcal{S} = \{\boldsymbol{s} : \mathbf{B}\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c} \text{ for some } \boldsymbol{y}\}$, and ensures $\boldsymbol{f}(t)$ is an approximate minimizer. Specifically, the optimality condition for $\boldsymbol{f}(t)$ is given by

$$\begin{aligned} \mu^t(\boldsymbol{f}, \boldsymbol{s}) &\stackrel{\text{def}}{=} \boldsymbol{s}/t + \nabla\phi(\boldsymbol{f}) = \mathbf{0} \\ (\boldsymbol{f}, \boldsymbol{s}) &\in \mathcal{F} \times \mathcal{S} \end{aligned} \tag{2.2}$$

where $\mu^t(\boldsymbol{f}, \boldsymbol{s})$ measures how close $\boldsymbol{f}$ is to the minimizer $\boldsymbol{f}(t)$. RIPM maintains $(\boldsymbol{f}, \boldsymbol{s})$ such that

$$\|\gamma^t(\boldsymbol{f}, \boldsymbol{s})\|_\infty \le \frac{1}{C\log m} \text{ where } \gamma^t(\boldsymbol{f}, \boldsymbol{s})_i = \frac{\mu^t(\boldsymbol{f}, \boldsymbol{s})_i}{(\nabla^2\phi(\boldsymbol{f}))_{ii}^{1/2}}, \tag{2.3}$$

for some universal constant C. The normalization term $(\nabla^2\phi)_{ii}^{1/2}$ makes the centrality measure $\|\gamma^t(\boldsymbol{f}, \boldsymbol{s})\|_\infty$ scale-invariant in $\boldsymbol{l}$ and $\boldsymbol{u}$.

The key subroutine Centering takes as input a point close to the central path $(\boldsymbol{f}(t_{\text{start}}), \boldsymbol{s}(t_{\text{start}}))$, and outputs another point on the central path $(\boldsymbol{f}(t_{\text{end}}), \boldsymbol{s}(t_{\text{end}}))$. Each step of the subroutine decreases t by a multiplicative factor of $(1 - \frac{1}{\sqrt{m}\log m})$ and moves $(\boldsymbol{f}, \boldsymbol{s})$ within $\mathcal{F} \times \mathcal{S}$ such that $\boldsymbol{s}/t + \nabla\phi(\boldsymbol{f})$ is smaller for the current t. [16] proved that even if each step is computed approximately, Centering still outputs a point close to $(\boldsymbol{f}(t_{\text{end}}), \boldsymbol{s}(t_{\text{end}}))$ using $\widetilde{O}(\sqrt{m}\log(t_{\text{end}}/t_{\text{start}}))$ steps. See Algorithm 1 for a simplified version.

RIPM calls Centering twice. The first call to Centering finds a feasible point by following the central path of the following modified linear program

$$\min_{\substack{\mathbf{B}^\top(\boldsymbol{f}^{(1)}+\boldsymbol{f}^{(2)}-\boldsymbol{f}^{(3)})=\boldsymbol{b} \\ \boldsymbol{l}\le\boldsymbol{f}^{(1)}\le\boldsymbol{u},\ \boldsymbol{f}^{(2)}\ge\mathbf{0},\ \boldsymbol{f}^{(3)}\ge\mathbf{0}}} \boldsymbol{c}^{(1)\top}\boldsymbol{f}^{(1)} + \boldsymbol{c}^{(2)\top}\boldsymbol{f}^{(3)} + \boldsymbol{c}^{(2)\top}\boldsymbol{f}^{(3)}$$

where $\boldsymbol{c}^{(1)} = \boldsymbol{c}$, and $\boldsymbol{c}^{(2)}, \boldsymbol{c}^{(3)}$ are some positive large vectors. The above modified linear program is chosen so that we know an explicit point on its central path, and any approximate minimizer to this new linear program gives an approximate central path point for the original problem. The second call to Centering finds an approximate solution by following the central path of the original linear program. Note that both calls run the same algorithm on essentially the same graph: The only difference is that in the first call to Centering, each edge e of G becomes three copies of the edge with flow value $\boldsymbol{f}_e^{(1)}, \boldsymbol{f}_e^{(2)}, \boldsymbol{f}_e^{(3)}$. Note that this edge duplication does not affect planarity.

We note that the IPM algorithm only requires access to $(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}})$, but not $(\boldsymbol{f}, \boldsymbol{s})$ during the main while loop. Hence, $(\boldsymbol{f}, \boldsymbol{s})$ can be implicitly maintained via any data structure. We only require $(\boldsymbol{f}, \boldsymbol{s})$ explicitly when returning the approximately optimal solution at the end of the algorithm Line 27.

Theorem 2.1. *Consider the linear program*

$$\min_{\mathbf{B}^\top\boldsymbol{f}=\boldsymbol{b},\ \boldsymbol{l}\le\boldsymbol{f}\le\boldsymbol{u}} \boldsymbol{c}^\top\boldsymbol{f}$$

[2] Although the min-cost flow problem can be written as a one-sided linear program, it is more convenience for the linear program solver to have both sides. Everything in this section works for general linear programs and hence we will not use the fact $m = O(n)$ in this subsection.

Algorithm 1 Robust Interior Point Method from [16]

procedure RIPM($\mathbf{B} \in \mathbb{R}^{m\times n}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u}, \epsilon$)
  Let $L = \|\boldsymbol{c}\|_2$ and $R = \|\boldsymbol{u} - \boldsymbol{l}\|_2$
  Define $\phi_i(x) \stackrel{\text{def}}{=} -\log(\boldsymbol{u}_i - x) - \log(x - \boldsymbol{l}_i)$

▷ Modify the linear program and obtain an initial (x, s) for modified linear program
  Let $t = 2^{21}m^5 \cdot \frac{LR}{128} \cdot \frac{R}{r}$
  Compute $\boldsymbol{f}_c = \arg\min_{\boldsymbol{l}\le \boldsymbol{f}\le \boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{f} + t\phi(\boldsymbol{f})$ and $\boldsymbol{f}_\circ = \arg\min_{\mathbf{B}^\top \boldsymbol{f}=\boldsymbol{b}} \|\boldsymbol{f} - \boldsymbol{f}_c\|_2$
  Let $\boldsymbol{f} = (\boldsymbol{f}_c, 3R + \boldsymbol{f}_\circ - \boldsymbol{f}_c, 3R)$ and $\boldsymbol{s} = (-t\nabla\phi(\boldsymbol{f}_c), \frac{t}{3R+\boldsymbol{f}_\circ - \boldsymbol{f}_c}, \frac{t}{3R})$
  Let the new matrix $\mathbf{B}^{\text{new}} \stackrel{\text{def}}{=} [\mathbf{B}; \mathbf{B}; -\mathbf{B}]$, the new barrier

$$\phi_i^{\text{new}}(x) = \begin{cases} \phi_i(x) & \text{if } i \in [m], \\ -\log x & \text{else.} \end{cases}$$

▷ Find an initial $(\boldsymbol{f}, \boldsymbol{s})$ for the original linear program
  $((\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \boldsymbol{f}^{(3)}), (\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \boldsymbol{s}^{(3)})) \leftarrow$ CENTERING($\mathbf{B}^{\text{new}}, \phi^{\text{new}}, \boldsymbol{f}, \boldsymbol{s}, t, LR$)
  $(\boldsymbol{f}, \boldsymbol{s}) \leftarrow (\boldsymbol{f}^{(1)} + \boldsymbol{f}^{(2)} - \boldsymbol{f}^{(3)}, \boldsymbol{s}^{(1)})$

▷ Optimize the original linear program
  $(\boldsymbol{f}, \boldsymbol{s}) \leftarrow$ CENTERING($\mathbf{B}, \phi, \boldsymbol{f}, \boldsymbol{s}, LR, \frac{\epsilon}{4m}$)
  return $\boldsymbol{f}$
end procedure

procedure CENTERING($\mathbf{B}, \phi, \boldsymbol{f}, \boldsymbol{s}, t_{\text{start}}, t_{\text{end}}$)
  Let $\alpha = \frac{1}{2^{20}\lambda}$ and $\lambda = 64\log(256m^2)$ where m is the number of rows in $\mathbf{B}$
  Let $t \leftarrow t_{\text{start}}$, $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}, \overline{\boldsymbol{s}} \leftarrow \boldsymbol{s}, \overline{t} \leftarrow t$
  Define $\mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}^{1/2}$
  while $t \ge t_{\text{end}}$ **do**
    Set $t \leftarrow \max((1 - \frac{\alpha}{\sqrt{m}})t, t_{\text{end}})$
    Update $h = -\alpha/\|\cosh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}))\|_2$ where γ is defined in Eq. (2.2)
    Update the diagonal weight matrix $\mathbf{W} = \nabla^2\phi(\overline{\boldsymbol{f}})^{-1}$
    Update the direction $\boldsymbol{v}$ where $\boldsymbol{v}_i = \sinh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}})_i)$
    Pick $\boldsymbol{v}^{\|}$ and $\boldsymbol{v}^{\perp}$ such that $\mathbf{W}^{-1/2}\boldsymbol{v}^{\|} \in \text{Range}(\mathbf{B})$, $\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}^{\perp} = \mathbf{0}$ and

$$\|\boldsymbol{v}^{\|} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\|_2 \le \alpha\|\boldsymbol{v}\|_2,$$
$$\|\boldsymbol{v}^{\perp} - (\mathbf{I} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}\|_2 \le \alpha\|\boldsymbol{v}\|_2$$

    Implicitly update $\boldsymbol{f} \leftarrow \boldsymbol{f} + h\mathbf{W}^{1/2}\boldsymbol{v}^{\perp}$, $\boldsymbol{s} \leftarrow \boldsymbol{s} + \overline{t}h\mathbf{W}^{-1/2}\boldsymbol{v}^{\|}$
    Explicitly maintain $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \le \alpha$ and $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \le \alpha$
    Update $\overline{t} \leftarrow t$ if $|\overline{t} - t| \ge \alpha\overline{t}$
  end while
  return $(\boldsymbol{f}, \boldsymbol{s})$
end procedure

with $\mathbf{B} \in \mathbb{R}^{m \times n}$. *We are given a scalar* $r > 0$ *such that* there exists *some interior point* $\boldsymbol{f}_\circ$ *satisfying* $\mathbf{B}^\top \boldsymbol{f}_\circ = \boldsymbol{b}$ *and* $\boldsymbol{l} + r \leq \boldsymbol{f}_\circ \leq \boldsymbol{u} - r$. *Let* $L = \|\boldsymbol{c}\|_2$ *and* $R = \|\boldsymbol{u} - \boldsymbol{l}\|_2$. *For any* $0 < \epsilon \leq 1/2$, *the algorithm* RIPM *finds* $\boldsymbol{f}$ *such that* $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{b}$, $\boldsymbol{l} \leq \boldsymbol{f} \leq \boldsymbol{u}$ *and*

$$\boldsymbol{c}^\top \boldsymbol{f} \leq \min_{\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{b},\, \boldsymbol{l} \leq \boldsymbol{f} \leq \boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{f} + \epsilon L R.$$

Furthermore, the algorithm has the following properties:

- *Each call of* Centering *involves* $O(\sqrt{m}\log(m)\log(\frac{mR}{\epsilon r}))$ *many steps, and* $\bar{t}$ *is only updated* $O(\log(m)\log(\frac{mR}{\epsilon r}))$ *times.*
- *In each step of* Centering, *the coordinate* i *in* $\mathbf{W}, \boldsymbol{v}$ *changes only if* $\overline{\boldsymbol{f}}_i$ *or* $\overline{\boldsymbol{s}}_i$ *changes.*
- *In each step of* Centering, $h\|\boldsymbol{v}\|_2 = O(\frac{1}{\log m})$.
- *Line 19 to Line 21 takes* $O(K)$ *time in total, where* K *is the total number of coordinate changes in* $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$.

Proof. The number of steps follows from Theorem A.1 in [15], with the parameter $w_i = \nu_i = 1$ for all i. The number of coordinate changes in $\mathbf{W}, \boldsymbol{v}$ and the runtime of Line 19 to Line 21 follows directly from the formula of $\mu^t(\boldsymbol{f}, \boldsymbol{s})_i$ and $\gamma^t(\boldsymbol{f}, \boldsymbol{s})_i$. For the bound for $h\|\boldsymbol{v}\|_2$, it follows from

$$h\|\boldsymbol{v}\|_2 \leq \alpha \frac{\|\sinh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}))\|_2}{\|\cosh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}))\|_2} \leq \alpha = O\left(\frac{1}{\log m}\right).$$

□

A key idea in our paper is the efficient computation of projection matrices required for the IPM. Recall from the definition of $\mathbf{P}_{\boldsymbol{w}}$ in Algorithm 1, Line 16, the true projection matrix is

$$\mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top \mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top \mathbf{W}^{1/2}.$$

We let $\mathbf{L}$ denote the weighted Laplacian where $\mathbf{L} = \mathbf{B}^\top \mathbf{W}\mathbf{B}$, so that

$$\mathbf{P}_{\boldsymbol{w}} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\mathbf{B}^\top \mathbf{W}^{1/2}. \tag{2.4}$$

Lemma 2.1. *To implement Line 22 in Algorithm 1, it suffices to find an approximate potential projection matrix* $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ *satisfying* $\left\|\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\right\|_{\text{op}} \leq \alpha$ *and* $\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} \in \text{Range}(\mathbf{B})$; *and a (not necessarily related) approximation flow projection matrix* $\widetilde{\mathbf{P}}^\perp_{\boldsymbol{w}}$ *satisfying* $\left\|\widetilde{\mathbf{P}}^\perp_{\boldsymbol{w}} - (\mathbf{I} - \mathbf{P}_{\boldsymbol{w}})\right\|_{\text{op}} \leq \alpha$ *and* $\mathbf{B}^\top \mathbf{W}^{1/2}\widetilde{\mathbf{P}}^\perp_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{0}$.

