ANALYSIS AND MAINTENANCE OF GRAPH LAPLACIANS VIA RANDOM WALKS

A Dissertation
Presented to
The Academic Faculty

By

Yu Gao

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Algorithms, Combinatorics and Optimization
College of Computing

Georgia Institute of Technology

August 2022

ANALYSIS AND MAINTENANCE OF GRAPH LAPLACIANS VIA RANDOM WALKS

Thesis committee:

Dr. Richard Peng
School of Computer Science
Georgia Institute of Technology

Dr. Matthew Baker
School of Mathematics
Georgia Institute of Technology

Dr. Zvi Galil
School of Computer Science
Georgia Institute of Technology

Dr. Jonathan A. Kelner
Department of Mathematics and CSAIL
Massachusetts Institute of Technology

Dr. Mohit Singh
School of Industrial & Systems Engineering
Georgia Institute of Technology

Date approved: May 9, 2022

In memory of my grandmother

ACKNOWLEDGMENTS

I would like to express my deepest thanks to my advisor, Richard Peng. He introduced me to many topics, problem solving skills, and management of research projects.

I would also like to thank Richard Peng, Matthew Baker, Zvi Galil, Jonathan A. Kelner, and Mohit Singh for being part of my dissertation committee and Yitong Yin for being the reader.

I would like to thank all my coauthors: David Durfee, Matthew Fahrbach, Wenhan Huang, Richard Peng, Tao Xiao, Junxing Wang, Saurabh Sawlani, Sushant Sachdeva, Timothy Chu, Anup B. Rao, Gramoz Goranci, Chirag Jain, Haowen Zhang, Srinivas Aluru, Digvijay Boob, Charalampos E. Tsourakakis, Di Wang, Jason Li, Danupon Nanongkai, Thatchaphol Saranurak, Sorrachai Yingchareonthawornchai, Julia Chuzhoy, Yihe Dong, Ilya Razenshteyn, Yang P. Liu, Sally Dong, Yin Tat Lee, Guanghao Ye, Yufan Huang, Jingbang Chen, Jan van den Brand, Arun Jambulapati, and Aaron Sidford.

I am grateful to be introduced theoretical computer science by many extraordinary teachers, Pinyan Lu, Yitong Yin, and others.

I would like to thank my programming contest coaches Wei Teng and Yong Yu for their guidance and encouragement. I would like to thank my teammates, friends, and student coaches from the competitive programming community, Qi He, Hanyuan Shi, Yanpei Liu, Wenhao Li, Haobin Ni, Xiaoxu Guo, Jingbo Shang, Zhijian Liu, Wenda Qiu, Matthew Fahrbach, Animesh Fatehpuria, Yuhao Du, Li Chen, and many others, for improving ourselves together.

I would like to thank Marcel Celaya, Wenhao Li, Pinyan Lu, Dana Randall, Chi Ho Yoen, Daniel D. Sleater, and Xiaorui Sun for the insightful discussions and comments on research presented in this thesis.

I would like to thank my friends and colleagues from both Georgia Tech and Shanghai Jiao Tong University for their help throughout my graduate studies.

Finally, I would like to thank my family, especially my parents and grandparents, for always helping and supporting in my endeavors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graph Laplacians arise in many natural and artificial contexts. They are linear systems associated with undirected graphs. They are equivalent to electric flows [1] which is a fundamental physical concept by itself and is closely related to other physical models, e.g., the Abelian sandpile model [2]. Many real-world problems can be modeled and solved via Laplacian linear systems, including semi-supervised learning [3, 4], graph clustering [5], and graph embedding [6, 7].

More recently, better theoretical understandings of Laplacians led to dramatic improvements across graph algorithms. The applications include dynamic connectivity problem, graph sketching, and most recently combinatorial optimization. For example, a sequence of papers starting from [9] improved the runtime for maximum flow and minimum cost flow in many different settings.

In this thesis, we present works that the analyze, maintain, and utilize Laplacian linear systems in both static and dynamic settings by representing them as *random walks*. This combinatorial representation leads to better bounds for Abelian sandpile model on grids, the first data structures for dynamic vertex sparsifiers and dynamic Laplacian solvers, and network flow algorithms on planar as well as general graphs.

Abelian Sandpile model The Abelian sandpile model was the first construction of a dynamic system displaying self-organized criticality [2]. Bak, Tang, and Wiesenfeld proposed this idea to explain several ubiquitous patterns in nature typically viewed as complex phenomena, such as the power laws observed in turbulent fluids, earthquakes, distribution of visible matter in the universe, solar flares, the fractal behavior of mountain landscapes and coastal lines, and the presence of pink noise in electrical networks and stellar luminosity. The Abelian sandpile model is a discrete process on a graph. Vertices receive grains of sand, and once the number of grains exceeds their degree, they topple by sending grains to their neighbors. A critical state in this process is intuitively a state where adding a single grain triggers massive chain topples. Unlike many physical models where a critical state is achieved only by fine tuning, the Abelian sandpile model reaches a critical state by itself after adding enough number of grains.

Choure and Vishwanathan [10] relate the Abelian sandpile model to random walks on the

underlying graph by linear programs. They proved that up to some constant factor, the flow of sand approaches an electric flow on the grid when the number of grains added to the system approaches infinity.

Following [10], through a more refined and global analysis of electrical potentials through random walks, we bound the maximum number of grains that can be added to the system before it necessarily reaches its steady-state behavior, or equivalently, a *recurrent state*. We give an $O(n^4 \log^4 n)$ upper bound and an $\Omega(n^4)$ lower bound for the transience class of the $n \times n$ grid. Our methods naturally extend to n^d-sized d-dimensional grids to give $O(n^{3d-2} \log^{d+2} n)$ upper bounds and $\Omega(n^{3d-2})$ lower bounds.

Dynamic Vertex Sparsification In order to solve problems on massive graphs, *graph sparsification* is a tool that has received a considerable amount of attention over the past two decades [11, 12]. A sparsifier of some graph G is a (smaller) graph H that preserves some properties of G up to some error threshold. Such sparsifiers are used to accelerate computations on large networks and to store them.

For solving Laplacian systems on large graphs, we focus on the *spectral sparsifiers*. A spectral sparsifier is a graph whose Laplacian quadratic form is approximately the same as that of the original graph on all real vector inputs. When the sparsifier is on the same vertex set as the original graph and has fewer edges, it is called an *edge sparsifier*. An edge sparsifier can be found in nearly linear time [13, 14]. They can reduce the number of edges in a graph to nearly the same as the number of vertices.

Many large networks, e.g., social networks, traffic networks, are already sparse. Thus, we also need *vertex sparsifiers* which reduce the number of vertices. When the sparsifier is on a strict subset of the vertex set of the original graph, it is a vertex sparsifier. We measure the quality of a vertex sparsifier by its spectral similarity to the Schur complement of the original Laplacian matrix on the subset. Thus, a vertex sparsifier can be viewed as an approximate Schur complement.

Many of the massive graphs in the real world are subject to frequent changes. Examples include the web graph and social networks. In such settings, we would like to maintain the solution to some problem undergo edge insertions, edge deletions, and updating edge weights in time faster than computing from scratch. Thus, a natural question is how to maintain vertex sparsifiers undergo graph modifications. We call an algorithm for this purpose a *dynamic vertex sparsifier*.

Most directly, a dynamic vertex sparsifier can maintain properties of an electric circuit, e.g., effective resistances and edge energies. It can also extend the applications of graph Laplacians to dynamic settings, for example, maintaining the results of a vertex embedding or vertex partitioning algorithms. Theoretically, dynamic vertex sparsification can accelerate the computation of maximum flow or minimum cost flow. Themselves are famous tools for many other graph algorithms.

We present our result that maintains an approximate Schur complement of any given graph with m edges in roughly $\widetilde{O}(m^{3/4})$ time per edge update through fast sampling of random walks [15, 16]. The result in [15] was the first data structure for maintaining key primitives from the Laplacian paradigm for graph algorithms, including dynamic all pair effective resistance and dynamic edge energy estimator, in sublinear time without assumptions on the underlying graph topologies. Using ideas developed from minimum cost flow, the subsequent paper [16] improved the runtime and generalized the result to maintaining solution to a dynamic Laplacian system. The key idea is that we represent electric flow and Schur complement as random walks instead of matrices. Such representation is easy to maintain as we can sample random walks to approximate a Schur complement and use basic data structures like balanced binary search trees to query or edit the sampled walks which are sequences of integers. By augmenting random vertices into the subset, we can bound the size of such representation. The data structures serve as a key component in the computation of maximum flows or minimum cost flows by the interior point method.

Minimum Cost Flow The maxflow problem asks to route the maximum amount of flow between two vertices in a graph such that the flow on any edge is at most its capacity. The more general minimum cost flow problem further asks for the minimum total cost of such a flow. They are studied extensively since the 1950's and have numerous applications in scheduling, image processing, and network science [17, 18]. Recently, most of the algorithms for maximum flow or minimum cost flow apply the interior point method and compute a sequence of slowly changing electrical flows [8, 9, 19, 20, 21, 22, 23, 24, 25]. In each iteration, we augment the current flow by an electric flow. To get improved runtimes, the algorithms either reduce the number of electric flow computations (iterations) or reduce the cost per computation.

We build data structures on our dynamic vertex sparsification algorithm to maintain the set of high energy edges in a dynamic electric network. Such data structures help approximately maintain a flow undergo electric flow augmentations. Our methods solve exact minimum cost flow on general graphs in $\widetilde{O}(m^{1.483})$ time [25, 26]. On planar graphs, we recursively partition the graph by balanced separators to form a *separator tree*. We then maintain information including Schur complements and flow values by lazy tags on the separator tree. By carefully batching the updates in different steps of the interior point method, such data structure implies $\widetilde{O}(m^{1/2+\alpha})$ time minimum cost flow algorithm for α-separable graphs [16]. Because a planar graph is 1/2-separable, the algorithm runs in nearly linear time on planar graphs.

CHAPTER 1
NEARLY TIGHT BOUNDS FOR SANDPILE TRANSIENCE ON GRID

This was joint work [27] with David Durfee, Matthew Fahrbach, and Tao Xiao. We use techniques from the theory of electrical networks to give nearly tight bounds for the transience class of the Abelian sandpile model on the two-dimensional grid up to polylogarithmic factors. The Abelian sandpile model is a discrete process on graphs that is intimately related to the phenomenon of self-organized criticality. In this process, vertices receive grains of sand, and once the number of grains exceeds their degree, they topple by sending grains to their neighbors. The transience class of a model is the maximum number of grains that can be added to the system before it necessarily reaches its steady-state behavior or, equivalently, a recurrent state. Through a more refined and global analysis of electrical potentials and random walks, we give an $O(n^4 \log^4 n)$ upper bound and an $\Omega(n^4)$ lower bound for the transience class of the $n \times n$ grid. Our methods naturally extend to n^d-sized d-dimensional grids to give $O(n^{3d-2} \log^{d+2} n)$ upper bounds and $\Omega(n^{3d-2})$ lower bounds.

1.1 Introduction

The Abelian sandpile model is the canonical dynamical system used to study *self-organized criticality.* In their seminal paper, Bak, Tang, and Wiesenfeld [2] proposed the idea of self-organized criticality to explain several ubiquitous patterns in nature typically viewed as complex phenomena, such as catastrophic events occurring without any triggering mechanism, the fractal behavior of mountain landscapes and coastal lines, and the presence of pink noise in electrical networks and stellar luminosity. Since their discovery, self-organized criticality has been observed in an abundance of disparate scientific fields [28, 29], including condensed matter theory [30], economics [31, 32], epidemiology [33], evolutionary biology [34], high-energy astrophysics [35, 36], materials science [37], neuroscience [38, 39], statistical physics [40, 41], seismology [42], and sociology [43]. A stochastic process is a self-organized critical system if it naturally evolves to highly imbalanced critical states where slight local disturbances can completely alter the current state. For example, when pouring grains of sand onto a table, the pile initially grows in a predictable way, but as it becomes steeper and more unstable, dropping a single grain can spontaneously cause an avalanche that affects the entire pile. Self-organized criticality differs from the critical point of a phase transition in statistical physics, because a self-organizing system does not rely on tuning an external parameter. Instead, it is insensitive to all parameters of the model and simply requires time to reach criticality, which is known as the transient period. Natural events empirically operate at a critical point between order and chaos, thus justifying our study of self-organized criticality.

Dhar [44] developed the *Abelian sandpile model* on finite directed graphs with a sink vertex to further understand self-organized criticality. The Abelian sandpile model, also known as a chip-firing game [45], on a graph with a sink is defined as follows. In each iteration a grain of sand is added to a non-sink vertex of the graph. While any non-sink vertex v contains at least $\deg(v)$ grains of sand, a grain is transferred from v to each of its neighbors. This is known as a *toppling*. When no vertex can be toppled, the state is *stable* and the iteration ends. The sink absorbs and destroys grains, and the presence of a sink guarantees that every toppling procedure eventually stabilizes. An important property of the Abelian sandpile model is that the order in which vertices topple does not affect the stable state. Therefore, as the process evolves it produces a sequence of stable states. From the theory of Markov chains, we say that a stable state is *recurrent* if it can be revisited; otherwise it is *transient*.

In the self-organized critical state of the Abelian sandpile model on a graph with a sink, transient states have zero probability and recurrent states occur with equal probability [44]. As a result, recurrent configurations model the steady-state behavior of the system. Thus, the natural algorithmic question to ask about self-organized criticality for the Abelian sandpile model is:

Question 1.1.1. *How long in the worst case does it take for the process to reach its steady-state behavior or, equivalently, a recurrent state?*

Starting with an empty configuration, if the vertex that receives the grain of sand is chosen uniformly at random in each step, Babai and Gorodezky [46] give a simple solution that is polynomial in the number of edges of the graph using a coupon collector argument. In the worst case, however, an adversary can choose where to place the grain of sand in each iteration. Babai and Gorodezky analyze the *transience class* of the model to understand its worst-case behavior, which is defined as the maximum number of grains that can be added to the empty configuration before the configuration necessarily becomes recurrent. An upper bound for the transience class of a model is an upper bound for the time needed to enter self-organized criticality.

1.1.1 Results

We give the first nearly tight bounds (up to polylogarithmic factors) for the transience class of the Abelian sandpile model on the $n \times n$ grid with all boundary vertices connected to the sink. This model was first studied in depth by Dhar, Ruelle, Sen, and Verma [47], and it has since been the most extensively studied Abelian sandpile model due to its role in algebraic graph theory, theoretical computer science, and statistical physics. Babai and Gorodezky [46] initially established that the transience class of the grid is polynomially bounded by $O(n^{30})$, which was unexpected because there are graphs akin to the grid with exponential transience classes. Choure and Vishwanathan [10] improved the upper bound for the transience class of the grid to $O(n^7)$ and gave a lower bound of $\Omega(n^3)$ by viewing the graph as an electrical network and relating the Abelian sandpile model to random walks on the underlying graph. Moreover, they conjectured that the transience class of the grid is $O(n^4)$, which we answer nearly affirmatively.

Theorem 1.1.2. *The transience class of the Abelian sandpile model on the* $n \times n$ *grid is* $O(n^4 \log^4 n)$.

Theorem 1.1.3. *The transience class of the Abelian sandpile model on the* $n \times n$ *grid is* $\Omega(n^4)$.

Our results establish how fast the system reaches its steady-state behavior in the adversarial case, and they corroborate empirical observations about natural processes exhibiting self-organized criticality. Our analysis directly generalizes to higher-dimensional cases, giving the following result.

Theorem 1.1.4. *For any integer* $d \geq 2$*, the transience class of the Abelian sandpile model on the* n^d*-sized* d*-dimensional grid is* $O(n^{3d-2} \log^{d+2} n)$ *and* $\Omega(n^{3d-2})$.

In addition to addressing the main open problem in [46] and [10], we begin to shed light on Babai and Gorodezky's inquiry about sequences of graphs that exhibit polynomially bounded transience classes. Specifically, for hypergrids (a family of locally finite graphs with high symmetry) we quantify how the transience class grows as a function of the size and local degree of the graph. When viewed through the lens of graph connectivity, such transience class bounds are surprising because grids have low algebraic connectivity, yet we are able to make global structural arguments using only the fact that grids have low maximum effective resistance when viewed as electrical networks. By doing this, we avoid spectral analysis of the grid and evade the main obstacle in Choure and Vishwanathan's analysis. Our techniques suggest that low effective resistance captures a different but similar phenomenon to high conductance and high edge expansion for stochastic processes on graphs. This distinction between the role of a graph's effective resistance and conductance could be an important step forward for building a theory for discrete diffusion processes analogous to the mixing time of Markov chains. We also believe our results have close connections to randomized, distributed optimization algorithms for flow problems [48, 49, 50, 51, 52, 53], where the dynamics of self-adjusting sandpiles (a Physarum slime mold in their model) are governed by electrical flows and resistances.

1.1.2 Techniques

Our approach is motivated by the method of Choure and Vishwanathan [10] for bounding the transience class of the Abelian sandpile model on graphs using electrical potential theory and the analysis of random walks. Viewing the graph as an electrical network with a voltage source at some vertex and a grounded sink, we give more accurate voltage estimates by carefully considering the geometry of the grid. We use several lines of symmetry to compare escape probabilities of random walks with different initial positions, resulting in a new technique for comparing vertex potentials. These geometric arguments can likely be generalized to other lattice-based graphs. As a result, we get empirically tight inequalities for the sum of all vertex potentials in the grid and the voltage drop between opposite corners of the network.

For many of our voltage bounds, we interpret a vertex potential as an escape probability and decouple the corresponding two-dimensional random walks on the grid into independent one-dimensional random walks on a path graph. Decoupling is the standout technique in this paper, because it allows us to apply classical results about simple symmetric random walks on $\mathbb{Z}$ (such as the reflection principle), which we extend as needed using conditional probability arguments. By reducing from two-dimensional random walks to one-dimensional walks, we utilize standard probabilistic tools including Stirling's approximation, Chernoff bounds, and the negative binomial distribution. Since we consider many different kinds of events in our analysis, Section 1.5 is an extensive collection of probability inequalities for symmetric t-step random walks on $\mathbb{Z}$ with various boundary conditions. We noticed that some of these inequalities are directly related to problems in enumerative combinatorics without closed-form solutions [54].

Lastly, we leverage well-known results about effective resistances of the $n \times n$ grid when viewed as an electrical network. We follow Choure and Vishwanathan in using the potential reciprocity theorem to swap the voltage source with any other non-sink vertex, but we use this theorem repeatedly with the fact that the effective resistance between any non-sink vertex and the sink is bounded between a constant and $O(\log n)$. This approach enables us to analyze tractable one-dimensional random walk problems at the expense of polylogarithmic factors.

1.2 Preliminaries

1.2.1 Abelian Sandpile Model

Let $G = (V, E)$ be an undirected multigraph. Throughout this paper all of the graphs we consider have a sink vertex denoted by v_{sink}. The *Abelian sandpile model* is a dynamical system on a graph G used to study the phenomenon of self-organized criticality. A *configuration* σ on G in the Abelian sandpile model is a vector of nonnegative integers indexed by the non-sink vertices such that $\sigma(v)$ denotes the number of grains of sand on vertex v. We say that a configuration is *stable* if $\sigma(v) < \deg(v)$ for all non-sink vertices and *unstable* otherwise. An unstable configuration σ moves towards stabilization by selecting a vertex v such that $\sigma(v) \geq \deg(v)$ and sending one grain of sand from v to each of its neighboring vertices. This event is called a *toppling* of v, and it creates a new configuration σ' such that $\sigma'(v) = \sigma(v) - \deg(v)$, $\sigma'(u) = \sigma(u) + 1$ for all vertices u adjacent to v, and $\sigma'(u) = \sigma(u)$ for all remaining vertices. This procedure eventually reaches a stable state because G has a sink. Moreover, the order in which vertices topple does not affect the final stable state. The initial configuration of the Abelian sandpile model is typically the zero vector, and in each iteration a grain of sand is placed at a vertex (chosen either deterministically or uniformly at random). The system evolves by stabilizing the configuration and then receiving another grain of sand.

A stable configuration σ is *recurrent* if the process can eventually return to σ. Any state that is not recurrent is *transient*. Note that once the system enters a recurrent state, it can never visit a transient state. Babai and Gorodezky [46] introduced the following notion to upper bound on the number of steps for the Abelian sandpile model to reach self-organized criticality.

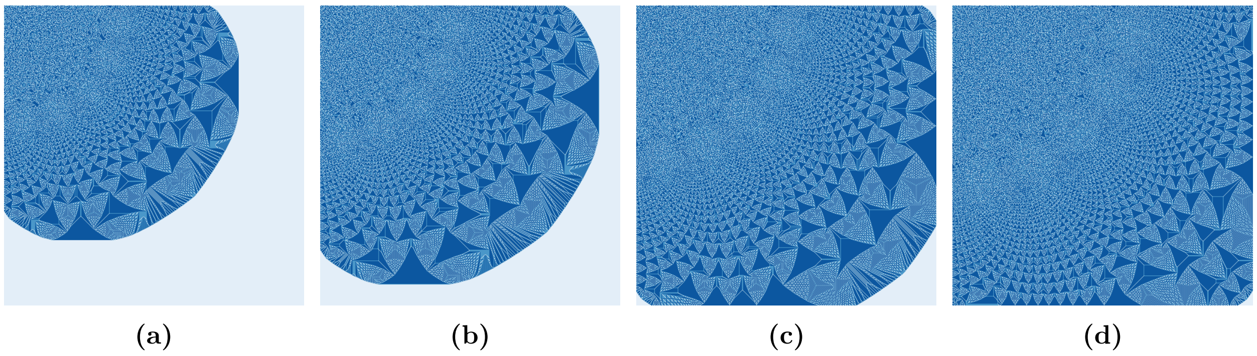

Figure 1.1: Configurations of the Abelian sandpile model on the 500×500 grid during its transience period after placing (a) 10^{10} (b) $2 \cdot 10^{10}$ (c) $4 \cdot 10^{10}$ (d) $8 \cdot 10^{10}$ grains of sand at $(1, 1)$.

Definition 1.2.1. The *transience class* of the Abelian sandpile model of G is the maximum number of grains that can be added to the empty configuration before the configuration necessarily becomes recurrent. We denote this quantity by $\mathrm{tcl}(G)$.

In Figure 1.1 we illustrate the transient configurations in the *transient period* of the Abelian sandpile model as it advances towards its critical state. We specifically show in this paper that by repeatedly placing grains of sand in the top-left corner of the grid, we maximize the length of the transience period up to a polylogarithmic factor.

In earlier related works, Björner, Lovász, and Shor [45] studied a variant of this process without a sink and characterized the conditions needed for stabilization to terminate. They also related the spectrum of the underlying graph to the rate at which the system converges. In the model we study, an observation by Dhar [44] and Kirchoff's theorem show that the stable recurrent states of the system are in bijection with the spanning trees of G. Choure and Vishwanathan [10] show that if every vertex in a configuration has toppled then the configuration is necessarily recurrent, which we use to bound the transience class. The Abelian sandpile model also has broad applications to algorithms and statistical physics, including a direct relation to the q-state Potts model and Markov chain Monte Carlo algorithms for sampling random spanning trees [44, 55, 56, 57, 58]. For a comprehensive survey on the Abelian sandpile model, see [59].

1.2.2 Random Walks on Graphs

A walk w on G is a sequence of vertices $w^{(0)}, w^{(1)}, \ldots, w^{(t_{\max})}$ such that every $w^{(t+1)}$ is a neighbor of $w^{(t)}$. We let $t_{\max} = |w|$ denote the length of the walk. A random walk is a process that begins at vertex $w^{(0)}$, and at each time step t transitions from $w^{(t)}$ to $w^{(t+1)}$ such that $w^{(t+1)}$ is chosen uniformly at random from the neighbors of $w^{(t)}$. Note that this definition naturally captures the effect of walking on a multigraph. We consider walks that continue until reaching a set of sink vertices. It will be convenient for our analysis to formally define these following families of walks.

Definition 1.2.2. For any set of starting vertices S and terminating vertices T in the graph G, let

$$\mathcal{W}(S \to T) \overset{\text{def}}{=} \left\{ w : w^{(0)} \in S,\ w^{(i)} \notin T \cup \{v_{\text{sink}}\} \text{ for } 0 \le i \le |w| - 1, \text{ and } w^{(|w|)} \in T \right\}$$

be the set of finite walks from S to T.

Observe that with this definition, walks w of length 0 are permissible if we have $w^{(0)} \in S \cap T$. Throughout the paper it will be convenient to consider random walks from one vertex u to another vertex v or the pair $\{v, v_{\text{sink}}\}$. We denote these cases by the notation $\mathcal{W}(u \to v) = \mathcal{W}(\{u\} \to \{v\})$. If walks on multiple graphs are being considered, we use $\mathcal{W}^G(u \to v)$ to denote the underlying graph. Lastly, we consider the set of nonterminating walks in our analysis, so it will be useful to define

$$\mathcal{W}(S) \overset{\text{def}}{=} \left\{ w \in \prod_{i=0}^{\infty} V : w^{(0)} \in S \text{ and } w^{(i)} \neq v_{\text{sink}} \text{ for any } i \ge 0 \right\},$$

which is the set of infinite walks from S. An analogous definition follows when $S = \{u\}$.

The focus of our study is the $n \times n$ grid graph, denoted by $\textsc{Square}_n$. Similar to previous works, we do not follow the usual graph-theoretic convention of using n to denote vertex count. We formally define the one-dimensional projection of $\textsc{Square}_n$ to be $\textsc{Path}_n$, which has the vertex set $\{1, 2, \ldots, n\} \cup \{v_{\text{sink}}\}$ and edges between i and $i+1$ for every $1 \le i \le n-1$, as well as two edges connecting v_{sink} to 1 and n. Thus, v_{sink} can be viewed as 0 and $n+1$. If we remove the sink (which can be thought of as letting $v_{\text{sink}} = \pm\infty$) then the resulting graph is the one-dimensional line with vertices $i \in \mathbb{Z}$ and edges between every pair $(i, i+1)$. We denote this graph by $\textsc{Line}$ and use the indices i, j, and k to represent its vertices. Analyzing random walks on $\textsc{Line}$ is critical to our analysis, and it will be useful to record the minimum and maximum position of t-step walks.

Definition 1.2.3. For an initial position $i \in \mathbb{Z}$ and walk $w \in \mathcal{W}(i)$ on $\textsc{Line}$, let the t-step minimum and maximum positions be

$$\min_{\le t}(w) \overset{\text{def}}{=} \min_{0 \le \widehat{t} \le t} w^{(\widehat{t})}$$

and

$$\max_{\le t}(w) \overset{\text{def}}{=} \max_{0 \le \widehat{t} \le t} w^{(\widehat{t})}.$$

We construct $\textsc{Square}_n$ similarly. Its vertices are $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\} \cup \{v_{\text{sink}}\}$, and its edges connect any pair of vertices that differ in one coordinate. Vertices on the boundary have edges connected to v_{sink} so that every non-sink vertex has degree 4. With this definition of $\textsc{Square}_n$, each corner vertex has two edges to v_{sink} and non-corner vertices on the boundary share one edge with v_{sink}. Since all vertices correspond to pairs of coordinates, we use the vector notation $\boldsymbol{u} = (\boldsymbol{u}_1, \boldsymbol{u}_2)$ to denote coordinates on the grid, as it easily extends to higher dimensions. Throughout the paper, boldfaced variables denote vectors. A t-step random walk on $\textsc{Square}_n$ naturally induces a $(t_{\max} + 1) \times 2$ matrix. We can decouple such a walk w into its horizontal and vertical components, using the notation w_1 for the change in position of the first coordinate and w_2 for the change in position of the second coordinate. In general we use the notation $w_{\widehat{d}}$ to index into one of the dimensions $1 \le \widehat{d} \le d$ of a d-dimensional walk. We do not record duplicate positions when the walk takes a step in a dimension different than $\widehat{d}$, so we have $|w| = |w_1| + |w_2| - 1$ when $d = 2$ since the initial vertex is present in both w_1 and w_2.

1.2.3 Electrical Networks

Vertex potentials are central to our analysis. They have close connections with electrical voltages and belong to the class of harmonic functions [1]. We analyze their relation to the transience class of general graphs. For any non-sink vertex u, we can define a unique potential vector $\boldsymbol{\pi}_u$ such that $\boldsymbol{\pi}_u(u) = 1$, $\boldsymbol{\pi}_u(v_{\text{sink}}) = 0$, and for all other vertices $v \in V \setminus \{u, v_{\text{sink}}\}$ we have

$$\boldsymbol{\pi}_u(v) = \frac{1}{\deg(v)} \sum_{x \sim v} \boldsymbol{\pi}_u(x),$$

where the sum is over the neighbors of v. Thus, $\boldsymbol{\pi}_u(v)$ denotes the potential at v when the boundary conditions are set to 1 at u and 0 at the sink. Since we analyze potential vectors in both $\textsc{Path}_n$ and $\textsc{Square}_n$, we use superscripts to denote the graph when context is unclear.

Choure and Vishwanathan showed that we can give upper and lower bounds on the transience class using potentials, which we rephrase in the following theorem.

Theorem 1.2.4 ([10])**.** *If G is a graph such that the degree of every non-sink vertex is bounded by a constant,*

$$\operatorname{tcl}(G) = O\left(\max_{u,v \in V \setminus \{v_{\text{sink}}\}} \left(\sum_{x \in V} \boldsymbol{\pi}_u(x) \right) \boldsymbol{\pi}_u(v)^{-1} \right)$$

and

$$\operatorname{tcl}(G) = \Omega\left(\max_{u,v \in V \setminus \{v_{\text{sink}}\}} \boldsymbol{\pi}_u(v)^{-1} \right).$$

All non-sink vertices have degree 4, so we can apply Theorem 1.2.4 to $\textsc{Square}_n$.

The following combinatorial interpretations of potentials as random walks is fundamental to our investigation of the transience class of $\textsc{Square}_n$. Note that we use boldfaced vector variables for non-sink vertices in $\textsc{Square}_n$ as they can be identified by their coordinates.

Fact 1.2.5 ([1])**.** *For any graph G and non-sink vertex u, the potential $\boldsymbol{\pi}_u(v)$ is the probability of a random walk starting at v and reaching u before v_{sink}.*

Lemma 1.2.6. *Let $\boldsymbol{u}$ be a non-sink vertex of* $\textsc{Square}_n$. *For any vertex $\boldsymbol{v}$, we have*

$$\boldsymbol{\pi}_{\boldsymbol{u}}\left(\boldsymbol{v}\right) = \sum_{w \in \mathcal{W}(\boldsymbol{v} \to \boldsymbol{u})} 4^{-|w|}.$$

We defer the proof of Lemma 1.2.6 to Section 1.7.1.

A systematic treatment of the connection between random walks and electrical networks can be found in the monograph by Doyle and Snell [1] or the survey by Lovász [60]. The following lemma is a key result for our investigation, which states that a voltage source and a measurement point can be swapped at the expense of a distortion in the potential equal to the ratio of the effective resistances between the sink and the two vertices. The effective resistance between a pair of vertices u and v, denoted as $\mathcal{R}_{\text{eff}}(u, v)$, can be formalized in several ways. In the electrical interpretation [1], effective resistance can be viewed as the voltage needed to send one unit of current from u to v if every edge in G is a unit resistor. For a linear algebraic definition of effective resistance see [61].

Lemma 1.2.7 ([10, Potential Reciprocity])**.** *Let G be a graph (not necessarily degree-bounded) with sink v_{sink}. For any pair of vertices u and v, we have*

$$\mathcal{R}_{\text{eff}}\left(v_{\text{sink}}, u\right) \boldsymbol{\pi}_u(v) = \mathcal{R}_{\text{eff}}\left(v_{\text{sink}}, v\right) \boldsymbol{\pi}_v(u).$$

This statement is particularly powerful for $\textsc{Square}_n$, because the effective resistance between any pair of vertices is bounded between a constant and $O(\log n)$. The following lemma makes use of a classical result that can be obtained using Thompson's principle of the electrical flow [1].

Lemma 1.2.8. *For any non-sink vertex $\boldsymbol{u}$ in* $\textsc{Square}_n$*,*

$$1/4 \leq \mathcal{R}_{\text{eff}}\left(v_{\text{sink}}, \boldsymbol{u}\right) \leq 2 \log n + 1.$$

We give the proof of Lemma 1.2.8 in Section 1.7.1. When used together, Lemma 1.2.7 and Lemma 1.2.8 imply the following result, which allows us to conveniently swap the source vertex when computing potentials.

Lemma 1.2.9. *For any non-sink vertices* $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *in* $\textsc{Square}_n$*, we have*

$$\boldsymbol{\pi_u}(\boldsymbol{v}) \leq (8 \log n + 4)\, \boldsymbol{\pi_v}(\boldsymbol{u})\,.$$

Voltages and flows on electrical networks are central to many recent developments in algorithmic graph theory (e.g. modern maximum flow algorithms and interior point methods [19, 21]). The convergence of many of these algorithms depend on the extremal voltage values of the electrical flow that they construct. As a result, we believe some of our techniques are relevant to the grid-based instantiations of these algorithms.

1.3 Upper Bounding the Transience Class

In this section we prove the upper bound in Theorem 1.1.2 for the transience class of the Abelian sandpile model on the square grid. Our proof follows the framework of Choure and Vishwanathan in that we use Theorem 1.2.4 to reduce the proof to bounding the following two quantities for any non-sink vertex $\boldsymbol{u} \in V(\textsc{Square}_n)$:

- We upper bound the potential sum $\sum_{\boldsymbol{v} \in V} \boldsymbol{\pi_u}(\boldsymbol{v})$.
- We lower bound the potential $\boldsymbol{\pi_u}(\boldsymbol{v})$ for all non-sink vertices $\boldsymbol{v}$.

By symmetry we assume without loss of generality that $\boldsymbol{u}$ is in the top-left quadrant of $\textsc{Square}_n$ (i.e., we have $1 \leq \boldsymbol{u}_1, \boldsymbol{u}_2 \leq \lceil n/2 \rceil$). The principal idea is to use reciprocity from Lemma 1.2.7 and effective resistance bounds from Lemma 1.2.8 to swap source vertices and bound $\boldsymbol{\pi_v}(\boldsymbol{u})$ instead, at the expense of a $O(\log n)$ factor. The second key idea is to interpret potentials as random walks using Fact 1.2.5 and then decouple two-dimensional walks on $\textsc{Square}_n$ into separate horizontal and vertical one-dimensional walks on $\textsc{Path}_n$. Using well-studied properties of one-dimensional random walks, we achieve nearly tight bounds on $\text{tcl}(\textsc{Square}_n)$.

We note that there is a natural trade-off in the choice of the source vertex $\boldsymbol{u}$. Setting $\boldsymbol{u}$ near the boundary decreases vertex potentials because a random walk has a higher probability of escaping to v_{sink} instead of $\boldsymbol{u}$. This improves the upper bound of the sum of vertex potentials, but it weakens the lower bound of the minimum vertex potential. For vertices $\boldsymbol{u}$ that are not near the boundary, the opposite is true. Therefore, we account for the choice of $\boldsymbol{u}$ in our bounds.

1.3.1 Upper Bounding the Potential Sum

Lemma 1.3.1. *For any non-sink vertex $\boldsymbol{u}$ in* $\textsc{Square}_n$*, we have*

$$\sum_{\boldsymbol{v} \in V} \boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) = O\left(\boldsymbol{u}_1 \boldsymbol{u}_2 \log^3 n\right).$$

Proof. We use Fact 1.2.5 and Lemma 1.2.6 to interpret vertex potentials as random walks. We can omit v_{sink} because any random walk starting there immediately terminates. By Lemma 1.2.9,

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) = O\left(\boldsymbol{\pi}_{\boldsymbol{v}}(\boldsymbol{u}) \log n\right),$$

so we apply the random walk interpretation to potentials starting at $\boldsymbol{u}$ instead of $\boldsymbol{v}$. Consider one such walk $w \in \mathcal{W}(\boldsymbol{u} \to \boldsymbol{v})$ and its one-dimensional decompositions w_1 and w_2. The probability of a walk from $\boldsymbol{u}$ reaching $\boldsymbol{v}$ is equal to the probability that two interleaved walks in $\textsc{Path}_n$ starting at $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are present on $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, respectively, at the same time before either hits their one-dimensional sink $v_{\text{sink}} = \{0, n+1\}$.

If we remove the restriction that these walks are present on $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ at the same time and only require that they visit $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ before hitting v_{sink}, then each of these less restricted walks w_d belongs to the class $\mathcal{W}^{\textsc{Path}_n}(\boldsymbol{u}_d \to \boldsymbol{v}_d)$. Viewing a walk w on $\textsc{Square}_n$ as infinite walk on the lattice $\mathbb{Z}^2$ induces independence between w_1 and w_2. Thus, we obtain the upper bound

$$\begin{aligned}
\boldsymbol{\pi}_{\boldsymbol{v}}(\boldsymbol{u}) &= \Pr_{w \sim \mathcal{W}^{\mathbb{Z}^2}(\boldsymbol{u})}\left[w \text{ hits } \boldsymbol{v} \text{ before leaving } \textsc{Square}_n\right] \\
&\leq \Pr_{w \sim \mathcal{W}^{\mathbb{Z}^2}(\boldsymbol{u})}[w_1 \text{ hits } \boldsymbol{v}_1 \text{ before } v_{\text{sink}} \text{ and } w_2 \text{ hits } \boldsymbol{v}_2 \text{ before } v_{\text{sink}}] \\
&= \Pr_{w \sim \mathcal{W}^{\mathbb{Z}^2}(\boldsymbol{u})}\left[w_1 \text{ hits } \boldsymbol{v}_1 \text{ before } v_{\text{sink}}\right] \cdot \Pr_{w \sim \mathcal{W}^{\mathbb{Z}^2}(\boldsymbol{u})}\left[w_2 \text{ hits } \boldsymbol{v}_2 \text{ before } v_{\text{sink}}\right] \\
&= \boldsymbol{\pi}_{\boldsymbol{v}_1}^{\textsc{Path}_n}(\boldsymbol{u}_1) \cdot \boldsymbol{\pi}_{\boldsymbol{v}_2}^{\textsc{Path}_n}(\boldsymbol{u}_2).
\end{aligned}$$

Summing over all choices of $\boldsymbol{v} = (\boldsymbol{v}_1, \boldsymbol{v}_2)$ gives

$$\sum_{\boldsymbol{v} \in V} \boldsymbol{\pi}_{\boldsymbol{v}}\left(\boldsymbol{u}\right) \leq \left(\sum_{\boldsymbol{v}_1=1}^{n} \boldsymbol{\pi}_{\boldsymbol{v}_1}^{\textsc{Path}_n}\left(\boldsymbol{u}_1\right)\right)\left(\sum_{\boldsymbol{v}_2=1}^{n} \boldsymbol{\pi}_{\boldsymbol{v}_2}^{\textsc{Path}_n}\left(\boldsymbol{u}_2\right)\right).$$

The potentials of vertices in $\textsc{Path}_n$ have the following closed-form solution, as shown in [1]:

$$\boldsymbol{\pi}_{\boldsymbol{v}_1}^{\textsc{Path}_n}\left(\boldsymbol{u}_1\right) = \begin{cases} \frac{n+1-\boldsymbol{u}_1}{n+1-\boldsymbol{v}_1} & \text{if } \boldsymbol{v}_1 \leq \boldsymbol{u}_1, \\ \frac{\boldsymbol{u}_1}{\boldsymbol{v}_1} & \text{if } \boldsymbol{v}_1 > \boldsymbol{u}_1. \end{cases}$$

Splitting the sum at $\boldsymbol{u}_1$ and using the fact that potentials are escape probabilities, we have

$$\sum_{\boldsymbol{v}_1=1}^{n} \boldsymbol{\pi}_{\boldsymbol{v}_1}^{\textsc{Path}_n}(\boldsymbol{u}_1) \leq \boldsymbol{u}_1 + \sum_{\boldsymbol{v}_1=\boldsymbol{u}_1+1}^{n} \frac{\boldsymbol{u}_1}{\boldsymbol{v}_1} = O(\boldsymbol{u}_1 \log n).$$

We similarly obtain an upper bound of $O(\boldsymbol{u}_2 \log n)$ in the other dimension. These bounds along with the initial $O(\log n)$ overhead from swapping $\boldsymbol{u}$ and $\boldsymbol{v}$ gives the desired upper bound. □

1.3.2 Lower Bounding the Minimum Potential

The more involved part of this paper proves a lower bound for the minimum vertex potential $\min_{\boldsymbol{v} \in V \setminus \{v_{\text{sink}}\}} \boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v})$ as a function of a fixed vertex $\boldsymbol{u} = (\boldsymbol{u}_1, \boldsymbol{u}_2)$. Recall that we assumed without loss of generality that $\boldsymbol{u}$ is in the top-left quadrant of $\textsc{Square}_n$. We first prove that the minimum potential occurs at vertex (n, n), the corner farthest from $\boldsymbol{u}$. Using Lemma 1.2.9 to swap $\boldsymbol{u}$ and (n, n) at the expense of a $\Omega(1/\log n)$ factor, we reduce the problem to giving a lower bound for $\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})$. Then we decompose walks $w \in \mathcal{W}(\boldsymbol{u} \to \{(n, n), v_{\text{sink}}\})$ into their one-dimensional walks $w_1 \in \mathcal{W}^{\textsc{Path}_n}(\boldsymbol{u}_1)$ and $w_2 \in \mathcal{W}^{\textsc{Path}_n}(\boldsymbol{u}_2)$, and we interpret $\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})$ as the probability that the individual processes w_1 and w_2 are present on n at the same time before either walk leaves the interval $[1, n]$. Walks on $\textsc{Line}$ that meet at n before leaving the interval $[1, n]$ are equivalent to walks on $\textsc{Path}_n$ that meet at n before terminating at v_{sink}. Lastly, we use conditional probabilities to analyze walks on $\textsc{Line}$ instead of walks on $\textsc{Path}_n$ in order to leverage well-known facts about simple symmetric random walks.

To lower bound the desired probability $\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})$, we show that a subset of $\mathcal{W}(\boldsymbol{u} \to (n,n))$ of interleaved one-dimensional walks starting at $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ that first reach n in approximately the same number of steps has a sufficient amount of probability mass. We prove this by observing that the distributions of the number of steps for the walks to first reach n without leaving the interval $[1,n]$ are concentrated around $(n-\boldsymbol{u}_1)^2$ and $(n-\boldsymbol{u}_2)^2$, respectively. Consequently, we show that this distribution is approximately uniform in an $\Theta(n^2)$ length interval, with each t-step having probability $\Omega(\boldsymbol{u}_1/n^3)$ and $\Omega(\boldsymbol{u}_2/n^3)$. We then use Chernoff bounds to show that both walks take approximately the same number of steps with constant probability. Combining these facts, we give the desired lower bound $\Omega(\boldsymbol{u}_1\boldsymbol{u}_2/n^4)$.

Opposite Corner Minimizes Potential

We first show that the corner vertex (n,n) has the minimum potential up to a constant factor. Viewing potentials as escape probabilities, we utilize the geometry of the grid to construct maps between sets of random walks that prove the potential of an interior vertex is greater than its axis-aligned projection to the boundary of the grid. We defer the proof of Lemma 1.3.2 to Section 1.7.2.

Lemma 1.3.2. *If $\boldsymbol{u}$ is a vertex in the top-left quadrant of $\textsc{Square}_n$, then for any non-sink vertex $\boldsymbol{v}$ we have*

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) \geq \frac{1}{16}\boldsymbol{\pi}_{\boldsymbol{u}}((n,n)).$$

Lower Bounding Corner Potential

By decomposing two-dimensional walks on $\textsc{Square}_n$ that start at $\boldsymbol{u}$ into one-dimensional walks on $\textsc{Line}$, our lower bound relies on showing that there is a $\Theta(n^2)$ length interval such that each one-dimensional walk of a fixed length in this interval has probability $\Omega(\boldsymbol{u}_1/n^3)$ or $\Omega(\boldsymbol{u}_2/n^3)$, respectively, of remaining above 0 and reaching n for the first time upon termination. For our purposes, lower bounds for this probability will suffice, and they follow from the following key property for one-dimensional walks that we prove in Section 1.5.

Lemma 1.3.3. *Let $n \in \mathbb{Z}_{\geq 1}$ and $1 \leq i \leq \lceil n/2 \rceil$ be any starting position. For any constant $c > 4$ and any $t \in \mathbb{Z}$ such that $n^2/c \leq t \leq n^2/4$ with $t \equiv n - i \pmod 2$, a simple symmetric random walk*

w on $\mathbb{Z}$ satisfies

$$\Pr_{w\sim\mathcal{W}^{\textsc{Line}}(i)}\left[w^{(t)}=n, \max_{\leq t}(w)=n, \text{and } \min_{\leq t}(w)\geq 1\right]\geq e^{-2-2c}\frac{i}{n^3}.$$

Using Lemma 1.3.3 with the following lemma, we give a lower bound for $\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})$, the probability that a walk starting from $\boldsymbol{u}$ reaches (n,n) before v_{sink}. Lemma 1.3.4 is a consequence of a Chernoff bound, and we defer its proof to Section 1.7.2.

Lemma 1.3.4. *For all $n\geq 10$, we have*

$$\min\left\{\frac{1}{2^n}\sum_{\substack{k=\lceil\frac{n}{4}\rceil\\ k\text{ odd}}}^{\lfloor\frac{3n}{4}\rfloor}\binom{n}{k}, \frac{1}{2^n}\sum_{\substack{k=\lceil\frac{n}{4}\rceil\\ k\text{ even}}}^{\lfloor\frac{3n}{4}\rfloor}\binom{n}{k}\right\}\geq\frac{2}{5}.$$

Lemma 1.3.5. *For all $n\geq 10$ and any vertex $\boldsymbol{u}$ in the top-left quadrant of $\textsc{Square}_n$, we have*

$$\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})\geq e^{-100}\frac{\boldsymbol{u}_1\boldsymbol{u}_2}{n^4}.$$

Proof. We decouple each walk $w\in\mathcal{W}(\boldsymbol{u}\to(n,n))$ into its horizontal walk $w_1\in\mathcal{W}^{\textsc{Line}}(\boldsymbol{u}_1)$ and vertical walk $w_2\in\mathcal{W}^{\textsc{Line}}(\boldsymbol{u}_2)$. The potential $\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})$ can be interpreted as the probability that w_1 and w_2 visit n at the same time before either leaves the interval $[1,n]$. We can further decompose t-step walks on $\textsc{Square}_n$ into those that take t_1 steps in the horizontal direction and t_2 in the vertical direction. Considering restricted instances where w_1 and w_2 visit n exactly once, we obtain the following bound by Lemma 1.2.6:

$$\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})\geq\sum_{\substack{w\sim\mathcal{W}(\boldsymbol{u}\to(n,n))\\ w_1\text{ hits } n\text{ exactly once}\\ w_2\text{ hits } n\text{ exactly once}}}4^{-|w|}. \tag{1.1}$$

Accounting for all the ways that two one-dimensional walks can be interleaved, the right hand side of (1.1) is

$$\sum_{t_1,t_2\geq 0} \frac{\binom{t_1+t_2}{t_1}}{4^{t_1+t_2}} \left(\#\text{ of } t_1\text{-step walks from } \boldsymbol{u}_1 \text{ that stay in } [1,n-1] \text{ and terminate at } n\right)$$
$$\cdot \left(\#\text{ of } t_2\text{-step walks from } \boldsymbol{u}_2 \text{ that stay in } [1,n-1] \text{ and terminate at } n\right).$$

Observing that

$$\Pr_{w_1\sim\mathcal{W}(\boldsymbol{u}_1)}\left[w_1^{(t_1)} = n,\ \max_{\leq t_1-1}(w) = n-1,\ \min_{\leq t_1-1}(w)\geq 1\right]$$
$$= \frac{\left(\#\text{ of } t_1\text{-step walks from } \boldsymbol{u}_1 \text{ that stay in } [1,n-1] \text{ and terminate at } n\right)}{2^{t_1}},$$

it follows from (1.1) that

$$\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u}) \geq \sum_{t_1,t_2\geq 0} \frac{\binom{t_1+t_2}{t_1}}{2^{t_1+t_2}} \Pr_{w_1\sim\mathcal{W}(\boldsymbol{u}_1)}\left[w_1^{(t_1)} = n,\ \max_{\leq t_1-1}(w) = n-1,\ \min_{\leq t_1-1}(w)\geq 1\right]$$
$$\cdot \Pr_{w_2\sim\mathcal{W}(\boldsymbol{u}_2)}\left[w_2^{(t_2)} = n,\ \max_{\leq t_2-1}(w) = n-1,\ \min_{\leq t_2-1}(w)\geq 1\right].$$

By our choice of n and $\boldsymbol{u}$, the right hand side of inequality above equals

$$\sum_{t_1,t_2\geq 5} \frac{\binom{t_1+t_2}{t_1}}{2^{t_1+t_2}} \left(\frac{1}{2}\Pr_{w_1\sim\mathcal{W}(\boldsymbol{u}_1)}\left[w_1^{(t_1-1)} = n-1,\ \max_{\leq t_1-1}(w) = n-1,\ \min_{\leq t_1-1}(w)\geq 1\right]\right)$$
$$\cdot \left(\frac{1}{2}\Pr_{w_2\sim\mathcal{W}(\boldsymbol{u}_2)}\left[w_2^{(t_2-1)} = n-1,\ \max_{\leq t_2-1}(w) = n-1,\ \min_{\leq t_2-1}(w)\geq 1\right]\right). \tag{2}$$

Letting $t = t_1 + t_2$, we further refine the set of two-dimensional walks so that $t \in [\frac{1}{5}n^2, \frac{1}{4}n^2]$ and $t_1, t_2 \in [\frac{1}{4}t, \frac{3}{4}t]$ while capturing a sufficient amount of probability mass for a useful lower bound. Note that the parities of t_1 and t_2 satisfy $t_1 \equiv n - \boldsymbol{u}_1 \pmod 2$ and $t_2 \equiv n - \boldsymbol{u}_2 \pmod 2$ for valid walks. Let I be an indexing of all such pairs (t_1, t_2). Working from (2), we have

$$\begin{aligned}
\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u}) &\geq \sum_{(t_1,t_2)\in I} \frac{\binom{t_1+t_2}{t_1}}{2^{t_1+t_2}} \left(\frac{1}{2}e^{-2-2(20)}\frac{\boldsymbol{u}_1}{n^3}\right)\left(\frac{1}{2}e^{-2-2(20)}\frac{\boldsymbol{u}_2}{n^3}\right) \\
&\geq e^{-84}\cdot\frac{\boldsymbol{u}_1\boldsymbol{u}_2}{4n^6} \sum_{\substack{t\in\left[\frac{n^2}{5},\frac{n^2}{4}\right] \\ t\equiv \boldsymbol{u}_1+\boldsymbol{u}_2 \pmod 2}} \frac{2}{5} \\
&\geq e^{-84}\cdot\frac{\boldsymbol{u}_1\boldsymbol{u}_2}{4n^6}\cdot\frac{n^2}{50}\cdot\frac{2}{5} \\
&\geq e^{-100}\cdot\frac{\boldsymbol{u}_1\boldsymbol{u}_2}{n^4}.
\end{aligned}$$

For the first inequality, we can apply Lemma 1.3.3 because

$$\frac{1}{20}n^2 \leq t_1, t_2 \leq \frac{3}{16}n^2.$$

For the second inequality, we group pairs (t_1, t_2) by their sum $t = t_1 + t_2$ and apply Lemma 1.3.4. The number of $t \in [\frac{1}{5}n^2, \frac{1}{4}n^2]$ with either parity restriction is at least $\lfloor\frac{1}{40}n^2\rfloor \geq \frac{1}{50}n^2$. □

1.3.3 Proof of Theorem 1.1.2

We now combine the upper bound for the sum of potentials given by Lemma 1.3.1 and the lower bounds in Section 1.3.2 to obtain the overall upper bound for the transience class of the grid.

Proof. For any $\boldsymbol{u} = (\boldsymbol{u}_1, \boldsymbol{u}_2)$ in the top-left quadrant of $\textsc{Square}_n$, we have

$$\begin{aligned}
\max_{\boldsymbol{u},\boldsymbol{v}\in V\setminus\{v_{\text{sink}}\}} \left(\sum_{\boldsymbol{x}\in V}\pi_{\boldsymbol{u}}(\boldsymbol{x})\right)\pi_{\boldsymbol{u}}(\boldsymbol{v})^{-1} &\leq \max_{\boldsymbol{u}\in V\setminus\{v_{\text{sink}}\}} \left(\sum_{\boldsymbol{x}\in V}\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{x})\right)\frac{16}{\boldsymbol{\pi}_{\boldsymbol{u}}((n,n))} \\
&= \max_{\boldsymbol{u}\in V\setminus\{v_{\text{sink}}\}} \left(\sum_{\boldsymbol{x}\in V}\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{x})\right)\frac{O(\log n)}{\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})} \\
&= \max_{\boldsymbol{u}\in V\setminus\{v_{\text{sink}}\}} O\left(\boldsymbol{u}_1\boldsymbol{u}_2\log^3 n\right) O\left(\frac{n^4\log n}{\boldsymbol{u}_1\boldsymbol{u}_2}\right) \\
&= O\left(n^4\log^4 n\right).
\end{aligned}$$

The first inequality follows from Lemma 1.3.2, the second from Lemma 1.2.9, and the third from

Lemma 1.3.5 and Lemma 1.3.1. The result follows from Theorem 1.2.4. □

1.4 Lower Bounding the Transience Class

In this section we lower bound $\mathrm{tcl}(\textsc{Square}_n)$ using techniques similar to those in Section 1.3. Since the lower bound in Theorem 1.2.4 considers the maximum inverse vertex potential over all pairs of non-sink vertices u and v, it suffices to upper bound $\pi_{(n,n)}((1,1))$. We lower bound vertex potentials by decomposing two-dimensional walks on $\textsc{Square}_n$ into one-dimensional walks on $\textsc{Line}$ and then upper bound the probability that a t-step walk on $\textsc{Line}$ starting at 1 and ending at n does not leave the interval $[1,n]$. More specifically, our upper bound for $\pi_{(n,n)}((1,1))$ follows from Lemma 1.4.1 (which we prove in Section 1.5) and Fact 1.4.2.

Lemma 1.4.1. *For all $n \geq 20$ and $t \geq n-1$, we have*

$$\Pr_{w \sim \mathcal{W}^{\textsc{Line}}(1)}\left[w^{(t)} = n, \max_{\leq t}(w) = n, \text{and } \min_{\leq t}(w) \geq 1\right] \leq \min\left\{\frac{e^{25}}{n^3}, 64\left(\frac{n}{t}\right)^3\right\}.$$

Fact 1.4.2. *For any nonnegative integer t_1, we have*

$$\sum_{t_2 \geq 0} \binom{t_1+t_2}{t_2} \frac{1}{2^{t_1+t_2}} = 2.$$

Proof. This follows directly from the negative binomial distribution. Observe that

$$\sum_{t_2 \geq 0} \binom{t_1+t_2}{t_2} \frac{1}{2^{t_1+t_2}} = 2 \sum_{t_2 \geq 0} \binom{(t_1+1)-1+t_2}{t_2} \frac{1}{2^{t_1+1}} \cdot \frac{1}{2^{t_2}} = 2,$$

as desired. □

By decoupling the two-dimensional walks in a way similar to the proof of Lemma 1.3.5, we apply Lemma 1.4.1 to the resulting one-dimensional walks to achieve the desired upper bound.

Lemma 1.4.3. *For all $n \geq 20$, we have*

$$\pi_{(n,n)}((1,1)) \leq 2 \max_t \left\{ \Pr_{w \sim \mathcal{W}(1)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right] \right\} \cdot \sum_{t \geq 0} \Pr_{w \sim \mathcal{W}(1)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right].$$

Proof. Analogous to our lower bound for $\pi_{(n,n)}((1,1))$, decouple each walk $w \in \mathcal{W}((1,1) \to (n,n))$ into its horizontal walk $w_1 \in \mathcal{W}^{\text{LINE}}(1)$ and its vertical walk $w_2 \in \mathcal{W}^{\text{LINE}}(1)$. We view $\boldsymbol{\pi}_{(n,n)}((1,1))$ as the probability that the walks w_1 and w_2 are present on n at the same time before either leaves the interval $[1, n]$. Letting t_1 be the length of w_1 and t_2 be the length of w_2, we relax the conditions on the one-dimensional walks and only require that w_1 and w_2 both are present on n at the final step $t = t_1 + t_2$. Note that now both walks could have previously been present on n at the same time before terminating. This gives the upper bound

$$\boldsymbol{\pi}_{(n,n)}((1,1)) \leq \sum_{t_1, t_2 \geq 0} \frac{\binom{t_1+t_2}{t_1}}{2^{t_1+t_2}} \Pr_{w_1 \sim \mathcal{W}(1)} \left[w_1^{(t_1)} = n, \max_{\leq t_1}(w) = n, \min_{\leq t_1}(w) \geq 1 \right] \cdot \Pr_{w_2 \sim \mathcal{W}(1)} \left[w_2^{(t_2)} = n, \max_{\leq t_2}(w) = n, \min_{\leq t_2}(w) \geq 1 \right].$$

Nesting the summations gives

$$\boldsymbol{\pi}_{(n,n)}((1,1)) \leq \sum_{t_1 \geq 0} \Pr_{w_1 \sim \mathcal{W}(1)} \left[w_1^{(t_1)} = n, \max_{\leq t_1}(w) = n, \min_{\leq t_1}(w) \geq 1 \right] \cdot \sum_{t_2 \geq 0} \frac{\binom{t_1+t_2}{t_1}}{2^{t_1+t_2}} \Pr_{w_2 \sim \mathcal{W}(1)} \left[w_2^{(t_2)} = n, \max_{\leq t_2}(w) = n, \min_{\leq t_2}(w) \geq 1 \right].$$

Using Fact 1.4.2, we can upper bound the inner sum by

$$\sum_{t_2 \geq 0} \frac{\binom{t_1+t_2}{t_1}}{2^{t_1+t_2}} \Pr \left[w_2^{(t_2)} = n, \max_{\leq t_2}(w) = n, \min_{\leq t_2}(w) \geq 1 \right] \leq 2 \max_{t_2} \left\{ \Pr \left[w_2^{(t_2)} = n, \max_{\leq t_2}(w) = n, \min_{\leq t_2}(w) \geq 1 \right] \right\}.$$

Factoring out this term from the initial expression completes the proof. □

The upper bound on the maximum term in the right hand side of Lemma 1.4.3 follows immediately from Lemma 1.4.1. Now we upper bound the summation in the right hand side of Lemma 1.4.3 using a simple Lemma 1.4.1.

Lemma 1.4.4. *If $n \geq 20$ and $w \sim \mathcal{W}^{\textsc{Line}}(1)$, we have*

$$\sum_{t \geq 0} \Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \leq \frac{e^{26}}{n}.$$

Proof. We first split the sum into

$$\begin{aligned}&\sum_{t \geq 0} \Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \\ &\quad= \sum_{n^2 \geq t \geq 0} \Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] + \sum_{t > n^2} \Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right].\end{aligned}$$

We will bound both terms by $O(1/n)$. The upper bound for the first term follows immediately from Lemma 1.4.1 and the fact that we are summing $n^2 + 1$ terms:

$$\sum_{n^2 \geq t \geq 0} \Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \leq \frac{e^{25}}{n}.$$

To upper bound the second summation, we again use Lemma 1.4.1. When $t > n^2$, we have

$$\Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \leq 64\left(\frac{n}{t}\right)^3.$$

Since $64(n/t)^3$ is a decreasing function in t,

$$64\left(\frac{n}{t}\right)^3 \leq \int_{t-1}^{t} 64\left(\frac{n}{t}\right)^3 \mathrm{d}t.$$

Therefore, we can bound the infinite sum by the integral

$$\sum_{t > n^2} \Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \leq \int_{n^2}^{\infty} 64\left(\frac{n}{t}\right)^3 \mathrm{d}t = \frac{32}{n},$$

which concludes the proof since we bounded both halves of the sum by $O(1/n)$. $\square$

1.4.1 Proof of Theorem 1.1.3

We can now easily combine the lemmas in this section with the bounds that relate vertex potentials to the lower bound for the transience class of $\textsc{Square}_n$.

Proof. Applying Lemma 1.4.3 and then Lemma 1.4.1 and Lemma 1.4.4, it follows that

$$\begin{aligned}
\pi_{(n,n)}((1,1)) &\le \max_t \left\{ \Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)} = n, \max_{\le t}(w) = n, \min_{\le t}(w) \ge 1\right]\right\} \\
&\quad \cdot 2\sum_{t\ge 0} \Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)} = n, \max_{\le t}(w) = n, \min_{\le t}(w) \ge 1\right] \\
&\le 2 \cdot \frac{e^{25}}{n^3} \cdot \frac{e^{26}}{n} \\
&\le \frac{e^{100}}{n^4}.
\end{aligned}$$

Therefore, $\pi_{(n,n)}((1,1))^{-1} = \Omega(n^4)$. By Theorem 1.2.4 it follows that $\operatorname{tcl}(\textsc{Square}_n) = \Omega(n^4)$. $\square$

1.5 Simple Symmetric Random Walks

Our proofs for upper and lower bounding the sandpile transience class on the grid heavily relied on decoupling two-dimensional walks into two independent one-dimensional walks since they are easier to analyze. This claim is immediately apparent when working with vertex potentials for one-dimensional walks on the path, which we used in the proof of Lemma 1.3.1.

However, we assumed two essential lemmas about one-dimensional walks to prove the lower and upper bound of the minimum vertex potential. Consequently, in this section we examine the probability

$$\Pr_{w\sim\mathcal{W}^{\textsc{Line}}(i)}\left[w^{(t)} = n, \max_{\le t}(w) = n, \min_{\le t}(w) \ge 1\right], \tag{1.2}$$

and we prove these necessary lower and upper bounds in Lemma 1.3.3 and Lemma 1.4.1 by extending previously known properties of simple symmetric random walks on $\mathbb{Z}$. The key ideas in these proofs are that: the position of a walk in one dimension follows the binomial distribution; the number of walks reaching a maximum position in a fixed number of steps has an explicit formula; and there are tight bounds for binomial coefficients via Stirling's approximation.

The properties we need do not immediately follow from previously known facts because we require conditions on both the minimum and maximum positions. Section 1.5.3 gives proofs of the known explicit expressions for the maximum and minimum position of a walk, along with several other useful facts that follow from this proof. In Section 1.5.4 we apply Stirling's bound to give accurate lower bounds on a range of binomial coefficients. Section 1.5.5 and Section 1.5.6 prove several necessary preliminary lower bound lemmas. We prove Lemma 1.3.3 at the end of Section 1.5.6. In Section 1.5.7 we give necessary upper bound lemmas and a proof of Lemma 1.4.1.

1.5.1 Lower Bounding (1.2)

To lower bound (1.2), we split the desired probability into the product of two probabilities using the definition of conditional probability. Then we prove lower bounds for each.

- In Lemma 1.5.6 we show for $t \in \Theta(n^2)$, the probability that a walk on $\mathbb{Z}$ starting at $1 \leq i \leq \lceil n/2 \rceil$ is $\Pr_{w \sim \mathcal{W}(i)} [\min_{\leq t}(w) \geq 1] = \Omega(i/n)$.
- In Lemma 1.5.8 and Lemma 1.5.7 we bound the probability that a walk starting at $1 \leq i \leq \lceil n/2 \rceil$ of length $t \in \Theta(n^2)$ reaches n at step t without going above n, conditioned on never dropping below 1:

$$\Pr\left[w^{(t)} = n, \max_{\leq t}(w) = n \,\middle|\, \min_{\leq t}(w) \geq 1 \right] = \Omega\left(\frac{1}{n^2}\right).$$

Lemma 1.3.3 immediately follows multiplying these two bounds together. This division allows us to separate proving a minimum and maximum, and in turn simplifies applying known bounds on binomial distributions. Specifically, Lemma 1.5.6 is an immediate consequence of explicit expressions for the minimum point of a walk and bounds on binomial coefficients, both of which will be given rigorous treatment in Section 1.5.3.