Proof. We simply observe that setting $\boldsymbol{v}^\parallel = \widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}$ and $\boldsymbol{v}^\perp = \widetilde{\mathbf{P}}^\perp_{\boldsymbol{w}}\boldsymbol{v}$ suffices. □

In finding these approximate projection matrices, we apply ideas from nested dissection and approximate Schur complements to the matrix $\mathbf{L}$.

2.2 Nested Dissection and Approximate Schur Complements In this subsection, we give the algorithm for maintaining a collection of approximate Schur complements. We first focus on a discussion using the two-layer nested dissection for planar graphs to highlight the key ideas, and then give the extension to the recursive partitioning scheme with $O(\log n)$-layers. Finally we explain how it relates to our goal of finding the approximate projection matrices for Lemma 2.1.

As we will discuss later, our LP formulation for the IPM uses a modified graph which includes two additional vertices and $O(n)$ additional edges to the original planar graph. Although the modified graph is no longer planar, it has only two additional vertices, which we can include in any balanced vertex separators of the original graph to obtain a balanced vertex separator of the modified graph. Hence the separator sizes are the same up to an additive constant, and we can apply the ideas of nested dissection as we would for planar graphs. As such, we assume the input graph G is a planar graph with n vertices for simplicity here.

From the well-known planar separator theorem [44], we know that we can decompose G into two edge-disjoint (but not vertex-disjoint) subgraphs H_1 and H_2 called *regions*, such that each subgraph has at most $2n/3$ vertices. Furthermore, let ∂H_i denote the *boundary* of region H_i, that is, the set of vertices $v \in H_i$ such that v is adjacent to some $u \notin H_i$. Then ∂H_i has size bounded by $O(\sqrt{n})$.

Let $C = H_1 \cup H_2$ denote the union of the boundaries, and let $F = V(G) \setminus C$ denote the remaining *interior* vertices. Note that C is a vertex separator of G, with size

$$|C| \leq |\partial H_1| + |\partial H_2| = O(\sqrt{n}).$$

The vertex subsets F and C give a natural partition of the vertices of G. Using block Cholesky decomposition, we can now write[3]

$$\mathbf{L}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{FF}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C)^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix}, \tag{2.5}$$

where $\mathbf{Sc}(\mathbf{L}, C) \stackrel{\text{def}}{=} \mathbf{L}_{CC} - \mathbf{L}_{CF}\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC}$ is the *Schur complement* of $\mathbf{L}$ onto vertex set C, and $\mathbf{L}_{FC} \in \mathbb{R}^{F \times C}$ is the $F \times C$-indexed submatrix of $\mathbf{L}$.

Algorithm 1 requires updating $\mathbf{L}^{-1}$ in every step; written as the above decomposition, we must in turn update the Schur complement $\mathbf{Sc}(\mathbf{L}, C)$ in every step. Hence, the update cost must be sub-linear in n. Computing $\mathbf{Sc}(\mathbf{L}, C)$ exactly takes $\Omega(|C|^2) = \Omega(n^2)$ time, which is already too expensive. Our key idea here is to maintain a collection of approximate Schur complements, where each individual Schur complement is of a smaller size based on the graph decomposition, and we can maintain all the Schur complements in amortized $\sqrt{n}$ time per IPM step.

We illustrate the intuition using a two-layer scheme: Let $\mathbf{L}[H_i]$ denote the Laplacian of the subgraph H_i. Since these regions are edge-disjoint, we can write the Laplacian $\mathbf{L}$ as the sum

$$\mathbf{L} = \mathbf{L}[H_1] + \mathbf{L}[H_2].$$

We show that based on this decomposition, we have

$$\mathbf{Sc}(\mathbf{L}, C) = \mathbf{Sc}(\mathbf{L}[H_1], C) + \mathbf{Sc}(\mathbf{L}[H_2], C).$$

Our data structure maintains a sparse *approximate Schur complement* $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C) \approx \mathbf{Sc}(\mathbf{L}[H_i], C)$ for each region i, which allows us to localize edge weight updates. Namely, if an edge in region i is updated, we only need to recompute one corresponding Schur complement term in the sum. Each term $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)$ can be computed in time nearly-linear to the size of H_i rather than n.

Furthermore, $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)$ is supported only on ∂H_i, which is of size $O(\sqrt{n})$. Hence, any *sparse* approximate Schur complement has only $\widetilde{O}(\sqrt{n})$ edges. Given these matrices, we can define the approximate Schur complement of $\mathbf{L}$ on C by

$$\widetilde{\mathbf{Sc}}(\mathbf{L}, C) \stackrel{\text{def}}{=} \widetilde{\mathbf{Sc}}(\mathbf{L}[H_1], \partial H_1) + \widetilde{\mathbf{Sc}}(\mathbf{L}[H_2], \partial H_2). \tag{2.6}$$

To extend the two-level scheme to more layers, we apply nested dissection recursively to each region using balanced vertex separators, until the regions are of constant size. The resulting hierarchical structure can be represented by a tree $\mathcal{T}$, which is known as the *separator tree* of G: Formally, each node of $\mathcal{T}$ is a *region* (edge-induced subgraph) H of G; we denote this by $H \in \mathcal{T}$. At a node H, we store subsets of vertices $\partial H, S(H), F_H \subseteq V(H)$, where ∂H is the set of *boundary vertices* that are incident to vertices outside H in G; $S(H)$ is the balanced vertex separator of H; and F_H is the set of *eliminated vertices* at H. Concretely, the nodes and associated vertex sets are defined recursively in a top-down way as follows:

1. The root of $\mathcal{T}$ is the node $H = G$, with $\partial H = \emptyset$ and $F_H = S(H)$.
2. A non-leaf node $H \in \mathcal{T}$ has exactly two children $D_1, D_2 \in \mathcal{T}$ that form an edge-disjoint partition of H, and their vertex sets intersect on the balanced separator $S(H)$ of H. Define $\partial D_1 = (\partial H \cup S(H)) \cap V(D_1)$, and similarly $\partial D_2 = (\partial H \cup S(H)) \cap V(D_2)$. Moreover, $F_H = S(H) \setminus \partial H$.

[3] To keep notation simple, $\mathbf{M}^{-1}$ will denote the Moore-Penrose pseudo-inverse for non-invertible matrices.

3. If a region H contains a single edge, then we stop the recursion and H becomes a leaf node. Further, we define $S(H) = \emptyset$ and $F_H = V(H) \setminus \partial H$. Note that by construction, each edge of G is contained in a unique leaf node.

LEMMA 2.2. *Using the above construction, we have that $\{F_H : H \in \mathcal{T}\}$ partition the vertex set $V(G)$.*

The *level* $\eta(H)$ of a node H is the number of edges in the path between H and the root. We call the maximum distance between a leaf node and the root the *height* of $\mathcal{T}$ and we denote it by η.

THEOREM 2.2. (SEPARATOR TREE CONSTRUCTION [18]) *Given a* modified planar graph *G, there is an algorithm that computes a separator tree $\mathcal{T}$ of G of height $\eta = O(\log n)$ in $O(n \log n)$ time.*

Our data structure involving the maintenance of approximate Schur complements crucially relies on bounding the number of affected nodes in the separator tree and their boundary when a subset of edges in the graph undergo weight changes. Concretely, for a node H in $\mathcal{T}$, let $\mathcal{P}_{\mathcal{T}}(H)$ be the set nodes on the path from H to the root of $\mathcal{T}$ including H. Given a set of K nodes $\mathcal{H} = \{H \mid H \in \mathcal{T}\}$, we define

$$\mathcal{P}_{\mathcal{T}}(\mathcal{H}) := \bigcup_{H \in \mathcal{H}} \mathcal{P}_{\mathcal{T}}(H)$$

to be the set of ancestor nodes of $\mathcal{H}$. Furthermore, we partition these nodes by their level in $\mathcal{T}$, and use $\mathcal{P}_{\mathcal{T}}(\mathcal{H}, i)$ to denote all the nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ at level i in $\mathcal{T}$, where by convention the root is at level η. Fakcharoenphol and Rao [18, Section 3.5] showed that the total number of boundary vertices from the nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ is $O(\sqrt{mK})$. Here, we use a slightly weaker bound that in addition requires bounding the number of separator vertices.

LEMMA 2.3. *Consider a planar graph G and its separator tree $\mathcal{T}$, a set $\mathcal{H}$ of K nodes in $\mathcal{T}$, and the union of node-to-root paths $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ for the nodes in $\mathcal{H}$. We have that*

$$\sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})} |\partial(H)| + |S(H)| \leq \widetilde{O}(\sqrt{mK}).$$

For a height-η separator tree, we generalize the set C from Eq. (2.5) to a sequence of sets $V(G) = C_{-1} \supset C_0 \supset \cdots \supset C_{\eta-1} \supset C_\eta = \emptyset$, and generalize F to the sets $F_0, \ldots, F_\eta$ partitioning $V(G)$, where $F_i \stackrel{\text{def}}{=} C_{i-1} \setminus C_i$. Concretely, Lemma 2.2 allows us to define F_i to be the union of F_H over all nodes H at level i in the separator tree, and $C_i = \cup_{j>i} F_j$.

Now, the decomposition from Eq. (2.5) can be extended as follows:

$$\mathbf{L}^{-1} = \mathbf{U}^{(0)\top} \cdots \mathbf{U}^{(\eta-1)\top} \begin{bmatrix} (\mathbf{Sc}(\mathbf{L}, C_{-1})_{F_0,F_0})^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & (\mathbf{Sc}(\mathbf{L}, C_{\eta-1})_{F_\eta,F_\eta})^{-1} \end{bmatrix} \mathbf{U}^{(\eta-1)} \cdots \mathbf{U}^{(0)}, \tag{2.7}$$

where the $\mathbf{U}^{(i)}$'s are upper triangular matrices with $\mathbf{U}^{(i)} \stackrel{\text{def}}{=} \mathbf{I} - \mathbf{Sc}(\mathbf{L}, C_{i-1})_{C_i,F_i} \left(\mathbf{Sc}(\mathbf{L}, C_{i-1})_{F_i,F_i}\right)^{-1}$, where we assume all matrices are $n \times n$ by padding zeroes when required. To efficiently compute parts of $\mathbf{L}^{-1}$, we use approximate Schur complements instead of exact ones everywhere in Eq. (2.7).

DEFINITION 2.1. (APPROXIMATE SCHUR COMPLEMENT) *Let G be a weighted graph with Laplacian $\mathbf{L}$, and let C be a set of boundary vertices in G. We say that a Laplacian matrix $\widetilde{\mathbf{Sc}}(\mathbf{L}, C) \in \mathbb{R}^{C \times C}$ is an* δ-approximate Schur complement *of $\mathbf{L}$ onto C if $\widetilde{\mathbf{Sc}}(\mathbf{L}, C) \approx_\delta \mathbf{Sc}(\mathbf{L}, C)$, where we use $\approx_\delta$ to mean an e^δ-spectral approximation.*

We use the following result as a black-box for computing sparse approximate Schur complements:

LEMMA 2.4. (APPROXSCHUR PROCEDURE [17]) *Let $\mathbf{L}$ be the weighted Laplacian of a graph with n vertices and m edges, and let C be a subset of boundary vertices of the graph. Let $\gamma = 1/n^3$ be the probability parameter. Given error parameter $\delta \in (0, 1/2)$, there is an algorithm* APPROXSCHUR$(\mathbf{L}, C, \delta)$ *that computes and outputs a δ-approximate Schur complement $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)$ that satisfies the following properties with probability at least $1 - \gamma$:*

Algorithm 2 Data structure to maintain Schur complements

private: member
  Graph G with incidence matrix $\mathbf{B}$
  Separator tree $\mathcal{T}$ of height η, and vertex sets F_H and ∂H at every node H in $\mathcal{T}$
  Weight vector $\boldsymbol{w}$ and diagonal matrix $\mathbf{W}$ used interchangeably
  Schur complement approximation factor δ
  A Laplacian $\mathbf{L}^{(H)}$ supported on $F_H \cup \partial H$ at every node H in $\mathcal{T}$

procedure INITIALIZE(G, $\boldsymbol{w} \in \mathbb{R}^m$, $\varepsilon_{\mathbf{P}} > 0$)
  $\mathbf{B} \leftarrow$ incidence matrix of G
  $\mathcal{T} \leftarrow$ separator tree of G of height η as in Theorem 2.2
  $\delta \leftarrow \varepsilon_{\mathbf{P}}/\eta$
  $\boldsymbol{w} \leftarrow \boldsymbol{w}$
  for $i = 0, \ldots, \eta$ **do**
    for each node H at level i in $\mathcal{T}$ **do**
      APPROXSCHURNODE(H)
    end for
  end for
end procedure

procedure REWEIGHT($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m$)
  $\mathcal{H} \leftarrow$ leaf nodes in $\mathcal{T}$ that contain all the edges in G whose weight has changed
  $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})}$
  $\mathcal{P}_{\mathcal{T}}(\mathcal{H}) \leftarrow$ set of all ancestor nodes of $\mathcal{H}$ in $\mathcal{T}$
  for $i = 0, \ldots, \eta$ **do**
    for each node H at level i in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ **do**
      APPROXSCHURNODE(H)
    end for
  end for
end procedure

procedure APPROXSCHURNODE($H \in \mathcal{T}$)
  if H is a leaf node **then**
    $\triangleright$ $\mathbf{B}[H]$ is the incidence matrix for the induced subgraph H with edge set $E(H)$
    $\mathbf{L}^{(H)} \leftarrow (\mathbf{B}[H])^\top \mathbf{W}_{E(H)} \mathbf{B}[H]$
  else
    Let D_1, D_2 be the children of H
    $\mathbf{L}^{(H)} \leftarrow$ APPROXSCHUR($\mathbf{L}^{(D_1)}, \partial D_1, \delta$) + APPROXSCHUR($\mathbf{L}^{(D_2)}, \partial D_2, \delta$) (Lemma 2.4)
  end if
end procedure

1. *The graph corresponding to* $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)$ *has* $O(\delta^{-2}|C|\log(n/\gamma))$ *edges.*
2. *The total running time is* $O(m\log^3(n/\gamma) + \delta^{-2}n\log^4(n/\gamma))$.