These proofs will also output a known explicit expression for the probability of the walk reaching n at step t, while only staying to its left. All that remains then is to condition the walk to not go to the left of 1. Note that 1 is in the opposite direction of n, with respect to the starting position i. We formally show that the probability of reaching n without going above n only improves if the walk cannot move too far in the wrong direction, but only for $t \leq (n - i + 1)^2$, thus giving the reason we need to upper bound t by $n^2/4$.

1.5.2 Upper Bounding (1.2)

The desired lemma only concerns walks starting at $i = 1$, which will be critical for our proof. The key idea will then be to split the walk in half and consider the probability that the necessary conditions are satisfied in the first $t/2$ steps and in the second $t/2$ steps. The midpoint of the walk at $t/2$ steps can be any point in the interval $[1, n]$, so we must sum over all these possible midpoints. Removing the upper and lower bound conditions, respectively, will then give the upper bound in Lemma 1.5.9:

$$\begin{aligned}
&\Pr_{w \sim \mathcal{W}(1)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right] \\
&\qquad \leq \sum_{i=1}^{n} \Pr_{w \sim \mathcal{W}(1)} \left[w^{(\lfloor \frac{t}{2} \rfloor)} = i, \min_{\leq \lfloor \frac{t}{2} \rfloor}(w) \geq 1 \right] \cdot \Pr_{w \sim \mathcal{W}(i)} \left[w^{(\lceil \frac{t}{2} \rceil)} = n, \max_{\leq \lceil \frac{t}{2} \rceil}(w) = n \right].
\end{aligned}$$

Due to the first $t/2$ step walk starting at 1 and the second $t/2$ step walk ending at n, the conditions $\min_{\leq t}(w) \geq 1$ for the first walk and $\max_{\leq \lceil \frac{t}{2} \rceil}(w) = n$ for the second walk will be the difficult property for each walk to satisfy, respectively. Next we apply facts proved in Section 1.5.3 to obtain expressions for each term within the summation. The remainder of the upper bound analysis will then focus on bounding those expressions.

1.5.3 Maximum Position of a Walk

As previously mentioned, our proofs mostly leverage well-known facts about the maximum/minimum position of a random walk along with corresponding bounds for these probabilities. This section will first give the result regarding maximum/minimum position of walks and a connection to Stirling's approximation.

Observe that if we are only concerned with a single end point, we can fix the starting location at 0 by shifting accordingly. In these cases, the following bounds are well known in combinatorics.

Fact 1.5.1 ([62])**.** *For any $t, n \in \mathbb{Z}_{\geq 0}$, we have*

$$\mathrm{Pr}_{w \sim \mathcal{W}(0)}\left[\max_{\leq t}(w) = n\right] = \begin{cases} \mathrm{Pr}\left[w^{(t)} = n\right] = \binom{t}{\frac{t+n}{2}}\frac{1}{2^t} & \text{if } t + n \equiv 0 \pmod{*}2, \\ \mathrm{Pr}\left[w^{(t)} = n+1\right] = \binom{t}{\frac{t+n+1}{2}}\frac{1}{2^t} & \text{if } t + n \equiv 1 \pmod{*}2. \end{cases}$$

Proof. For any $k \leq n$, consider a walk $w \in \mathcal{W}(0)$ that satisfies $w^{(t)} = k$ and $\max_{\leq t}(w) \geq n$. Let t^* be the first time that $w^{(t^*)} = n$, and construct the walk m ending at $2n - k$ such that

$$m^{(\widehat{t})} = \begin{cases} w^{(\widehat{t})} & \text{if } 0 \leq \widehat{t} \leq t^*, \\ 2n - w^{(\widehat{t})} & \text{if } t^* < \widehat{t} \leq t. \end{cases}$$

This reflection map is a bijection, so for $k \leq n$ we have

$$\mathrm{Pr}_{w \sim \mathcal{W}(0)}\left[w^{(t)} = k, \max_{\leq t}(w) \geq n\right] = \mathrm{Pr}_{w \sim \mathcal{W}(0)}\left[w^{(t)} = 2n - k\right].$$

Subtracting the probability of the maximum position being at least $n + 1$ gives

$$\begin{aligned} &\mathrm{Pr}_{w \sim \mathcal{W}^{\textsc{Line}}(0)}\left[w^{(t)} = k \text{ and } \max_{\leq t}(w) = n\right] \\ &\qquad = \mathrm{Pr}_{w \sim \mathcal{W}^{\textsc{Line}}(0)}\left[w^{(t)} = 2n - k\right] - \mathrm{Pr}_{w \sim \mathcal{W}^{\textsc{Line}}(0)}\left[w^{(t)} = 2(n+1) - k\right]. \end{aligned}$$

Summing over all $k \leq n$, we have

$$\mathrm{Pr}_{w \sim \mathcal{W}^{\textsc{Line}}(0)}\left[\max_{\leq t}(w) = n\right] = \mathrm{Pr}_{w \sim \mathcal{W}^{\textsc{Line}}(0)}\left[w^{(t)} = n\right] + \mathrm{Pr}_{w \sim \mathcal{W}^{\textsc{Line}}(0)}\left[w^{(t)} = n+1\right].$$

Considering the parity of t and n completes the proof. □

The proof above contains two intermediate expressions for probabilities similar to the ones we want to bound.

Fact 1.5.2. *For any integers $n \geq 0$ and $k \leq n$, we have*

$$\Pr_{w \sim \mathcal{W}(0)}\left[w^{(t)} = k, \max_{\leq t}(w) \geq n\right] = \Pr_{w \sim \mathcal{W}(0)}\left[w^{(t)} = 2n - k\right].$$

Fact 1.5.3. *Let $t, n \in \mathbb{Z}_{\geq 0}$. For any integer $k \leq n$,*

$$\Pr_{w \sim \mathcal{W}(0)}\left[w^{(t)} = k, \max_{\leq t}(w) = n\right] = \begin{cases} \binom{t}{\frac{t+2n-k}{2}} \frac{1}{2^t} \left(\frac{4n-2k+2}{t+2n-k+2}\right) & \textit{if } t + k \equiv 0 \pmod{*}2, \\ 0 & \textit{if } t + k \equiv 1 \pmod{*}2. \end{cases}$$

Proof. Using Fact 1.5.1 and analyzing the parity of the walks gives

$$\begin{aligned} \binom{t}{\frac{t+2n-k}{2}} \frac{1}{2^t} - \binom{t}{\frac{t+2n-k+2}{2}} \frac{1}{2^t} &= \binom{t}{\frac{t+2n-k}{2}} \frac{1}{2^t} - \frac{t-2n+k}{t+2n-k+2} \binom{t}{\frac{t+2n-k}{2}} \frac{1}{2^t} \\ &= \binom{t}{\frac{t+2n-k}{2}} \frac{1}{2^t} \left(\frac{4n-2k+2}{t+2n-k+2}\right), \end{aligned}$$

as desired. □

1.5.4 Lower Bounding Binomial Coefficients

Ultimately, our goal is to give strong lower bounds on closely related probabilities to the ones above. To do so, we need to use various bounds on binomial coefficients that are consequences of Stirling's approximation.

Fact 1.5.4 (Stirling's Approximation)**.** *For any positive integer n, we have*

$$\sqrt{2\pi} \leq \frac{n!}{\sqrt{n}\left(\frac{n}{e}\right)^n} \leq e.$$

An immediate consequence of this is a concentration bound on binomial coefficients.

Fact 1.5.5. *Let $c, n \in \mathbb{R}_{>0}$ such that $c\sqrt{n} < n$. For any $k \in [(n - c\sqrt{n})/2, (n + c\sqrt{n})/2]$, we have*

$$\binom{n}{k} \geq e^{-1-c^2} \cdot \frac{2^n}{\sqrt{n}}.$$

Proof. We directly substitute Stirling's approximation to the definition of binomial coefficients to get

$$\begin{aligned}
\binom{n}{\frac{n-c\sqrt{n}}{2}} &= \frac{n!}{\left(\frac{n-c\sqrt{n}}{2}\right)!\left(\frac{n+c\sqrt{n}}{2}\right)!} \\
&\geq \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{e\sqrt{\frac{n-c\sqrt{n}}{2}}\left(\frac{n-c\sqrt{n}}{2e}\right)^{\frac{n-c\sqrt{n}}{2}} e\sqrt{\frac{n+c\sqrt{n}}{2}}\left(\frac{n+c\sqrt{n}}{2e}\right)^{\frac{n+c\sqrt{n}}{2}}} \\
&\geq \frac{2\sqrt{2\pi}}{e^2\sqrt{n}} \cdot \frac{2^n}{\left(1-\frac{c^2}{n}\right)^{\frac{n}{2}}\left(1-\frac{c}{\sqrt{n}}\right)^{-\frac{c\sqrt{n}}{2}}\left(1+\frac{c}{\sqrt{n}}\right)^{\frac{c\sqrt{n}}{2}}} \\
&\geq \frac{2\sqrt{2\pi}}{e^{2+c^2}} \cdot \frac{2^n}{\sqrt{n}} \\
&\geq e^{-1-c^2} \cdot \frac{2^n}{\sqrt{n}},
\end{aligned}$$

as desired. □

1.5.5 Lower Bounding the Minimum Position

We now bound the probability of the minimum position of a walk in $\mathcal{W}(i)$ being at least 1 after t steps.

Lemma 1.5.6. *For any positive integer n, initial position $1 \leq i \leq \lceil n/2 \rceil$, and constant $c > 4$, if we have $t \in [n^2/c, n^2/4]$, then*

$$\Pr_{w \sim \mathcal{W}(i)}\left[\min_{\leq t}(w) \geq 1\right] \geq e^{-1-c} \cdot \frac{i}{n}.$$

Proof. First observe that

$$\Pr_{w \sim \mathcal{W}(i)}\left[\min_{\leq t}(w) \geq 1\right] = \sum_{k=1}^{i} \Pr_{w \sim \mathcal{W}(i)}\left[\min_{\leq t}(w) = k\right].$$

By symmetry, this sum is

$$\sum_{k=0}^{i-1} \Pr_{w \sim \mathcal{W}(0)}\left[\max_{\leq t}(w) = k\right].$$

For each $0 \le k \le i-1$, Fact 1.5.1 implies that

$$\Pr_{w\sim\mathcal{W}(0)}\left[\max_{\le t}(w) = k\right] \in \left\{ \binom{t}{\frac{t+k}{2}}\frac{1}{2^t}, \binom{t}{\frac{t+k+1}{2}}\frac{1}{2^t} \right\}.$$

By assumption $k \le k+1 \le i \le n \le \sqrt{ct}$, so applying Fact 1.5.5 gives

$$\begin{aligned}\min\left\{ \binom{t}{\frac{t+k}{2}}\frac{1}{2^t}, \binom{t}{\frac{t+k+1}{2}}\frac{1}{2^t} \right\} &\ge \binom{t}{\frac{t+\sqrt{ct}}{2}}\frac{1}{2^t} \\ &\ge e^{-1-c}\frac{1}{\sqrt{t}} \\ &\ge e^{-1-c}\frac{1}{n},\end{aligned}$$

because $t \le n^2/4$. Summing over $0 \le k \le i-1$ gives the desired bound. □

1.5.6 Lower Bounding the Final and Maximum Position

Similarly, we can use binomial coefficient approximations to bound the probability of a t-step walk terminating at n while never moving to a position greater than n.

Lemma 1.5.7. *For any initial position* $1 \le i \le \lceil n/2 \rceil$ *and any* $\max\{n, n^2/c\} \le t \le n^2/4$ *with* $t \equiv n-i \pmod 2$*, we have*

$$\Pr_{w\sim\mathcal{W}(i)}\left[\max_{\le t}(w) = n, w^{(t)} = n\right] \ge e^{-1-c}\cdot\frac{1}{n^2}.$$

Proof. By symmetry we rewrite the probability as

$$\Pr_{w\sim\mathcal{W}(0)}\left[\max_{\le t}(w) = n-i, w^{(t)} = n-i\right].$$

Fact 1.5.3 gives that this probability equals to

$$\frac{1}{2^t}\binom{t}{\frac{t+n-i}{2}}\frac{2(n-i+1)}{t+n-i+2}.$$

We can separately bound the last two terms according to the assumptions on t and i. Setting $i = 0$

minimizes $\binom{t}{(t+n-i)/2}$ for all $i \geq 0$. Setting $i = \lceil n/2 \rceil$ in the numerator, $i = 0$ in the denominator, and $t = n^2/4$ minimizes $(2(n-i+1))/(t+n+2)$. It follows that

$$\frac{1}{2^t}\binom{t}{\frac{t+n}{2}} \cdot \frac{2(\lfloor n/2 \rfloor + 1)}{n^2/4 + n + 2} \geq \frac{1}{2^t}\binom{t}{\frac{t+n}{2}}\frac{n}{n^2}.$$

We reapply Fact 1.5.5 with the observation that $n \leq \sqrt{ct}$ to get

$$\frac{1}{2^t}\binom{t}{\frac{t+\sqrt{ct}}{2}}\frac{1}{n} \geq e^{-1-c} \cdot \frac{1}{n^2},$$

as desired. $\square$

It remains to condition upon the minimum of a walk. This hinges upon the following statement about moving in the wrong direction only decreasing the probability a walk starting at some $1 \leq i \leq \lceil n/2 \rceil$ ending at n without ever going past n.

Lemma 1.5.8. *For any* $1 \leq i \leq \lceil n/2 \rceil$*, at any step* $t \leq n^2/4$ *with* $t \equiv n - i \pmod 2$*, we have*

$$\Pr_{w \sim \mathcal{W}(i)}\left[w^{(t)} = n, \max_{\leq t}(w) = n\right] \geq \Pr_{w \sim \mathcal{W}(i)}\left[w^{(t)} = n, \max_{\leq t}(w) = n \,\middle|\, \min_{\leq t}(w) < 1\right].$$

Proof. Condition on $\min_{\leq t}(w) < 1$ and consider the first time $\widehat{t}$ the walk hits 0. This means $i \equiv \widehat{t} \pmod 2$ and in turn $n \equiv t - \widehat{t} \pmod 2$. The probability of $\max_{\leq t}(w) = w^{(t)} = n$ via the walk in steps $\widehat{t}+1, \ldots, t$ is then at most

$$\Pr_{w \sim \mathcal{W}(0)}\left[w^{(t-\widehat{t})} = \max_{\leq t-\widehat{t}}(w) = n\right].$$

Note that we have inequality since it is possible that we already have $\max_{\leq \widehat{t}}(w) > n$. Therefore, it suffices to show for any n and any $1 \leq \widehat{t} \leq t$ we have

$$\Pr_{w \sim \mathcal{W}(0)}\left[w^{(t-\widehat{t})} = \max_{\leq t-\widehat{t}}(w) = n\right] \leq \Pr_{w \sim \mathcal{W}(i)}\left[w^{(t)} = \max_{\leq t}(w) = n\right].$$

There are two variables that are shifted from one side of the inequality to the other, the starting position of the walk and the number of steps. In order to prove the inequality, we will show that both taking more steps and starting further to the right will only improve the probability of ending at n and not going above n.

We begin by showing that taking more steps will only improve this probability:

$$\begin{aligned}
&\Pr_{w\sim\mathcal{W}(0)}\left[w^{\left(t-\widehat{t}\right)}=\max_{\leq t-\widehat{t}}(w)=n\right]\\
&\qquad\leq \max\left\{\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t-1)}=\max_{\leq t-1}(w)=n\right],\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n\right]\right\}.
\end{aligned}$$

There is no guarantee that $t\equiv n \pmod 2$, so we consider t or $t-1$ steps depending on parity. We are guaranteed that $t-1\geq t-\widehat{t}$ since $\widehat{t}\geq 1$, so without loss of generality, we assume $t\equiv\widehat{t} \pmod 2$ and show

$$\Pr_{w\sim\mathcal{W}(0)}\left[w^{\left(t-\widehat{t}\right)}=\max_{\leq t-\widehat{t}}(w)=n\right]\leq \Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n\right].$$

Note that the proof is equivalent when $t-1\equiv\widehat{t} \pmod 2$.

Using Fact 1.5.3, we have the explicit probability

$$\Pr_{w\sim\mathcal{W}(0)}\left[\max_{\leq t}(w)=w^{(t)}=n\right]=\frac{1}{2^t}\binom{t}{\frac{t+n}{2}}\frac{2n+2}{t+n+2}.$$

Substituting t by $t-2$ into the equation above and comparing the right hand sides gives

$$\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t-2)}=\max_{\leq t-2}(w)=n\right]\leq \Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n\right],$$

because we know by assumption that

$$\begin{aligned}
\frac{1}{2^{t-2}}\binom{t-2}{\frac{t+n}{2}-1}\frac{2n+2}{t+n}&\leq\frac{1}{2^t}\binom{t}{\frac{t+n}{2}}\frac{2n+2}{t+n+2}\\
\frac{(t-2)!}{\left(\frac{t+n-2}{2}\right)!\left(\frac{t-n-2}{2}\right)!}\frac{1}{t+n}&\leq\frac{1}{4}\frac{t!}{\left(\frac{t+n}{2}\right)!\left(\frac{t-n}{2}\right)!}\frac{1}{t+n+2}\\
\frac{1}{t+n}&\leq\frac{t(t-1)}{(t+n)(t-n)(t+n+2)}\\
3t&\leq n^2+2n.
\end{aligned}$$

Inductively applying this argument inductively for $t-2$ proves the inequality.

To complete our proof, it now suffices to show

$$\max\left\{\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t-1)}=\max_{\leq t-1}(w)=n\right],\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n\right]\right\} \\ \leq \Pr_{w\sim\mathcal{W}(i)}\left[w^{(t)}=\max_{\leq t}(w)=n\right],$$

which we prove similarly. First rewrite the right hand side using the fact that

$$\Pr_{w\sim\mathcal{W}(i)}\left[w^{(t)}=\max_{\leq t}(w)=n\right]=\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n-i\right],$$

and initially assume that $t\equiv n \pmod 2$, which implies $n\equiv n-i \pmod 2$. Again, using the explicit formula from Fact 1.5.3 and substituting n by $n-2$ gives

$$\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n\right]\leq\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n-2\right],$$

when $t+2\leq n^2$, which true by assumption and can be inductively applied until $n=(n-i+2)$ because $(n-i+2)\geq\lceil n/2\rceil+1$. Unfortunately, we cannot entirely apply the same proof when $t-1\equiv n \pmod 2$ because this implies $n\not\equiv n-i \pmod 2$. Applying the same proof as for $t\equiv n \pmod 2$ we can obtain

$$\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t-1)}=\max_{\leq t-1}(w)=n\right]\leq\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t-1)}=\max_{\leq t-1}(w)=n-i+1\right],$$

because $(t-1)+2\leq(n-i+3)^2$.

Therefore, we can conclude the proof by showing

$$\Pr_{w\sim\mathcal{W}(0)}\left[w^{(t-1)}=\max_{\leq t-1}(w)=n-i+1\right]\leq\Pr_{w\sim\mathcal{W}^{\textsc{Line}}(0)}\left[w^{(t)}=\max_{\leq t}(w)=n-i\right].$$

This is then true when

$$n-i\leq\frac{t}{t-(n-i)}\cdot(n-i+1),$$

which holds for $n - i \geq 0$. $\square$

An immediate corollary of this Lemma 1.5.8 is that if we condition on the walk not going to the left of 1, it only becomes more probable to reach n without going above n. Now we prove the main result of this section.

Lemma 1.3.3. *Let $n \in \mathbb{Z}_{\geq 1}$ and $1 \leq i \leq \lceil n/2 \rceil$ be any starting position. For any constant $c > 4$ and any $t \in \mathbb{Z}$ such that $n^2/c \leq t \leq n^2/4$ with $t \equiv n - i \pmod 2$, a simple symmetric random walk w on $\mathbb{Z}$ satisfies*

$$\Pr_{w \sim \mathcal{W}^{\textsc{Line}}(i)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \text{and } \min_{\leq t}(w) \geq 1 \right] \geq e^{-2-2c} \frac{i}{n^3}.$$

Proof. Consider any starting position $1 \leq i \leq \lceil n/2 \rceil$ and any time $n^2/c \leq t \leq n^2/4$ with $t \equiv n - i \pmod 2$. By the definition of conditional probability we have

$$\begin{aligned}\Pr_{w \sim \mathcal{W}(i)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right] &= \Pr_{w \sim \mathcal{W}(i)} \left[w^{(t)} = n, \max_{\leq t}(w) = n \,\middle|\, \min_{\leq t}(w) \geq 1 \right] \\ &\quad \cdot \Pr_{w \sim \mathcal{W}(i)} \left[\min_{\leq t}(w) \geq 1 \right].\end{aligned}$$

Lemma 1.5.6 shows that the second term is at least $\exp(-1 - c))i/n$. Taking the probability under $\min_{\leq t}(w) \geq 1$ (i.e., the complementary event of $\min_{\leq t}(w) < 1$) in Lemma 1.5.8 allows us to upper bound the first term using Lemma 1.5.7 by

$$\begin{aligned}\Pr_{w \sim \mathcal{W}(i)} \left[w^{(t)} = n, \max_{\leq t}(w) = n \,\middle|\, \min_{\leq t}(w) \geq 1 \right] &\geq \Pr_{w \sim \mathcal{W}(i)} \left[w^{(t)} = n, \max_{\leq t}(w) = n \right] \\ &\geq e^{-1-c} \frac{1}{n^2}.\end{aligned}$$

Putting these together then gives

$$\begin{aligned}\Pr_{w \sim \mathcal{W}(i)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right] &\geq \left(e^{-1-c} \frac{i}{n} \right) \cdot \left(e^{-1-c} \frac{1}{n^2} \right) \\ &= e^{-2-2c} \cdot \frac{i}{n^3},\end{aligned}$$

which completes the proof. □

1.5.7 Upper Bounding the Final, Maximum, and Minimum Position

We begin by splitting every t step walk in half, and instead consider the probability of each walk satisfying the given conditions. In order to give upper bounds of these probabilities, we will relax the requirements, allowing us to more easily relate the probabilities to previously known facts about one-dimensional walks that we proved in Section 1.5.3. Furthermore, by splitting the walk in half we now have to consider all possible midpoints in $[1, n]$.

Lemma 1.5.9. *For all integers $1 \leq n \leq t$, we have*

$$\begin{aligned}
&\Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \\
&\qquad \leq \sum_{i=1}^{n} \Pr_{w\sim\mathcal{W}(1)}\left[w^{(\lfloor \frac{t}{2} \rfloor)} = i, \min_{\leq \lfloor \frac{t}{2} \rfloor}(w) \geq 1\right] \cdot \Pr_{w\sim\mathcal{W}(i)}\left[w^{(\lceil \frac{t}{2} \rceil)} = n, \max_{\leq \lceil \frac{t}{2} \rceil}(w) = n\right].
\end{aligned}$$

Proof. By subdividing the walk roughly in half, we consider all possible positions of a walk after half of its steps such that the walk satisfies the maximum and minimum conditions. The second half of the walk must end at n, which implies the maximum position of the walk must be at least n. Thus, the first half of the walk only needs to not go above n. Accordingly, we can write

$$\begin{aligned}
&\Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1\right] \\
&\qquad = \sum_{i=1}^{n} \Pr_{w\sim\mathcal{W}(1)}\left[w^{(\lfloor \frac{t}{2} \rfloor)} = i, \max_{\leq \lfloor \frac{t}{2} \rfloor}(w) \leq n, \min_{\leq \lfloor \frac{t}{2} \rfloor}(w) \geq 1\right] \\
&\qquad\qquad \cdot \Pr_{w\sim\mathcal{W}(i)}\left[w^{(\lceil \frac{t}{2} \rceil)} = n, \max_{\leq \lceil \frac{t}{2} \rceil}(w) = n, \min_{\leq \lceil \frac{t}{2} \rceil}(w) \geq 1\right].
\end{aligned}$$

Removing conditions that the walks must satisfy cannot decrease the probability, so our upper bound follows. □

From Fact 1.5.3 we can obtain explicit expressions for each inner term of the summation, which we then simplify into a strong bound on the summation in the following lemma.

Lemma 1.5.10. *For all integers $1 \le n \le t$, we have*

$$\Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)} = n, \max_{\le t}(w) = n, \min_{\le t}(w) \ge 1\right] \le \sum_{i=1}^{n}\left(\frac{16i(n-i+1)}{t^2}\right)\binom{\lfloor\frac{t}{2}\rfloor}{\frac{\lfloor\frac{t}{2}\rfloor+i-1}{2}}\frac{1}{2^{\lfloor\frac{t}{2}\rfloor}}\cdot\binom{\lceil\frac{t}{2}\rceil}{\frac{\lceil\frac{t}{2}\rceil+(n-i+1)-1}{2}}\frac{1}{2^{\lceil\frac{t}{2}\rceil}}.$$

Proof. Apply the upper bound from Lemma 1.5.9 and examine each inner term in the summation. By the symmetry of walks, there must be an equivalent number of $\lfloor t/2\rfloor$ step walks with endpoints 1 and i that never walk below 1 versus those that never walk above i. Thus

$$\Pr_{w\sim\mathcal{W}(1)}\left[\min_{\le\lfloor\frac{t}{2}\rfloor}(w) \ge 1, w^{(\lfloor\frac{t}{2}\rfloor)} = i\right] = \Pr_{w\sim\mathcal{W}(1)}\left[\max_{\le\lfloor\frac{t}{2}\rfloor}(w) \le i, w^{(\lfloor\frac{t}{2}\rfloor)} = i\right].$$

Shifting the start of the walk to 0 allows us to apply Fact 1.5.3, because $\max_{\le\lfloor\frac{t}{2}\rfloor}(w) \le i$ is equivalent to $\max_{\le\lfloor\frac{t}{2}\rfloor}(w) = i$ if the walk must end at i. Therefore,

$$\Pr_{w\sim\mathcal{W}(1)}\left[\min_{\le t}(w) \ge 1, w^{(\lfloor\frac{t}{2}\rfloor)} = i\right] = \left(\frac{2i}{\lfloor\frac{t}{2}\rfloor+i+1}\right)\binom{\lfloor\frac{t}{2}\rfloor}{\frac{\lfloor\frac{t}{2}\rfloor+i-1}{2}}\frac{1}{2^{\lfloor\frac{t}{2}\rfloor}},$$

when the parity is correct and 0 otherwise. This works as an upper bound. Similarly, by shifting the start to 0 and applying Fact 1.5.3 we have

$$\Pr_{w\sim\mathcal{W}(i)}\left[w^{(\lceil\frac{t}{2}\rceil)} = n, \max_{\le\lceil\frac{t}{2}\rceil}(w) = n\right] = \left(\frac{2(n-i+1)}{\lceil\frac{t}{2}\rceil+(n-i+1)+1}\right)\binom{\lceil\frac{t}{2}\rceil}{\frac{\lceil\frac{t}{2}\rceil+(n-i+1)-1}{2}}\frac{1}{2^{\lceil\frac{t}{2}\rceil}}.$$

Applying Lemma 1.5.9, we now have expressions for the term inside the summation, so

$$\begin{aligned}
&\Pr_{w\sim\mathcal{W}(1)}\left[w^{(\lfloor\frac{t}{2}\rfloor)} = i, \min_{\le\lfloor\frac{t}{2}\rfloor}(w) \ge 1\right]\cdot\Pr_{w\sim\mathcal{W}(i)}\left[w^{(\lceil\frac{t}{2}\rceil)} = n, \max_{\le\lceil\frac{t}{2}\rceil}(w) = n\right]\\
&= \left(\frac{2i}{\lfloor\frac{t}{2}\rfloor+i+1}\right)\binom{\lfloor\frac{t}{2}\rfloor}{\frac{\lfloor\frac{t}{2}\rfloor+i-1}{2}}\frac{1}{2^{\lfloor\frac{t}{2}\rfloor}}\cdot\left(\frac{2(n-i+1)}{\lceil\frac{t}{2}\rceil+(n-i+1)+1}\right)\binom{\lceil\frac{t}{2}\rceil}{\frac{\lceil\frac{t}{2}\rceil+(n-i+1)-1}{2}}\frac{1}{2^{\lceil\frac{t}{2}\rceil}}\\
&\le \left(\frac{16i(n-i+1)}{t^2}\right)\binom{\lfloor\frac{t}{2}\rfloor}{\frac{\lfloor\frac{t}{2}\rfloor+i-1}{2}}\frac{1}{2^{\lfloor\frac{t}{2}\rfloor}}\cdot\binom{\lceil\frac{t}{2}\rceil}{\frac{\lceil\frac{t}{2}\rceil+(n-i+1)-1}{2}}\frac{1}{2^{\lceil\frac{t}{2}\rceil}},
\end{aligned}$$

which we upper bound by the fact that $(\lfloor\frac{t}{2}\rfloor+i+1)(\lceil\frac{t}{2}\rceil+(n-i+1)+1) \ge t^2/4$. This completes the proof. $\square$

The following lemma gives a upper bound for the inner expression from Lemma 1.5.10 by bounding the binomial coefficients with the central binomial coefficients and using Stirling's approximation.

Lemma 1.5.11. *For any integer* $1 \leq in$*, we have*

$$\left(\frac{16i(n-i+1)}{t^2}\right)\binom{\lfloor \frac{t}{2} \rfloor}{\frac{\lfloor \frac{t}{2} \rfloor + i - 1}{2}}\frac{1}{2^{\lfloor \frac{t}{2} \rfloor}}\binom{\lceil \frac{t}{2} \rceil}{\frac{\lceil \frac{t}{2} \rceil + (n-i+1) - 1}{2}}\frac{1}{2^{\lceil \frac{t}{2} \rceil}} \leq 64\frac{n^2}{t^3}.$$

Proof. Given that $1 \leq i \leq n$, we can crudely upper bound $i(n-i+1)$ by n^2. Additionally, we can will use Stirling's approximation on the central binomial coefficient to upper bound our binomial coefficients.

$$\binom{\lceil \frac{t}{2} \rceil}{\frac{\lceil \frac{t}{2} \rceil + (n-i+1) - 1}{2}} \leq 2^{\lceil \frac{t}{2} \rceil} \cdot \frac{1}{\sqrt{\lceil \frac{t}{2} \rceil}},$$

and

$$\binom{\lfloor \frac{t}{2} \rfloor}{\frac{\lfloor \frac{t}{2} \rfloor + i - 1}{2}} \leq 2^{\lfloor \frac{t}{2} \rfloor} \cdot \frac{1}{\sqrt{\lfloor \frac{t}{2} \rfloor}}.$$

The exponential terms will cancel and

$$\frac{1}{\sqrt{\lceil \frac{t}{2} \rceil}} \cdot \frac{1}{\sqrt{\lfloor \frac{t}{2} \rfloor}} \leq \frac{4}{t},$$

giving our desired bound. $\square$

The upper bound in Lemma 1.5.11 is not sufficient for t that are asymptotically less than n^2, so for these t we need to give a more detailed analysis. Therefore, we more carefully examine the binomial coefficients that are significantly smaller than the central coefficient for small t. Consequently, the exponential term will not be sufficiently canceled by the binomial coefficient for values of t that are asymptotically smaller than n^2. More specifically, we show that the function of t on the right hand side of Lemma 1.5.10 is increasing in t up until approximately n^2.

In the following lemma we consider even length walks for simplicity. The proof for odd length walks follows analogously.

Lemma 1.5.12. *For $n \geq 20$ and any integer $1 \leq i \leq n$, for all $t \leq n^2/40$ we have*

$$\frac{16i(n-i+1)}{(2t)^2}\frac{1}{2^{2t}}\binom{t}{\frac{t+i-1}{2}}\binom{t}{\frac{t+(n-i+1)-1}{2}} \leq \frac{16i(n-i+1)}{(2t+4)^2}\frac{1}{2^{2t+4}}\binom{t+2}{\frac{t+2+i-1}{2}}\binom{t+2}{\frac{t+2+(n-i+1)-1}{2}},$$

where we consider walks of length $2t$ and $2t+4$ to ensure that $(2t)/2$ and $(2t+4)/2$ have the same parity.

Proof. Canceling like terms implies that the desired inequality is equivalent to

$$\frac{1}{t^2}\binom{t}{\frac{t+i-1}{2}}\binom{t}{\frac{t+(n-i+1)-1}{2}} \leq \frac{1}{(t+2)^2}\cdot\frac{1}{16}\cdot\binom{t+2}{\frac{t+2+i-1}{2}}\binom{t+2}{\frac{t+2+(n-i+1)-1}{2}}.$$

Examining the binomial coefficients shows that

$$\binom{t}{\frac{t+i-1}{2}}\frac{(t+2)(t+1)}{\left(\frac{t+1+i}{2}\right)\left(\frac{t+3-i}{2}\right)} = \binom{t+2}{\frac{t+2+i-1}{2}},$$

and

$$\binom{t}{\frac{t+(n-i+1)-1}{2}}\frac{(t+2)(t+1)}{\left(\frac{t+2+(n-i)}{2}\right)\left(\frac{t+2-(n-i)}{2}\right)} = \binom{t+2}{\frac{t+2+(n-i+1)-1}{2}}.$$

Using these identities, our desired inequality equals

$$\frac{1}{t^2} \leq \frac{16^{-1}}{(t+2)^2}\frac{(t+2)(t+1)}{\left(\frac{t+1+i}{2}\right)\left(\frac{t+3-i}{2}\right)}\frac{(t+2)(t+1)}{\left(\frac{t+2+(n-i)}{2}\right)\left(\frac{t+2-(n-i)}{2}\right)}.$$

Further cancellation of like terms and moving the denominator on each side into the numerator on the other side implies that our desired inequality is equivalent to

$$(t+1+i)(t+3-i)(t+2+(n-i))(t+2-(n-i)) \leq t^2(t+1)^2.$$

It is straightforward to see that

$$(t+1+i)(t+3-i)$$

is maximized by $i = 1$ and

$$(t+2+(n-i))(t+2-(n-i))$$

is maximized by $n - i = 0$. Furthermore, it must be true that either $i \geq n/2$ or $n - i \geq n/2$, so we can upper bound the left hand side of our inequality by substituting $n/2$ for i or $n - i$, and setting the other terms to the value that maximizes the product. Hence,

$$(t+1+i)(t+3-i)(t+2+(n-i))(t+2-(n-i)) \leq (t+2)^2 \left(t+3+\frac{n}{2}\right)\left(t+3-\frac{n}{2}\right).$$

To prove our desired inequality it now suffices to show $(t+2)^2\left(t+3+n/2\right)\left(t+3-n/2\right) \leq t^2(t+1)^2$, which is equivalent to

$$\left(t+3+\frac{n}{2}\right)\left(t+3-\frac{n}{2}\right) \leq t^2\left(1-\frac{1}{t+2}\right)^2.$$

Expanding both sides of the inequality and rearranging terms yields

$$6t+9+\frac{2t^2}{t+2}-\left(\frac{t}{t+2}\right)^2 \leq \frac{n^2}{4}.$$

Given that $2t^2/(t+2) \leq 2t$, it suffices to show that $8t + 9 \leq n^2/4$, which is true when $t \leq n^2/40$ and $n \geq 20$. $\square$

We can now prove the main upper bound result of this section using the recently developed bounds for the right hand side of the expression in Lemma 1.5.10.

Lemma 1.4.1. *For all $n \geq 20$ and $t \geq n-1$, we have*

$$\Pr_{w\sim\mathcal{W}^{\textsc{Line}}(1)}\left[w^{(t)} = n, \max_{\leq t}(w) = n, \text{and } \min_{\leq t}(w) \geq 1\right] \leq \min\left\{\frac{e^{25}}{n^3}, 64\left(\frac{n}{t}\right)^3\right\}.$$

Proof. Applying Lemma 1.5.10 and Lemma 1.5.11 gives

$$\Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)}=n, \max_{\leq t}(w)=n, \min_{\leq t}(w)\geq 1\right] \leq \sum_{i=1}^{n} 64\left(\frac{n^2}{t^3}\right),$$

which immediately gives the upper bound $64(n/t)^3$. Similarly, Lemmas 1.5.10 and 1.5.12 imply that for $t \leq n^2/40$,

$$\begin{aligned}&\Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)}=n, \max_{\leq t}(w)=n, \min_{\leq t}(w)\geq 1\right]\\ &\qquad\leq \sum_{i=1}^{n}\left(\frac{16i(n-i+1)}{T^2}\right)\binom{\lfloor\frac{T}{2}\rfloor}{\frac{\lfloor\frac{T}{2}\rfloor+i-1}{2}}\frac{1}{2^{\lfloor\frac{T}{2}\rfloor}}\cdot\binom{\lceil\frac{T}{2}\rceil}{\frac{\lceil\frac{T}{2}\rceil+(n-i+1)-1}{2}}\frac{1}{2^{\lceil\frac{T}{2}\rceil}},\end{aligned}$$

where $T = n^2/40$. We then use Lemma 1.5.11 and sum from 1 to n to obtain

$$\begin{aligned}\Pr_{w\sim\mathcal{W}(1)}\left[w^{(t)}=n, \max_{\leq t}(w)=n, \min_{\leq t}(w)\geq 1\right] &\leq \sum_{i=1}^{n}\frac{64n^2}{T^3}\\ &\leq \frac{64(40)^3}{n^3}\\ &\leq \frac{e^{25}}{n^3},\end{aligned}$$

for all $t \leq n^2/40$. Using the fact that $64(n/t)^3$ is a decreasing function in t, we have

$$64\left(\frac{n}{t}\right)^3 \leq \frac{e^{25}}{n^3},$$

for all $t \geq n^2/40$, which is the desired bound. $\square$

1.6 Extension to Higher Dimensions

Now we show how to extend our analysis to upper and lower bound the transience class of d-dimensional grids.

Theorem 1.1.4. *For any integer $d \geq 2$, the transience class of the Abelian sandpile model on the n^d-sized d-dimensional grid is $O(n^{3d-2}\log^{d+2} n)$ and $\Omega(n^{3d-2})$.*

We denote by d-$\textsc{Cube}_n$ the d-dimensional hypercube grid with n^d vertices, and construct it analogously to $\textsc{Square}_n$. Its vertex set is $\{1, 2, \ldots, n\}^d \cup \{v_{\text{sink}}\}$ and its edges connect any pair of vertices that differ in one coordinate. Vertices on the boundary have additional edges connecting to v_{sink} so that every non-sink vertex has degree $2d$. We use the vector notation $\boldsymbol{u} = (\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_d)$ to identify non-sink vertices. We can decouple a walk w on d-$\textsc{Cube}_n$ into one-dimensional walks $w_1, w_2, \ldots, w_d$, so that each step of a random walk on d-$\textsc{Cube}_n$ can be understood as choosing a random direction with probability $1/d$ and then a step in the corresponding one-dimensional walk with probability $1/2$.

Our bounds for the two-dimensional grid heavily relied on decoupling walks into interleaved one-dimensional walks, and applying bounds from Section 1.5 for simple symmetric walks. Generalizing these bounds to d-dimensional hypercubes follows comparably and only requires simple extensions of our lemmas for the two-dimensional grids. Therefore, we will reference the necessary lemmas from previous sections and show the minor modifications needed to give analogous lemmas for the d-dimensional grid. The upper bound proof requires several key lemmas and is more involved, whereas extending the lower bound only requires one simple addition to our proof in Section 1.4.

1.6.1 Upper Bounding the Transience Class

Since Theorem 1.2.4 from [10] relies on non-sink vertices having constant degree, our assumptions that d is constant and that all non-sink vertices have degree $2d$. In addition to utilizing properties of one-dimensional walks, specifically Lemma 1.3.3 proven in Section 1.5, the proof of our upper bound relies on four key lemmas:

- Lemma 1.2.9 — The source vertex can be swapped with a any non-sink vertex while only losing a $O(\log n)$ approximation factor in the potential.
- Lemma 1.3.1 — An upper bound on the sum of all vertex potentials by factoring the expression into one-dimensional vertex potentials.
- Lemma 1.3.2 — For any vertex, the opposite corner vertex minimizes the potential up to a constant.

- Lemma 1.3.5 — A lower bound on the vertex potential $\boldsymbol{\pi}_{(n,n)}(\boldsymbol{u})$, for any $\boldsymbol{u}$ in the top-right quadrant of $\textsc{Square}_n$.

Now we describe how to extend each of these lemmas to constant dimensions. These results almost immediately follow from decoupling walks into one-dimensional walks.

Lemma 1.6.1. *For any pair of non-sink vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ in d-$\textsc{Cube}_n$, we have*

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) \leq (8\log n + 4)\,\boldsymbol{\pi}_{\boldsymbol{v}}(\boldsymbol{u}).$$

Proof. This is a consequence of Rayleigh's monotonicity theorem. Fix an underlying $n \times n$ subgraph of the hypercube with corners at the source and sink, and set the rest of the resistors to infinity. The upper bound for the $n \times n$ grid is an upper bound for the hypercube. □

Our lemma analogous to Lemma 1.3.1 follows from Lemma 1.6.1 and decoupling walks into one-dimension.

Lemma 1.6.2. *For any non-sink vertex $\boldsymbol{u}$ in d-$\textsc{Cube}_n$,*

$$\sum_{\boldsymbol{v}\in V} \pi_{\boldsymbol{u}}(\boldsymbol{v}) = O\left(\log n \prod_{i=1}^{d} \boldsymbol{u}_i \log n\right).$$

Proof. Follow the proof structure of Lemma 1.3.1. □

We can also generalize our proof of Lemma 1.3.2 to higher dimensions, because we work with each dimension independently.

Lemma 1.6.3. *If $\boldsymbol{u}$ is a non-sink vertex of d-$\textsc{Cube}_n$ such that $1 \leq \boldsymbol{u}_i \leq \lceil n/2 \rceil$ for all $1 \leq i \leq d$, then*

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) \geq \left(\frac{1}{2d}\right)^d \boldsymbol{\pi}_{\boldsymbol{u}}((n, n, \ldots, n)).$$

Proof. Extend the proof of Lemma 1.3.2 by reflecting walks across the $(d-1)$-dimensional hyperplane perpendicular to the chosen axis instead of a line. □

Lastly, we generalize Lemma 1.3.5, where the key idea was to considers walks of length $\Theta(n^2)$ and show that there is a constant fraction such that both dimensions have taken $\Theta(n^2)$ steps, which allows us to apply Lemma 1.3.3 for each possible walk. To do this, we essentially union bound Lemma 1.3.4 over d dimensions, which shows that $\Theta(n^2)$ walk lengths take $\Theta(n^2)$ steps in each direction with probability at least 2^{-d}.

Lemma 1.6.4. *For $n \geq 10$ and $\boldsymbol{u} \in V(d\text{-}\textsc{Cube}_n)$ such that $1 \leq \boldsymbol{u}_i \leq \lceil n/2 \rceil$ for $1 \leq i \leq d$, we have*

$$\boldsymbol{\pi}_{(n,n,\ldots,n)}(\boldsymbol{u}) = \Omega\left(\frac{\prod_{i=1}^d \boldsymbol{u}_i}{n^{3d-2}}\right).$$

Proof. Decouple walks $w \in \mathcal{W}(\boldsymbol{u} \to (n,n,\ldots,n))$ into one-dimensional walks $w_i \in \mathcal{W}^{\textsc{Line}}(\boldsymbol{u}_i)$, and view $\boldsymbol{\pi}_{(n,n,\ldots,n)}(\boldsymbol{u})$ as the probability that each walk w_i is present on n at the same time before any leaves the interval $[1,n]$. If each walk takes $t_1, t_2, \ldots, t_d$ steps, respectively, then the total number of possible interleavings of these walks is the multinomial

$$\binom{t_1 + t_2 + \cdots + t_d}{t_1, t_2, \ldots, t_d}.$$

Just as before, we can obtain the lower bound

$$\boldsymbol{\pi}_{(n,n,\ldots,n)}(\boldsymbol{u}) \geq \sum_{t_1,t_2,\ldots,t_d \geq 0} \frac{\binom{t_1+t_2+\cdots+t_d}{t_1,t_2,\ldots,t_d}}{d^{t_1+t_2+\cdots+t_d}} \prod_{i=1}^{d} \frac{1}{2} \Pr\left[w_i^{(t_i-1)} = n-1, \max_{\leq t_i - 1}(w) = n-1, \min_{\leq t_i-1}(w) \geq 1\right].$$

To apply Lemma 1.3.3 to each walk, we need each t_i to be in the interval $[n^2/c, n^2/4]$, for $c = 16d$. Then we consider all walks of length

$$\frac{n^2}{8} \leq t \leq \frac{n^2}{4},$$

where $t = t_1 + t_2 + \cdots + t_d$, and show that a constant fraction of these walks satisfy $t_i \geq n^2/c$ with t_i having the correct parity. Note that we can ignore the parity conditions by simply lower bounding the probability of all having correct parity by 4^{-d}. It then remains to show that all walks satisfy the inequality $t_i \geq n^2/c$ with constant probability.

Consider the probability that $t_1 \geq n^2/c$. The other dimensions follow identically. Letting each dimension take at least n^2/c steps introducing dependence, so we instead consider the probability that $t_1 \geq n^2/c$ and condtion on $t_2, t_3, \ldots, t_d \geq n^2/c$ (which can only decrease the probability of the event $t_1 \geq n^2/c$). This is equivalent to fixing n^2/c steps in each of those directions and randomly choosing all remaining steps with probability $1/d$ for each direction. The remaining number of steps is then at least dn^2/c by our assumption that $t \geq n^2/8$. Therefore, the expected number of steps in the first dimension is at least n^2/c, which implies $t_1 \geq n^2/c$ with probability at least $1/2$. Multiplying this probability over all dimensions gives $t_i \geq n^2/c$ with probability at least 2^{-d}.

Thus, there are $O(n^2)$ values of t that we can decompose into one-dimensional walks, each occurring with constant probability. Applying Lemma 1.3.3 to each decomposition and summing

$$\Omega\left(\prod_{i=1}^{d} \frac{\boldsymbol{u}_i}{n^3}\right)$$

over $O(n^2)$ possible walks proves the claim. □

Now we prove $\operatorname{tcl}(d\text{-}\textsc{Cube}_n) = O(n^{3d-2}\log^{d+2} n)$ using Theorem 1.2.4. For any $\boldsymbol{u} = (\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_d)$ in the top-left orthant of d-$\textsc{Cube}_n$, it follows that

$$\begin{aligned}
\max_{\boldsymbol{u},\boldsymbol{v} \in V\setminus\{v_{\text{sink}}\}} \left(\sum_{\boldsymbol{x}\in V} \pi_{\boldsymbol{u}}(\boldsymbol{x})\right) \pi_{\boldsymbol{u}}(\boldsymbol{v})^{-1} &\leq \max_{\boldsymbol{u} \in V\setminus\{v_{\text{sink}}\}} \left(\sum_{\boldsymbol{x}\in V} \boldsymbol{\pi}_{\boldsymbol{u}}\left(\boldsymbol{x}\right)\right) \frac{(2d)^d}{\boldsymbol{\pi}_{\boldsymbol{u}}\left((n)^d\right)} \\
&= \max_{\boldsymbol{u} \in V\setminus\{v_{\text{sink}}\}} \left(\sum_{\boldsymbol{x}\in V} \boldsymbol{\pi}_{\boldsymbol{u}}\left(\boldsymbol{x}\right)\right) \frac{O\left(\log n\right)}{\boldsymbol{\pi}_{(n)^d}\left(\boldsymbol{u}\right)} \\
&= \max_{\boldsymbol{u} \in V\setminus\{v_{\text{sink}}\}} O\left(\log n \prod_{i=1}^{d} \boldsymbol{u}_i \log n\right) O\left(\frac{n^{3d-2}\log n}{\prod_{i=1}^{d} \boldsymbol{u}_i}\right) \\
&= O\left(n^{3d-2}\log^{d+2} n\right).
\end{aligned}$$

1.6.2 Lower Bounding the Transience Class

Extending our lower bound to d-dimensional hypergrids is a simple consequence of decoupling d-dimensional walks into one-dimensional walks, because we only need to generalize the upper bound in Lemma 1.4.3 to

$$\pi_{(n)^d}\left((1)^d\right) \leq \max_t \left\{ \Pr_{w \sim \mathcal{W}(1)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right] \right\}$$
$$\cdot\, d \sum_{t \geq 0} \Pr_{w \sim \mathcal{W}(1)} \left[w^{(t)} = n, \max_{\leq t}(w) = n, \min_{\leq t}(w) \geq 1 \right],$$

by replacing the negative binomial distribution with the negative multinomial distribution.

Fact 1.6.5. *For any nonnegative integer t_1, we have*

$$\sum_{t_2,\ldots,t_d \geq 0} \binom{t_1 + t_2 + \cdots + t_d}{t_1, t_2, \ldots, t_d} \frac{1}{d^{t_1+t_2+\cdots+t_d}} = d.$$

Proof. Consider the proof of Fact 1.4.2 using the negative multinomial distribution. □

Thus, we can apply Lemma 1.4.1 and Lemma 1.4.4 to show

$$\pi_{(n)^d}\left((1)^d\right) = O\left(\left(\frac{1}{n^3}\right)^{d-1} \frac{1}{n}\right) = O\left(n^{-3d+2}\right).$$

By Theorem 1.2.4, we have $\mathrm{tcl}(d\text{-}\textsc{Cube}_n) = \Omega(n^{3d-2})$.

1.7 Omitted Proofs

1.7.1 Omitted Proofs in Section 1.2

In this section, we prove a relationship between voltage potentials and the probability of a random walk escaping at the source instead of the sink.

Lemma 1.2.6. *Let $\boldsymbol{u}$ be a non-sink vertex of $\textsc{Square}_n$. For any vertex $\boldsymbol{v}$, we have*

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) = \sum_{w \in \mathcal{W}(\boldsymbol{v} \to \boldsymbol{u})} 4^{-|w|}.$$

Proof. By definition, we have

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) = \frac{\sum_{w \in \mathcal{W}(\boldsymbol{v} \to \boldsymbol{u})} 4^{-|w|}}{\sum_{w \in \mathcal{W}(\boldsymbol{v} \to \{\boldsymbol{u}, v_{\text{sink}}\})} 4^{-|w|}}.$$

For any $\boldsymbol{v} \in V(\textsc{Square}_n)$, let

$$f(\boldsymbol{v}) = \sum_{w \in \mathcal{W}(\boldsymbol{v} \to \{\boldsymbol{u}, v_{\text{sink}}\})} 4^{-|w|}$$

be the normalizing constant for $\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v})$. It follows that $f(\boldsymbol{u}) = 1$ and $f(v_{\text{sink}}) = 1$, because the only such walk for each has length 0. For all other $\boldsymbol{v} \in V(\textsc{Square}_n) \setminus \{\boldsymbol{u}, v_{\text{sink}}\}$, we have

$$f(\boldsymbol{v}) = \frac{1}{4} \sum_{\boldsymbol{x} \sim \boldsymbol{v}} f(\boldsymbol{x}).$$

Therefore, $f(\boldsymbol{v})$ is a harmonic function with constant boundary values, so $f(\boldsymbol{v}) = 1$ for all vertices $\boldsymbol{v} \in V(\textsc{Square}_n)$. $\square$

We also verify that the effective resistance between v_{sink} and any internal vertex is bounded between $\Omega(1)$ and $O(\log n)$ using a triangle inequality for effective resistances and the fact that the effective resistance between opposite corners in an $n \times n$ resistor network is $\Theta(\log n)$. This proof easily generalizes to any pair of vertices in $\textsc{Square}_n$.

Proposition 1.7.1 ([63])**.** *Let G be an $n \times n$ network of unit resistors. If u and v are vertices at opposite corner vertices, then* $\log(n-1)/2 \le \mathcal{R}_{\text{eff}}(u, v) \le 2\log n$.

Lemma 1.2.8. *For any non-sink vertex $\boldsymbol{u}$ in $\textsc{Square}_n$,*

$$1/4 \le \mathcal{R}_{\text{eff}}(v_{\text{sink}}, \boldsymbol{u}) \le 2\log n + 1.$$

Proof. We first prove the lower bound

$$1/4 \le \mathcal{R}_{\text{eff}}(v_{\text{sink}}, \boldsymbol{u}).$$

The effective resistance between v_{sink} and $\boldsymbol{u}$ is the reciprocal of the total current flowing into the circuit when $\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{u}) = 1$ and $\boldsymbol{\pi}_{\boldsymbol{u}}(v_{\text{sink}}) = 0$. Since $\boldsymbol{\pi}_{\boldsymbol{u}}$ is a harmonic function, we have $\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) \ge 0$ for all $\boldsymbol{v} \in V(\textsc{Square}_n)$. Moreover, $\deg(\boldsymbol{u}) = 4$, so

$$\mathcal{R}_{\text{eff}}(v_{\text{sink}}, \boldsymbol{u}) = \left(\sum_{\boldsymbol{v} \sim \boldsymbol{u}} \boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{u}) - \boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) \right)^{-1} \ge \frac{1}{4}.$$

For the upper bound, we use Rayleigh's monotonicity law, Proposition 1.7.1, and the triangle inequality for effective resistances to show that

$$\mathcal{R}_{\text{eff}}(v_{\text{sink}}, \boldsymbol{u}) \leq 2\log n + 1,$$

for n sufficiently large. Rayleigh's monotonicity law [1] states that if the resistances of a circuit are increased, the effective resistance between any two points can only increase. The following triangle inequality for effective resistances is given in [64]:

$$\mathcal{R}_{\text{eff}}(u, v) \leq \mathcal{R}_{\text{eff}}(u, x) + \mathcal{R}_{\text{eff}}(x, v).$$

Define H to be the subgraph of $\textsc{Square}_n$ obtained by deleting v_{sink} and all edges incident to v_{sink}. Let m be the largest positive integer such that $\boldsymbol{u}_1 + i \leq n$ and $\boldsymbol{u}_2 + j \leq n$ for all $0 \leq i, j < m$, and let $H(\boldsymbol{u})$ be the subgraph of H induced by the vertex set

$$\{(\boldsymbol{u}_1 + i, \boldsymbol{u}_2 + j) : 0 \leq i, j < m\}.$$

We can view $H(\boldsymbol{u})$ as the largest square resistor network in H such that $\boldsymbol{u}$ is the top-left vertex. Let $\boldsymbol{v} = [\boldsymbol{u}_1 + m - 1, \boldsymbol{u}_2 + m - 1]$ be the bottom-right vertex in $H(\boldsymbol{u})$. Using infinite resistors to remove every edge in $E(\textsc{Square}_n) \setminus E(H(\boldsymbol{u}))$, we have

$$\mathcal{R}_{\text{eff}}^{\textsc{Square}_n}(\boldsymbol{v}, \boldsymbol{u}) \leq \mathcal{R}_{\text{eff}}^{H(\boldsymbol{u})}(\boldsymbol{v}, \boldsymbol{u})$$

by Rayleigh's monotonicity law. Proposition 1.7.1 implies that

$$\mathcal{R}_{\text{eff}}^{H(\boldsymbol{u})}(\boldsymbol{v}, \boldsymbol{u}) \leq 2\log n$$

since $m \leq n$. The vertex $\boldsymbol{v}$ is incident to v_{sink} in $\textsc{Square}_n$, so Rayleigh's monotonicity law gives

$$\mathcal{R}_{\text{eff}}^{\textsc{Square}_n}(v_{\text{sink}}, \boldsymbol{v}) \leq 1.$$

By the triangle inequality for effective resistances, we have

$$\mathcal{R}_{\text{eff}}(v_{\text{sink}}, \boldsymbol{u}) \leq \mathcal{R}_{\text{eff}}(v_{\text{sink}}, \boldsymbol{v}) + \mathcal{R}_{\text{eff}}(\boldsymbol{v}, \boldsymbol{u}) \leq 2 \log n + 1,$$

which completes the proof. □

1.7.2 Omitted Proofs in Section 1.3

We use the random walk interpretation of voltage to prove Lemma 1.3.2. The key idea is that the voltage on the boundary opposite of $\boldsymbol{u}$ along any axis is less by a constant factor. This projection can be iterated along an axis in each dimension.

Lemma 1.3.2. *If $\boldsymbol{u}$ is a vertex in the top-left quadrant of* $\textsc{Square}_n$*, then for any non-sink vertex $\boldsymbol{v}$ we have*

$$\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v}) \geq \frac{1}{16} \boldsymbol{\pi}_{\boldsymbol{u}}((n, n)).$$

Proof. We use Lemma 1.2.6 to decompose $\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{v})$ as a sum of probabilities of walks, and then construct maps for all $1 \leq \boldsymbol{v}_1, \boldsymbol{v}_2 \leq n$ to show

$$\boldsymbol{\pi}_{\boldsymbol{u}}((\boldsymbol{v}_1, \boldsymbol{v}_2)) \geq \max\left\{ \frac{1}{4} \boldsymbol{\pi}_{\boldsymbol{u}}((n, \boldsymbol{v}_2)),\ \frac{1}{4} \boldsymbol{\pi}_{\boldsymbol{u}}((\boldsymbol{v}_1, n)) \right\}.$$

We begin by considering the first dimension:

$$\boldsymbol{\pi}_{\boldsymbol{u}}((\boldsymbol{v}_1, \boldsymbol{v}_2)) \geq \frac{\boldsymbol{\pi}_{\boldsymbol{u}}((n, \boldsymbol{v}_2))}{4}.$$

Let ℓ_{hor} be the horizontal line of reflection passing through $(\lceil (\boldsymbol{v}_1 + n)/2 \rceil, 1)$ and $(\lceil (\boldsymbol{v}_1 + n)/2 \rceil, n)$ in $\mathbb{Z}^2$, and let $\boldsymbol{u}^*$ be the reflection of $\boldsymbol{u}$ over ℓ_{hor}. Note that $\boldsymbol{u}^*$ may be outside of the $n \times n$ grid. Next, define the map

$$f : \mathcal{W}((n, \boldsymbol{v}_2) \to \boldsymbol{u}) \to \mathcal{W}((\boldsymbol{v}_1, \boldsymbol{v}_2) \to \boldsymbol{u})$$

as follows. For any walk $w \in \mathcal{W}((n, \boldsymbol{v}_2) \to \boldsymbol{u})$:

1. Start the walk $f(w)$ at $(\boldsymbol{v}_1, \boldsymbol{v}_2)$, and if $n - \boldsymbol{v}_1$ is odd move to $(\boldsymbol{v}_1 + 1, \boldsymbol{v}_2)$.

2. Perform w but make opposite vertical moves before the walk hits ℓ_{hor}, so that the partial walk is a reflection over ℓ_{hor}.

3. After hitting ℓ_{hor} for the first time, continue performing w, but now use the original vertical moves.

4. Terminate this walk when it first reaches $\boldsymbol{u}$.

Denote the preimage of a walk $w' \in \mathcal{W}((\boldsymbol{v}_1, \boldsymbol{v}_2) \to \boldsymbol{u})$ under f to be

$$f^{-1}(w') = \{w \in \mathcal{W}((n, \boldsymbol{v}_2) \to \boldsymbol{u}) : f(w) = w'\}.$$

We claim that for any $w' \in \mathcal{W}^{\text{SQUARE}_n}((\boldsymbol{v}_1, \boldsymbol{v}_2) \to \boldsymbol{u})$,

$$\frac{1}{4} \sum_{w \in f^{-1}(w')} 4^{-|w|} \le 4^{-|w'|}.$$

If $f^{-1}(w') = \emptyset$ the claim is true, so assume $f^{-1}(w') \ne \emptyset$. We analyze two cases. If w' hits ℓ_{hor}, then $f^{-1}(w')$ contains exactly one walk w of length $|w'|$ or $|w'| - 1$. If w' does not hit ℓ_{hor}, then

$$f^{-1}(w') = \{w \in \mathcal{W}((n, \boldsymbol{v}_2) \to \boldsymbol{u}) : w \text{ is a reflection of } w' \text{ over } \ell_{\text{hor}} \text{ before } w \text{ hits } \boldsymbol{u}^*\}.$$

It follows that any walk $w \in f^{-1}(w')$ can be split into $w = w_1 w_2$, where w_1 is the unique walk from $(n, \boldsymbol{v}_2)$ to $\boldsymbol{u}^*$ that is a reflection of w', and w_2 is a walk from $\boldsymbol{u}^*$ to $\boldsymbol{u}$ that avoids v_{sink} and hits $\boldsymbol{u}$ exactly once upon termination. Clearly w_1 has length $|w'|$ or $|w'| - 1$, and the set of admissible w_2 is $\mathcal{W}(\boldsymbol{u}^* \to \boldsymbol{u})$. Therefore,

$$\begin{aligned}\frac{1}{4} \sum_{w \in f^{-1}(w')} 4^{-|w|} &= 4^{-|w_1|-1} \sum_{w_2 \in \mathcal{W}(\boldsymbol{u}^* \to \boldsymbol{u})} 4^{-|w_2|} \\ &= 4^{-|w_1|-1} \boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{u}^*) \\ &\le 4^{-|w'|},\end{aligned}$$

since $\boldsymbol{\pi}_{\boldsymbol{u}}(\boldsymbol{u}^*)$ is an escape probability. Summing over all $w' \in \mathcal{W}((\boldsymbol{v}_1, \boldsymbol{v}_2) \to \boldsymbol{u})$, it follows from

Lemma 1.2.6 and the previous inequality that

$$\begin{aligned}\boldsymbol{\pi}_{\boldsymbol{u}}\left((\boldsymbol{v}_1, \boldsymbol{v}_2)\right) &= \sum_{w' \in \mathcal{W}((\boldsymbol{v}_1,\boldsymbol{v}_2)\to \boldsymbol{u})} 4^{-|w'|} \\ &\geq \sum_{w' \in \mathcal{W}((\boldsymbol{v}_1,\boldsymbol{v}_2)\to \boldsymbol{u})} \frac{1}{4} \sum_{w \in f^{-1}(w')} 4^{-|w|} \\ &\geq \frac{1}{4}\boldsymbol{\pi}_{\boldsymbol{u}}\left((n, \boldsymbol{v}_2)\right),\end{aligned}$$

because every $w \in \mathcal{W}((n, \boldsymbol{v}_2) \to \boldsymbol{u})$ is the preimage of a $w' \in \mathcal{W}((\boldsymbol{v}_1, \boldsymbol{v}_2) \to \boldsymbol{u})$.

Similarly, we can show that $\boldsymbol{\pi}_{\boldsymbol{u}}\left((\boldsymbol{v}_1, \boldsymbol{v}_2)\right) \geq \boldsymbol{\pi}_{\boldsymbol{u}}\left((\boldsymbol{v}_1, n)\right)/4$ for all $1 \leq \boldsymbol{v}_1 \leq n$ by reflecting walks over the vertical line from $(1, \lceil (n + \boldsymbol{v}_2)/2 \rceil)$ to $(n, \lceil (n + \boldsymbol{v}_2)/2 \rceil)$. Combining inequalities proves the claim. ☐

Lastly, we give a constant lower bound for the probability of an n-step simple symmetric walk being sufficiently close to its starting position by using the recursive definition of binomial coefficients and a Chernoff bound for symmetric random variables.