We maintain a collection of δ-approximate Schur complements in our data structures carefully, crucially making use of the separator tree and the transitive property of Schur complements, so that altogether, we maintain an approximation of the block-diagonal matrix in Eq. (2.7) as edge weights undergo updates throughout the IPM. Specifically, we show:

THEOREM 2.3. (MAINTENANCE OF SCHUR COMPLEMENTS) *Given a modified planar graph* G *with* m *edges and its separator tree* $\mathcal{T}$*, there exists a deterministic data structure that maintains the edge weights* $\boldsymbol{w}$ *from the RIPM and a collection of Schur complements, and supports the following procedures:*

- INITIALIZE($G, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \varepsilon_{\mathbf{P}} > 0$)*: Given a graph* G*, initial weights* $\boldsymbol{w}$*, projection matrix approximation accuracy* $\varepsilon_{\mathbf{P}}$*, preprocess in* $\widetilde{O}(m)$ *time.*
- REWEIGHT($\boldsymbol{w} \in \mathbb{R}^m_{>0}$ *given implicitly as a set of changed coordinates): Set the current weight to* $\boldsymbol{w}$ *and updates the relevant Schur complements in* $\widetilde{O}(\delta^{-2}\sqrt{mK})$ *time, where* K *is the number of coordinates changed in* $\boldsymbol{w}$*.*
- *Access to a Laplacian* $\mathbf{L}^{(H)}$ *at every node* H *of* $\mathcal{T}$ *in time nearly-linear in* $|\partial H \cup F_H|$*.*

Furthermore, the data structure maintains the $\mathbf{L}^{(H)}$*'s so that for each level* $i \geq 0$ *of* $\mathcal{T}$*,*

$$\mathbf{L}^{(i)} \stackrel{\text{def}}{=} \sum_{H \text{ at level } i} \mathbf{L}^{(H)} \approx_{i\delta} \mathbf{Sc}(\mathbf{L}, C_{i-1}), \tag{2.8}$$

where we use $\approx_\delta$ *to mean an* e^δ*-spectral approximation, and we assume all* $\mathbf{L}^{(H)}$*'s are of the same dimension by padding zeros.*

Let us now define

$$\widetilde{\mathbf{\Gamma}} \stackrel{\text{def}}{=} \begin{bmatrix} (\mathbf{L}^{(0)}_{F_0,F_0})^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & (\mathbf{L}^{(\eta)}_{F_\eta,F_\eta})^{-1} \end{bmatrix} \tag{2.9}$$

to be the block-diagonal matrix of Schur complements in Eq. (2.7), except we replace the exact Schur complements with approximate ones maintained by the data structure given by Eq. (2.8). Similarly, let $\mathbf{\Pi}^{(i)} \stackrel{\text{def}}{=} \mathbf{I} - \mathbf{L}^{(i)}_{C_i,F_i}\left(\mathbf{L}^{(i)}_{F_i,F_i}\right)^{-1}$ be the corresponding approximation of $\mathbf{U}^{(i)} = \mathbf{I} - \mathbf{Sc}(\mathbf{L}, C_{i-1})_{C_i,F_i}(\mathbf{Sc}(\mathbf{L}, C_{i-1})_{F_i,F_i})^{-1}$.

With the data structure for maintaining the Schur complements, we can therefore maintain $\widetilde{\mathbf{\Gamma}}$ and the associated $\mathbf{\Pi}^{(i)}$'s. Finally, we arrive at the approximate projection matrix

$$\widetilde{\mathbf{P}}_{\boldsymbol{w}} = \mathbf{W}^{1/2}\mathbf{B}^\top\mathbf{\Pi}^{(0)\top}\cdots\mathbf{\Pi}^{(\eta-1)\top}\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2} \approx_{\eta\delta} \mathbf{P}_{\boldsymbol{w}}. \tag{2.10}$$

This is in fact what we want as the approximate slack projection matrix, and also a crucial component of the flow projection matrix.

2.3 Maintaining Vector Approximation In the flow and slack maintenance data structures, one key operation is to maintain vectors $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ that are close to $\boldsymbol{f}, \boldsymbol{s}$ throughout the RIPM. In this section, we will give a meta data structure that solves this in a more general setting. The data structure involves three steps; the first two are similar to [16] and our key contribution is the last step:

1. We maintain an approximate vector by detecting coordinates with large changes. In step k, for every ℓ such that $2^\ell|k$, we consider all coordinates of the approximate vector that did not change in the last 2^ℓ steps. If any of them is off by more than $\frac{\delta}{2\lceil\log m\rceil}$ from the true vector, it is updated. We can prove that each coordinate of the approximate vector has additive error at most δ. The number of updates to the approximate vector will be roughly $O(2^{2\ell_k})$ where 2^{ℓ_k} is the largest power of 2 that divides k. This guarantees that K-sparse updates only happen $\sqrt{m/K}$ times.

2. We detect coordinates with large changes via a random sketch. We can sample an edge with probability proportional to its amount of change, given access to sum of the probabilities of edges in any region of the decomposition tree.

3. We show how to maintain random sketches for a large family of operators, the *forest operators*. To maintain the random sketch of $\mathbf{M}\boldsymbol{z}$ for some $\mathbf{M}$ with forest structure, i.e. when $\mathbf{M}$ is the concatenation of operators on tree paths, we store intermediate results for every subtrees at their roots. To modify the operator on some edge or operand on some coordinates, we only need to update nodes on several tree paths. We will apply this data structure to the separator tree where the cost of updating a node H will be proportional to its separator size, so that a K-sparse update costs roughly $\widetilde{O}(\sqrt{mK})$.

We begin with the formal definition of *forest operator*. Its structure is motivated by the separator tree from the previous section.

DEFINITION 2.2. (FOREST OPERATOR) *Given a rooted forest $\mathcal{T}$ where each vertex v is associated with a disjoint set S_v and each upward*[4] *edge v_1v_2 is associated with a matrix $\mathbf{M}_{v_1v_2} \in \mathbb{R}^{S_{v_1} \times S_{v_2}}$. For each upward path $p = (u_1, u_2, \cdots, u_z)$, we define*

$$\mathbf{M}_p = \mathbf{M}_{u_1u_2}\mathbf{M}_{u_2u_3}\cdots\mathbf{M}_{u_{z-1}u_z}.$$

Let $L = \bigcup_{leaf\ l} S_l$ and $R = \bigcup_{root\ r} S_r$. We define the forest operator $\overline{\mathbf{M}} \in \mathbb{R}^{L \times R}$ by

$$(\overline{\mathbf{M}})_{S_l \times S_r} = \mathbf{M}_{l \to r} \text{ for all } l \in L, r \in R$$

where $l \to r$ denotes the downward path from root r to leaf l. If there is no path from l to r, we set $\mathbf{M}_{l\to r} \in \mathbb{R}^{S_l \times S_r}$ to be the zero matrix.

The complexity of a forest operator can be parameterized as follows:

DEFINITION 2.3. (COMPLEXITY OF FOREST OPERATOR) *For any $k \le m$, we define $T(k)$ be the maximum cost of computing $\boldsymbol{u}^\top \mathbf{M}_e + \mathbf{M}_e \boldsymbol{v}$ for k different $\boldsymbol{u}, \boldsymbol{v}$ and k* distinct *edges e.*

For the operator induced by the flow and slack variables, we have $T(k) = \widetilde{O}(\sqrt{mk})$ where m is the number of edges in the input graph.

Now, we are ready to state the main result:

THEOREM 2.4. (VECTOR MAINTENANCE FOR FOREST OPERATOR) *Given a constant degree forest $\mathcal{T}$ with height η, a forest operator $\overline{\mathbf{M}}$ on $\mathcal{T}$ with complexity T. There exists a randomized data structure (Algorithms 3 to 5) that supports the following procedures against adversarial inputs:*

- INITIALIZE($\overline{\mathbf{M}}, \boldsymbol{z}_1, \boldsymbol{z}_2, c_1, c_2, \rho > 0, \delta > 0, \beta > 0$)*: Initialize the data structure with initial vector $\boldsymbol{v} = \overline{\mathbf{M}}(c_1\boldsymbol{z}_1 + c_2\boldsymbol{z}_2)$, target step accuracy δ, success probability $1 - \rho$, speed limit β in $O(m\eta^2 \log m \log(\frac{m}{\rho}))$ time.*
- APPROXIMATE($\overline{\mathbf{M}}, \boldsymbol{z}_1, \boldsymbol{z}_2, c_1, c_2, \mathbf{D}$)*: Output a vector $\overline{\boldsymbol{v}}$ such that $\|\mathbf{D}^{1/2}(\boldsymbol{v} - \overline{\boldsymbol{v}})\|_\infty \le \delta$ for the current vector $\boldsymbol{v} = \overline{\mathbf{M}}(c_1\boldsymbol{z}_1 + c_2\boldsymbol{z}_2)$ and the current diagonal scaling $\mathbf{D}$. We assume $\boldsymbol{z}_1, \boldsymbol{z}_2, \mathbf{D}$ are given implicitly as a set of changed coordinates.*
- EXACT()*: Output the current vector $\boldsymbol{v} = \overline{\mathbf{M}}(c_1\boldsymbol{z}_1 + c_2\boldsymbol{z}_2)$ in $O(\eta^2 \log(\frac{m}{\rho}) \cdot T((\eta + \log m) \cdot m))$ time.*

Suppose $\|\boldsymbol{v}^{(k+1)} - \boldsymbol{v}^{(k)}\|_{\mathbf{D}^{(k+1)}} \le \beta$ for all k, where $\mathbf{D}^{(k)}$ and $\boldsymbol{v}^{(k)}$ are the $\mathbf{D}$ and $\boldsymbol{v}$ at the k-th step. We use k-th step to denote the state of the data structure after the k-th APPROXIMATE*. 0-th step means the state after* INITIALIZE*. At the k-th step, we have the following:*

- *Excluding the coordinates i where $\mathbf{D}^{(k)}_{ii} \ne \mathbf{D}^{(k-1)}_{ii}$, there are*

$$N_k \stackrel{\text{def}}{=} O(2^{2\ell_k}(\beta/\delta)^2 \log^2 m)$$

coordinates in $\overline{\boldsymbol{v}}$ that are changed, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$.

[4]The edge v_1v_2 is upward if v_2 is closer to a root than v_1.

- *The cost of the k-th* Approximate *call is*

$$\Theta(\eta^2 \log(\frac{m}{\rho}) \log m) \cdot T(O(\eta + \log m) \cdot (N_k + \Delta))$$

where $\Delta = |\{e : \mathbf{M}_e \neq \mathbf{M}_e^{(\text{last})}\}| + |\{\text{root } r : \boldsymbol{z}_{d,S_r} \neq \boldsymbol{z}_{d,S_r}^{(\text{last})} \text{ for some } d = 1 \text{ or } 2\}| + |\{i : \mathbf{D}_{ii} \neq \mathbf{D}_{ii}^{(\text{last})}\}|$.

Algorithm 3 Data structure MaintainVector

private : member
  Accuracy target $\delta > 0$
  Current step k and approximate vector $\overline{\boldsymbol{v}} \in \mathbb{R}^m$
  Previous inputs $\{\boldsymbol{v}^{(j)} \in \mathbb{R}^m\}_{j=0}^k$ and diagonal matrices $\{\mathbf{D}^{(j)} \in \mathbb{R}^{m \times m}\}_{j=1}^k$

procedure Initialize($\boldsymbol{v} \in \mathbb{R}^m, \delta > 0$)
  $\overline{\boldsymbol{v}} \leftarrow \boldsymbol{v}$, $\delta \leftarrow \delta$, $k \leftarrow 0$
end procedure

procedure Approximate($\boldsymbol{v} \in \mathbb{R}^m, \mathbf{D} \in \mathbb{R}_{>0}^{m \times m}$)
  $k \leftarrow k + 1$, $\boldsymbol{v}^{(k)} \leftarrow \boldsymbol{v}$, $\mathbf{D}^{(k)} \leftarrow \mathbf{D}$.
  if $k \geq 2$ **then**
    $\overline{\boldsymbol{v}}_i \leftarrow \boldsymbol{v}_i^{(k)}$ for all i such that $\mathbf{D}_{ii}^{(k)} \neq \mathbf{D}_{ii}^{(k-1)}$ and $\overline{\boldsymbol{v}}_i \neq \boldsymbol{v}_i^{(k-1)}$.
  end if
  for $\ell = 0, 1, \cdots, \lceil \log m \rceil$ **do**
    if $k = 0 \mod 2^\ell$ and $k \geq 2^\ell$ **then**
      $I_\ell^{(k)} \stackrel{\text{def}}{=} \{i \in [m] : (\mathbf{D}_{ii}^{(k)})^{1/2} \cdot |\boldsymbol{v}_i^{(k)} - \boldsymbol{v}_i^{(k-2^\ell)}| \geq \frac{\delta}{2\lceil \log m \rceil}$ ▷ Implemented in Algorithm 4
        and $\bar{\boldsymbol{v}}_i$ has not been updated in Line 13 or Line 24 since the $(k - 2^\ell)$-th step$\}$
      $I \leftarrow I \cup I_\ell^{(k)}$
      if $\ell = \lceil \log m \rceil$ **then** $I \leftarrow [m]$
      end if
    end if
  end for
  $\overline{\boldsymbol{v}}_i \leftarrow \boldsymbol{v}_i^{(k)}$ for all $i \in I$
  return $\overline{\boldsymbol{v}}$
end procedure

procedure Exact()
  Let $\{\boldsymbol{u}_H\}_{H \in \text{ forest } \mathcal{T}}$ be temporary variables
  Let $\boldsymbol{u}_R = (c_1 \boldsymbol{z}_1 + c_2 \boldsymbol{z}_2)|_{S_R}$ for each root R in the forest
  for each node $H \in \mathcal{T}$ by height in decreasing order **do**
    Let P be the parent of H
    $\boldsymbol{u}_H \leftarrow \mathbf{M}_{H,P} \boldsymbol{u}_P$
  end for
  return $\{\boldsymbol{u}_H\}_{\text{leaf } H}$
end procedure

2.4 Maintaining Flow and Slack In this section, we outline the data structures for maintaining the flow and slack solutions $\boldsymbol{f}, \boldsymbol{s}$ as needed in Algorithm 1, Line 23. To this end, we make crucial use of ideas from nested dissection and approximate Schur complements.

As we will discuss later, our LP formulation for the IPM uses a modified graph, which includes two additional vertices and $O(n)$ additional edges to the original graph. Therefore, the following data structures work with these *modified planar graph*.

Algorithm 4 Implementation of Line 17 in Algorithm 3

Let $\overline{\mathbf{D}}$ be the diagonal matrix such that

$$\overline{\mathbf{D}}_{ii} = \begin{cases} \mathbf{D}_{ii}^{(k)} & \text{if } \overline{\boldsymbol{v}}_i^{(j)} = \overline{\boldsymbol{v}}_i \text{ for } j \geq k - 2^\ell \\ 0 & \text{otherwise} \end{cases}.$$

Let $\boldsymbol{q} = \overline{\mathbf{D}}^{1/2}\boldsymbol{v}^{(k)} - \overline{\mathbf{D}}^{1/2}\boldsymbol{v}^{(k-2^\ell)}$.
Let $\mathcal{T}$ be a height-η tree with n leaves corresponding to the coordinates of $\boldsymbol{q}$.
Let the number of samples $N = \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho))$ where ρ is the failure probability.
Let the sketch dimension $w = \Theta(\eta^2 \log(\frac{m}{\rho}))$.
Let $\boldsymbol{\Phi} \in \mathbb{R}^{w \times m}$ be a random matrix with each entries samples independently from $N(0, \frac{1}{w})$.
Let I be the set of candidate coordinates.
For any node $u \in \mathcal{T}$, let $\mathcal{D}(u)$ be the set of leaf nodes in the subtree under u, and let $\mathbf{I}_{\mathcal{D}(u)}$ be the diagonal matrix with 1 on the entries indexed by $\mathcal{D}(u)$ and zero otherwise.
for $j = 1, \cdots, N$ **do**
    ▷ Sample a coordinate u with probability proportionally to $\boldsymbol{q}_i^2$.
    while TRUE **do**
        $u \leftarrow \text{root}(\mathcal{T})$, $p_u \leftarrow 1$.
        while u is not a leaf node **do**
            Sample a child u' of u with probability

$$\mathbf{P}(u \to u') \stackrel{\text{def}}{=} \frac{\|\boldsymbol{\Phi}\mathbf{I}_{\mathcal{D}(u')}\boldsymbol{q}\|_2^2}{\sum_{u'' \text{ is a child of } u} \|\boldsymbol{\Phi}\mathbf{I}_{\mathcal{D}(u'')}\boldsymbol{q}\|_2^2}$$

            $p_u \leftarrow p_u \cdot \mathbf{P}(u \to u')$
            $u \leftarrow u'$
        end while
        With probability $p_{\text{accept}} \stackrel{\text{def}}{=} \boldsymbol{q}_u^2/(2 \cdot p_u \cdot \|\Phi\boldsymbol{q}\|_2^2)$, **break**
    end while
    $I \leftarrow I \cup \{u\}$ ▷ u corresponds to a coordinate of $\boldsymbol{q}$
end for
return $\{i \in I \text{ such that } |\boldsymbol{q}_i| \geq \frac{\delta}{2\lceil \log m \rceil}\}$.

Algorithm 5 Data Structure for Sketch Maintenance

private : member
  Forest Operator $\{\mathbf{M}_e\}_{\text{edge } e \text{ in } \mathcal{T}}$.
  Rooted tree $\overline{\mathcal{T}}$.
  Sketch matrix $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$.
  Current vector $\boldsymbol{z} \in \mathbb{R}^R$.
  Partial Sketch $\{\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_u}\}_{\text{vertex } u \text{ in } \mathcal{T}}$ where $\widehat{\mathbf{M}}_u = \sum_{\text{leaf } l} \mathbf{M}_{l \to u}$ and $\boxed{\cdots}$ denotes the memoization of $\cdots$
  Sketched value $\{\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_r\boldsymbol{z}}\}_{\text{root } r \text{ in } \mathcal{T} \text{ or vertex } r \in \overline{\mathcal{T}} \setminus \mathcal{T}}$.

procedure INITIALIZE(a forest $\mathcal{T}$, $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$)
  Set $\mathbf{M}_e, \boldsymbol{z}, \boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_u}, \boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_r\boldsymbol{z}}$ to 0 with correct sizes.
  Set $\overline{\mathcal{T}}$ to a rooted tree obtained by connecting the roots of forest $\mathcal{T}$ via a binary tree with height $O(\log m)$.
  Set $\mathbf{\Phi} \leftarrow \mathbf{\Phi}$.
end procedure

procedure UPDATE($\{\mathbf{M}_e^{(\text{new})}\}_{e \in \delta_M}, \{\boldsymbol{z}_i^{(\text{new})}\}_{i \in \delta_z}$)
  Update $\mathbf{M}_e$ to $\mathbf{M}_e^{(\text{new})}$ for $e \in \delta_M$, $\boldsymbol{z}_i$ to $\boldsymbol{z}_i^{(\text{new})}$ for $i \in \delta_z$.
  Let the affected vertices $V_{\text{aff}} = \left(\bigcup_{(u,v) \in \delta_M} \{v \text{ and its ancestors in } \overline{\mathcal{T}}\}\right) \cup \{\delta_z \text{ and its ancestors in } \overline{\mathcal{T}}\}$.
  for $v \in V_{\text{aff}}$ sorted by height of v in increasing order **do**
    If $v \in \mathcal{T}$, **then** set $\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_v} = \sum_{\text{child } u \text{ of } v} \boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_u}\mathbf{M}_{uv}$.
    If v is a root of $\mathcal{T}$, **then** set $\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_v\boldsymbol{z}} = \boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_v} \cdot \boldsymbol{z}$.
    If $v \in \overline{\mathcal{T}} \setminus \mathcal{T}$, **then** set $\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_v\boldsymbol{z}} = \sum_{\text{child } u \text{ of } v} \boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_v\boldsymbol{z}}$.
  end for
end procedure

procedure MEMOIZEDQUERY($i \in \mathcal{T}$)
  If i is a root, **then return** $\boldsymbol{z}_i$.
  if $\boldsymbol{z}_j$ and $\mathbf{M}_{uv}$ for all ancestors (u, v) and j of i have not been changed since the last call of MEMOIZEDQUERY(i) **then**
    return the result of the last MEMOIZEDQUERY(i) .
  else
    Let $p(i)$ be the parent of i.
    Call MEMOIZEDQUERY($p(i)$) and let $\boldsymbol{u}$ be the result. ▷ $\boldsymbol{u} = \mathbf{M}_{p(i) \to r}\boldsymbol{z}$.
    return $\mathbf{M}_{(i,p(i))}\boldsymbol{u}$.
  end if
end procedure

procedure ESTIMATE($u \in \overline{\mathcal{T}}$)
  if u is a root of $\mathcal{T}$ or $u \in \overline{\mathcal{T}} \setminus \mathcal{T}$ **then**
    return $\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_u\boldsymbol{z}}$
  else
    Let $\boldsymbol{u}$ be the result of MEMOIZEDQUERY(u).
    return $\boxed{\mathbf{\Phi}\widehat{\mathbf{M}}_u}\boldsymbol{u}$.
  end if
end procedure

procedure QUERY($i \in [m]$)
  return MEMOIZEDQUERY(i).
end procedure

In all procedures in these data structures, we assume inputs are given by the set of changed coordinates and their values, *compared to the previous input*. Similarly, we output a vector by the set of changed coordinates and their values, compared to the previous output. While the data structures for maintaining flow and slack are randomized, they are guaranteed to work against an adaptive adversary that is allowed to see the entire internal state of the data structure, including the random bits.

Finally, in these data structures where we write $\mathbf{L}^{-1}\boldsymbol{x}$ for some Laplacian $\mathbf{L}$ and vector $\boldsymbol{x}$, we imply the use of an SDD-solver as a black box in nearly-linear time:

THEOREM 2.5. ([54, 30]) *There is a randomized algorithm which is an ε-approximate Laplacian system solver for the any input n-vertex m-edge graph and $\varepsilon \in (0,1)$ and has the following runtime $O(m\text{poly}(\log\log n)\log(1/\varepsilon))$.*

2.4.1 Maintaining the Intermediate Vector $\boldsymbol{z}$ To work with $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ in Eq. (2.10), we will consider it as the composition of two operators applied sequentially. Let $\widetilde{\mathbf{P}}^U \stackrel{\text{def}}{=} \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}$ in Eq. (2.10), so that $\widetilde{\mathbf{P}}_{\boldsymbol{w}} = \mathbf{\Pi}^{(0)\top}\cdots\mathbf{\Pi}^{(\eta-1)\top}\widetilde{\mathbf{P}}^U$. We first show how to maintain the intermediate vector

$$\boldsymbol{z} \stackrel{\text{def}}{=} \widetilde{\mathbf{P}}^U\boldsymbol{v}.$$

Since $\boldsymbol{v}$ is updated at every step of IPM with a set of changed coordinates and their values, we will maintain $\boldsymbol{z}$ as a sum of two terms $\boldsymbol{z} \stackrel{\text{def}}{=} c\cdot\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})}$ to facilitate sparse updates, where $\boldsymbol{z}^{(\text{prev})} = \widetilde{\mathbf{P}}^U\boldsymbol{v}^{(\text{prev})}$ with $\boldsymbol{v}^{(\text{prev})}$ being the $\boldsymbol{v}$ from the previous iteration, and $\boldsymbol{z}^{(\text{sum})}$ is the remaining accumulation of $\boldsymbol{z}$ throughout the algorithm. Specifically, we show:

THEOREM 2.6. *Given a modified planar graph G with m edges, there exists a deterministic data structure that maintains the edge weights $\boldsymbol{w}$ from the RIPM and vector $\boldsymbol{z} \stackrel{\text{def}}{=} c\cdot\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})}$, and supports the following procedures:*

- INITIALIZE($G, \boldsymbol{v}\in\mathbb{R}^m, \boldsymbol{w}\in\mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0$)*: Given a graph G, initial vector $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target projection matrix accuracy $\epsilon_{\mathbf{P}}$, the data structure preprocesses in $\widetilde{O}(m)$ time.*
- REWEIGHT($\boldsymbol{w}\in\mathbb{R}^m_{>0}$ *given implicitly as a set of changed coordinates): Sets the current weight to $\boldsymbol{w}$ and updates the representation of $\boldsymbol{z}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$, compared to the last input to* REWEIGHT *or* MOVE*. Updates $\boldsymbol{z}^{(\text{prev})}|_{F_H}$ and $\boldsymbol{z}^{(\text{sum})}|_{F_H}$ for at most $\widetilde{O}(K)$ nodes H.*
- SPARSEVECTORPROJECT($\boldsymbol{d}\in\mathbb{R}^n$)*: Computes $\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\boldsymbol{d}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of non-zero coordinates in $\boldsymbol{d}$.*
- MOVE($\alpha\in\mathbb{R}$, $\boldsymbol{v}\in\mathbb{R}^n$ *given implicitly as a set of changed coordinates): Maintains $\boldsymbol{z}\leftarrow\boldsymbol{z}+\alpha\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{v}$ compared to the last input to* REWEIGHT *or* MOVE*. Updates $\boldsymbol{z}^{(\text{prev})}|_{F_H}$ and $\boldsymbol{z}^{(\text{sum})}|_{F_H}$ for at most $\widetilde{O}(K)$ nodes H.*

When MOVE($\alpha,\boldsymbol{v}$) is called, we decompose $\boldsymbol{v} = \boldsymbol{v}^{(\text{prev})} + \Delta\boldsymbol{v}$, where $\boldsymbol{v}^{(\text{prev})}$ is the $\boldsymbol{v}$ from the previous IPM step, and $\Delta\boldsymbol{v}$ is the change for the current step. We update $\boldsymbol{z}$ to reflect the change of $\alpha\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}(\boldsymbol{v}^{(\text{prev})}+\Delta\boldsymbol{v})$, and we also update the representation of $\boldsymbol{z} = c\cdot\boldsymbol{z}^{(\text{prev})}+\boldsymbol{z}^{(\text{sum})}$ so that at the beginning of the next IPM, $\boldsymbol{z}^{(\text{prev})} = \widetilde{\mathbf{P}}^U\boldsymbol{v}^{(\text{prev})}$ is maintained.