Lemma 1.3.4. *For all $n \geq 10$, we have*

$$\min\left\{ \frac{1}{2^n} \sum_{\substack{k=\lceil \frac{n}{4} \rceil \\ k \text{ odd}}}^{\lfloor \frac{3n}{4} \rfloor} \binom{n}{k}, \; \frac{1}{2^n} \sum_{\substack{k=\lceil \frac{n}{4} \rceil \\ k \text{ even}}}^{\lfloor \frac{3n}{4} \rfloor} \binom{n}{k} \right\} \geq \frac{2}{5}.$$

Proof. First observe that for $n \geq 10$, we have

$$\frac{1}{2^n} \sum_{\substack{k=\lceil \frac{n}{4} \rceil \\ k \text{ odd}}}^{\lfloor \frac{3n}{4} \rfloor} \binom{n}{k} \geq \frac{1}{2^n} \sum_{k \in \left(\frac{n-1}{4}, \frac{3(n-1)}{4}\right)} \binom{n-1}{k}$$

and

$$\frac{1}{2^n} \sum_{\substack{k=\lceil \frac{n}{4} \rceil \\ k \text{ even}}}^{\lfloor \frac{3n}{4} \rfloor} \binom{n}{k} \geq \frac{1}{2^n} \sum_{k \in \left(\frac{n-1}{4}, \frac{3(n-1)}{4}\right)} \binom{n-1}{k}.$$

To see this, use the parity restriction and expand the summands as

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Let $X_1, X_2, \ldots, X_{n-1}$ be independent Bernoulli random variables such that $\Pr[X_i = 0] = 1/2$ and $\Pr[X_i = 1] = 1/2$. Let $S_{n-1} = X_1 + X_2 + \cdots + X_{n-1}$ and $\mu = E[S_{n-1}] = (n-1)/2$. Using a Chernoff bound, we have

$$\begin{aligned}\frac{1}{2^n} \sum_{k \in \left(\frac{n-1}{4}, \frac{3(n-1)}{4}\right)} \binom{n-1}{k} &= \frac{1}{2}\left(1 - \Pr\left[|S_{n-1} - \mu| \geq \frac{1}{2}\mu\right]\right) \\ &\geq \frac{1}{2} - e^{-(n-1)/24} \\ &\geq \frac{2}{5},\end{aligned}$$

for $n \geq 60$. Checking the remaining cases numerically when $10 \leq n < 60$ proves the claim. □

CHAPTER 2
DYNAMIC SPECTRAL VERTEX SPARSIFIERS AND APPLICATIONS

This section is based on a joint work [26] with Jan van den Brand, Arun Jambulapati, Yin Tat Lee, Yang P. Liu, Richard Peng, and Aaron Sidford. In Section 2.5, we elaborate on faster dynamic Laplacian solver, which was implicit in [26]. As a summary, we make several advances broadly related to the maintenance of electrical flows in weighted graphs undergoing dynamic resistance updates, including:

1. More efficient dynamic spectral vertex sparsification, achieved by faster length estimation of random walks in weighted graphs using Morris counters [Morris 1978, Nelson-Yu 2020].
2. Direct reductions from
   (a) detecting edges with large energy in dynamic electric flows, and
   (b) solving dynamic Laplacian linear systems

   to dynamic spectral vertex sparsifiers.
3. A procedure for turning algorithms for estimating a sequence of vectors under updates from an oblivious adversary to one that tolerates adaptive adversaries via the Gaussian-mechanism from differential privacy.

Combining these pieces with modifications to prior robust interior point frameworks gives an algorithm that on graphs with m edges computes a mincost flow with edge costs and capacities in $[1, U]$ in time $\widetilde{O}(m^{3/2-1/58}\log^2 U)$.

2.1 Introduction

The maximum flow (maxflow) problem asks to route the maximum amount of flow between two vertices s and t in a directed graph G such that the amount of flow on every edge is at most its capacity. The more general minimum cost flow (mincost flow) problem asks to route a fixed demands in a directed graph G without sending more flow on any edge than its capacity, while minimizing a linear cost. Together these well-studied problems cover a wide range of combinatorial and numerical problems, including maximum cardinality bipartite matching, minimum s-t cut, shortest paths in graphs with negative edge length, and optimal transport (see e.g. [23, 24]).

While classical algorithms for these problems revolved around using augmenting paths or cycle primitives (such as blocking flows) [65, 66, 67, 68], the last decade has seen significant runtime improvements for maxflow and mincost flow in various settings based on *electrical flows*. For a graph G with vertex set V and edge set E, with edge resistances $\boldsymbol{r} \in \mathbb{R}^E_{\geq 0}$, and a demand vector $\boldsymbol{d} \in \mathbb{R}^E$, the electric flow on G is the flow that routes a fixed demand while minimizing the energy $\sum_{e \in E} \boldsymbol{r}_e \boldsymbol{f}_e^2$. Better maxflow algorithms have been given in several regimes using eletrical flows and stronger primitives [69, 70], including unit capacity graphs [8, 21, 22, 71, 72, 73], approximate maxflow on undirected graphs [19, 20, 74, 75, 76, 77, 78], and dense graphs [23, 24, 79].

However, it has been particularly challenging to obtain running time improvements for solving mincost flow and maxflow to high precision in sparse capacitated graphs. Recently, [25] gave an $\widetilde{O}(m^{1.5-1/328} \log U)$ time algorithm for sparse graphs with capacitaties bounded by U, the first improvement over $\widetilde{O}(m^{1.5} \log U)$ for maxflow on sparse graphs with arbitrary, polynomially bounded capacities. Their improvement involved an intricate interaction of dynamic data structures for electrical flows, a modification of the standard interior point method (IPM) outer loop which builds an maxflow via $\sqrt{m}$ approximate electric flows [80, 81], and sketching techniques. Additionally, issues relating to randomness in data structures and combinatorial reasoning about errors resulting from random walks required careful analysis that signficantly increase the runtime and resulted in the small improvement over $m^{1.5}$.

Our main result is an algorithm that while has a similar high-level picture as [25], significantly simplifies the major pieces described previously and their interactions. Specifically, we give a general purpose sketching tool for electrical flows, a graph theoretic (instead of algebraic) way of constructing the random walks at the core of the data structure, and handle randomness dependencies using ideas from differnetial privacy. Further, we provide modifications to prior robust interior point frameworks to do ℓ_2-based recentering (Definition 2.8.5) within a robust IPM via additional spectral vertex sparsification techniques. As a result, we achieve a faster runtime than in [25]. Also, as a result of our simplified electric flow data structure, our algorithm and IPM seamlessly extend to mincost flow. In constrast, the data structure complications in [25] restricted their algorithm to be applied to maxflow. [1]

Theorem 2.1.1. *There is an algorithm which given any m-edge directed graph G with integral capacities in $[1, U]$, feasible demand vector $\boldsymbol{d} \in \mathbb{Z}^E$, and an integral cost vector $\boldsymbol{c} \in [-U, U]^E$, computes a flow $\boldsymbol{f}$ that routes demand $\boldsymbol{d}$, satisfies the capacity constraints, and minimizes $\boldsymbol{c}^\top \boldsymbol{f}$. The algorithm succeeds whp. and runs in time $\widetilde{O}(m^{3/2-1/58} \log^2 U)$.*

Overall, this paper simplifies the key pieces of [25] and, as a result, clarifies the important components used to achieve faster maxflow algorithms via dynamic electric flows. Further, we consider each of these pieces to be interesting in their own right: dynamic maintenance of Schur complements, sketching and maintenance of high energy edges in dynamic electric flows, and understanding reductions between adaptive and oblivious adversaries. Ultimately, this paper be read independently of [25] and the proofs are simpler and more natural in many cases.

2.1.1 Key Algorithmic Pieces

Here we cover the key algorithmic pieces underlying our algorithm. The first major piece is a faster algorithm for generating random walks of a fixed length from a vertex, a core primitive in all random-walk based approaches to dynamic electric flows [15, 25]. Our second key contribution is an algorithm for detecting large energy edges in electric flows for graphs with dynamically changing resistances and demands based on a direct reduction to dynamic spectral vertex sparsifiers (Schur complements). As our data structures for maintaining dynamic electric flows naturally work only against oblivious adversaries, we develop an approach that black-box reduces dynamic electric flows against adaptive adversaries to the same problem against oblivious adversaries, at the cost of a small runtime increase.

[1][82], in FOCS 2021, claims an improvement to sparse capacitated mincost flow in the title. A preprint was recently made available at https://arxiv.org/abs/2111.10368v1. The results in this paper were derived independently: we defer detailed comparisons to a future version.

Faster generation of random walks and Schur complements. All previous algorithms for dynamic electric flows [15, 25] require dynamic maintenance of spectral vertex sparsifiers or Schur complements, which approximate the electric flow and potentials onto a smaller set of terminal vertices. The Schur complement is generated by sampling several random walks from vertices v with exit probabilities proportional to inverse resistances until the walk visited a fixed number L of distinct vertices, and by estimating the sum of resistances of edges along the walk.

Because there may be edges with very large or small resistances, a naïve simulation may get stuck for polynomially many steps. Consequently, [15] gave an algorithm for this based on taking high powers of the random walk matrix (which [25] applied in a black-box fashion). This generated large factors in the runtime of the data structures. We give an approach to signficantly speed up the sampling of vertices and length estimation by applying a *Morris counter* from the streaming/sketching literature [83], and reducing the problem to solving a sequence of electric flow computations (Laplacian systems) as opposed to the more expensive matrix multiplication operations of [15]. This signficant runtime improvement immediately translates to our dynamic electric flow data structure described above, which directly uses dynamic Schur complements.

Simplified electric flow heavy hitter. To design our dynamic algorithm for detecting edges with large energy in dynamic electric flows, we maintain an ℓ_2 heavy hitter sketch of the electric flow vector. The algorithm of [25] maintained this sketch by using a dynamic spectral vertex sparsifier or Schur complement, which approximates the electric flow and potentials on a smaller set of terminal vertices, and several random walks for "moving" the heavy-hitter sketch vector to the terminal set. This latter piece (maintaining random walks for moving the heavy hitter vector) introduced several complications into the analysis and generated a large overall running time for the data structure. On the other hand, our algorithm is more directly based on spectral approximations. In particular, we show how to dynamically maintain the result of moving the heavy hitter vector onto the terminal set by simply calling another dynamic Schur complement data structure, and carefully reasoning about spectral approximations to bound how that affects the resulting error.

Simplified IPM outer loop. As a result of our more linear algebraic approach to maintaining electric flows, our data structure for detecting large energy edges in electric flows works for both dynamic resistance changes and dynamic demands, while the algorithm of [25] required restricting to only s-t flows. Our generalization also allows us to use a more standard and efficient robust IPM (from [84]) to implement the outer loop utilizing the data structure, while [25] had to redesign the IPM to carefully only use s-t electric flows to interact with their data structure. Our robust IPM implements an additional batching, or ℓ_2-based recentering step, by computing the changes on a small subset of edges to higher accuracy by using spectral vertex sparsifiers.

Black-box reduction from adaptive to oblivious adversaries. As we are applying randomized data structure inside an algorithmic outer loop, their previous responses may affect future updates. This is referred to an *adaptive adversary* in the literature. On the other hand, our data structures which are based on random walks naturally only work against *oblivious adversaries*, where the input sequence does not depend on the outputs and randomness of the data structure. The algorithm of [25] handled this issue in their data structures by carefully controlling the total number of adaptive phases of their algorithm before snapping back to a deterministic state.

Our approach on the other hand is more black-box, and gives a more general approach for converting data structures against oblivious adversaries to handle adaptive queries. We build a LOCATOR which returns a superset of edges with large energies, and several EVALUATORS with differing accuracy parameters which separately estimate the energies of the edges. By leveraging ideas from the Gaussian-mechanism from differential privacy [85] we show how to apply the EVALUATOR data structures to simulate estimating adding Gaussian noise to the true energy vector that we wish to output. We simulate this by making several queries to the EVALUATORS, where we query the least accurate EVALUATORS most often, and only query more accurate EVALUATORS when the estimate of the energy vector is close to certain thresholds and we require finer estimates to decide how to round. Because we are simulating adding noise to the true output, the algorithm succeeds against an adaptive adversary.

2.1.2 Related Work

We briefly survey the lines of work most relevant to our results, and refer the reader to [25] for more comprehensive discussion. Recently, [86] gave a mincost flow algorithm on planar graphs running in nearly linear time. Similar to our paper, it is based on the robust IPM framework of [84] and dynamic Schur complements. However, [86] relies on the fact that the terminal set C is small due to the existence of planar separators, while our paper relies on the fact that C is slowly changing.

Data structures for IPMs. IPMs are a powerful framework which reduces linear programming with m variables to a sequence of $\widetilde{O}(\sqrt{m})$ linear system solutions [87]. For maxflow and mincost flow, these linear systems correspond to computing electrical flows, and Daitch-Spielman [9] leveraged this observation to give a $\widetilde{O}(m^{1.5} \log U)$ mincost flow algorithm. Recently, several works have leveraged the key fact dating back to early works of Karmarkar [80] and Vaidya [81] that the linear systems change slowly and only need to be solved approximately, both in the context of linear programs [24, 88, 89, 90, 91, 92, 93, 94] and mincost flows [23, 24, 25].

Dynamic electrical flows. Recent works applying dynamic data structures to IPMs for maxflow require maintaining various properties of electrical flows on dynamically changing graphs. The improvements on dense graphs [23, 24] required dynamically maintaining spectral sparsifiers of the Laplacian in $\widetilde{O}(1)$ time per edge update and $\widetilde{O}(n)$ per query, as well as detecting edges with large electrical energies in $\widetilde{O}(n)$ time per query. Both of these pieces were done using dynamic expander decompositions [95, 96, 97, 98, 99, 100]. The work of [25] desired *sublinear* time per query and hence required dynamically maintaining Schur complements, whose study was initiated in [15] to dynamically maintain approximate effective resistances.

Adaptivity and differential privacy. There has been significant work towards building techniques to apply oblivious data structures in the context of an algorithmic outer loop, which requires adaptivity. To date, most approaches to this problem involve either making the algorithm deterministic [101, 102, 103, 104, 105, 106], or resparsifying [100], both of which heavily leverage properties provided by dynamic expander decompositions [95, 96, 97, 98, 99, 100]. Our work takes a different perspective and instead more carefully analyzes whether the adversary can learn any randomness leaked from the distribution of our output vector. This perspective is motivated by ideas from differential privacy, and in fact our key result is an adaptation of the Gaussian mechanism [85] which simulates adding unbiased Gaussian noise to the true output vector by using a sequence of oblivious estimates. Our recursive scheme is also broadly related to the idea of multilevel Monte Carlo [107, 108] and its recent applications in leveraging approximate optimization procedures to obtain nearly unbiased estimates of minimizers [109].

2.1.3 General Notation

We use plaintext to denote scalars, bold lowercase for vectors, and bold uppercase for matrices. For resistances $\boldsymbol{r}$ and conductances $\boldsymbol{w} \stackrel{\text{def}}{=} \boldsymbol{r}^{-1}$, the corresponding capital matrices are diagonal matrices with the vector entries on the diagonal, i.e. $\mathbf{R} \stackrel{\text{def}}{=} \text{diag}(\boldsymbol{r})$ and $\mathbf{W} \stackrel{\text{def}}{=} \text{diag}(\boldsymbol{w})$. As our algorithm heavily use approximations, we will use $\tilde{\cdot}$ to denote the approximate versions of true variables.

We use $\widetilde{O}(\cdot)$ to suppress logarithmic factors in m and $\widetilde{\Omega}(\cdot)$ to suppress inverse logarithmic factors in m. For vectors $\boldsymbol{x}, \boldsymbol{y}$ we sometimes let $\boldsymbol{xy}$ denote the entry-wise product of $\boldsymbol{x}, \boldsymbol{y}$, so $(\boldsymbol{xy})_i \stackrel{\text{def}}{=} \boldsymbol{x}_i \boldsymbol{y}_i$. Similarly, we let $(\boldsymbol{x}/\boldsymbol{y})_i \stackrel{\text{def}}{=} \boldsymbol{x}_i/\boldsymbol{y}_i$. We say that an event holds with high probability (whp.) if for any constant $C > 0$, the event succeeds with probability at least $1 - n^{-C}$ by adjusting parameters. We let $[n] = \{1, 2, \ldots, n\}$. We denote the (unweighted) degree of a vertex v as $\deg(()v)$.

We say that a symmetric matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ is positive semidefinite (PSD) if $\boldsymbol{x}^\top \mathbf{A} \boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$. For PSD matrices $\mathbf{A}, \mathbf{B}$ we write $\mathbf{A} \preceq \mathbf{B}$ if $\mathbf{B} - \mathbf{A}$ is PSD. For positive real numbers a, b we write $a \approx_\gamma b$ to denote $\exp(-\gamma) b \leq a \leq \exp(\gamma) b$. For PSD matrices $\mathbf{A}, \mathbf{B}$ we write $\mathbf{A} \approx_\gamma \mathbf{B}$ if $\exp(-\gamma)\mathbf{B} \preceq \mathbf{A} \preceq \exp(\gamma)\mathbf{B}$.

2.1.4 Organization

In Section 2.2, we give a technical overview of each of our improvements to each of the key components of [25]: faster sampling of Schur complements, operator-based electric flow heavy hitters, and black-box reduction of adaptive to oblivious adversaries. We also overview the robust IPM we use. In Section 2.3 we give preliminaries for maxflow, mincost flow, and electric flows that we require for the remainder of our paper. In Section 2.4 we give our algorithm for faster sampling of random walks and Schur complements, and we combine this with an operator-based heavy hitter in Section 2.6 to give a faster algorithm for detecting edges with large energy. In Section 2.7 we show how to black-box reduce adaptive to oblivious adversaries for the problem of estimating dynamic vectors. We give our robust IPM in Section 2.8, which is an adaptation of that in [84], and additional tools to apply it. Finally, we combine all pieces and compute the final runtime in Section 2.9.

2.2 Overview

Here w,e provide a technical overview of our contributions.

2.2.1 Overview of Faster Schur Complements via the Morris walk

Our data structures, as in [25], heavily rely on dynamically maintaining spectral vertex sparsifiers (Schur complements) of G, which approximate the inverse spectral form of G onto a subset of the vertices. This was achieved using the algorithm of [15], which showed how to dynamically maintain an approximate Schur complement under edge resistance updates. The main primitive behind the dynamic Schur complement data structure was a procedure to sample random walks from a vertex with exit probability proportional to inverse resistances, i.e. the probability of going from a vertex v to a neighbor u is given by

$$\frac{\boldsymbol{r}_{vu}^{-1}}{\sum_{w \text{ neighbor of } v} \boldsymbol{r}_{vw}^{-1}}.$$

For this walk and a parameter L, we must run the walk until the total degree of visited vertices is L, and to estimate the total *resistive length* of the walk up to a $(1+\epsilon)$-factor, where resistive length refers to the sum of resistances of edges on the walk. Directly simulating the random walk is not efficient enough, because there may be polynomially large and small resistances, which cause the walks to "get stuck" on a small set of edges, without visiting new vertices. Thus it can take a long time to visit L distinct vertices. Despite this, [15] showed how to sample the walk and resistive length in $\widetilde{O}(L^4\epsilon^{-2})$ time per vertex, and this large runtime directly led to the fact that [25] only achieved a small $1/328$ improvement in the exponent.

Interestingly, if one is only interested in obtaining distinct vertices on the walk (and not the resistive length) until the total degree is L, this can be done in $\widetilde{O}(L^2)$ time by solving a Laplacian system (corresponding to computing an electric flow) for each of at most L steps to compute the next exit vertex. However, the approach of [15] which also computed the resistive length, i.e. the sum of resistances of edges on the walk, was based on matrix-powering/matrix multiplication, and instead had a larger $\widetilde{O}(L^4\epsilon^{-2})$ runtime. Our main idea is to resolve this runtime discrepancy between sampling the distinct vertices and computing a $(1+\epsilon)$-resistive length estimate by giving an algorithm that computes both quantities by solving a sequence of Laplacian systems. In total, we solve $\widetilde{O}(L+\epsilon^{-2})$ systems, for a runtime of $\widetilde{O}(L^2+L\epsilon^{-2})$. In our settings, L will generally be $\Omega(\epsilon^{-2})$, so our runtime is $\widetilde{O}(L^2)$, matching the time to generate the first L vertices using a sequence of Laplacian systems, and significantly improving over the $\widetilde{O}(L^4\epsilon^{-2})$ runtime of [15, 25].

Our algorithm for this task is derived from the Morris counter [83, 110], a probabilistic algorithm for maintaining low-space approximations to a counter N undergoing increments. For simplicity of exposition, we assume our input graph has integer, polynomially-bounded edge weights. Our algorithm intuitively begins by running a random walk in G. However, we replace the naive procedure for computing the resistive length with a Morris counter. More precisely, assume we have run a random walk starting from a vertex u for k steps and have estimated the resistive length of the walk via the Morris counter. To estimate the resistive length of this walk after a further step, we simply sample one new step of the walk: if we sampled an edge of resistance length w, we increment the Morris counter w times. In this way, the Morris counter enables us to maintain estimates of the resistive length of a random walk.

We use the following properties of Morris counters as shown in [83]. First, they take discrete values of $\frac{1}{a}\left((1+a)^i-1\right)$ for some real number $a>0$ and integers $i\geq 0$, and if $a=\epsilon^2/\text{poly}\log n$, the value of the Morris counter is always a $(1+\epsilon)$-approximation of the true value with high probability. In particular, for graphs with polynomially bounded weights, the Morris counters takes at most $\widetilde{O}(a^{-1})=\widetilde{O}(\epsilon^{-2})$ distinct values.

Our key insight we can simulate incrementing of the Morris counter and the sequence of vertices visited as a random walk on a "lifted" graph with $\widetilde{O}(\epsilon^{-1})$ layers. Each time we explore a "new" neighbor in this lifted space, we either find a new unexplored vertex along the random walk or increment the Morris counter. However, there are only $\widetilde{O}(\epsilon^{-2})$ distinct values of the counter: thus we must explore only an additional $\widetilde{O}(\epsilon^{-2})$ distinct vertices in this lifted space to obtain the desired guarantee on the number of new vertices seen. We obtain our final algorithm by replacing the explicit random walks with a subroutine based on Laplacian linear system solvers: in this way our final complexity of $\widetilde{O}(L^2 + L\epsilon^{-2})$ follows. Overall, this provides a graph-theoretic approach for estimating lengths of random walks on graphs, as opposed to the previous algorithm in [15] which was based on matrix mutiplication.

2.2.2 Overview of Operator-based Electric Flow Heavy Hitters

Our next major improvement over [25] is a data structure that detects large energy edges in electric flows on graphs with dynamic resistances and demands by direct reduction to maintaining dynamic Schur complements. To be precise, we give a data structure that on a graph G with dynamically changing resistances and demands, solves a *electric flow heavy hitter* problem, by returning a set S of $O(\epsilon^{-2})$ edges containing all edges e with at least ϵ^2 fraction of the electric energy, i.e. $\boldsymbol{r}_e \boldsymbol{f}_e^2 \geq \epsilon^2 \sum_{e \in E} \boldsymbol{r}_e \boldsymbol{f}_e^2$ where $\boldsymbol{f}$ is the electric flow vector. [25] gave a data structure that solved this problem in sublinear time per resistance update and query as a core piece of their algorithm. We give improved runtimes for solving this problem and shed light on its complexity by directly reducing to dynamically maintaining Schur complements.

Note that the dynamic electric flow heavy hitter problem is equivalent to detecting large coordinates of the vector $\mathbf{R}^{1/2}\boldsymbol{f}$ compared to its ℓ_2 norm. Hence, it is natural to apply an ℓ_2 heavy-hitter sketch [111], which at a high-level consists of $\widetilde{O}(\epsilon^{-2})$ Johnson-Lindenstrauss ℓ_2 sketches. In total, this consists of maintaining the value of $\boldsymbol{q}^\top \mathbf{R}^{1/2}\boldsymbol{f}$ for $\widetilde{O}(\epsilon^{-2})$ random sketch vectors $\boldsymbol{q} \in \{-1, 0, 1\}^E$. The flow $\boldsymbol{f}$ can be represented as $\boldsymbol{f} = \mathbf{R}^{-1}\mathbf{B}\boldsymbol{\phi}$ for electric potentials $\boldsymbol{\phi}$ and edge-vertex incidence matrix $\mathbf{B}$, so

$$\boldsymbol{q}^\top \mathbf{R}^{1/2}\boldsymbol{f} = \langle \mathbf{B}^\top \mathbf{R}^{-1/2}\boldsymbol{q}, \boldsymbol{\phi} \rangle.$$

Let $\boldsymbol{y} = \mathbf{B}^\top \mathbf{R}^{-1/2}\boldsymbol{q}$, so that we focus on maintaining $\boldsymbol{y}^\top \boldsymbol{\phi}$. However, $\boldsymbol{\phi}$ is still a $|V|$-dimensional vector, so in order to achieve sublinear time [25] used a smaller terminal set C to estimate $\boldsymbol{y}^\top \boldsymbol{\phi}$.

In particular, they write $\boldsymbol{\phi} = \mathcal{H}_C \boldsymbol{\phi}_C$ where $\boldsymbol{\phi}_C$ is the restriction of $\boldsymbol{\phi}$ to C, and $\mathcal{H}_C \in \mathbb{R}^{V(G) \times C}$ is the *harmonic extension* (Definition 2.5.2) operator which extends $\boldsymbol{\phi}_C$ to $\boldsymbol{\phi}$ using that for any vertex v, $\boldsymbol{\phi}_v$ is the average of its neighbors, weighted proportional to inverse resistances. This way, $\boldsymbol{y}^\top \boldsymbol{\phi} = \langle \mathcal{H}_C^\top \boldsymbol{y}, \boldsymbol{\phi}_C \rangle$. Assuming that we can approximate maintain $\boldsymbol{\phi}_C$ (which we discuss towards of the end of this section's overview), it suffices to maintain $\mathcal{H}_C^\top \boldsymbol{y}$.

Our major difference from [25] is in how we maintain $\mathcal{H}_C^\top \boldsymbol{y}$. While [25] used the combinatorial interpretation of the operator $\mathcal{H}_C^\top$ as using random walks to "move" the mass from vector $\boldsymbol{y}$ onto C, we use the spectral fact (Lemma 2.5.4) that

$$\mathcal{H}_C = \mathbf{L}^\dagger \begin{bmatrix} 0 & \mathbf{SC}(\mathbf{L}, C) \end{bmatrix},$$

where $\mathbf{L}$ is the graph Laplacian and $\mathbf{SC}(\mathbf{L}, C)$ is the Schur complement of $\mathbf{L}$ onto C. Thus, we get

$$\mathcal{H}_C^\top \boldsymbol{y} = \begin{bmatrix} 0 \\ \mathbf{SC}(\mathbf{L}, C) \end{bmatrix} \mathbf{L}^\dagger \boldsymbol{y}.$$

Thus, we could optimistically precompute $\mathbf{L}^\dagger \boldsymbol{y}$ and then compute $\mathcal{H}_C^\top \boldsymbol{y}$ as long as we can dynamically maintain the Schur complement $\mathbf{SC}(\mathbf{L}, C)$, which is a size C object. The remaining issue is that the Laplacian $\mathbf{L}$ may change because the resistances change. However, the operator $\mathcal{H}_C^\top$ does not depend on the resistances of edges completely inside C (by definition), so we may actually let $\mathbf{L}$ be the Laplacian of the *original graph* as long as all endpoints of edges with resistance changes are added to C. Finally we are able to show that using an approximate Schur complement in place of $\mathbf{SC}(\mathbf{L}, C)$ still suffices for our data structures (Lemma 2.6.5).

Finally we discuss the (approximate) maintenance of the potential $\boldsymbol{\phi}_C$. For an electric flow $\boldsymbol{f}$ routing demand $\boldsymbol{d}$, i.e. $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d}$, the potentials $\boldsymbol{\phi}_C$ are given by

$$\boldsymbol{\phi}_C = \mathbf{SC}(\mathbf{L}, C)^\dagger \mathcal{H}_C^\top \boldsymbol{d}.$$

In other words, we first "move" the demands to the terminal set using $\mathcal{H}_C^\top$ just as above, and then invert the Schur complement on it. Therefore we can maintain $\boldsymbol{\phi}_C$ as follows: maintain $\mathcal{H}_C^\top \boldsymbol{d}$

approximately as above, and then also approximate maintain $\mathbf{SC}(\mathbf{L}, C)$ using an approximate Schur complement data structure. In all, this reduces the maintenance of the heavy hitter vector to three calls to an approximate Schur complement oracle.

2.2.3 Overview of Reduction from Adaptive to Oblivious Adversaries

The dynamic electric flow data structures built in Section 2.2.2 naïvely only work against oblivious adversaries, i.e. the inputs must be independent of the outputs and randomness of the data structure. In [25] this was handled by carefully designing the data structures to utilize the fact that the IPM central path is a deterministic object. However, we take a more general approach, by applying ideas from the Gaussian-mechanism from differential privacy [85] to build versions of these data structures that work directly against adaptive adversaries, allowing them to be applied within the interior point outer loop. In fact, we give a generic reduction for estimating vectors against adaptive adversaries to oblivious adversaries.

Consider an oblivious data structure that outputs vectors $\overline{\boldsymbol{v}} \in \mathbb{R}^m$ that are supposed to approximate a true underlying vector $\boldsymbol{v} \in \mathbb{R}^m$. In our dynamic electric flow setting, this corresponds to a data structure which detects edges with large electric energy, and approximates their flow values. Consider what would happen if instead of $\overline{\boldsymbol{v}}$, our algorithm uses $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{v}, \sigma^2)$ for small enough σ (i.e. the vector $\boldsymbol{v}$ with some Gaussian noise added to it). If σ is small enough, then $\boldsymbol{z}$ would be an accurate approximation for our algorithm to work. Additionally, the vector $\boldsymbol{z}$ obviously does not depend on the internal randomness of the data structure, since it is defined with respect to $\boldsymbol{v}$, not the approximation $\overline{\boldsymbol{v}}$. Unfortunately, computing $\boldsymbol{z}$ by computing $\boldsymbol{v}$ and adding noise is rather inefficient since $\boldsymbol{v}$ is the exact solution, not an approximation. We now explain how to obtain vector $\boldsymbol{z}$ more efficiently from oblivious estimate vectors $\overline{\boldsymbol{v}}$ by using the Gaussian-mechanism from differential privacy [85].

Specifically, it is known that for any $\sigma > 0$ there is small enough $\alpha > 0$ such that if d is the density function of $\mathcal{N}(\boldsymbol{v}, \sigma^2)$ and $\overline{d}$ is the density function of $\mathcal{N}(\overline{\boldsymbol{v}}, \sigma^2)$, then $\overline{d}(\boldsymbol{x}) \leq \exp(\alpha) d(\boldsymbol{x})$ for all $\boldsymbol{x}$.[2] For example, Figure 2.1 shows density function $d(\boldsymbol{x})$ and the scaled density function

[2]This is actually only true for $\boldsymbol{x} \in D$ for some event D that holds whp. We ignore this here for simplicity.

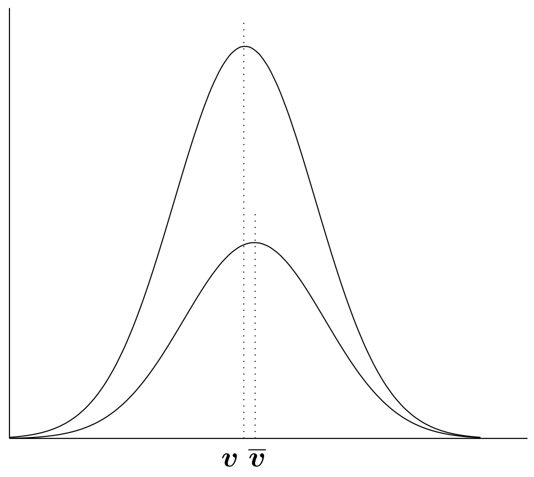

Figure 2.1: Density function d of $\mathcal{N}(\boldsymbol{v}, \sigma^2)$, and density function $\overline{d}$ of $\mathcal{N}(\overline{\boldsymbol{v}}, \sigma^2)$ scaled by some $\exp(-\alpha)$, $\alpha > 0$ so that $\overline{d}(\boldsymbol{x})\exp(-\alpha) \leq d(\boldsymbol{x})$.

$\exp(-\alpha)\cdot\overline{d}(\boldsymbol{x})$ for the 1-dimensional case. Note that we can pick a random $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{v}, \sigma^2)$ by picking uniformly at random a point below the curve of $d(\overline{\boldsymbol{x}})$ and returning the x-coordinate. We can also split this sampling scheme into two phases: (i) With probability $1-\exp(-\alpha)$, sample from the area between the two curves. (ii) Alternatively, with probability $\exp(-\alpha)$ sample from the area below the bottom curve $\exp(-\alpha)\overline{d}(\boldsymbol{x})$ in Figure 2.1.

When case (i) happens, we handle it directly by computing $\boldsymbol{v}$ exactly (which is expensive), which gives us the distributions d and $\overline{d}$ explicitly. However, note that if α is close to 0, then this case only occurs infrequently: with probability $1 - \exp(-\alpha) = O(\alpha)$, which balances out the expensive cost of computing $\boldsymbol{v}$. On the other hand, case (ii), which occurs with probability $\exp(-\alpha)$, corresponds to flipping an unbalanced coin and with probability $\exp(-\alpha)$ we sample a $\boldsymbol{z}' \sim \mathcal{N}(\overline{\boldsymbol{v}}, \sigma^2)$. So with probability $\exp(-\alpha)$ we do not need to know/compute the exact vector $\boldsymbol{v}$ in order to obtain a sample with distribution $\mathcal{N}(\boldsymbol{v}, \sigma^2)$ and just knowing the approximate result $\overline{\boldsymbol{v}}$ already suffices.