To accomplish this, we first compute the sparse vector $\Delta\boldsymbol{z} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\Delta\boldsymbol{v} = \text{SPARSEVECTORPROJECT}(\mathbf{B}^\top\mathbf{W}^{1/2}\Delta\boldsymbol{v})$. Then, we set $c\leftarrow c+\alpha, \boldsymbol{z}^{(\text{prev})}\leftarrow\boldsymbol{z}^{(\text{prev})}+\Delta\boldsymbol{z}, \boldsymbol{z}^{(\text{sum})}\leftarrow\boldsymbol{z}^{(\text{sum})}-c\cdot\Delta\boldsymbol{z}$. It can be easily checked that $\boldsymbol{z}^{(\text{new})} = \boldsymbol{z}+\alpha\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$; that is, we perform the correct update and maintain the correct representation.

Let us discuss the run-time of SPARSEVECTORPROJECT and use the two-layer nested dissection setup for intuition. Let $\boldsymbol{d} \stackrel{\text{def}}{=} \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$, then we have

$$\boldsymbol{z} \stackrel{\text{def}}{=} \begin{bmatrix} \mathbf{L}_{FF}^{-1} & \mathbf{0} \\ \mathbf{0} & \widetilde{\mathbf{Sc}}(\mathbf{L},C)^{-1} \end{bmatrix}\begin{bmatrix} \mathbf{I} & 0 \\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix}\boldsymbol{d}.$$

The only difficult part for the next left matrix multiplication is $-\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1}$. However, we note that $\mathbf{L}_{FF}$ is block-diagonal with two blocks, each corresponding to a region generated during nested dissection. Hence, we can solve the Laplacians on the two subgraphs separately. Next, we note that the two terms of $\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1}\boldsymbol{d}$ are both fed into $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)^{-1}$, and we solve this Laplacian in time linear in the size of $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)$. The rest of the terms are not the bottleneck in the overall run-time. In the more general nested-dissection setting with $O(\log n)$ layers, we solve a sequence of Laplacians corresponding to the regions given by paths in the separator tree. We can bound the run-time of these Laplacian solves by the size of the corresponding regions for the desired overall run-time.

We use this partial computation $\boldsymbol{z}$ in the data structures for maintaining $\boldsymbol{f}$ and $\boldsymbol{s}$ below.

Algorithm 6 Data structure to maintain the intermediate vector $\boldsymbol{z}$, Part 1

private: member
  Approximate Schur complements Data Structure `ApxSc` (Theorem 2.3)
  Vectors $\boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})}$
  Coefficient c
  Vector $\boldsymbol{v}^{(\text{prev})}$
  Diagonal weight matrix $\mathbf{W}$ (and vector $\boldsymbol{w}$ used interchangeably)

procedure InitializeSparseVectorProject($G, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \varepsilon_{\mathbf{P}} > 0$)
  `ApxSc`.Initialize($G, \boldsymbol{w}, \varepsilon_{\mathbf{P}}$)
  $\mathbf{W} \leftarrow \text{diag}(\boldsymbol{w})$
  $\boldsymbol{v}^{(\text{prev})} \leftarrow \boldsymbol{v}$
  $\boldsymbol{z}^{(\text{prev})} \leftarrow$ SparseVectorProject($\mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}$)
  $\boldsymbol{z}^{(\text{sum})} \leftarrow \mathbf{0}$
  $c \leftarrow 1$
end procedure

procedure SparseVectorProject($\boldsymbol{d} \in \mathbb{R}^m$)
  $\mathcal{H} \leftarrow$ set of leaf nodes in $\mathcal{T}$ containing any edge with a non-zero coordinate in $\boldsymbol{d}$
  $\mathcal{P}_{\mathcal{T}}(\mathcal{H}) \leftarrow$ set of all ancestor nodes of $\mathcal{H}$ in $\mathcal{T}$
▷ $\boldsymbol{z}$ is a sparse vector throughout this procedure
  $\boldsymbol{z} \leftarrow \boldsymbol{d}$
  for $i = 0, \ldots, \eta$ **do**
    for each node H at level i in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ **do**
      $\boldsymbol{z}|_{F_H} \leftarrow (\mathbf{L}^{(H)}_{F_H, F_H})^{-1} \boldsymbol{z}|_{F_H}$
      $\boldsymbol{z}|_{\partial H} \leftarrow \boldsymbol{z}|_{\partial H} - \mathbf{L}^{(H)}_{\partial H, F_H} \cdot \boldsymbol{z}|_{F_H}$
    end for
  end for
  return $\boldsymbol{z}$
end procedure

procedure Move($\alpha \in \mathbb{R}, \boldsymbol{v}^{(\text{new})} \in \mathbb{R}^m$)
  $\Delta \boldsymbol{v} \leftarrow \boldsymbol{v}^{(\text{new})} - \boldsymbol{v}^{(\text{prev})}$
  $\boldsymbol{v}^{(\text{prev})} \leftarrow \boldsymbol{v}^{(\text{new})}$
  $\Delta \boldsymbol{z} \leftarrow$ SparseVectorProject($\mathbf{B}^\top \mathbf{W}^{1/2} \Delta \boldsymbol{v}$)
  $c \leftarrow c + \alpha$
  $\boldsymbol{z}^{(\text{prev})} \leftarrow \boldsymbol{z}^{(\text{prev})} + \Delta \boldsymbol{z}$
  $\boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c \cdot \Delta \boldsymbol{z}$
end procedure

2.4.2 Maintaining Slack Now, we discuss how to maintain the slack solution under the update $\boldsymbol{s} \leftarrow \boldsymbol{s} + \bar{t} h \mathbf{W}^{-1/2} \widetilde{\mathbf{P}}_{\boldsymbol{w}} \boldsymbol{v}$ as needed in Algorithm 1, Line 23. This is accomplished by a slack maintenance data structure

Algorithm 7 Data structure to maintain the intermediate vector $\boldsymbol{z}$, Part 2

procedure REWEIGHT($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m_{>0}$)
  $\Delta \boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})} - \boldsymbol{w}$ ▷ We use $\mathbf{W}$ and $\boldsymbol{w}$ interchangeably for ease of notation
  $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})}$
  $\mathcal{H} \leftarrow$ set of leaf nodes in $\mathcal{T}$ that contain all the edges of G whose weight has changed
  $\mathcal{P}_{\mathcal{T}}(\mathcal{H}) \leftarrow$ set of all ancestor nodes of $\mathcal{H}$
  $\Delta \boldsymbol{z} \leftarrow \text{SPARSEVECTORPROJECT}(\mathbf{B}^\top(\Delta \mathbf{W})^{1/2}\boldsymbol{v}^{(\text{prev})})$
  $\boldsymbol{z}^{(\text{prev})} \leftarrow \boldsymbol{z}^{(\text{prev})} + \Delta \boldsymbol{z}$
  $\boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c \cdot \Delta \boldsymbol{z}$
  ▷ At this point, $\boldsymbol{z}^{(\text{prev})} = \widetilde{\boldsymbol{\Gamma}}\boldsymbol{\Pi}^{(\eta-1)} \dots \boldsymbol{\Pi}^{(0)}\boldsymbol{d}^{(\text{new})}$, where $\widetilde{\boldsymbol{\Gamma}}$ and the $\boldsymbol{\Pi}^{(i)}$'s are based on the old weights

  for $i = 0, \dots, \eta$ **do**
    for each node H at level i in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ **do**
      $\boldsymbol{z}^{(\text{prev})}|_{F_H} \leftarrow \mathbf{L}^{(H)}_{F_H,F_H} \cdot \boldsymbol{z}^{(\text{prev})}|_{F_H}$
      $\boldsymbol{z}^{(\text{sum})}|_{F_H} \leftarrow \mathbf{L}^{(H)}_{F_H,F_H} \cdot \boldsymbol{z}^{(\text{sum})}|_{F_H}$
    end for
  end for
  ▷ At this point $\boldsymbol{z}^{(\text{prev})} = \boldsymbol{\Pi}^{(\eta-1)} \dots \boldsymbol{\Pi}^{(0)}\boldsymbol{d}^{(\text{new})}$, where the $\boldsymbol{\Pi}^{(i)}$'s are based on the old weights

  $\mathcal{A} \leftarrow \bigcup_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})} F_H$
  $\boldsymbol{y}|_{\mathcal{A}} \leftarrow \boldsymbol{z}^{(\text{prev})}|_{\mathcal{A}}$
  for $i = 0, \dots, \eta$ **do**
    for each node H at level i in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ **do**
      $\boldsymbol{z}^{(\text{prev})}|_{\partial H} \leftarrow \boldsymbol{z}^{(\text{prev})}|_{\partial H} + \mathbf{L}^{(H)}_{\partial H,F_H} \cdot (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{y}|_{F_H}$ ▷ Remove the old computation
    end for
  end for

  ApxSc.REWEIGHT($\boldsymbol{w}^{(\text{new})}$)
  ▷ At this point, all $\mathbf{L}^{(H)}$'s are based on the new weights

  for $i = 0, \dots, \eta$ **do**
    for each node H at level i in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ **do**
      $\boldsymbol{z}^{(\text{prev})}|_{\partial H} \leftarrow \boldsymbol{z}^{(\text{prev})}|_{\partial H} - \mathbf{L}^{(H)}_{\partial H,F_H} \cdot (\mathbf{L}^{(H)}_{F_H,F_H})^{-1} \cdot \boldsymbol{z}^{(\text{prev})}|_{F_H}$ ▷ Add the new computation
      $\boldsymbol{z}^{(\text{sum})}|_{F_H} \leftarrow \boldsymbol{z}^{(\text{sum})}|_{F_H} - c \cdot (\boldsymbol{z}^{(\text{prev})}|_{F_H} - \boldsymbol{y}|_{F_H})$
      $\boldsymbol{z}^{(\text{prev})}|_{F_H} \leftarrow (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{z}^{(\text{prev})}|_{F_H}$
      $\boldsymbol{z}^{(\text{sum})}|_{F_H} \leftarrow (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{z}^{(\text{sum})}|_{F_H}$
    end for
  end for
end procedure

(Algorithm 8) that simultaneously maintains the approximate slack $\overline{\boldsymbol{s}}$ and the edge weights $\boldsymbol{w}$ from the IPM. A single IPM step calls the procedures Reweight, Move, Approximate in this order once.

Theorem 2.7. (Slack Maintenance) *Given a modified planar graph G with m edges, there exists a randomized data structure that implicitly maintains the flow solution $\boldsymbol{s}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{s}}$ as well as the edge weights $\boldsymbol{w}$, and supports the following procedures with high probability in m against an adaptive adversary:*

- Initialize($G, \boldsymbol{s} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0$)*: Given a graph G, initial vector $\boldsymbol{s}$, initial weight $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$ and target output accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m)$ time.*
- Reweight($\boldsymbol{w} \in \mathbb{R}^m_{>0}$, *given implicitly as a set of changed weights): Set the current weight to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time where K is the number of coordinates changed in $\boldsymbol{w}$ (compared to the last input to* Reweight*).*
- Move($\alpha \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ *given implicitly as a set of changed coordinates): Set $\boldsymbol{s} \leftarrow \boldsymbol{s} + \alpha \mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}$ for some matrix $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ with $\|\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\|_{\mathrm{op}} \le \epsilon_{\mathbf{P}}$ and $\mathrm{Range}(\widetilde{\mathbf{P}}_{\boldsymbol{w}}) = \mathrm{Range}(\mathbf{P}_{\boldsymbol{w}})$. The time is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ where K is the number of coordinates changed in $\boldsymbol{v}$, compared to the last input to* Move*.*
- Approximate() $\to \mathbb{R}^m$*: Update the vector $\overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \le \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{s}$.*
- Exact() $\to \mathbb{R}^m$*: Output the current vector $\boldsymbol{s}$ in $\widetilde{O}(m)$ time.*

Suppose we have T steps and in each step we call Reweight*,* Move *and* Approximate *in order. If $\alpha\|\boldsymbol{v}\|_2 \le \beta$ for all calls to* Move*, then*

- *at the k-th* Approximate*, $O(N_k \stackrel{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates in $\overline{\boldsymbol{s}}$ are changed, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$. The time for the k-th* Approximate *is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_k)})$ where K is the number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$, compared to the last input to* Approximate *(or to the initial state if $k = 1$).*

To implement this data structure, we use the definition of $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ from Eq. (2.10). Since this is the original $\mathbf{P}_{\boldsymbol{w}}$ with the Schur complement terms replaced by approximations, we have $\widetilde{\mathbf{P}}_{\boldsymbol{w}} \approx \mathbf{P}_{\boldsymbol{w}}$ using spectral approximations. Also, $\mathrm{Range}(\widetilde{\mathbf{P}}_{\boldsymbol{w}}) = \mathrm{Range}(\mathbf{W}^{1/2}\mathbf{B}) = \mathrm{Range}(\mathbf{P}_{\boldsymbol{w}})$ by definition.

Theorem 2.6 shows how to maintain the intermediate vector $\boldsymbol{z}$ defined as

$$\boldsymbol{z} = \widetilde{\boldsymbol{\Gamma}}\boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}.$$

This is a useful partial computation, as we observe that the full slack update is

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \alpha \mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} = \boldsymbol{s} + \alpha \mathbf{W}^{-1/2}\mathbf{W}^{1/2}\mathbf{B}\boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}\boldsymbol{z}.$$

The remaining expression $\mathbf{B}\boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}\boldsymbol{z}$ can in fact be written as a forest operator applied to $\boldsymbol{z}$. The forest operator $\overline{\mathbf{M}}$ operates on the forest $\overline{\mathcal{T}} = \{T_H : H \in \mathcal{T}\}$, where T_H is the subtree of $\mathcal{T}$ rooted at H with an additional child node D_N for each leaf node N. Recall that

$$\boldsymbol{\Pi}^{(i)\top} = \mathbf{I} - \left(\mathbf{L}^{(i)}_{F_i,F_i}\right)^{-1}\mathbf{L}^{(i)}_{F_i,C_i}.$$

We decompose based on nodes H at level i, as a key step to reformulating $\boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}$ as a forest operator.

For every level i of $\mathcal{T}$, let $\{\bar{C}(H) : H \text{ at level } i \text{ in } \mathcal{T}\}$ be an arbitrary partition of C_i satisfying $\bar{C}(H) \subseteq \partial H$ for each H. Let $\mathbf{I}_{\bar{C}(H)}$ be the diagonal matrix with 1 on the diagonal entries indexed by $\bar{C}(H)$, and 0 otherwise.

For a non-leaf node $H \in \mathcal{T}$ with parent P, for all trees $T \in \overline{\mathcal{T}}$ containing the upward edge (H, P), we define, as part of the forest operator,

$$\mathbf{M}_{H,P} \stackrel{\text{def}}{=} \mathbf{I}_{\bar{C}(H)} - \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1}\mathbf{L}^{(H)}_{F_H,\partial H}. \tag{2.11}$$

Algorithm 8 Slack Maintenance, Main Algorithm

Private: member
  Sketch maintenance data structure for forest operator slackSketch (Theorem 2.4, Algorithm 3)
  Intermediate vector $\boldsymbol{z}$ maintenance data structure maintainZ (Theorem 2.6)
  Forest operator $\overline{\mathbf{M}} = \mathbf{B}^\top \mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(\eta-1)\top}$
  Forest coefficient c which refer to c maintained by maintainZ
  Forest vectors $\boldsymbol{z}_1, \boldsymbol{z}_2$ which refer to $\boldsymbol{z}^{(\text{sum})}$ and $\boldsymbol{z}^{(\text{prev})}$ maintained by maintainZ respectively
  Slack vector $\tilde{\boldsymbol{s}}_0, \overline{\boldsymbol{s}}_1$

procedure INITIALIZE($G, \boldsymbol{s} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \varepsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0$)
  Compute $\overline{\mathbf{M}}$ by Eqs. (2.11) and (2.12)
  $\hat{c} \leftarrow 0$
  $\tilde{\boldsymbol{s}}_0 \leftarrow \mathbf{0}, \overline{\boldsymbol{s}}_1 \leftarrow \boldsymbol{s}$
  $\mathbf{W} \leftarrow \operatorname{diag}(\boldsymbol{w})$
  slackSketch.INITIALIZE($\mathbf{M}, \boldsymbol{z}_1, \boldsymbol{z}_2, c_1, c_2, n^{-5}, \varepsilon_{\mathbf{P}}, \beta$)
  maintainZ.INITIALIZE($G, \boldsymbol{v}, \boldsymbol{w}, \varepsilon_{\mathbf{P}}$) ▷ maintainZ.INITIALIZE initializes $\boldsymbol{z}_1, \boldsymbol{z}_2$ and c_2
end procedure

procedure REWEIGHT($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m$)
  Compute the new forest operator $\overline{\mathbf{M}}^{(\text{new})}$ using the new weights $\boldsymbol{w}^{(\text{new})}$
  $\tilde{\boldsymbol{s}_0} \leftarrow \tilde{\boldsymbol{s}_0} - (\overline{\mathbf{M}}^{(\text{new})} - \overline{\mathbf{M}})(\boldsymbol{z}_1 + c_1 \boldsymbol{z}_2)$
  $\overline{\mathbf{M}} \leftarrow \overline{\mathbf{M}}^{(\text{new})}$
  $\mathbf{W} \leftarrow \operatorname{diag}(\boldsymbol{w}^{(\text{new})})$
  maintainZ.REWEIGHT($\boldsymbol{w}^{(\text{new})}$) ▷ maintainZ.REWEIGHT updates $\boldsymbol{z}_1, \boldsymbol{z}_2$ and c_2
end procedure

procedure MOVE($\alpha, \boldsymbol{v}^{(\text{new})} \in \mathbb{R}^m$)
  maintainZ.MOVE($\alpha, \boldsymbol{v}^{(\text{new})}$) ▷ maintainZ.MOVE updates $\boldsymbol{z}_1, \boldsymbol{z}_2$ and c_2
end procedure

procedure APPROXIMATE( )
  $\overline{\boldsymbol{s}}_1 \leftarrow$ slackSketch.APPROXIMATE($\overline{\mathbf{M}}, \boldsymbol{z}_1, \boldsymbol{z}_2, c_1, c_2, \mathbf{W}$)
  return $\overline{\boldsymbol{s}}_1 + \tilde{\boldsymbol{s}}_0$
end procedure

procedure EXACT( )
  $\tilde{\boldsymbol{s}}_1 \leftarrow$ slackSketch.EXACT()
  return $\tilde{\boldsymbol{s}}_1 + \tilde{\boldsymbol{s}}_0$
end procedure

For a leaf node $H \in \mathcal{T}$, for all trees $T \in \overline{\mathcal{T}}$ containing H, we define

$$\mathbf{M}_{N_H,H} \stackrel{\text{def}}{=} \mathbf{B}[H]. \tag{2.12}$$

With this forest operator, we may then directly invoke Theorem 2.4 to maintain an approximation of $\boldsymbol{s}$.

2.4.3 Maintaining Flow Now, we discuss how to maintain the flow solution under the update $\boldsymbol{f} \leftarrow \boldsymbol{f} + h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}\boldsymbol{v}$ as needed in Algorithm 1, Line 23. This is accomplished by a flow maintenance data structure that simultaneously maintains the approximate slack $\overline{\boldsymbol{f}}$ and the edge weights $\boldsymbol{w}$ from the IPM. A single IPM step calls the procedures Reweight, Move, Approximate in this order once.

Theorem 2.8. (Flow Maintenance) *Given a modified planar graph G with m edges, there exists a randomized data structure (Algorithm 9) that implicitly maintains the flow solution $\boldsymbol{f}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{f}}$ as well as the edge weights $\boldsymbol{w}$, and supports the following procedures with high probability in m against an adaptive adversary:*

- Initialize($G, \boldsymbol{f} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0$)*: Given a graph G, initial vector $\boldsymbol{f}$, initial weight $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$ and target output accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time.*
- Reweight($\boldsymbol{w} \in \mathbb{R}^m_{>0}$ *given implicitly as a set of changed weights): Set the current weight to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$ compared to the last input to* Reweight.
- Move($\alpha \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ *given implicitly as a set of changed coordinates): Set $\boldsymbol{f} \leftarrow \boldsymbol{f} + \alpha\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}\boldsymbol{v}$ for some matrix $\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}$ with $\|\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp} - (\mathbf{I} - \mathbf{P}_{\boldsymbol{w}})\|_{\text{op}} \le \epsilon_{\mathbf{P}}$ and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp} = \mathbf{0}$. The time is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$, where K is the number of coordinates changed in $\boldsymbol{v}$, compared to the last input to* Move.
- Approximate() $\to \mathbb{R}^m$*: Update the vector $\overline{\boldsymbol{f}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \le \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{f}$.*
- Exact() $\to \mathbb{R}^m$*: Output the current vector $\boldsymbol{f}$ in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time.*

Suppose we have T steps and in each step we call Reweight, Move *and* Approximate *in order. If $\alpha\|\boldsymbol{v}\|_2 \le \beta$ for all calls to* Move, *then*

- *at the k-th* Approximate, *$O(N_k \stackrel{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates in $\overline{\boldsymbol{f}}$ are changed, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$. The time for the k-th* Approximate *is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_k)})$ where K is the number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$, compared to the last input to* Approximate *(or to the initial state if $k = 1$).*

To implement Move($\alpha, \boldsymbol{v}$), we must define $\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}$ such that $\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp} \approx \mathbf{I} - \mathbf{P}_{\boldsymbol{w}}$ and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp} = \mathbf{0}$. Rather than writing $\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}$ in some closed form, observe that it suffices to find some *weighted flow* $\tilde{\boldsymbol{f}} \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$ satisfying $\mathbf{B}^\top\mathbf{W}^{1/2}\tilde{\boldsymbol{f}} = \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$. (It is a weighted flow as it is obtained by multiplying the weights $\mathbf{W}$ to some flow.) Indeed, with such an $\tilde{\boldsymbol{f}}$, we can define $\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}\boldsymbol{v} \stackrel{\text{def}}{=} \boldsymbol{v} - \tilde{\boldsymbol{f}}$, and check that it satisfies $\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}\boldsymbol{v} = \boldsymbol{v} - \tilde{\boldsymbol{f}} \approx (\mathbf{I} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}$, as well as $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}^{\perp}\boldsymbol{v} = \mathbf{0}$. Hence, we maintain $\boldsymbol{f} = \hat{\boldsymbol{f}} - \tilde{\boldsymbol{f}}$. For each IPM step, we update them as follows:

$$\hat{\boldsymbol{f}} \leftarrow \hat{\boldsymbol{f}} + \alpha\mathbf{W}^{1/2}\boldsymbol{v} \qquad \text{and} \qquad \tilde{\boldsymbol{f}} \leftarrow \tilde{\boldsymbol{f}} + \alpha\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v},$$

where $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ satisfies $\left\|\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\right\|_{\text{op}} \le \epsilon_{\mathbf{P}}$, and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$. Maintaining the term $\hat{\boldsymbol{f}}$ is straightforward; as such, we turn our focus to $\tilde{\boldsymbol{f}}$.

Let us define demands on vertices by $\boldsymbol{d} \stackrel{\text{def}}{=} \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$. Unwrapping the definition of $\mathbf{P}_{\boldsymbol{w}}$, we see that the first condition is $\tilde{\boldsymbol{f}} \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$. Combinatorially, this says we want some weighted flow close to the weighted electrical flow routing demand $\boldsymbol{d}$. Suppose we had $\tilde{\boldsymbol{f}} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$, then we see immediately that $\mathbf{B}^\top\mathbf{W}^{1/2}\tilde{\boldsymbol{f}} = \mathbf{B}^\top\mathbf{W}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d} = \boldsymbol{d}$, so that the second condition is also satisfied. To realize the approximation, we make use of Eq. (2.7) and approximate Schur complements to *implicitly* produce some $\widetilde{\mathbf{L}}^{-1} \approx \mathbf{L}^{-1}$. Hence, one

Algorithm 9 Flow Maintenance, Main Algorithm

Private: member
  Sketch Maintenance Data Structure flowSketch (Theorem 2.4, Algorithm 3)
  Intermediate Vector $\boldsymbol{z}$ Maintenance Data Structure maintainZ (Theorem 2.6)
  Forest Operator $\overline{\mathbf{M}}$
  Forest coefficient c_1 which is always 1
  Forest coefficient c_2 which refer to c maintained by maintainZ
  Forest vectors $\boldsymbol{z}_1, \boldsymbol{z}_2$ which refer to $\boldsymbol{z}^{(\text{sum})}$ and $\boldsymbol{z}^{(\text{prev})}$ maintained by maintainZ respectively
  Flow vector $\tilde{\boldsymbol{f}}_0, \overline{\boldsymbol{f}}_1$
  Flow coefficient $\hat{c}$
  Flow vector $\hat{\boldsymbol{f}}_0, \hat{\boldsymbol{f}}_1$

procedure INITIALIZE($G, \boldsymbol{f} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \varepsilon_{\mathbf{P}} > 0, \bar{\epsilon} > 0$)
  Compute $\overline{\mathbf{M}}$ by Eqs. (2.13) and (2.14)
  $\hat{c} \leftarrow 0$
  $\hat{\boldsymbol{f}}_0 \leftarrow \boldsymbol{f}, \tilde{\boldsymbol{f}}_0 \leftarrow \mathbf{0}, \overline{\boldsymbol{f}}_1 \leftarrow \mathbf{0}$
  $\mathbf{W} \leftarrow \operatorname{diag}(\boldsymbol{w})$
  flowSketch.INITIALIZE($\mathbf{M}, \boldsymbol{z}_1, \boldsymbol{z}_2, c_1, c_2, n^{-5}, \varepsilon_{\mathbf{P}}, \beta$)
  maintainZ.INITIALIZE($G, \boldsymbol{v}, \boldsymbol{w}, \varepsilon_{\mathbf{P}}$) ▷ maintainZ.INITIALIZE initializes $\boldsymbol{z}_1, \boldsymbol{z}_2$ and c_2
  $\hat{\boldsymbol{f}}_1 \leftarrow \mathbf{W}^{1/2}\boldsymbol{v}$
end procedure