Now, this scheme can be extended recursively to handle case (ii), i.e. sampling from $\boldsymbol{z} \sim \mathcal{N}(\overline{\boldsymbol{v}}, \sigma^2)$. We can use the same scheme again via some $\overline{\boldsymbol{v}}'$, i.e. sampling from $\mathcal{N}(\overline{\boldsymbol{v}}', \sigma^2)$ with probability $\exp(-\alpha)$ instead of $\mathcal{N}(\overline{\boldsymbol{v}}, \sigma^2)$. This leads to another speed-up because of the following reason: the probability $\exp(-\alpha)$ depends on the approximation quality of $\overline{\boldsymbol{v}}'$ compared to $\boldsymbol{v}$. We want to use a large α in order to reduce the probability of computing $\boldsymbol{v}$, but this requires $\overline{\boldsymbol{v}}'$ to be a better approximation. Thus, we are able to compute higher accuracy approximations (which take more runtime) less frequently, and this leads to a speedup. Overall, by using this scheme, our data structures will work against an adaptive adversary because the output has distribution $\mathcal{N}(\boldsymbol{v}, \sigma^2)$, i.e. a distribution that is independent of the internal randomness of the data structures.

2.2.4 Overview of IPM Outer Loop

Here we overview how we apply the above primitives in a robust IPM to give an algorithm for algorithm, which reduces solving maxflow to computing a sequence of $\widetilde{O}(\sqrt{m})$ approximate electric flows. The IPM of [25] required several nonstandard modifications, including restricting to using s-t flows, which resulted in using more than $\widetilde{O}(\sqrt{m})$ steps, and overall higher runtime. On the other hand, our algorithm is based on the more standard robust IPM of [84], with an additional procedure that allows for recentering in the context of a robust IPM that allows us to control errors that accumulate over longer periods of time.

We start by briefly introducing a standard robust IPM setup for the mincost flow problem based on [84] (in Section 2.8 we change notation slightly to work with general linear programs)

$$\min_{\boldsymbol{f} \in \mathbb{R}^m : \mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d} \text{ and } \boldsymbol{\ell} \le \boldsymbol{f} \le \boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{f}, \tag{2.1}$$

where $\boldsymbol{c} \in \mathbb{R}^E$ is the cost vector, and $\boldsymbol{\ell}, \boldsymbol{u} \in \mathbb{R}^E$ are lower/upper capacities on edges. For $e \in E$ and real number $f \in \mathbb{R}$, define the logarithmic barrier function $\phi_e(f) \stackrel{\text{def}}{=} -\log(f - \boldsymbol{\ell}_e) - \log(\boldsymbol{u}_e - f)$, and for flow $\boldsymbol{f} \in \mathbb{R}^E$ define $\phi(\boldsymbol{f}) \stackrel{\text{def}}{=} \sum_{e \in E} \phi_e(\boldsymbol{f}_e)$. For a path parameter μ that decreases towards 0 over the course of $\widetilde{O}(\sqrt{m})$ steps, the robust IPM maintains an approximate minimizer to the expression

$$\boldsymbol{f}_\mu \stackrel{\text{def}}{=} \min_{\boldsymbol{f} \in \mathbb{R}^m : \mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d} \text{ and } \boldsymbol{\ell} \le \boldsymbol{f} \le \boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{f} + \mu \phi(\boldsymbol{f}) \,. \tag{2.2}$$

Since ϕ is convex, the KKT conditions for (2.2) give that there is a vector $\boldsymbol{y}$ such that $\boldsymbol{c} + \mu \nabla \phi(\boldsymbol{f}_\mu) = \mu \mathbf{B} \boldsymbol{y}$. Thus there is a vector $\boldsymbol{s}_\mu \in \mathbb{R}^E$ such that $\mathbf{B}\boldsymbol{y} + \boldsymbol{s}_\mu = \boldsymbol{c}/\mu$ and $\boldsymbol{s}_\mu + \nabla \phi(\boldsymbol{f}_\mu) = 0$. In this way, we define a *μ-centered point* as a pair $(\boldsymbol{f}, \boldsymbol{s})$ such that $\mathbf{B}\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c}/\mu$ for some $\boldsymbol{y} \in \mathbb{R}^V$ and $\|\nabla^2 \phi(\boldsymbol{f})^{-1/2}(\boldsymbol{s} + \nabla \phi(\boldsymbol{f}))\|_\infty \le 1/64$. The robust IPM maintains μ-centered points throughout by tracking the potential function

$$\sum_{e \in E} \cosh\left(\lambda \phi_e''(\boldsymbol{f}_e)^{-1/2}(\boldsymbol{s}_e + \phi_e'(\boldsymbol{f}_e))\right)$$

for $\lambda = 128 \log(16m)$. Now IPM steps are taken to simulate gradient descent steps on the potential to keep it small, and hence maintain μ-centered points at all times.

Because our data structures work in time sublinear in the number of vertices, the flows we maintain during the robust IPM are stored implicitly, even without the ability to query in $\widetilde{O}(1)$ time the "true flow" on an edge e. Further, the error of our flow estimate from the true value accumulates over the steps of our method. Hence we require a procedure to recompute a feasible μ-centered flow in a robust IPM every k steps in $\widetilde{O}(m)$ time for some $k = m^{\Omega(1)}$ (Theorem 2). To see why this could be possible, note that the true step per iteration is an electric flow with some resistances and demands. Additionally, over the course of k steps, these resistances and demands will only change at most $\mathrm{poly}(k)$ total times. Thus, we can put all edges whose resistance or demand changed into a terminal set C and compute an ϵ-approximate Schur complement onto C. [112] shows that such a Schur complement (onto a slightly larger set) can be computed in time $\widetilde{O}(m + |C|/\epsilon^2) = \widetilde{O}(m + \mathrm{poly}(k)/\epsilon^2) = \widetilde{O}(m)$ for some $k = m^{\Omega(1)}$. Leveraging this, we show that we can recover a centered point in the context of a robust IPM in $\widetilde{O}(m)$ time every k steps.

Overall, our algorithm splits the $\widetilde{O}(\sqrt{m})$ robust IPM steps in $\widetilde{O}(\sqrt{m}/k)$ batches of k steps. Within each batch, we ensure that at most $\mathrm{poly}(k)$ edges have their resistances change in the graph G (but there may be more resistance updates in between batches). Each step in the batch is maintained using the dynamic electric flow heavy hitter data structure we built, as described in Sections 2.2.1 to 2.2.3. At the end of each batch, we use the approximate recentering procedure described in the previous paragraph. Combining these pieces along with the standard bound that over T IPM steps, at most $\widetilde{O}(T^2)$ resistances change by a constant factor, gives our final runtimes.

2.3 Preliminaries

We give preliminaries on maxflow, mincost flow, electric flows, and Schur complements.

Maxflow and mincost flow. Throughout, we let $G = (V, E)$ be our graph with $n = |V|$ vertices and $m = |E|$ edges. We let $\mathbf{B} \in \mathbb{R}^{E \times V}$ denote the edge-vertex incidence matrix of G. Additionally, we let $\boldsymbol{\ell}, \boldsymbol{u} \in \mathbb{Z}^E$ denote the lower/upper capacities on edges in G. We assume that $\|\boldsymbol{\ell}\|_\infty, \|\boldsymbol{u}\|_\infty \le U$. A *flow* $\boldsymbol{f} \in \mathbb{R}^E$ is any assignment of real numbers of edges of G. We say that a flow $\boldsymbol{f}$ is *feasible* if $\boldsymbol{\ell}_e \le \boldsymbol{f}_e \le \boldsymbol{u}_e$ for all $e \in E$. We say that $\boldsymbol{f}$ routes the demand $\boldsymbol{d} \in \mathbb{R}^V$ if $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d}$.

The maximum flow problem asks to find a feasible flow routing the maximum multiple of a demand $\boldsymbol{d}$ (generally assumed to be s-t). Written linear algebraically, this asks to find the largest F^* such that there is a flow $\boldsymbol{f}$ satisfying $\mathbf{B}^\top \boldsymbol{f} = F^* \boldsymbol{d}$ and $\boldsymbol{\ell}_e \le \boldsymbol{f}_e \le \boldsymbol{u}_e$ for all $e \in E$. The minimum cost flow problem asks to minimize a linear cost $\boldsymbol{c}$ over flows routing a fixed demand $\boldsymbol{d}$. Linear algebraically, this can be written as

$$\min_{\substack{\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d} \\ \boldsymbol{\ell}_e \le \boldsymbol{f}_e \le \boldsymbol{u}_e \text{ for all } e \in E}} \boldsymbol{c}^\top \boldsymbol{f}.$$

We work with mincost flow throughout, as it is known to generalize maxflow. We also focus on finding high-accuracy solutions in runtime depending logarithmically on U and $\|\boldsymbol{c}\|_\infty$, as it is known that this suffices to get an exact solution with linear time overhead [9, 24].

Electric flows and Schur complements. Electric flows are ℓ_2-minimization analogues of maxflow on undirected graphs, and are used in all current state-of-the-art high accuracy maxflow algorithms [22, 23, 24, 25, 73] based on IPMs. On a graph G with resistances $\boldsymbol{r}$, the electric flow routing demand $\boldsymbol{d}$ is given by

$$\arg\min_{\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d}} \sum_{e \in E} \boldsymbol{r}_e \boldsymbol{f}_e^2. \tag{2.3}$$

The minimizer in (2.3) is given by the solution to a linear system: $\boldsymbol{f} = \mathbf{R}^{-1}\mathbf{B}(\mathbf{B}^\top \mathbf{R}^{-1} \mathbf{B})^\dagger \boldsymbol{d}$. The matrix $\mathbf{B}^\top \mathbf{R}^{-1} \mathbf{B}$ is known as the *Laplacian* of G, which can be solved in nearly-linear time [13, 14, 113, 114, 115, 116, 117, 118, 119]. Precisely, solving a Laplacian system gives high accuracy *vertex potentials*, defined as $\boldsymbol{\phi} = (\mathbf{B}^\top \mathbf{R}^{-1} \mathbf{B})^\dagger \boldsymbol{d}$.

Theorem 2.3.1. *Let G be a graph with n vertices and m edges. Let $\boldsymbol{r} \in \mathbb{R}^E_{>0}$ denote edge resis-*

tances. For any demand vector $\boldsymbol{d}$ *and* $\epsilon > 0$ *there is an algorithm which computes in* $\widetilde{O}(m \log \epsilon^{-1})$ *time* potentials $\boldsymbol{\phi}$ *such that* $\|\boldsymbol{\phi} - \boldsymbol{\phi}^*\|_{\mathbf{L}} \le \epsilon \|\boldsymbol{\phi}^*\|_{\mathbf{L}}$*, where* $\mathbf{L} = \mathbf{B}^\top \mathbf{R}^{-1} \mathbf{B}$ *is the Laplacian of* G*, and* $\boldsymbol{\phi}^* = \mathbf{L}^\dagger \boldsymbol{d}$ *are the true potentials determined by the resistances* $\boldsymbol{r}$*.*

For notational convenience, we define the *conductances* $\boldsymbol{w} \stackrel{\text{def}}{=} \boldsymbol{r}^{-1}$, and let $\mathbf{L}(\boldsymbol{w}) \stackrel{\text{def}}{=} \mathbf{B}^\top \mathbf{W} \mathbf{B}$.

Several of our algorithms want to solve Laplacian systems in *sublinear time.* This can be done in the following natural sense: instead of returning the full potential vector $\boldsymbol{\phi}$, we only wish to determine $\boldsymbol{\phi}$ restricted to a subset of vertices $C \subseteq V$. This is captured by a *Schur complement*, which is defined as $\mathbf{SC}(\mathbf{L}, C) \stackrel{\text{def}}{=} \mathbf{L}_{CC} - \mathbf{L}_{CF} \mathbf{L}_{FF}^{-1} \mathbf{L}_{CC}$, where $F = V \backslash C$ and $\mathbf{L}_{FF}, \mathbf{L}_{CF}, \mathbf{L}_{FC}, \mathbf{L}_{CC}$ are blocks of the Laplacian $\mathbf{L}$ corresponding to rows/columns in F, C. Schur complements satisfy two key properties which are essential for our algorithm: they are also graph Laplacians, and they are directly related to $\mathbf{L}^\dagger$ via the Cholesky factorization.

Lemma 2.3.2 (Cholesky factorization)**.** *For a connected graph* G *with Laplacian* $\mathbf{L} \in \mathbb{R}^{V \times V}$*, subset* $C \subseteq V$*, and* $F \stackrel{\text{def}}{=} V \backslash C$*,*

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

The matrix $\begin{bmatrix} -\mathbf{L}_{FF}^{-1} \mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix}$ appearing in the Cholesky factorization corresponds to mapping the potentials on C back to the whole graph via a *harmonic extension.* In other words, a random walk on G, with exit probabilities proportional to conductances is a martingale on potentials. We give a more formal definition and properties later in Section 2.6.

Finally, it is very useful intuition that electric flows are inherently connected with the following random walk on G: a vertex v goes to a neighbor u with probability proportinal to conductance (inverse resistances), i.e. $\frac{\boldsymbol{w}_{uv}}{\sum_{w \in N(v)} \boldsymbol{w}_{wv}}$, where $N(v)$ are the neighbors of v in G. This random walk is the one used to define the harmonic extension, and also is used more directly in our algorithm for sampling Schur complements (see Lemma 2.4.11). Throughout, any mention of random walks refers to this random walk.

2.4 Improved Dynamic Schur Complements

In this section, we give our main algorithm for maintaining a Schur complement in a dynamic graph. Our main result is the following (see Theorem 2.4.10 for a more precise statement):

Theorem 2.4.1 (Dynamic Schur complement (informal))**.** *There is a data structure that supports the following operations against oblivious adversaries given a graph $G = (V, E)$ with dynamic edge conductances $\boldsymbol{w} \in \mathbb{R}^{E(G)}$ and parameters $\beta < \epsilon^2 < 1$.*

- *Initialize*$(G, \boldsymbol{w}, \epsilon, \beta)$*. Initializes the data structure with accuracy parameter ϵ, and chooses a set of $O(\beta m)$ terminals C. $\overline{\boldsymbol{w}}$ is initialized as $\boldsymbol{w}$. Runtime: $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$.*
- *AddTerminal*(v)*. Makes v a terminal, i.e. $C \leftarrow C \cup \{v\}$. Runtime: amortized $\widetilde{O}(\beta^{-2}\epsilon^{-2})$.*
- *Update*$(e, \overline{\boldsymbol{w}}^{(\text{new})})$*. Under the guarantee that both endpoints of e are terminals in C, updates $\overline{\boldsymbol{w}}_e \leftarrow \overline{\boldsymbol{w}}^{(\text{new})}$. Runtime: amortized $\widetilde{O}(1)$.*
- *SC(). Returns a Laplacian $\widetilde{\mathbf{SC}}$ with $\widetilde{O}(\beta m \epsilon^{-2})$ edges which $(1+\epsilon)$-spectrally approximates the Schur complement of $\mathbf{L}$ with terminal set C in time $\widetilde{O}(\beta m \epsilon^{-2})$.*

All outputs and runtimes are correct with high probability if $|C| = O(\beta m)$ at all times and there are at most $O(\beta m)$ total calls to Update*.*

Our proof is organized in two parts. In Section 2.4.1 we give an algorithm to efficiently generate useful attributes of a random walk in graphs with polynomially bounded edge weights. Next, in Section 2.4.2 we describe how to use these walk attributes to maintain a Schur complement under modifications to the terminal set and edge weights.

2.4.1 Approximate Random Walks with Morris Counters

Our main contribution in this section is an improved algorithm to sample random walks in weighted graphs, based on the Morris counter of [83, 110]. The main technical result of this section is the following:

Theorem 2.4.2 (Morris Walk). *Let $G = (V, E, w, \ell)$ be a graph with edge weights w and edge lengths ℓ bounded between 1 and $n^{O(1)}$. For any vertex u, and parameters $L, \epsilon \geq 0$, Algorithm 3 with high probability runs in $\widetilde{O}(L^2 + L\epsilon^{-2})$ time and generates the following attributes of a random walk $\mathcal{W}_u$ in G which starts from u, samples the edges it traverses with probabilities proportional to w_e, and stops when $\sum_{v \in \mathcal{W}_u} deg(()v) > L^3$:*

- *$u_1, u_2, \ldots,$ the $O(L)$ distinct vertices of $\mathcal{W}_u$ in order of their encounter.*
- *For each u_i, δ_{u_i} is a $(1+\epsilon)$-approximation of*

$$\sum_{k=1}^{f_i - 1} \ell_{(u_k, u_{k+1})},$$

  where f_i is the index of the first visit of u_i in $\mathcal{W}_u$.

Our algorithm is based on simulating random walks in a graph by repeatedly solving linear systems, a technique that has been used in prior work on sampling random spanning trees and dynamically maintaining Schur complements [15, 25, 120, 121, 122, 123]. However, a difficulty in applying this approach to our setting is the need to estimate the length of the resulting random walk. We address this issue by appealing to an approximate counter algorithm to estimate the length of prefixes of the walk by simulating random walks on a larger graph.

To aid our exposition, we begin by recalling a variant of the Morris counter algorithm and an improved analysis of such from [83], which we present in Algorithm 1.

Theorem 2.4.3 (Modification of Theorem 1.2 from [83]). *Consider an instantiation of Algorithm 1 for parameters ϵ, δ, where INCREMENT() has been called N times after one call to INITCOUNTER() . Then APPROXVAL() returns a value $\widehat{N}$ satisfying $\mathbb{E}[\widehat{N}] = N$ and*

$$(1 - \epsilon)N \leq \widehat{N} \leq (1 + \epsilon)N$$

with probability $1 - \delta$.

[3]Here, deg() denotes the *unweighted* degree of a vertex in G.

Algorithm 1 Morris Counter MORRIS

```
global variables
    $X$: current counter value
    $a \geq 0$: accuracy parameter


procedure INITCOUNTER() $(\epsilon, \delta)$
    $X = 0$.
    $a \leftarrow \frac{\epsilon^2}{8\log(1/\delta)}$.
end procedure

procedure INCREMENT()                ▷ Probabilistically updates the Morris counter $X$
    with probability $(1+a)^{-X}$ do
        $X = X + 1$
    end
end procedure

procedure APPROXVAL()    ▷ Returns unbiased estimator for number of times INCREMENT()
was called.
    return $\frac{1}{a}\left((1+a)^X - 1\right)$
end procedure
```

The above theorem may be recovered directly from the analysis of Section 2.2 in [83]. We will employ this algorithm in a white-box fashion to estimate the length of a random walk in a graph. To do this, we condense the behavior of the counter over a collection of w increments into an explicit probability distribution:

Definition 2.4.4 (Morris Increment Probabilities). Given a parameter $a \geq 0$, for integers Y, Z we define the Morris increment probabilities

$$p_{a,Y}^{Z}(\ell) = \Pr\left(\text{MORRIS}.X = Z \text{ after processing } \ell \text{ INCREMENT() calls } |\text{MORRIS}.X = Y \text{originally}\right)$$

We remark that these probabilities may be nontrivial to compute. However, in our algorithms we only require the ability to sample a Z with probability proportional to $p_{a,Y}^{Z}(\ell)$: we will later show how this may be done efficiently.

Definition 2.4.5 (Layer Graph). For weighted graph $G = (V, E, w, \ell)$ and parameter $a \geq 0$, the

Algorithm 2 Conceptual Morris Walk

```
1: procedure ConceptualMorrisWalk($G, L, \epsilon, u, c$)
2:     $a = \frac{\epsilon^2}{8\log(n^{1+c})}$
3:     $\widehat{G} = a-$layer graph of $G$, $u_{rw} = (u, 0)$
4:     $S_{\text{visited}} = [u]$
5:     while $\sum_{v \in S_{\text{visited}}} \deg()G(v) \leq L$ do
6:         $(v_{rw}, i_{rw}) \leftarrow$ random neighbor of $u_{rw}$ in $\widehat{G}$
7:         if $v_{rw} \notin S_{\text{visited}}$ then
8:             $S_{\text{visited}} = [S_{\text{visited}}; v_{rw}]$
9:             $\delta_{v_{rw}} = \frac{1}{8}\left((1+a)^{i_{rw}} - 1\right)$
10:        end if
11:        $u_{rw} = (v_{rw}, i_{rw})$
12:    end whilereturn $S_{\text{visited}}, \{\delta\}$
13: end procedure
```

a-layer graph is an (infinite) weighted directed graph $\widehat{G}$ with vertex set $\widehat{V} = V \otimes \{0, 1, \ldots, \}$[4] and edge set $\widehat{E}$ constructed in the following fashion: For each edge $(u, v) \in E$ of weight w and length ℓ, each $0 \leq i$, and each $i \leq j$, add a directed edge $(u, i) \to (v, j)$ of weight $w \cdot p_{a,i}^{j}\left(\left\lfloor \frac{8}{a}\ell \right\rfloor\right)$ to $\widehat{E}$.

We remark that although the layer graph as defined is infinite, we only access finite subgraphs of it in our algorithms. Our proof strategy in this section is in two parts. First, we describe an idealized algorithm (Algorithm 2) that directly runs a random walk in a graph and generates an output matching the requirements of Theorem 2.4.2. We then provide an efficient variant (Algorithm 3) which with high probability returns an output matching that of Algorithm 2 in distribution: the result follows.

Theorem 2.4.6. *Let $G = (V, E, w, \ell)$ be a graph with edge weights w and edge lengths ℓ bounded between 1 and $n^{O(1)}$. For parameters $L, \epsilon, c \geq 0$ and starting vertex u, Algorithm 2 with high probability returns the following attributes of $\mathcal{W}_u$, a random walk in G which starts from u, samples the edges it traverses with probabilities proportional to w_e, and stops once $\sum_{v \in \mathcal{W}_u} deg()G(v) \geq L$:*

- *A set S of the first $O(L)$ distinct vertices in $\mathcal{W}_u$, in order of encounter*
- *With probability $1 - n^{-c}$, values $\{\delta\}$ such that for each $v \in S$, δ_v $(1 + \epsilon)$-approximates the*

[4]We use the notation $A \otimes B$ to denote the Cartesian product of A and B: it consists of all tuples (i, j) for $i \in A$ and $j \in B$.

length in $\mathcal{W}_u$ (measured with respect to ℓ) from u to the first encounter of v.

Proof. Let $a = \frac{\epsilon^2}{8\log(n^{1+c})}$, and let $\widehat{\mathcal{W}}$ be the ordred collection of vertices $(v_{rw}, i_{rw}) \in \widehat{G}$ encountered on Line 6: note that these vertices form a random walk on $\widehat{G}$ by construction. We define an auxillary random walk $\mathcal{W}$ in G as follows: if the k^{th} in $\widehat{\mathcal{W}}$ is (v, i), the k^{th} node in $\mathcal{W}$ is v. We will show the following two facts:

- $\mathcal{W}$ is a random walk in G which starts from u, samples its edges with probabilities proportional to w_e, and stops once $\sum_{v\in\mathcal{W}} \deg()G(v) \geq L$.
- For any k, let the k^{th} node in $\widehat{\mathcal{W}}$ be (v_k, i_k), and let R_k be inductively defined by $R_1 = 0$, $R_{i+1} = R_i + \left\lfloor \frac{8}{a}\ell_{(v_i,v_{i+1})}\right\rfloor$. Then i_k is distributed as Morris$(a).X$ after processing R_k increments.

The first of these claims follows immediately: if we sample a random neighbor of (v, i) in $\widehat{G}$, the probability that it is of the form (v', j) for some j is simply

$$\frac{\sum_{j=0}^{\infty} w_{(v',v)} p_{a,i}^{j}\left(\left\lfloor \frac{8}{a}\ell_{(v',v)}\right\rfloor\right)}{\sum_{x\in V}\sum_{j=0}^{\infty} w_{(x,v)} p_{a,i}^{j}\left(\left\lfloor \frac{8}{a}\ell_{(x,v)}\right\rfloor\right)} = \frac{w_{(v',v)}}{\sum_{w\in V} w_{(x,v)}}.$$

Thus the k^{th} element of $\mathcal{W}$ is a neighbor of the $(k-1)^{st}$ sampled proportional to w: since $\mathcal{W}$ starts from u the claim follows.

For the second claim, we proceed by induction on k. The claim is trivially true for $k = 1$ (as the first node in $\widehat{\mathcal{W}}$ is $(u, 0)$). It remains to show the induction step. Let the k^{th} node of $\widehat{\mathcal{W}}$ be (v_k, i_k): by the induction hypothesis i_k is distributed as Morris$(a).X$ after R_k increments. Now conditioned on the value of v_{k+1}, we have

$$\begin{aligned}
&\Pr\left(i_{k+1} = x\right) = \sum_{y=0}^{x} \Pr\left(i_{k+1} = x | i_k = y\right)\Pr\left(i_k = y\right)\\
&= \sum_{y=0}^{x} \frac{w_{(v_k,v_{k+1})} p_{a,y}^{x}\left(\left\lfloor \frac{8}{a}\ell_{(v_k,v_{k+1})}\right\rfloor\right)}{\sum_{z=0}^{\infty} w_{(v_k,v_{k+1})} p_{a,y}^{z}\left(\left\lfloor \frac{8}{a}\ell_{(v_k,v_{k+1})}\right\rfloor\right)}\Pr\left(i_k = y\right) = \sum_{y=0}^{x} p_{a,y}^{x}\left(\left\lfloor \frac{8}{a}\ell_{(v_k,v_{k+1})}\right\rfloor\right)\Pr\left(i_k = y\right).
\end{aligned}$$

But by the induction hypothesis, each term of the expression is the probability that Morris$(a).X$ equals y after R_k increments and also equals y after a further $\left\lfloor \frac{8}{a}\ell_{(v_k,v_{k+1})}\right\rfloor$ increments. Since

$R_{k+1} = R_k + \left\lfloor \frac{8}{a}\ell_{(v_k,v_{k+1})} \right\rfloor$ conditioned on the value of v_{k+1}, the claim follows by the law of total probability.

We finally show how these claims imply the theorem. First, note that S consists of the vertices in $\mathcal{W}$ in the order of their encounter: since $\mathcal{W}$ is a random walk in G the correctness of S follows. Next, for each $v \in S$ let $u_v = (v, i_v)$ be the value of u_{rw} set on Line 6 where v was first encountered. Observe that each edge in G has weight at least 1: thus

$$\left(1 - \frac{a}{8}\right)\ell_{(v_k,v_{k+1})} \le \ell_{(v_k,v_{k+1})} - \frac{a}{8} \le \frac{a}{8}\left\lfloor \frac{8}{a}\ell_{(v_k,v_{k+1})} \right\rfloor \le \ell_{(v_k,v_{k+1})}.$$

Thus for any k,

$$\left(1 - \frac{a}{8}\right)\frac{8}{a}\sum_{i=0}^{k}\ell_{(v_i,v_{i+1})} \le R_k \le \frac{8}{a}\sum_{i=0}^{k}\ell_{(v_i,v_{i+1})}.$$

By the second claim, we see that i_v is distributed as $\textsc{Morris}(a).X$ after processing N_{uv} increments, where $N_{uv} = R_k$ if k is the smallest index where v appears in $\mathcal{W}$. This number of increments is larger than $\frac{8}{a}$: by Theorem 2.4.3 we thus have

$$\Pr\left(\left|\frac{1}{a}\left((1+a)^{i_v} - 1\right) - \frac{8}{a}N_{uv}\right| \ge \frac{8\epsilon}{a}N_{uv}\right) \le 1 - n^{-3}.$$

But now, $\frac{a}{8}N_{uv}$ is within a $1 + \frac{a}{4} \le 1 + \epsilon$ factor of $\mathcal{L}_{uv}$, the length of $\mathcal{W}$ from u to the first visit of v. Thus,

$$\Pr\left(|\delta_v - \mathcal{L}_{uv}| \ge 2\epsilon\mathcal{L}_{uv}\right) \le 1 - n^{-C-1}.$$

As there are at most n vertices in S, the claim follows by scaling down ϵ and union bounding over these failure probabilities. □

With Theorem 2.4.6 in hand, we prove the main result of this section by giving an efficient implementation of Algorithm 2. Our algorithm works by simulating a random walk over the a-layer graph $\widehat{G}$ using a Laplacian linear system solver. We will employ the following (standard) lemma on the hitting probabilities of a random walk in undirected graphs:

Lemma 2.4.7 (Corollary of Lemma 5.6 from [25]). *Let $G = (V, E, w)$ be a weighted undirected* *graph, and let x be any vertex in V. For any $C \subseteq V$, the probability that a random walk starting from x first enters C at a vertex y is given by*

$$-\left[\mathbf{L}_{C,V\setminus C}\mathbf{L}_{C,C}^{-1}\boldsymbol{\chi}_x\right]_y.$$

Thus, in $\widetilde{O}(|E|)$ time we may sample a vertex $y \in C$ with probability equal to a random walk starting from x first entering C at y.

We will use this fact within the framework of ConceptualMorrisWalk to replace the explicit sampling of random walk (which may take $\mathrm{poly}(n, W)$ time) with a computationally efficient subroutine. We now describe the graphs on which we apply Lemma 2.4.7: In the below, $N_G(S)$ denotes the vertices which are neighbors of S but do not themselves belong to S.

Definition 2.4.8 ((a, ι, S)-shortcut graph). Let $G = (V, E, w, \ell)$ be an undirected graph with edge weights w and edge lengths ℓ. Let $a \geq 0$ be a parameter, and let $\iota \geq 0$ be an integer. For $S \subseteq V$, we define the (a, ι, S)-shortcut graph $H = (V_H, E_H, w_H)$ as follows:

- For each $v \in S \cup N_G(S)$, add v to V_H.
- For each edge $e = (u, v) \in E$ of weight w and length ℓ with $u, v \in S$, add vertices v_u^+, u_v^+ to V_H, an edge (u, v) of weight $w \cdot p_{a,\iota}^{\iota}\left(\left\lfloor \frac{8}{a}\ell \right\rfloor\right)$ to E_H, and edges $(u, v_u^+), (u_v^+, v)$ of weight $w \cdot \left(1 - p_{a,\iota}^{\iota}\left(\left\lfloor \frac{8}{a}\ell \right\rfloor\right)\right)$ to E_H.
- For each edge $(u, v) \in E$ of weight w and length ℓ with $u \in S$, $v \in N(S)$, add a vertex v_u^+ to V_H, an edge (u, v) of weight $w \cdot p_{a,\iota}^{\iota}\left(\left\lfloor \frac{8}{a}\ell \right\rfloor\right)$ to E_H, and an edge (u, v_u^+) of weight $w \cdot \left(1 - p_{a,\iota}^{\iota}\left(\left\lfloor \frac{8}{a}\ell \right\rfloor\right)\right)$ to E_H.

Let V_S^+ denote the set of vertices of the form $v_u^+ \in H$ for $v \in S$, N_S denote vertices of the form $v \in H$ for $v \in N_G(S)$, and N_S^+ denote vertices of the form $v_u^+ \in H$ for $v \in N_G(S)$.

Note that computing the shortcut graph defined above only requires computing Morris increment probabilities of the form $p_{a,\iota}^{\iota}(s)$. We will show that this admits a simple closed form, and that we may sample a variable proportional to the increment probabilities efficiently.

Lemma 2.4.9. *Given a parameter $a \geq 0$ and integers Y, ℓ, the Morris increment probabilities (Definition 2.4.4) satisfy*

$$p_{a,Y}^{Y}(\ell) = \left(1 - (1+a)^{-Y}\right)^{\ell}.$$

In addition, we may sample an integer $Z \geq Y + 1$ such that

$$\Pr\left(Z = \Gamma\right) = \frac{p_{a,Y}^{\Gamma}(\ell)}{1 - p_{a,Y}^{Y}(\ell)}$$

in time $\widetilde{O}(Z - Y)$.

Proof. For the first claim, note that each call to Increment() increments Morris.X with probability $(1+a)^{-Y}$. The probability that ℓ such increments fails to increase Morris.X is therefore $\left(1 - (1+a)^{-Y}\right)^{\ell}$ as claimed.