procedure REWEIGHT($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m$)
  Compute the new forest operator $\overline{\mathbf{M}}^{(\text{new})}$ using the new weights $\boldsymbol{w}^{(\text{new})}$
  maintainZ.REWEIGHT($\boldsymbol{w}^{(\text{new})}$)
  $\tilde{\boldsymbol{f}}_0 \leftarrow \tilde{\boldsymbol{f}}_0 - (\overline{\mathbf{M}}^{(\text{new})} - \overline{\mathbf{M}})(\boldsymbol{z}_1 + c_1\boldsymbol{z}_2)$ ▷ maintainZ.REWEIGHT updates $\boldsymbol{z}_1, \boldsymbol{z}_2$ and c_2
  $\overline{\mathbf{M}} \leftarrow \overline{\mathbf{M}}^{(\text{new})}$
  $\mathbf{W} \leftarrow \operatorname{diag}(\boldsymbol{w}^{(\text{new})})$
  $\hat{\boldsymbol{f}}_1^{(\text{new})} \leftarrow \mathbf{W}^{1/2}v$
  $\hat{\boldsymbol{f}}_0 \leftarrow \hat{\boldsymbol{f}}_0 + \hat{c} \cdot (\hat{\boldsymbol{f}}_1 - \hat{\boldsymbol{f}}_1^{(\text{new})})$
  $\hat{\boldsymbol{f}}_1 \leftarrow \hat{\boldsymbol{f}}_1^{(\text{new})}$
end procedure

procedure MOVE($\alpha, \boldsymbol{v}^{(\text{new})} \in \mathbb{R}^m$)
  maintainZ.MOVE($\alpha, \boldsymbol{v}^{(\text{new})}$) ▷ maintainZ.MOVE updates $\boldsymbol{z}_1, \boldsymbol{z}_2$ and c_2
  $\hat{\boldsymbol{f}}_1^{(\text{new})} \leftarrow \mathbf{W}^{1/2}v$
  $\hat{\boldsymbol{f}}_0 \leftarrow \hat{\boldsymbol{f}}_0 + \hat{c} \cdot (\hat{\boldsymbol{f}}_1 - \hat{\boldsymbol{f}}_1^{(\text{new})})$
  $\hat{\boldsymbol{f}}_1 \leftarrow \hat{\boldsymbol{f}}_1^{(\text{new})}$
  $\hat{c} \leftarrow \hat{c} + \alpha$
end procedure

procedure APPROXIMATE( )
  $\overline{\boldsymbol{f}}_1 \leftarrow$ flowSketch.APPROXIMATE($\overline{\mathbf{M}}, \boldsymbol{z}_1, \boldsymbol{z}_2, c_1, c_2, \mathbf{W}$).
  return $(\hat{\boldsymbol{f}}_0 + \hat{c} \cdot \hat{\boldsymbol{f}}_1) - (\overline{\boldsymbol{f}}_1 + \tilde{\boldsymbol{f}}_0)$
end procedure

procedure EXACT( )
  $\tilde{\boldsymbol{f}}_1 \leftarrow$ flowSketch.EXACT()
  return $(\hat{\boldsymbol{f}}_0 + \hat{c} \cdot \hat{\boldsymbol{f}}_1) - (\tilde{\boldsymbol{f}}_1 + \tilde{\boldsymbol{f}}_0)$
end procedure

important fact about our construction is that when the Schur complements are exact, our flow $\tilde{\boldsymbol{f}}$ agrees with the true electrical flow of the demand.

To construct $\tilde{\boldsymbol{f}}$ meeting the demand $\boldsymbol{d}$, we also make use of the intermediate $\boldsymbol{z}$ from Theorem 2.6. The first step of our algorithm is decomposing $\boldsymbol{d}$ using the separator tree, such that we have a demand term $\boldsymbol{d}^{(H)}$ for each region H from the separator tree, and furthermore, $\boldsymbol{d}^{(H)} = \mathbf{L}^{(H)} \cdot \boldsymbol{z}|_{F_H}$, for some Laplacian $\mathbf{L}^{(H)}$ supported on the region H, and $\boldsymbol{z}|_{F_H}$ is the sub-vector of $\boldsymbol{z}$ indexed by vertices eliminated at H. This construction allows us to route each demand $\boldsymbol{d}^{(H)}$ by electric flows using only the corresponding region H, rather than the entire graph. The recursive nature of the decomposition allows us to bound the overall run-time. To show that the resulting flow $\tilde{\boldsymbol{f}}$ indeed is close to the electric flow, one key insight is that the decomposed demands we construct are almost orthogonal when $\widetilde{\mathbf{Sc}} \approx \mathbf{Sc}$. Hence, routing them separately by electrical flows gives a good approximation to the true electrical flow of the whole demand.

Let us illustrate this partially using the two-layer nested dissection scheme from Section 2.2: Suppose we have a demand term $\boldsymbol{d}$ *that is non-zero only on vertices of* C. Then, observe that

$$\boldsymbol{z} = \begin{bmatrix} \mathbf{L}_{FF}^{-1} & \mathbf{0} \\ \mathbf{0} & \widetilde{\mathbf{Sc}}(\mathbf{L}, C)^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix} \boldsymbol{d}$$

Looking at the subvector indexed by C on both sides, we have that

$$\widetilde{\mathbf{Sc}}(\mathbf{L}, C) \cdot \boldsymbol{z} = \boldsymbol{d}$$

where we abuse the notation to extend $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)$ from $C \times C$ to $[n] \times [n]$ by padding zeros. Using Eq. (2.6), we have

$$\left(\widetilde{\mathbf{Sc}}(\mathbf{L}[H_1], C) + \widetilde{\mathbf{Sc}}(\mathbf{L}[H_2], C)\right) \cdot \boldsymbol{z} = \boldsymbol{d}$$

This gives a decomposition of the demand $\boldsymbol{d}$ into demand terms $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$. Crucially, each demand $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$ is supported on the vertices of the region H_i, hence we choose to route the flow accordingly. That is, we have that $\tilde{\boldsymbol{f}}_i$ is the electric flow on the subgraph H_i that satisfies the demand $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$; more concretely, $\tilde{\boldsymbol{f}}_i = \mathbf{W}^{1/2}\mathbf{B}[H_i]\mathbf{L}[H_i]^{-1}\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$. Finally, we will let the output be $\tilde{\boldsymbol{f}} = \sum \tilde{\boldsymbol{f}}_i$. By construction, this $\boldsymbol{f}$ satisfies $\mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{f} = \boldsymbol{d} = \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}$.

In order to use the data structure for the sketch maintenance given in Theorem 2.4 to maintain $\boldsymbol{f}$, we need to define the forest operator. The separator tree and the decomposition of flows naturally give rise to construction of the forest operator $\overline{\mathbf{M}}$: It consists of the forest $\overline{\mathcal{T}} = \{T_H : H \in \mathcal{T}\}$, where T_H is the subtree of $\mathcal{T}$ rooted at H with an additional child node D_N for each leaf node N.

Recall from the definition of $\mathbf{L}^{(H)}$, for a node $H \in \mathcal{T}$ with children D_1, D_2, we have

$$\mathbf{L}^{(H)} = \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial D_1) + \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial D_2).$$

For a non-leaf node $H \in \mathcal{T}$ with child D, for all trees $T \in \overline{\mathcal{T}}$ containing the upward edge (D, H), we define, as part of the forest operator,

$$\mathbf{M}_{D,H} \stackrel{\text{def}}{=} (\mathbf{L}^{(D)})^{-1}\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D)}, \partial D). \tag{2.13}$$

For a leaf node $H \in \mathcal{T}$, for all trees $T \in \overline{\mathcal{T}}$ containing H, we define

$$\mathbf{M}_{D_H,H} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}[H], \tag{2.14}$$

which correspond to the operation of routing the flow on H.

With this forest operator, we may then directly invoke Theorem 2.4 to maintain an approximation of $\boldsymbol{f}$.

2.5 Main Proof We are now ready to prove our main result. Algorithm 10 presents the implementation of RIPM Algorithm 1 using our data structures.

We first prove a lemma about how many coordinates change in $\boldsymbol{w}$ and $\overline{\boldsymbol{v}}$ in each step. This is useful for bounding the complexity of each iteration.

Algorithm 10 Implementation of Robust Interior Point Method

procedure CENTERINGIMPL($\mathbf{B}, \phi, \boldsymbol{f}, \boldsymbol{s}, t_{\text{start}}, t_{\text{end}}$)
  Let G be the graph on n vertices and m edges with incidence matrix $\mathbf{B}$
  Let $\mathcal{F}$ and $\mathcal{S}$ be the data structure for flow and slack maintenance (Theorem 2.8, Theorem 2.7)
  Let $\alpha = \frac{1}{2^{20}\lambda}$ and $\lambda = 64\log(256m^2)$
  Let $t \leftarrow t_{\text{start}}$, $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}, \overline{\boldsymbol{s}} \leftarrow \boldsymbol{s}, \overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}, \overline{\boldsymbol{s}} \leftarrow \boldsymbol{s}, \bar{t} \leftarrow t$, $\mathbf{W} \leftarrow \nabla^2\phi(\overline{\boldsymbol{f}})^{-1}$
  $\mathcal{F}$.INITALIZE($G, \boldsymbol{f}, \boldsymbol{w}, \alpha, \frac{\alpha}{2}$)
  $\mathcal{S}$.INITALIZE($G, \boldsymbol{s}, \boldsymbol{w}, \alpha, \frac{\alpha}{2}$)
  ▷ Define $\mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}^{1/2}$
  while $t \geq t_{\text{end}}$ **do**
    Set $t \leftarrow \max\{(1 - \frac{\alpha}{\sqrt{m}})t, t_{\text{end}}\}$
    Update $h = -\alpha/\|\cosh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}))\|_2$ where γ defined in Eq. (2.2)
    Update the diagonal weight matrix $\mathbf{W} = \nabla^2\phi(\overline{\boldsymbol{f}})^{-1}$
    $\mathcal{F}$.REWEIGHT($\boldsymbol{w}$) ▷ Update the implicit representation of $\boldsymbol{f}$ with new weights
    $\mathcal{S}$.REWEIGHT($\boldsymbol{w}$) ▷ Update the implicit representation of $\boldsymbol{s}$ with new weights
    Update the direction $\boldsymbol{v}$, where $\boldsymbol{v}_i = \sinh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}})_i)$
    $\mathcal{F}$.MOVE($h, \boldsymbol{v}$) ▷ Update $\boldsymbol{f} \leftarrow \boldsymbol{f} + h\mathbf{W}^{1/2}\boldsymbol{v}^\perp$ with $\boldsymbol{v}^\perp \approx (\mathbf{I} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}$
    $\mathcal{S}$.MOVE($\bar{t}h, \boldsymbol{v}$) ▷ Update $\boldsymbol{s} \leftarrow \boldsymbol{s} + \bar{t}h\mathbf{W}^{-1/2}\boldsymbol{v}^\parallel$ with $\boldsymbol{v}^\parallel \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$
    $\overline{\boldsymbol{f}} \leftarrow \mathcal{F}$.APPROXIMATE() ▷ Maintain $\overline{\boldsymbol{f}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \leq \alpha$
    $\overline{\boldsymbol{s}} \leftarrow \mathcal{S}$.APPROXIMATE() ▷ Maintain $\overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \leq \alpha$
    Update $\bar{t} \leftarrow t$ if $|\bar{t} - t| \geq \alpha\bar{t}$
    If $\bar{t}$ changed, $\mathcal{S}$.INITALIZE($G, \mathcal{S}$.EXACT(), $\boldsymbol{w}, \alpha, \frac{\alpha}{8\lceil\log m\rceil}$)
  end while
  Return ($\mathcal{F}$.EXACT(), $\mathcal{S}$.EXACT())
end procedure

LEMMA 2.5. *In the k-th iteration of the* CENTERINGIMPL *algorithm, there are $O(2^{2\ell_k}\log^2 m)$ coordinates in $\boldsymbol{w}$ and $\boldsymbol{v}$ changed, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$.*

Proof. Since both $\boldsymbol{w}$ and $\boldsymbol{v}$ are an entry-wise function of $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ and $\bar{t}$, it suffices to bound the number of changes of these variables.

Note that $\bar{t}$ changes every $\sqrt{m}/(C\log m)$ iterations for some constant C. By choosing the constant in the algorithm, we can assume $\sqrt{m}/(C\log m) = 2^\ell$ for some integer. When $\bar{t}$ changes, every coordinates of $\boldsymbol{w}$ and $\boldsymbol{v}$ change. This is allowed because $O(2^{2\ell}\log^2 m) = \widetilde{O}(m)$.

Now, we bound the changes of $\overline{\boldsymbol{f}}$. At the end of the $(k-1)$-th iteration, we have $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_2 \leq 2h\|\boldsymbol{v}\|_2 = O(\frac{1}{\log m})$ by Theorem 2.1. Then, $\overline{\boldsymbol{f}}$ satisfies the assumption in Theorem 2.8 with $\beta = O(\frac{1}{\log m})$ and $\overline{\epsilon} = \Theta(\frac{1}{\log m})$. Hence, Theorem 2.8 guarantees that there are $O(2^{2\ell_k}\log^2 m)$ coordinates in $\overline{\boldsymbol{f}}$ that changes at the k-th iteration. The proof for $\overline{\boldsymbol{s}}$ is similar. ∎

THEOREM 2.9. (MAIN RESULT) *We are given a directed planar graph $G = (V, E)$ with n vertices. Assume that the demands $\boldsymbol{d}$, edge capacities $\boldsymbol{u}$ and costs $\boldsymbol{c}$ are all integers and bounded by M in absolute value. Then we can compute a minimum cost flow satisfying demand $\boldsymbol{d}$ in $O(n\log^{O(1)} n\log M)$ expected time.*

Proof. The proof is structured as follows. We first write the minimum cost flow problem as a linear program of the form Eq. (2.1). We prove the linear program has an interior point and is bounded, so to satisfy the assumptions in Theorem 2.1. Then, we implement the IPM algorithm using the data structures from Section 2.4. Finally, we bound the cost of each operations of the data structures.