For the second claim, we describe an algorithm to sample from the claimed distribution. We first observe that the desired distribution is precisely the value of Morris.X after processing ℓ increments, conditioned on

- The initial value of Morris.X was Y.
- The final value of Morris.X is strictly larger than Y.

We will sample from this distribution by implicitly simulating the Morris counter algorithm itself: we repeatedly sample from the distribution over the number of Increment() calls required to increase Morris.X, and return the final value of Morris.X after ℓ simulated increments were processed. For the below, we let *Geom*(p) denote the geometric random variable over $\{1, 2, \dots\}$ with failure probability p and let *Geom*$^{k}(p)$ denote *Geom*(p) conditioned on the output being at most k: note that both distributions may be sampled from in $\widetilde{O}(1)$ time.

Assume that $Y' =$ Morris.X at some point. Let $p_{Y'}$ be a random variable representing the number of Increment() calls required to increase Morris.X: note that

$$\Pr\left(p_{Y'} > s\right) = p_{a,Y'}^{Y'}(s)$$

by definition. By the closed-form representation of these probabilities, we may therefore conclude that $p_{Y'}$ is distributed as $Geom((1+a)^{-Y'})$.

By the definition of MORRIS, it is therefore clear that we may sample Z $p^Z_{a,Y}(\ell)$ by repeating the following operations:

- Initialize a running increment counter $\ell' = 0$ and a counter value $Z = Y$.
- Generate a sample $k_Z \sim Geom((1+a)^{-Z})$ and set $\ell' = \ell' + k_Z$.
- If $\ell' > \ell$, return Z. Else, increment Z by 1 and go back to the previous line.

To sample Z conditioned on $Z \neq Y$, it is thus sufficient to sample the first $k_Y \sim Geom^{\ell}((1+a)^{-Z})$ to ensure Z is not incremented 0 times. To bound the running time, we additionally observe that the number of geometric and truncated geometric random variables sampled is proportional to $Z - Y$: as the total work performed is $\widetilde{O}(1)$ times this the claim follows. □

Proof of Theorem 2.4.2. Our proof proceeds in two steps. We will first show that the vertices added to S and the values $\{\delta\}$ have the same distribution as the output of Algorithm 2. We will then bound the runtime of the algorithm.

Let $\widehat{G}$ be the a-layer graph of G, and fix a parameter ι and visited set S during a single iteration of Algorithm 3. We consider the subgraph $\widehat{G}_{\iota,S}$ consisting of all directed edges with tail of the form (v,ι) for $v \in S$. Consider the process of running a random walk from $(v,\iota) \in \widehat{G}_{\iota,S}$ until a vertex not of the form (u,ι) with $u \in S$ is reached. It is self-evident that the only such vertices in $\widehat{G}_{\iota,S}$ belong to three classes:

- (v,ι) where $v \in N_G(S)$
- (v,ι') where $v \in S$ and $\iota' > \iota$
- (v,ι') where $v \in N_G(S)$ and $\iota' > \iota$.

Let C denote the collection of vertices of this type. Note that the subgraph of $\widehat{G}_{\iota,S}$ induced on vertices of the form (v,ι) for $v \in S$ is essentially undirected (since each directed edge (x,y) is

Algorithm 3 Morris Walk

```
procedure MorrisWalk($(G, L, \epsilon, u)$)
    $a = \frac{\epsilon^2}{8\log(n^3)}$
    $S = [u]$, $\iota = 0$, $u_{rw} = u$
    while $\sum_{v \in S} \deg()G(v) \leq L$ do
        $G^{\iota}_S \leftarrow (a, \iota, S)$-shortcut graph for $G$ (Definition 2.4.8)
        $C = V^+_S \cup N_S \cup N^+_S$
        $x \leftarrow$ vertex sampled with probability a random walk starting from $u_{rw}$ in $G^{\iota}_S$ first enters $C$ at $x$ (Lemma 2.4.7)
        if $x \in N_S$                                        ▷ Added new vertex to $S$ then
            $S = [S; x]$                                      ▷ Interpret $x$ as a vertex in $G$
            $\delta_x = \frac{1}{8}\left((1+a)^{\iota} - 1\right)$
            $u_{rw} = x$
        end if
        if $x \in V^+_S$                                      ▷ Incremented $\iota$ then
            $v^+_s \overset{\text{def}}{=} x$
            $\ell \overset{\text{def}}{=}$ length of edge $(s, v) \in G$
            $\iota \leftarrow \iota'$ sampled with probability $\propto p^{\iota'}_{a,\iota}(\left\lfloor \frac{8}{a}\ell \right\rfloor)$, conditioned on $\iota' > \iota$ (Lemma 2.4.9)
            $u_{rw} = v$
        end if
        if $v \in N^+_S$                    ▷ Incremented $\iota$ and added vertex to $S$ then
            $v^+_s \overset{\text{def}}{=} x$
            $\ell \overset{\text{def}}{=}$ length of edge $(s, v) \in G$
            $S = [S, v]$
            $\iota \leftarrow \iota'$ sampled with probability $\propto p^{\iota'}_{a,\iota}(\left\lfloor \frac{8}{a}\ell \right\rfloor)$, conditioned on $\iota' > \iota$ (Lemma 2.4.9)
            $\delta_v = \frac{1}{8}\left((1+a)^{\iota} - 1\right)$
            $u_{rw} = v$
        end if
    end while
    return $S, \{\delta\}$
end procedure
```

matched by a directed edge (y, x) of the same weight). Let $G_{\iota,S}$ be the graph obtained by replacing these parallel directed edges with an undirected edge of the same weight, and by removing edge directions from all other edges. It is clear that the probability distribution over vertices that a random walk starting from (u, ι) for $u \in S$ enters C at is induced by a Laplacian linear system solve via Lemma 2.4.7. By direct calculation, it may be verified that these sampling probabilities are equivalent to the sampling performed on Line 7: when sampling the number of increments to the counter, Algorithm 3 simply samples the event that the counter is incremented at least once and then samples from the appropriate conditional distribution for the true number of increments to apply.

We now bound the running time of our algorithm. We observe that the termination condition of the while loop ensures that G_S^ι never contains more than $O(L)$ edges: thus the call to Lemma 2.4.7 on on Line 7 can be implemented in $\widetilde{O}(L)$ time. We additionally see via the remaining operations in the loop that each linear system we solve ensures that we either add a new vertex to S or increase the value of ι. Next, we note that since G's weights and lengths are polynomially-bounded, the total length of a random walk which covers the entirety of G is bounded by poly(n). Thus for any v in the returned set S, δ_v is a $(1+\epsilon)$-approximation to a quantity which is also bounded by poly(n). But this implies that the variable ι satisfies

$$(1+a)^\iota \leq \text{poly}(n) \implies \iota \leq \widetilde{O}\left(\epsilon^{-2}\right)$$

with high probability. Thus at most $\widetilde{O}(\epsilon^{-2})$ calls to Lemma 2.4.7 can increase the value of ι: as the while loop must terminate after adding L vertices to S it follows that Algorithm 3 solves at most $\widetilde{O}(L+\epsilon^{-2})$ linear systems with high probability. Finally, the only remaining nontrivial computation of the algorithm is performed on Line 16 and Line 23. But as $\iota \leq \widetilde{O}(\epsilon^{-2})$ by Lemma 2.4.9 these lines cost $\widetilde{O}(\epsilon^{-2})$ amortized over the whole algorithm. The claimed runtime follows. □

2.4.2 Improved Dynamic Schur Complement

Here we provide our main result regarding the dynamic maintenance of Schur complements under edge resistance changes in G. We achieve this by plugging in our improved algorithm Theorem 2.4.2 for estimating lengths of random walks visiting a fixed number of vertices into previous frameworks [15, 25]. Below, the additional operation InitialSC maintains the approximate Schur complement ignoring edge updates, but still tracking terminal additions. It is useful for our dynamic Evaluator and Locator data structures in Section 2.6. We use the notation $\mathbf{SC}_{\mathcal{H}}$ for the approximation as it eventually gets used to approximately compute a harmonic extension $\mathcal{H}$.

Theorem 2.4.10 (Dynamic Schur complement)**.** *There is a data structure DynamicSC that sup-*

ports the following operations against oblivious adversaries given a graph $G = (V, E)$ with dynamic edge conductances $\boldsymbol{w} \in \mathbb{R}^{E(G)}$ and parameters $\beta < \epsilon^2 < 1$.

- *INITIALIZE$(G, \boldsymbol{w}, \epsilon, \beta)$. Initializes the data structure with accuracy parameter ϵ, and chooses a set of $O(\beta m)$ terminals C. $\overline{\boldsymbol{w}}$ is initialized as $\boldsymbol{w}$. Runtime: $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$.*
- *ADDTERMINAL(v). Makes v a terminal, i.e. $C \leftarrow C \cup \{v\}$. Runtime: amortized $\widetilde{O}(\beta^{-2}\epsilon^{-2})$.*
- *UPDATE$(e, \overline{\boldsymbol{w}}^{(\text{new})})$. Under the guarantee that both endpoints of e are terminals in C, updates $\overline{\boldsymbol{w}}_e \leftarrow \overline{\boldsymbol{w}}^{(\text{new})}$. Runtime: amortized $\widetilde{O}(1)$.*
- *SC(). Returns a Laplacian $\widetilde{\mathbf{SC}} \approx_\epsilon \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ with $\widetilde{O}(\beta m\epsilon^{-2})$ edges in time $\widetilde{O}(\beta m\epsilon^{-2})$.*
- *INITIALSC(). Returns a Laplacian $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ with $\widetilde{O}(\beta m\epsilon^{-2})$ edges in time $\widetilde{O}(\beta m\epsilon^{-2})$. Let Z be the set of edges which were input to UPDATE after initialization. Define $\boldsymbol{w}_{\overline{Z}}$ as $(\boldsymbol{w}_{\overline{Z}})_e = 0$ for $e \in Z$ and $(\boldsymbol{w}_{\overline{Z}})_e = \boldsymbol{w}_e$ otherwise. Then $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ satisfies*

$$\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) - \epsilon\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C) \preceq \widetilde{\mathbf{SC}}_{\mathcal{H}} \preceq \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) + \epsilon\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C). \tag{2.4}$$

All outputs and runtimes are correct whp. if $|C| = O(\beta m)$ at all times and there are at most $O(\beta m)$ total calls to UPDATE.

We note that we could achieve the tighter approximation guarantee in (2.4) for the operation SC(). However, we do not need use it in this paper (eg. Section 2.6) and therefore, do not state it.

We require the following process which samples Schur complements using random walks.

Lemma 2.4.11 (Schur complement approximation, [15] Theorem 3.1)**.** *Let $G = (V, E, \boldsymbol{r})$ be an undirected, weighted multigraph with a subset of vertices C. Furthermore, let $\epsilon \in (0, 1)$ and let $\rho = 1000\epsilon^{-2}\log n$. Let H be an initially empty graph with vertices C, and for each edge $e = (u, v) \in E(G)$ repeat the following procedure ρ times.*

1. *Simulate a random walk from u until it hits C at c_1.*
2. *Simulate a random walk from v until it hits C at c_2.*

3. Combine these random walks (along with edge $e = (u, v)$) to form a walk W.

4. Add edge (c_1, c_2) to H with resistance $\rho \sum_{e \in W} \boldsymbol{r}_e$.

The resulting graph H satisfies $\mathbf{L}(H) \approx_\epsilon \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)$ with probability at least $1 - n^{-10}$.

Finally, we require a dynamic spectral sparsification procedure.

Lemma 2.4.12 ([25, Lemma 4.10])**.** *There is a data structure that supports insertions and deletions of edges on a graph G which have underlying conductances/resistances in amortized $\widetilde{O}(\log U)$ time per operation. Additionally, it can output a $(1+\epsilon)$-spectral sparsifier of G in $\widetilde{O}(n\epsilon^{-2} \log U)$ time.*

Now, we can show Theorem 2.4.10 exactly as done in [15, 25] by sampling random walks using Lemma 2.4.11 and shortcutting them as terminals get added.

Proof of Theorem 2.4.10. We explain how to implement each operation in Theorem 2.4.10.

Initialize: Randomly sample an initial terminal set C of size $O(\beta m)$. From each edge $e = (u, v) \in G$, sample $\rho = \widetilde{O}(\epsilon^{-2})$ random walks from u, v to C as in Lemma 2.4.11, and record $(1+\epsilon)$ approximations of the sums of resistances of all prefixes. Note that these walks visit $\widetilde{O}(\beta^{-1})$ distinct vertices whp. Initialize the data structure $D^{(\mathrm{s})}$ in Lemma 2.4.12. Based on these random walks, add edges to C using the data structure $D^{(\mathrm{s})}$. Additionally, maintain a set Z of updated edges, whose original and final conductances we track explicitly.

The runtime of Initialize is dominated by the time to sample the random walks, which is $\widetilde{O}(m\epsilon^{-2}(\beta^{-2} + \beta^{-1}\epsilon^{-2})) = \widetilde{O}(m\epsilon^{-2}\beta^{-2})$ by Theorem 2.4.2 (the length $L = \widetilde{O}(\beta^{-1})$) and $\beta < \epsilon^2$.

AddTerminal(v): Update $C \leftarrow C \cup \{v\}$ and shortcut all walks passing through v. The total length of all walks is $\widetilde{O}(m\epsilon^{-2}\beta^{-1})$, so over the course of $O(\beta m)$ terminal insertions, the amortized runtime is $\widetilde{O}(m\epsilon^{-2}\beta^{-1}/(\beta m)) = \widetilde{O}(\beta^{-2}\epsilon^{-2})$. Finally, pass all edge insertions/deletions in C to $D^{(\mathrm{s})}$.

Update($e, \overline{\boldsymbol{w}}^{(\mathrm{new})}$): Delete the edge e (do not insert an edge with conductance $\overline{\boldsymbol{w}}^{(\mathrm{new})}$), and pass the deletion to $D^{(\mathrm{s})}$. Update $Z \leftarrow Z \cup \{e\}$. From now on, the algorithm explicitly stores in memory the original and current conductances of edge e. Clearly, the update time is $\widetilde{O}(1)$.

SC(): Call $D^{(\mathrm{s})}$ to output a $(1+\epsilon)$-approximation of $\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C)$ with high probability. The approximation guarantee follows from Lemma 2.4.11 and the guarantee of Theorem 2.4.2 that the total resistive length of each random walk is correct up to $(1+\epsilon/10)$ with high probability. Finally, add the edges $e \in Z$ back in with the current conductances. The runtime is $\widetilde{O}(\beta m \epsilon^{-2})$ by Lemma 2.4.12 as $|C| = O(\beta m)$.

InitialSC(): Same as SC(), except we add back edges in Z with their original conductances. The tighter approximation holds because the algorithm is returning a $(1+\epsilon)$-approximation of $\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C)$ and the edges $e \in Z$ that are added in contribute no error. $\square$

2.5 Dynamic Laplacian Solver in Sub-linear Time

In this section we extend our dynamic approximate Schur complement algorithm to obtain a dynamic Laplacian solver. Specifically, our goal is to design a data-structure that maintains a solution to the Laplacian system $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$ under updates to both the underlying graph and the demand vector vector $\boldsymbol{b}$ while being able to query a few entries of the solution vector. For the sake of exposition, in what follows we assume that the underlying graph is always connected.

Theorem 2.5.1 (Dynamic Laplacian Solver). *For a graph $G = (V, E)$ with dynamic edge conductances $\overline{\boldsymbol{w}} \in \mathbb{R}_{\geq 0}^{E(G)}$ and a dynamic vector $\overline{\boldsymbol{b}} \in \mathbb{R}^{V(G)}$, there is a data structure (Algorithm 4) that supports the following operations against an oblivious adversary for parameters $\beta < \epsilon^2 < 1$.*

- *Initialize$(G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \beta, \epsilon)$. Initializes the data structure in time $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$ with an empty set $Z \leftarrow \emptyset$ of marked edges. Initialize $\overline{\boldsymbol{w}}$ as $\boldsymbol{w}$ and $\overline{\boldsymbol{b}}$ as $\boldsymbol{b}^{(\mathrm{init})}$.*
- *UpdateB$(v, \overline{\boldsymbol{b}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{b}}_v \leftarrow \overline{\boldsymbol{b}}^{(\mathrm{new})}$ in amortized $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *UpdateW$(e, \overline{\boldsymbol{w}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{w}}_e \leftarrow \overline{\boldsymbol{w}}^{(\mathrm{new})}$ in amortized $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *Solve(). For $C \subseteq V$ with $|C| = O(\beta m)$ and $Z \subseteq E(C)$, returns in $\widetilde{O}(\beta m \epsilon^{-2})$ time a vector $\widetilde{\boldsymbol{x}}$ satisfying $\exists \boldsymbol{x}^{(\mathrm{full})}, \widetilde{\boldsymbol{x}}^{(\mathrm{full})}$ such that $\widetilde{\boldsymbol{x}}_C^{(\mathrm{full})} = \widetilde{\boldsymbol{x}}$, $\mathbf{L}(\overline{\boldsymbol{w}})\boldsymbol{x}^{(\mathrm{full})} = \overline{\boldsymbol{b}}$, and $\left|\widetilde{\boldsymbol{x}}^{(\mathrm{full})} - \boldsymbol{x}^{(\mathrm{full})}\right|_{\mathbf{L}(\overline{\boldsymbol{w}})} \leq \epsilon \|\overline{\boldsymbol{b}}\|_2$.*

Runtimes and output correctness hold w.h.p. if there are at most $O(\beta m)$ calls to UpdateB and UpdateW in total.

In Section 2.6.1, we will introduce the harmonic extension and use it to approximate $\boldsymbol{x}_C$ where C is the set of terminals. In Section 2.5.2, we show how to build a dynamic Laplacian solver by dynamic Schur complement.

2.5.1 Harmonic Extension

A key notion we use throughout is the harmonic extension, which is a linear operator that maps the potentials restricted to a terminal set to the full electric potentials $\boldsymbol{\phi}$. We use $\boldsymbol{T}$ to denote the projection orthogonal to the all-ones vector.

Definition 2.5.2 (Harmonic extension). For a graph $G = (V, E)$ with edge conductances $\boldsymbol{w} \in \mathbb{R}^E_{>0}$ and $C \subseteq V(G)$, define the *harmonic extension* operator $\mathcal{H}_C \in \mathbb{R}^{V(G)\times C}$ as

$$\mathcal{H}_C \stackrel{\text{def}}{=} \begin{bmatrix} -\mathbf{L}(\boldsymbol{w})^{-1}_{FF}\mathbf{L}(\boldsymbol{w})_{FC}\boldsymbol{T} \\ \boldsymbol{T} \end{bmatrix}.$$

Harmonic extension allows us to solve on the terminal set and then lift the solution back.

Lemma 2.5.3. *Let $\boldsymbol{x}_T$ be a solution vector such that $\mathbf{SC}(G,T)\boldsymbol{x}_T = \mathcal{H}_C^\top \boldsymbol{b}$. Then there exists an extension $\boldsymbol{x}$ of $\boldsymbol{x}_T$ such that $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$.*

Proof. We assume without loss of generality that the underlying graph G is connected. Consider the following extended linear system

$$\begin{bmatrix} \boldsymbol{L}_{[F,F]} & \boldsymbol{L}_{[F,T]} \\ \mathbf{0} & \mathbf{SC}(G,T) \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_F \\ \boldsymbol{x}_T \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_F & \mathbf{0} \\ & \mathcal{H}_C^\top \end{bmatrix} \begin{bmatrix} \boldsymbol{b}_F \\ \boldsymbol{b}_T \end{bmatrix}$$

Using the definitions of Schur complement and projection matrix, we can rewrite the above equation as follows:

$$\begin{bmatrix} \boldsymbol{L}_{[F,F]} & \boldsymbol{L}_{[F,T]} \\ \mathbf{0} & \boldsymbol{L}_{[T,T]} - \boldsymbol{L}_{[T,F]}\boldsymbol{L}^{-1}_{[F,F]}\boldsymbol{L}_{[F,T]} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_F \\ \boldsymbol{x}_T \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_F & \mathbf{0} \\ -\boldsymbol{L}_{[T,F]}\boldsymbol{L}^{-1}_{[F,F]} & I_T \end{bmatrix} \begin{bmatrix} \boldsymbol{b}_F \\ \boldsymbol{b}_T \end{bmatrix}$$

Multiplying both sides from the left with

$$\begin{bmatrix} \boldsymbol{I}_F & \mathbf{0} \\ \boldsymbol{L}_{[T,F]}\boldsymbol{L}_{[F,F]}^{-1} & I_T \end{bmatrix},$$

we get that

$$\begin{bmatrix} \boldsymbol{L}_{[F,F]} & \boldsymbol{L}_{[F,T]} \\ \boldsymbol{L}_{[T,F]} & \boldsymbol{L}_{[T,T]} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_F \\ \boldsymbol{x}_T \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_F \\ \boldsymbol{b}_T \end{bmatrix} \quad \text{or } \boldsymbol{L}\boldsymbol{x} = \boldsymbol{b},$$

what we wanted to show. □

By Lemma 2.5.3, it is sufficient to maintain a solution $\boldsymbol{x}_T = \mathbf{SC}(G,T)^{\dagger}\mathcal{H}_C^{\top}\boldsymbol{b}$ dynamically. Since Theorem 2.4.10 already allows us to maintain a dynamic Schur complement, we need to devise a routine that maintains the projection $\mathcal{H}_C^{\top}\boldsymbol{b}$ of $\boldsymbol{b}$ under vertex additions to the terminal set. In the following, we shall see that maintaining $\mathbf{SC}(G,T)^{\dagger}\mathcal{H}_C^{\top}\boldsymbol{b}$ can be efficiently reduced to maintaining dynamic Schur complement. We then prove Theorem 2.5.1 by a simple program built on the dynamic Schur complement data structure (Theorem 2.4.10).

Note that the harmonic extension does not depend on edges with both endpoints in C. Leveraging this yields the following alternative characterization of the harmonic extension. These properties are crucial for our data structures as they maintain a growing terminal set where are resistance changes are on edges completely inside the terminal set. In this section, we use $\overline{\boldsymbol{w}}$ to denote modified conductances and $\boldsymbol{w}$ to denote initial conductances.

Lemma 2.5.4 (Alternate definition of harmonic extension)**.** *For a graph $G = (V,E)$ with edge conductances $\boldsymbol{w} \in \mathbb{R}^E_{>0}$ and $C \subseteq V$, let $\widetilde{G}$ be a graph with the same edge set as G whose conductances $\widetilde{\boldsymbol{w}}$ agree with $\boldsymbol{w}$* except *potentially on edges with both endpoints inside C. Then*

$$\mathcal{H}_C = \mathbf{L}(\widetilde{\boldsymbol{w}})^{\dagger} \begin{bmatrix} 0 \\ \mathbf{SC}(\mathbf{L}(\widetilde{\boldsymbol{w}}), C) \end{bmatrix}. \tag{2.5}$$

Proof. By Definition 2.5.2, the harmonic extension does not depend on the edges inside C. Hence, we can simply show the lemma for the Laplacian $\mathbf{L} = \mathbf{L}(\widetilde{\boldsymbol{w}})$. By the Cholesky factorization

(Lemma 2.3.2), we have

$$\begin{aligned}\mathbf{L}^{\dagger}\begin{bmatrix}0\\ \mathbf{SC}(\mathbf{L},C)\end{bmatrix} &= \begin{bmatrix}\mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC}\\ 0 & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{L}_{FF}^{-1} & 0\\ 0 & \mathbf{SC}(\mathbf{L},C)^{\dagger}\end{bmatrix}\begin{bmatrix}\mathbf{I} & 0\\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I}\end{bmatrix}\begin{bmatrix}0\\ \mathbf{SC}(\mathbf{L},C)\end{bmatrix}\\ &= \begin{bmatrix}\mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC}\\ 0 & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{L}_{FF}^{-1} & 0\\ 0 & \mathbf{SC}(\mathbf{L},C)^{\dagger}\end{bmatrix}\begin{bmatrix}0\\ \mathbf{SC}(\mathbf{L},C)\end{bmatrix}\\ &= \begin{bmatrix}\mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC}\\ 0 & \boldsymbol{T}\end{bmatrix}\begin{bmatrix}0\\ \boldsymbol{T}\end{bmatrix} = \mathcal{H}_C.\end{aligned}$$

□

This is why we use the notation $\mathcal{H}_C$ without reference to G – when we use $\mathcal{H}_C$ in our dynamic data structures all changed edges will lie inside C. Consequently, the actual graph (beyond initialization) does not affect $\mathcal{H}_C$.

Now we are ready to decompose the projection $\mathcal{H}_C^{\top}\boldsymbol{b}$.

Lemma 2.5.5 (Approximate solution)**.** *Let G be a graph with weights $\overline{\boldsymbol{w}} \in \mathbb{R}^E$ which differ from weights $\boldsymbol{w} \in \mathbb{R}^E$ except on an edge subset $Z \subseteq E(G)$. Let $C \subseteq V(G)$ contain all endpoints of edges in Z. Let $\boldsymbol{w}_{\overline{Z}} \in \mathbb{R}^E$ be defined as $(\boldsymbol{w}_{\overline{Z}})_e = 0$ for $e \in Z$ and $(\boldsymbol{w}_{\overline{Z}})_e = \boldsymbol{w}_e$ otherwise. Let $\widetilde{\mathbf{SC}} \approx_{\epsilon} \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ and let $\widetilde{\mathcal{H}} = \mathbf{L}(\boldsymbol{w})^{\dagger}\begin{bmatrix}0\\ \widetilde{\mathbf{SC}}_{\mathcal{H}}\end{bmatrix}$ for some $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ satisfying*

$$\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) - \epsilon\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C) \preceq \widetilde{\mathbf{SC}}_{\mathcal{H}} \preceq \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) + \epsilon\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C). \tag{2.6}$$

Then the vectors $\boldsymbol{x}_C = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}\mathcal{H}_C^{\top}\boldsymbol{b}$ and $\widetilde{\boldsymbol{x}_C} = \widetilde{\mathbf{SC}}^{\dagger}\widetilde{\mathcal{H}}^{\top}\boldsymbol{b}$ in $\mathbb{R}^C$ satisfy $\|\boldsymbol{x}_C - \widetilde{\boldsymbol{x}_C}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq 3\epsilon\|\boldsymbol{b}\|_{\mathbf{L}(\overline{\boldsymbol{w}})^{\dagger}}$.

Proof. We extract $\widetilde{\boldsymbol{x}}$ by swapping the $\widetilde{\mathcal{H}}$ by $\widetilde{\mathcal{H}}_C$ in $\boldsymbol{x}$:

$$\begin{aligned}\boldsymbol{x} &= \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} - \widetilde{\mathbf{SC}}^{\dagger}\right)\mathcal{H}_C^{\top}\boldsymbol{b} + \widetilde{\mathbf{SC}}^{\dagger}\mathcal{H}_C^{\top}\boldsymbol{b}\\ &= \widetilde{\boldsymbol{x}} + \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} - \widetilde{\mathbf{SC}}^{\dagger}\right)\mathcal{H}_C^{\top}\boldsymbol{b} + \widetilde{\mathbf{SC}}^{\dagger}\left(\mathcal{H}_C^{\top} - \widetilde{\mathcal{H}}^{\top}\right)\boldsymbol{b}.\end{aligned}$$

Hence

$$\boldsymbol{x} - \widetilde{\boldsymbol{x}} = \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} - \widetilde{\mathbf{SC}}^{\dagger}\right) \mathcal{H}_C^{\top} \boldsymbol{b} + \widetilde{\mathbf{SC}}^{\dagger} \left(\mathcal{H}_C^{\top} - \widetilde{\mathcal{H}}^{\top}\right) \boldsymbol{b}. \tag{2.7}$$

We bound both terms separately. For the first term,

$$\left\| \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} - \widetilde{\mathbf{SC}}^{\dagger}\right) \mathcal{H}_C^{\top} \boldsymbol{b} \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \overset{(i)}{\leq} \epsilon \left\| \mathcal{H}_C^{\top} \boldsymbol{b} \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}}$$
$$\overset{(ii)}{\leq} \epsilon \left\| \boldsymbol{b} \right\|_{\mathbf{L}(\overline{\boldsymbol{w}})^{\dagger}} .$$

where (i) follows from $\widetilde{\mathbf{SC}} \approx_{\epsilon} \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ and Lemma 2.6.2 for $\mathbf{X} = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}$ and $\mathbf{Y} = \widetilde{\mathbf{SC}}^{\dagger}$, and (ii) follows from Lemma 2.6.1 and the fact that $\mathbf{L}_{F,F}^{-1}$ is positive definite. For the second term,

$$\left\| \widetilde{\mathbf{SC}}^{\dagger} \left(\mathcal{H}_C^{\top} - \widetilde{\mathcal{H}}^{\top}\right) \boldsymbol{b} \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \leq 2 \left\| \left(\mathcal{H}_C^{\top} - \widetilde{\mathcal{H}}^{\top}\right) \boldsymbol{b} \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}}$$
$$\leq 2 \left\| \left(\mathcal{H}_C^{\top} - \widetilde{\mathcal{H}}^{\top}\right) \boldsymbol{b} \right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C)^{\dagger}} \overset{(i)}{\leq} 2\epsilon \left\| \left[\mathbf{L}(\boldsymbol{w})^{\dagger} \boldsymbol{b}\right]_C \right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C)}$$
$$\overset{(ii)}{\leq} 2\epsilon \left\| \left[\mathbf{L}(\boldsymbol{w})^{\dagger} \boldsymbol{b}\right]_C \right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)} \overset{(iii)}{\leq} 2\epsilon \left\| \boldsymbol{b} \right\|_{\mathbf{L}(\overline{\boldsymbol{w}})^{\dagger}} ,$$

where (i) follows from Lemma 2.5.4 and (2.6), (ii) follows from $\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) - \mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C) = \mathbf{L}(G[C])$ being positive semidefinite, and (iii) follows from the fact that the Schur complement is spectrally smaller than the Laplacian: $\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) \preceq \mathbf{L}(\boldsymbol{w})$. □

2.5.2 Dynamic Laplacian Solver

In this section, we show how to maintain the approximate solution vector $\widetilde{\boldsymbol{x}}$ (Lemma 2.5.5) that approximates the solution $\boldsymbol{x}$. This data structure can also be used for $\boldsymbol{\psi}$ (Corollary 2.6.4).

Theorem 2.5.1 (Dynamic Laplacian Solver)**.** *For a graph $G = (V, E)$ with dynamic edge conductances $\overline{\boldsymbol{w}} \in \mathbb{R}_{\geq 0}^{E(G)}$ and a dynamic vector $\overline{\boldsymbol{b}} \in \mathbb{R}^{V(G)}$, there is a data structure (Algorithm 4) that supports the following operations against an oblivious adversary for parameters $\beta < \epsilon^2 < 1$.*

- *Initialize$(G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \beta, \epsilon)$. Initializes the data structure in time $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$ with an empty set $Z \leftarrow \emptyset$ of marked edges. Initialize $\overline{\boldsymbol{w}}$ as $\boldsymbol{w}$ and $\overline{\boldsymbol{b}}$ as $\boldsymbol{b}^{(\mathrm{init})}$.*

- *$\textsc{UpdateB}(v, \overline{\boldsymbol{b}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{b}}_v \leftarrow \overline{\boldsymbol{b}}^{(\mathrm{new})}$ in amortized $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*

- *$\textsc{UpdateW}(e, \overline{\boldsymbol{w}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{w}}_e \leftarrow \overline{\boldsymbol{w}}^{(\mathrm{new})}$ in amortized $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*

- *$\textsc{Solve}()$. For $C \subseteq V$ with $|C| = O(\beta m)$ and $Z \subseteq E(C)$, returns in $\widetilde{O}(\beta m \epsilon^{-2})$ time a vector $\widetilde{\boldsymbol{x}}$ satisfying $\exists \boldsymbol{x}^{(\mathrm{full})}, \widetilde{\boldsymbol{x}}^{(\mathrm{full})}$ such that $\widetilde{\boldsymbol{x}}_C^{(\mathrm{full})} = \widetilde{\boldsymbol{x}}$, $\mathbf{L}(\overline{\boldsymbol{w}})\boldsymbol{x}^{(\mathrm{full})} = \overline{\boldsymbol{b}}$, and $\left|\widetilde{\boldsymbol{x}}^{(\mathrm{full})} - \boldsymbol{x}^{(\mathrm{full})}\right|_{\mathbf{L}(\overline{\boldsymbol{w}})} \leq \epsilon \|\overline{\boldsymbol{b}}\|_2$.*

Runtimes and output correctness hold w.h.p. if there are at most $O(\beta m)$ calls to $\textsc{UpdateB}$ and $\textsc{UpdateW}$ in total.