To write down the min-cost flow problem as a linear program of the form Eq. (2.1), we add extra vertices s and t. For every vertex v with $\boldsymbol{d}_v \leq 0$, we add a directed edge from s to v with capacity $-\boldsymbol{d}_v$ and cost 0. For every vertex v with $\boldsymbol{d}_v \geq 0$, we add a directed edge from v to t with capacity $\boldsymbol{d}_v$ and cost 0. Then, we add a directed edge from t to s with capacity $2nM$ and cost $-2nM$.

The cost and capacity on the $t \to s$ edge is chosen such that the minimum cost flow problem on the original graph is equivalent to the minimum cost circulation on this new graph. Namely, if the minimum cost circulation in this new graph satisfies all the demand $\boldsymbol{d}_v$, then this circulation (ignoring the flow on the new edges) is the minimum cost flow in the original graph.

Since Theorem 2.1 requires an "interior" point in the polytope, we first remove all directed edges e through which no flow from s to t can pass. To do this, we simply check, for every directed edge $e = (v_1, v_2)$, if s can reach v_1 and if v_2 can reach t. This can be done in $O(m)$ time by a BFS from s and a reverse BFS from t. With this preprocessing, we write the minimum cost circulation problem as the following linear program

$$\min_{\mathbf{B}^\top \boldsymbol{f} = \mathbf{0},\, \boldsymbol{l}^{\text{new}} \le \boldsymbol{f} \le \boldsymbol{u}^{\text{new}}} (\boldsymbol{c}^{\text{new}})^\top \boldsymbol{f}$$

where $\mathbf{B}$ is the signed incidence matrix of the new graph, $\boldsymbol{c}^{\text{new}}$ is the new cost vector (with cost on extra edges), and $\boldsymbol{l}^{\text{new}}, \boldsymbol{u}^{\text{new}}$ are the new capacity constraints. If an edge e has only one direction, we set $\boldsymbol{l}_e^{\text{new}} = 0$ and $\boldsymbol{u}_e^{(\text{new})} = \boldsymbol{u}_e$, otherwise, we orient the edge arbitrarily and set $-\boldsymbol{l}_e^{\text{new}} = \boldsymbol{u}_e^{\text{new}} = \boldsymbol{u}_e$.

Now, we bound the parameters L, R, r in Theorem 2.1. Clearly, $L = \|\boldsymbol{c}^{\text{new}}\|_2 = O(Mm)$ and $R = \|\boldsymbol{u}^{\text{new}} - \boldsymbol{l}^{\text{new}}\|_2 = O(Mm)$. To bound r, we prove that there is an "interior" flow $\boldsymbol{f}$ in the polytope $\mathcal{F}$. We construct this $\boldsymbol{f}$ by $\boldsymbol{f} = \sum_{e \in E} \boldsymbol{f}^{(e)}$, where $\boldsymbol{f}^{(e)}$ is a circulation passing through edges e and (t, s) with flow value $1/(4m)$. All such circulations exist because of the removal preprocessing. This $\boldsymbol{f}$ satisfies the capacity constraints because all capacities are at least 1. This shows $r \ge \frac{1}{4m}$.

The requirements of Theorem 2.1 for $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$ are satisfied by the guarantees of Theorem 2.8 and Theorem 2.7. Hence, Theorem 2.1 shows that we can find a circulation $\boldsymbol{f}$ such that $(\boldsymbol{c}^{\text{new}})^\top \boldsymbol{f} \le \text{OPT} - \frac{1}{2}$ by setting $\epsilon = \frac{1}{CM^2m^2}$ for some large constant C. Note that $\boldsymbol{f}$, when restricted to the original graph, is almost a flow routing the required demand with flow value off by at most $\frac{1}{2nM}$. This is because sending extra k units of fractional flow from s to t gives extra negative cost $\le -knM$. Now we can round $\boldsymbol{f}$ to an integral flow $\boldsymbol{f}^{\text{int}}$ with same or better flow value using no more than $\widetilde{O}(m)$ time [31]. Since $\boldsymbol{f}^{\text{int}}$ is integral with flow value at least the total demand minus $\frac{1}{2}$, $\boldsymbol{f}^{\text{int}}$ routes the demand completely. Again, since $\boldsymbol{f}^{\text{int}}$ is integral with cost at most $\text{OPT} - \frac{1}{2}$, $\boldsymbol{f}^{\text{int}}$ must have the minimum cost.

Finally, we bound the run-time of Theorem 2.1. At each step of Centering, we maintain the implicit update step of $\boldsymbol{f}$ and $\boldsymbol{s}$ using Move in Section 2.4; we update $\mathbf{W}$ in the data structures using Reweight; and we construct the explicit $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$ using Approximate. We return the true $(\boldsymbol{f}, \boldsymbol{s})$ by Exact. The total cost of Centering is dominated by Move, Reweight, and Approximate.

Since we call Move, Reweight and Approximate in order in each step and the runtime for Move, Reweight are both dominated by the runtime for Approximate, we bound the runtime for Approximate only. Theorem 2.1 guarantees that there are $T = O(\sqrt{m} \log n \log(nM))$ total calls. Lemma 2.5 shows that at the k-th call, the number of coordinates changed in $\mathbf{W}, \boldsymbol{v}$ is bounded by $O(2^{2\ell_k} \log^2 m)$. This number is bound by $O(N_k)$. Using this, we apply Theorem 2.8 and Theorem 2.7 to show that the time of the k-step is

$$\widetilde{O}\left(\epsilon_{\mathbf{P}}^{-2} \sqrt{m} \cdot \sqrt{\min(2^{2\ell_k} \log^2 m, m)}\right) = \widetilde{O}\left(\min(\sqrt{m} 2^{\ell_k}, m)\right)$$

where we use $\epsilon_{\mathbf{P}} = 1/C \log m$ as required in Theorem 2.1. Hence, the total time for Move and Reweight over T calls is

$$\widetilde{O}\left(\sum_{k=1}^{T} \min(\sqrt{m} \cdot 2^{\ell_k}, m)\right).$$

Note that $\sum_{k=1}^{T} 2^{\ell_k} = O(T \log T)$: We bound the sum by splitting it into segments of length $2^\ell = \Theta(\sqrt{m})$. In each segment, we bound the sum by $O(m \log m)$. Since there are $T/2^\ell = O(\log n \log(nM))$ segments, the total time for Move and Reweight is $\widetilde{O}(m \log(nM))$. ∎

References

[1] Deeksha Adil, Brian Bullins, Rasmus Kyng, and Sushant Sachdeva. Almost-Linear-Time Weighted ℓ_p-norm Solvers in Slightly Dense Graphs via Sparsification. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *LIPIcs*, pages 9:1–9:15, 2021.

[2] Deeksha Adil and Sushant Sachdeva. Faster p-norm minimizing flows, via smoothed q-norm problems. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 892–910. SIAM, 2020.

[3] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. 1988.

[4] Mudabir Kabir Asathulla, Sanjeev Khanna, Nathaniel Lahn, and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite perfect matching in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 457–476. SIAM, 2018.

[5] Kyriakos Axiotis, Aleksander Mądry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 93–104. IEEE Computer Society, 2020.

[6] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. *arXiv preprint arXiv:2101.07149*, 2021. To appear in 2021 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS).

[7] Glencora Borradaile. *Exploiting planarity for network flow and connectivity problems*. Brown University, 2008.

[8] Glencora Borradaile and Philip N. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *J. ACM*, 56(2):9:1–9:30, 2009.

[9] Glencora Borradaile, Philip N. Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM J. Comput.*, 46(4):1280–1303, 2017.

[10] Erin W. Chambers, Jeff Erickson, and Amir Nayyeri. Homology flows, cohomology cuts. *SIAM J. Comput.*, 41(6):1605–1634, 2012.

[11] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 273–282, 2011.

[12] Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *Journal of the ACM (JACM)*, 68(1):1–39, 2021.

[13] Michael B Cohen, Aleksander Mądry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\widetilde{O}(m^{10/7} \log w)$ time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 752–771. SIAM, 2017.

[14] Samuel I Daitch and Daniel A Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 451–460, 2008.

[15] Sally Dong, Yin Tat Lee, and Guanghao Ye. A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path. *arXiv preprint arXiv:2011.05365v2*, 2020.

[16] Sally Dong, Yin Tat Lee, and Guanghao Ye. A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, pages 1784–1797. Association for Computing Machinery, 2021.

[17] David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. Sampling random spanning trees faster than matrix multiplication. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 730–742. ACM, 2017.

[18] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006.

[19] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[20] Yu Gao, Yang P. Liu, and Richard Peng. Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao. *arXiv preprint arXiv:2101.07233*, 2021. To appear in 2021 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS).

[21] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numer. Math.*, 50(4):377–404, February 1987.

[22] Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate schur complement on separable graphs. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[23] Keith D Gremban. *Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems*. PhD thesis, Carnegie Mellon University, 1996.

[24] Refael Hassin and Donald B Johnson. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM Journal on Computing*, 14(3):612–624, 1985.

[25] Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1):3–23, 1997.

[26] Baihe Huang, Shunhua Jiang, Zhao Song, and Runzhou Tao. Solving tall dense sdps in the current matrix multiplication time. *arXiv preprint arXiv:2101.08208*, 2021.

[27] Hiroshi Imai and Kazuo Iwano. Efficient sequential and parallel algorithms for planar minimum cost flow. In *Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings*, volume 450 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 1990.

[28] Alon Itai and Yossi Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.

[29] Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 313–322. ACM, 2011.

[30] Arun Jambulapati and Aaron Sidford. Ultrasparse ultrasparsifiers and faster laplacian system solvers. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 540–559. SIAM, 2021.

[31] Donggu Kang and James Payor. Flow rounding. *arXiv preprint arXiv:1507.08139*, 2015.

[32] Haim Kaplan and Yahav Nussbaum. Min-cost flow duality in planar networks. *arXiv preprint arXiv:1306.6728*, 2013.

[33] Adam Karczmarz and Piotr Sankowski. Min-cost flow in unit-capacity planar graphs. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 66:1–66:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[34] Tarun Kathuria, Yang P. Liu, and Aaron Sidford. Unit capacity maxflow in almost $o(m^{4/3})$ time. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 119–130, 2020.

[35] Jonathan A Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on discrete algorithms*, pages 217–226. SIAM, 2014.

[36] Samir Khuller, Joseph Naor, and Philip Klein. The lattice structure of flow in planar graphs. *SIAM Journal on Discrete Mathematics*, 6(3):477–490, 1993.

[37] Valerie King, Satish Rao, and Rorbert Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[38] Rasmus Kyng. *Approximate Gaussian Elimination*. PhD thesis, Yale University, 2017. Available at: http://rasmuskyng.com/rjkyng-dissertation.pdf.

[39] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A. Spielman. Sparsified cholesky and multigrid solvers for connection laplacians. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 842–850. ACM, 2016.

[40] Rasmus Kyng, Richard Peng, Sushant Sachdeva, and Di Wang. Flows in almost linear time via adaptive preconditioning. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 902–913, 2019.

[41] Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians-fast, sparse, and simple. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 573–582. IEEE, 2016.

[42] Nathaniel Lahn and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite matching in minor-free graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 569–588. SIAM, 2019.

[43] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979.

[44] RJ Lipton and ROBERT ENDRE Tarjan. A planar separator theorem. *SIAM Journal of Applied Mathematics*, 36(2):177–189, 1979.

[45] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 253–262. IEEE, 2013.

[46] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016.

[47] Gary L. Miller and Joseph Naor. Flow in planar graphs with multiple sources and sinks. *SIAM J. Comput.*, 24(5):1002–1017, 1995.

[48] Gary L. Miller and Richard Peng. Approximate maximum flow on separable undirected graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1151–1170. SIAM, 2013.

[49] James Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the Twentieth annual*

ACM symposium on Theory of Computing, pages 377–387, 1988.

[50] John H Reif. Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM Journal on Computing*, 12(1):71–81, 1983.

[51] Jonah Sherman. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 263–269. IEEE, 2013.

[52] Jonah Sherman. Area-convexity, linf regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 452–460, 2017.

[53] Aaron Sidford and Kevin Tian. Coordinate methods for accelerating linf regression and faster approximate maximum flow. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 922–933. IEEE, 2018.

[54] Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90, 2004.

[55] Robert E Tarjan. An efficient planarity algorithm. Technical report, STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, 1971.

[56] Balachandran Vaidyanathan and Ravindra K Ahuja. Fast algorithms for specially structured minimum cost flow problems with applications. *Operations research*, 58(6):1681–1696, 2010.

[57] Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 259–278. SIAM, 2020.

[58] Jan van den Brand. Unifying matrix data structures: Simplifying and speeding up iterative algorithms. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 1–13. SIAM, 2021.

[59] Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, MDPs, and l1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021.

[60] Jan van den Brand, Yin Tat Lee, Aaron Sidford, and Zhao Song. Solving tall dense linear programs in nearly linear time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 775–788, 2020.

[61] Karsten Weihe. Maximum (s,t)-flows in planar networks in $O(|v| \log |v|)$ time. *Journal of Computer and System Sciences*, 55(3):454–475, 1997.