Proof. The pseudocode for the proof of Lemma 2.6.6 is in Algorithm 4. At a high-level, it simply maintains $\widetilde{\boldsymbol{x}}$ as in Lemma 2.5.5. We start by analyzing the correctness of the algorithm, then move the runtime.

Correctness. We only need to analyze the $\textsc{Solve}()$ operation. In this proof, we first show the bound $\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq \frac{\epsilon}{3}\left(\|\overline{\boldsymbol{b}}\|_2 + \|\boldsymbol{b}^{(\mathrm{init})}\|_2\right)$. This suffices to give

$$\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq \epsilon \|\overline{\boldsymbol{b}}\|_2$$

because we can build $\widetilde{O}(1)$ copies of the data structure. For $-\widetilde{O}(1) \leq j \leq \widetilde{O}(1)$, the j-th instance initializes and answers queries only when $\|\overline{\boldsymbol{b}}\|_2 \in (2^j, 2^{j+1}]$. Updates are passed to all instances. When the number of updates exceeds $O(\beta m)$ for an instance but it cannot be initialized because $\|\overline{\boldsymbol{b}}\|_2$ does not fall in its range, it simply ignore following updates until it can be initialized. This only increases the runtime by $\widetilde{O}(1)$ factors.

Let $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ be the value of $D^{(\mathrm{sc})}.\textsc{InitialSC}()$ returned in line 28 of Algorithm 4. It satisfies condition (2.6) by the guarantees of Theorem 2.4.10. Also, the returned vector $\widetilde{\boldsymbol{x}}$ of the procedure

Solve() in Algorithm 4 is defined as

$$\begin{aligned}
\widetilde{\boldsymbol{x}} &= \widetilde{\mathbf{SC}}^{\dagger} \begin{bmatrix} \widetilde{\mathbf{SC}}_{\mathcal{H}} & 0 \end{bmatrix} \boldsymbol{d}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^{\dagger} \left(\overline{\boldsymbol{b}} - \boldsymbol{b}^{(\mathrm{init})} \right) \\
&= \widetilde{\mathbf{SC}}^{\dagger} \begin{bmatrix} \widetilde{\mathbf{SC}}_{\mathcal{H}} & 0 \end{bmatrix} \mathbf{L}(\boldsymbol{w})^{\dagger} \boldsymbol{b}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^{\dagger} \left(\overline{\boldsymbol{b}} - \boldsymbol{b}^{(\mathrm{init})} \right) \\
&= \widetilde{\mathbf{SC}}^{\dagger} \widetilde{\mathcal{H}}^{\top} \boldsymbol{b}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^{\dagger} \left(\overline{\boldsymbol{b}} - \boldsymbol{b}^{(\mathrm{init})} \right).
\end{aligned}$$

Additionally, because $\boldsymbol{b}^{(\mathrm{init})} - \overline{\boldsymbol{b}}$ is supported on Z, the true $\boldsymbol{x}$ can be written as

$$\begin{aligned}
\boldsymbol{x} &= \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} \mathcal{H}_C^{\top} \overline{\boldsymbol{b}} \\
&= \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} \mathcal{H}_C^{\top} \boldsymbol{b}^{(\mathrm{init})} + \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} (\overline{\boldsymbol{b}} - \boldsymbol{b}^{(\mathrm{init})}).
\end{aligned}$$

Hence we get that

$$\begin{aligned}
& \|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \\
\leq \; & \left\| \widetilde{\mathbf{SC}}^{\dagger} \widetilde{\mathcal{H}}^{\top} \boldsymbol{b}^{(\mathrm{init})} - \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} \mathcal{H}_C^{\top} \boldsymbol{b}^{(\mathrm{init})} \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \\
+ \; & \left\| \widetilde{\mathbf{SC}}^{\dagger} \left(\overline{\boldsymbol{b}} - \boldsymbol{b}^{(\mathrm{init})} \right) - \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger} (\overline{\boldsymbol{b}} - \boldsymbol{b}^{(\mathrm{init})}) \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \\
\overset{(i)}{\leq} \; & 3\epsilon \|\boldsymbol{v}^{(\mathrm{init})}\|_2 + \epsilon \|\overline{\boldsymbol{v}} - \boldsymbol{v}^{(\mathrm{init})}\|_2 \leq 4\epsilon \left(\|\overline{\boldsymbol{v}}\|_2 + \|\boldsymbol{v}^{(\mathrm{init})}\|_2 \right),
\end{aligned}$$

where (i) follows from Lemma 2.5.5 for the first term, and $\widetilde{\mathbf{SC}} \approx_{\epsilon} \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ from the guarantee of Theorem 2.4.10 for the second term. Because Algorithm 4 set $\epsilon \leftarrow \epsilon/10$ in Line 2 and by the $\widetilde{O}(1)$ copies for different $\|\overline{\boldsymbol{b}}\|_2$, this suffices to give

$$\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \leq \epsilon \|\overline{\boldsymbol{b}}\|_2.$$

By Lemma 2.5.3, We know there exists a vector $\boldsymbol{x}^{(\mathrm{full})}$ such that $\boldsymbol{x}_C^{(\mathrm{full})} = \boldsymbol{x}$ and $\mathbf{L}(\overline{\boldsymbol{w}}) \boldsymbol{x}^{(\mathrm{full})} = \overline{\boldsymbol{b}}$.

We let $\widetilde{\boldsymbol{x}}^{(\text{full})} = \boldsymbol{x}^{(\text{full})}_{V\setminus C} + \widetilde{\boldsymbol{x}}$. Then

$$\left\|\widetilde{\boldsymbol{x}}^{(\text{full})} - \boldsymbol{x}^{(\text{full})}\right\|_{\mathbf{L}(\overline{\boldsymbol{w}})} = \|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{L}(\overline{\boldsymbol{w}})} = \|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq \epsilon\|\overline{\boldsymbol{b}}\|_2$$

where the second equation is because

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

and $\widetilde{\boldsymbol{x}} - \boldsymbol{x}$ is supported on C.

Runtime. The runtimes of UpdateB, UpdateW are trivially the same as the runtime of AddTerminal from Theorem 2.4.10 and Mark, respectively. The runtimes of Mark and Initialize follows from the AddTerminal and Initialize operations respectively of Theorem 2.4.10. The runtime of Solve is $\widetilde{O}(\beta m \epsilon^{-2})$ by the runtime guarantees of SC and InitialSC of Theorem 2.4.10, and the fact that $\widetilde{\mathbf{SC}}$ and $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ all have $\widetilde{O}(\beta m \epsilon^{-2})$ edges, and hence solving or multiplying by them costs $\widetilde{O}(\beta m \epsilon^{-2})$ time. □

2.6 Data Structures for Dynamic Electrical Flows

The goal of this section is to apply the dynamic Schur complement data structure of Theorem 2.4.10 to give algorithms that dynamically maintain electric potentials and edges with large electric energies in dynamic electrical flows. In Section 2.6.1, we use the harmonic extension to decompose the energy vector we need to maintain for the outer IPM. In Section 2.6.2, we show how to maintain a potential vector which is a key component for the following subsections. In Section 2.6.3, we build the Evaluator that estimates the energy of any edge. In Section 2.6.4, we build the Locator that returns a superset of edges with large energies.

Algorithm 4 Dynamic Laplacian Solver

```
procedure $\textsc{Initialize}(G, \boldsymbol{w}, \boldsymbol{b}^{(\text{init})}, \beta, \epsilon)$
    $\epsilon \leftarrow \epsilon/12$.
    Let $D^{(\text{sc})}$ be a instance of the dynamic Schur complement data structure of Theorem 2.4.10.

    $D^{(\text{sc})}.\textsc{Initialize}(G, \boldsymbol{w}, \epsilon, \beta)$.
    $\overline{\boldsymbol{b}} \leftarrow \boldsymbol{b}^{(\text{init})}$.    ▷ $\boldsymbol{b}^{(\text{init})}$ is the initial vector and $\overline{\boldsymbol{b}}$ to denote the current vector $\boldsymbol{b}$ throughout the algorithm.
    $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w}$.    ▷ We use $\boldsymbol{w}$ to denote the initial vector and $\overline{\boldsymbol{w}}$ to denote the current vector $\boldsymbol{w}$ throughout the algorithm.
    $\boldsymbol{d}^{(\text{init})} \leftarrow \textsc{Solve}(\mathbf{L}(\boldsymbol{w}), \boldsymbol{b}^{(\text{init})})$.
    $Z \leftarrow \emptyset$.    ▷ Marked edges.
end procedure
procedure $\textsc{Mark}(e)$
    $D^{(\text{sc})}.\textsc{AddTerminal}(u)$.
    $D^{(\text{sc})}.\textsc{AddTerminal}(v)$.
    $Z \leftarrow Z \cup \{e\}$.
    $D^{(\text{sc})}.\textsc{Update}(e, \boldsymbol{w}_e)$.    ▷ Make sure the $D^{(\text{sc})}$ puts edge $e$ in $Z$.
end procedure
procedure $\textsc{UpdateB}(v, \boldsymbol{b}^{(\text{new})})$
    $D^{(\text{sc})}.\textsc{AddTerminal}(v)$.
    $\overline{\boldsymbol{b}}_v \leftarrow \boldsymbol{b}^{(\text{new})}$.
end procedure
procedure $\textsc{UpdateW}(e, \boldsymbol{w}^{(\text{new})})$
    $\textsc{Mark}(e)$.
    $D^{(\text{sc})}.\textsc{Update}(e, \boldsymbol{w}^{(\text{new})})$.
    $\overline{\boldsymbol{w}}_e \leftarrow \boldsymbol{w}^{(\text{new})}$.
end procedure
procedure $\textsc{QueryPotential}$
    $\widetilde{\mathbf{SC}} \leftarrow D^{(\text{sc})}.\text{SC}()$.
    $\widetilde{\boldsymbol{x}} \leftarrow \textsc{Solve}\left(\widetilde{\mathbf{SC}}, \left[D^{(\text{sc})}.\textsc{InitialSC}() \quad 0\right] \boldsymbol{d}^{(\text{init})}\right)$.
    $\widetilde{\boldsymbol{x}} \leftarrow \widetilde{\boldsymbol{x}} + \textsc{Solve}\left(\widetilde{\mathbf{SC}}, \overline{\boldsymbol{b}} - \boldsymbol{b}^{(\text{init})}\right)$.
    return $\widetilde{\boldsymbol{x}}$.
end procedure
```

2.6.1 Harmonic Extension

A key notion we use throughout is the harmonic extension (Definition 2.5.2), which is a linear operator that maps the potentials restricted to a terminal set to the full electric potentials $\boldsymbol{\phi}$.

The inverse of the Laplacian can be represented by a contribution from the Schur complement, plus $\mathbf{L}_{FF}^{-1}$. This is essentially just a restatement of the Cholesky factorization (Lemma 2.3.2).

Lemma 2.6.1. *Let $G = (V, E, \boldsymbol{w})$ be a graph. Then*

$$\mathbf{L}(\boldsymbol{w})^\dagger = \mathcal{H}_C \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)^\dagger \mathcal{H}_C^\top + \begin{bmatrix} \mathbf{L}(\boldsymbol{w})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}.$$

Proof. The Cholesky factorization (Lemma 2.3.2) says that

$$\mathbf{L}(\boldsymbol{w})^\dagger = \begin{bmatrix} \mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ 0 & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{FF}^{-1} & 0 \\ 0 & \mathbf{SC}(\mathbf{L}, C)^\dagger \end{bmatrix} \begin{bmatrix} \mathbf{I} & 0 \\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix}.$$

As $\boldsymbol{T}\mathbf{SC}(\mathbf{L}, C)^\dagger \boldsymbol{T} = \mathbf{SC}(\mathbf{L}, C)^\dagger$, the equation above is equal to

$$\begin{aligned}
&\begin{bmatrix} \begin{bmatrix} \mathbf{I} \\ 0 \end{bmatrix} & \mathcal{H}_C \end{bmatrix} \begin{bmatrix} \mathbf{L}_{FF}^{-1} & 0 \\ 0 & \mathbf{SC}(\mathbf{L}, C)^\dagger \end{bmatrix} \begin{bmatrix} \begin{bmatrix} \mathbf{I} & 0 \end{bmatrix} \\ \mathcal{H}_C^\top \end{bmatrix} \\
&= \mathcal{H}_C \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathcal{H}_C^\top + \begin{bmatrix} \mathbf{I} \\ 0 \end{bmatrix} \mathbf{L}(\boldsymbol{w})_{F,F}^{-1} \begin{bmatrix} \mathbf{I} & 0 \end{bmatrix} \\
&= \mathcal{H}_C \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathcal{H}_C^\top + \begin{bmatrix} \mathbf{L}(\boldsymbol{w})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}
\end{aligned}$$

□

Let $\boldsymbol{v} \in \mathbb{R}^E$ be a (dynamic) vector. To implement the outer IPM, we must be able to maintain a heavy-hitter sketch on the following vector

$$\mathbf{\Pi}\boldsymbol{v} \overset{\text{def}}{=} \mathbf{\Pi}(\overline{\boldsymbol{w}})\boldsymbol{v} = \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathbf{L}(\overline{\boldsymbol{w}})^\dagger \mathbf{B}^\top \overline{\mathbf{W}}^{1/2}\boldsymbol{v}.$$

For this, we decompose $\mathbf{\Pi}\boldsymbol{v}$ into three terms. Let $\widehat{\boldsymbol{v}}$ be any vector that agrees with $\boldsymbol{v}$ in $F = V \setminus C$.

Then We first decompose $\mathbf{\Pi}(\boldsymbol{v})$ by Lemma 2.6.1.

$$\begin{aligned}
\mathbf{\Pi}(\boldsymbol{v}) =& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathbf{L}(\overline{\boldsymbol{w}})^{\dagger}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} \\
=& \overline{\mathbf{W}}^{1/2}\mathbf{B}\left(\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top} + \begin{bmatrix} \mathbf{L}(\overline{\boldsymbol{w}})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}\right)\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} \qquad \text{(by Lemma 2.6.1)} \\
=& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} + \overline{\mathbf{W}}^{1/2}\mathbf{B}\begin{bmatrix} \mathbf{L}(\overline{\boldsymbol{w}})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\widehat{\boldsymbol{v}}.
\end{aligned}$$

Since $\boldsymbol{w}_C$ does not affect the value of the second term, we have

$$\begin{aligned}
& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} + \overline{\mathbf{W}}^{1/2}\mathbf{B}\begin{bmatrix} \mathbf{L}(\overline{\boldsymbol{w}})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\widehat{\boldsymbol{v}} \\
=& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} + \mathbf{W}^{1/2}\mathbf{B}\begin{bmatrix} \mathbf{L}(\boldsymbol{w})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}}.
\end{aligned}$$

Then we use Lemma 2.6.1 in the other direction to get

$$\begin{aligned}
& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} + \mathbf{W}^{1/2}\mathbf{B}\begin{bmatrix} \mathbf{L}(\boldsymbol{w})_{F,F}^{-1} & 0 \\ 0 & 0 \end{bmatrix}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}} \\
=& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} + \mathbf{W}^{1/2}\mathbf{B}\left(\mathbf{L}(\boldsymbol{w})^{\dagger} - \mathcal{H}_C\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)^{\dagger}\mathcal{H}_C^{\top}\right)\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}} \\
& \qquad \text{(by Lemma 2.6.1)} \\
=& \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} - \mathbf{W}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}} \\
& + \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}}.
\end{aligned}$$

We will use

$$\boldsymbol{\Pi v} = \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} - \mathbf{W}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}} + \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}} \tag{2.8}$$

in two cases

- where $\widehat{\boldsymbol{v}} = \boldsymbol{v}$, and
- where $\boldsymbol{v}$ being the current vector and $\widehat{\boldsymbol{v}}$ being the initial $\boldsymbol{v}$.

At a high level, our approach will use several spectral approximations of the RHS of (2.8). We will replace $\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ with an approximate Schur complement using Theorem 2.4.10. Additionally, we will replace $\mathcal{H}_C$ and $\mathcal{H}_C^{\top}$ by replacing the Schur complements in (2.5) with approximate Schur complements given by Theorem 2.4.10.

We now focus on approximating the "right" of the first two terms of the RHS (2.8), i.e. the induced potentials on C

$$\boldsymbol{\phi} = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} \tag{2.9}$$

and

$$\boldsymbol{\psi} = \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\mathbf{W}^{1/2}\widehat{\boldsymbol{v}}. \tag{2.10}$$

Lemma 2.6.3 below defines the approximation of $\boldsymbol{\phi}$ that our data structures maintain. To analyze the quality of the approximation, we will need a standard spectral approximation inequality, proven for completeness.

Lemma 2.6.2 (Spectral approximation of differences)**.** *For PSD matrices $\mathbf{X} \approx_{\epsilon} \mathbf{Y}$, we have that*

$$(\mathbf{X} - \mathbf{Y})\mathbf{X}^{\dagger}(\mathbf{X} - \mathbf{Y}) \preceq \epsilon^2\mathbf{X}.$$

Proof. The desired inequality follows from $\left\|\mathbf{I} - \mathbf{X}^{\dagger/2}\mathbf{Y}\mathbf{X}^{\dagger/2}\right\|_2 \leq \epsilon$ and multiplying the LHS and RHS by $\mathbf{X}^{1/2}$ on the left and right. Now, this follows because $\mathbf{X} \approx_{\epsilon} \mathbf{Y}$ implies that $(1-\epsilon)\mathbf{I} \preceq \mathbf{X}^{\dagger/2}\mathbf{Y}\mathbf{X}^{\dagger/2} \preceq (1+\epsilon)\mathbf{I}$ as desired. □

Lemma 2.6.3 (Approximate potential)**.** *Let G be a graph with weights $\overline{\boldsymbol{w}} \in \mathbb{R}^E$ which differ from weights $\boldsymbol{w} \in \mathbb{R}^E$ except on an edge subset $Z \subseteq E(G)$. Let $C \subseteq V(G)$ contain all endpoints of edges in Z. Let $\boldsymbol{w}_{\overline{Z}} \in \mathbb{R}^E$ be defined as $(\boldsymbol{w}_{\overline{Z}})_e = 0$ for $e \in Z$ and $(\boldsymbol{w}_{\overline{Z}})_e = \boldsymbol{w}_e$ otherwise. Let $\widetilde{\mathbf{SC}} \approx_\epsilon \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ and let $\widetilde{\mathcal{H}} = \mathbf{L}(\boldsymbol{w})^\dagger \begin{bmatrix} 0 \\ \widetilde{\mathbf{SC}}_{\mathcal{H}} \end{bmatrix}$ for some $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ satisfying*

$$\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) - \epsilon \mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C) \preceq \widetilde{\mathbf{SC}}_{\mathcal{H}} \preceq \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) + \epsilon \mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C). \tag{2.11}$$

Then the vectors $\boldsymbol{\phi} = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v}$ ((2.9)) and $\widetilde{\boldsymbol{\phi}} = \widetilde{\mathbf{SC}}^\dagger \mathbf{B}^\top (\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}) \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \widetilde{\mathcal{H}}^\top \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}$ in $\mathbb{R}^C$ satisfy $\left\| \boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}} \right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)} \le 3\epsilon \|\boldsymbol{v}\|_2$.

Proof. We first calculate that

$$\begin{aligned}
\boldsymbol{\phi} &= \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger \right) \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} \\
&= \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger \right) \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathcal{H}_C^\top \mathbf{B}^\top \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2} \right) \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathcal{H}_C^\top \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v} \\
&\overset{(i)}{=} \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger \right) \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}_{:,C}^\top \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2} \right) \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathcal{H}_C^\top \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}
\end{aligned}$$

(i) is because $\mathcal{H}_C^\top \mathbf{B}^\top \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2} \right) = \mathbf{B}_{:,C}^\top \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2} \right)$ as $\overline{\boldsymbol{w}} = \boldsymbol{w}$ except on C. We extract $\widetilde{\boldsymbol{\phi}}$ by swapping the $\widetilde{\mathcal{H}}$ by $\widetilde{\mathcal{H}}_C$ in the last term:

$$\begin{aligned}
\boldsymbol{\phi} &= \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger \right) \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}_{:,C}^\top \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2} \right) \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \mathcal{H}_C^\top \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v} \\
&= \widetilde{\boldsymbol{\phi}} + \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger \right) \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \left(\mathcal{H}_C^\top - \widetilde{\mathcal{H}}^\top \right) \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}.
\end{aligned}$$

Hence

$$\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}} = \left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger \right) \mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2} \boldsymbol{v} + \widetilde{\mathbf{SC}}^\dagger \left(\mathcal{H}_C^\top - \widetilde{\mathcal{H}}^\top \right) \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}. \tag{2.12}$$

We bound both terms separately. For the first term,

$$\begin{aligned}&\left\|\left(\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger - \widetilde{\mathbf{SC}}^\dagger\right)\mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2}\boldsymbol{v}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \overset{(i)}{\leq} \epsilon \left\|\mathcal{H}_C^\top \mathbf{B}^\top \overline{\mathbf{W}}^{1/2}\boldsymbol{v}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^\dagger}\\ \overset{(ii)}{\leq}\ & \epsilon \left\|\mathbf{B}^\top \overline{\mathbf{W}}^{1/2}\boldsymbol{v}\right\|_{\mathbf{L}(\overline{\boldsymbol{w}})^\dagger} \leq \epsilon \|\boldsymbol{v}\|_2.\end{aligned}$$

where (i) follows from $\widetilde{\mathbf{SC}} \approx_\epsilon \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ and Lemma 2.6.2 for $\mathbf{X} = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger$ and $\mathbf{Y} = \widetilde{\mathbf{SC}}^\dagger$, and (ii) follows from Lemma 2.6.1 and the fact that $\mathbf{L}_{F,F}^{-1}$ is positive definite. For the second term,

$$\begin{aligned}&\left\|\widetilde{\mathbf{SC}}^\dagger\left(\mathcal{H}_C^\top - \widetilde{\mathcal{H}}^\top\right)\mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq 2\left\|\left(\mathcal{H}_C^\top - \widetilde{\mathcal{H}}^\top\right)\mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^\dagger}\\ \leq\ & 2\left\|\left(\mathcal{H}_C^\top - \widetilde{\mathcal{H}}^\top\right)\mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}),C)^\dagger} \overset{(i)}{\leq} 2\epsilon\left\|\left[\mathbf{L}(\boldsymbol{w})^\dagger \mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}\right]_C\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}),C)}\\ \overset{(ii)}{\leq}\ & 2\epsilon\left\|\left[\mathbf{L}(\boldsymbol{w})^\dagger \mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}\right]_C\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)} \overset{(iii)}{\leq} 2\epsilon\|\boldsymbol{v}\|_2,\end{aligned}$$

where (i) follows from Lemma 2.5.4 and (2.11), (ii) follows from $\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) - \mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}), C) = \mathbf{L}(G[C])$ being positive semidefinite, and (iii) follows from the fact that the Schur complement is spectrally smaller than the Laplacian: $\mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C) \preceq \mathbf{L}(\boldsymbol{w})$. □

By Lemma 2.6.3 with $\overline{\boldsymbol{w}} = \boldsymbol{w}$, we can approximate the other potential vector in the RHS of (2.8).

Corollary 2.6.4. *Let* $\boldsymbol{\psi} = \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)^\dagger \mathcal{H}_C^\top \mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}$ *((2.10)) and* $\widetilde{\boldsymbol{\psi}} = \widetilde{\mathbf{SC}}(\mathbf{L}(\boldsymbol{w}), C)^\dagger \widetilde{\mathcal{H}}^\top \mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}$ *in* $\mathbb{R}^C$ *where* $\widetilde{\mathbf{SC}}(\mathbf{L}(\boldsymbol{w}), C)$ *satisfies* $\widetilde{\mathbf{SC}}(\mathbf{L}(\boldsymbol{w}), C) \approx_\epsilon \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)$*. We have* $\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)} \leq 3\epsilon\|\boldsymbol{v}\|_2$.

Proof. Apply Lemma 2.6.3 with $\overline{\boldsymbol{w}} = \boldsymbol{w}$, $\boldsymbol{\psi} = \boldsymbol{\phi}$ and $\widetilde{\boldsymbol{\psi}} = \widetilde{\boldsymbol{\phi}}$. The first term of $\boldsymbol{\psi}$

$$\widetilde{\mathbf{SC}}^\dagger \mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\boldsymbol{v}$$

equals 0 because $\overline{\mathbf{W}} = \mathbf{W}$. □

We can use our approximate potential $\widetilde{\boldsymbol{\phi}}$, $\widetilde{\boldsymbol{\psi}}$ in Lemma 2.6.3 and Corollary 2.6.4 to define a full approximate projection of $\mathbf{\Pi}\boldsymbol{v}$. Our starting point is that $\mathbf{\Pi}\boldsymbol{v} = \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\boldsymbol{\phi} + \mathbf{W}^{1/2}\mathbf{B}\mathcal{H}_C\boldsymbol{\psi}$ for $\boldsymbol{\phi}$,

$\boldsymbol{\psi}$ as in Lemma 2.6.3 and Corollary 2.6.4.

Lemma 2.6.5 (Approximate projection)**.** *Let $\boldsymbol{w}, \overline{\boldsymbol{w}}, Z, \boldsymbol{w}_{\overline{Z}}, \widetilde{\mathcal{H}}, \boldsymbol{\phi}, \widetilde{\boldsymbol{\phi}}$ be as in Lemma 2.6.3, let $\boldsymbol{\psi}, \widetilde{\boldsymbol{\psi}}$ be as in Corollary 2.6.4, and let*

$$\begin{aligned}\widetilde{\boldsymbol{\Pi}}\boldsymbol{v} = &\left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \mathbf{W}^{1/2}\mathbf{B}\widetilde{\mathcal{H}}\widetilde{\boldsymbol{\phi}} \\ &+ \mathbf{W}^{1/2}\mathbf{B}\widetilde{\mathcal{H}}\widetilde{\boldsymbol{\psi}} \\ &+ \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}\end{aligned}$$

where $\widetilde{\boldsymbol{\phi}}$ is padded with zeroes for computing $\mathbf{B}\widetilde{\boldsymbol{\phi}}$. Then

$$\left\|\boldsymbol{\Pi}\boldsymbol{v} - \widetilde{\boldsymbol{\Pi}}\boldsymbol{v}\right\|_2 \leq 2\epsilon\|\boldsymbol{v}\|_2 + (1+\epsilon)\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} + (1+\epsilon)\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)}.$$

Proof. We first prove the first two terms of $\widetilde{\boldsymbol{\Pi}}\boldsymbol{v}$

$$\widetilde{\boldsymbol{\Pi}}_{\phi}(\boldsymbol{v}) \stackrel{\text{def}}{=} \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \mathbf{W}^{1/2}\mathbf{B}\widetilde{\mathcal{H}}\widetilde{\boldsymbol{\phi}}$$

approximates $\boldsymbol{\Pi}_{\phi}(\boldsymbol{v}) \stackrel{\text{def}}{=} \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\boldsymbol{\phi}$. Specifically,

$$\left\|\boldsymbol{\Pi}_{\phi}(\boldsymbol{v}) - \widetilde{\boldsymbol{\Pi}}_{\phi}(\boldsymbol{v})\right\|_2 \leq \epsilon\|\boldsymbol{v}\|_2 + (1+\epsilon)\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)}.$$

We start by calculating

$$\begin{aligned}\boldsymbol{\Pi}_{\phi}(\boldsymbol{v}) &= \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\boldsymbol{\phi} = \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\left(\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right) + \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\widetilde{\boldsymbol{\phi}} \\ &= \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\left(\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right) + \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\mathcal{H}_C\widetilde{\boldsymbol{\phi}} + \mathbf{W}^{1/2}\mathbf{B}\mathcal{H}_C\widetilde{\boldsymbol{\phi}} \\ &\stackrel{(i)}{=} \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\left(\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right) + \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \mathbf{W}^{1/2}\mathbf{B}\mathcal{H}_C\widetilde{\boldsymbol{\phi}} \\ &= \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\left(\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right) + \mathbf{W}^{1/2}\mathbf{B}(\mathcal{H}_C - \widetilde{\mathcal{H}})\widetilde{\boldsymbol{\phi}} + \widetilde{\boldsymbol{\Pi}}_{\phi}(\boldsymbol{v}),\end{aligned}$$

where (i) follows from $\left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\mathcal{H}_C = \left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}$ as $\overline{\boldsymbol{w}} = \boldsymbol{w}$ outside C. Hence

$$\boldsymbol{\Pi}_\phi(\boldsymbol{v}) - \widetilde{\boldsymbol{\Pi}}_\phi(\boldsymbol{v}) = \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\left(\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right) + \mathbf{W}^{1/2}\mathbf{B}(\mathcal{H}_C - \widetilde{\mathcal{H}})\widetilde{\boldsymbol{\phi}}. \tag{2.13}$$

We bound both terms of (2.13) separately. For the first term, note that

$$\left\|\overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\left(\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right)\right\|_2 \overset{(i)}{\leq} \left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)},$$

where (i) follows from properties of $\mathcal{H}_C$. For the second term of (2.13),

$$\left\|\mathbf{W}^{1/2}\mathbf{B}(\mathcal{H}_C - \widetilde{\mathcal{H}})\widetilde{\boldsymbol{\phi}}\right\|_2 \overset{(i)}{\leq} \epsilon\left\|\widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}_{\overline{Z}}),C)} \leq \epsilon\left\|\widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \overset{(ii)}{\leq} \epsilon\|\boldsymbol{v}\|_2 + \left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)}$$

where (i) follows from (2.11), and (ii) is because

$$\|\widetilde{\boldsymbol{\phi}}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq \|\widetilde{\boldsymbol{\phi}} - \boldsymbol{\phi}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} + \|\boldsymbol{\phi}\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq \left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} + \|\boldsymbol{v}\|_{\mathbf{P}_{\overline{\boldsymbol{w}}}} \leq \|\boldsymbol{v}\|_2$$

because $\mathbf{P}_{\overline{\mathbf{W}}}$ is an orthogonal projection matrix. Summing these errors completes the proof for

$$\left\|\boldsymbol{\Pi}_\phi(\boldsymbol{v}) - \widetilde{\boldsymbol{\Pi}}_\phi(\boldsymbol{v})\right\|_2 \leq \epsilon\|\boldsymbol{v}\|_2 + (1+\epsilon)\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)}. \tag{2.14}$$

We then define

$$\widetilde{\boldsymbol{\Pi}}_\psi(\boldsymbol{v}) \overset{\text{def}}{=} \left(\mathbf{W}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \mathbf{W}^{1/2}\mathbf{B}\widetilde{\mathcal{H}}\widetilde{\boldsymbol{\phi}}$$

which is the third term of $\boldsymbol{\Pi v}$ and

$$\boldsymbol{\Pi}_\psi(\boldsymbol{v}) \overset{\text{def}}{=} \overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\boldsymbol{\psi}.$$

Then, by the proof above with $\overline{\boldsymbol{w}}$ replaced by $\boldsymbol{w}$ (and $\overline{\mathbf{W}}$ replaced by $\mathbf{W}$), we get

$$\left\|\boldsymbol{\Pi}_\psi(\boldsymbol{v}) - \widetilde{\boldsymbol{\Pi}}_\psi(\boldsymbol{v})\right\|_2 \leq \epsilon\|\boldsymbol{v}\|_2 + (1+\epsilon)\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)}. \tag{2.15}$$

Recall that

$$\begin{aligned}\mathbf{\Pi}\boldsymbol{v} =&\overline{\mathbf{W}}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\overline{\mathbf{W}}^{1/2}\boldsymbol{v} - \mathbf{W}^{1/2}\mathbf{B}\mathcal{H}_C\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}\\ &+\mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}.\end{aligned}$$

Its first two terms are approximated respectively by $\widetilde{\mathbf{\Pi}}_{\phi}(\boldsymbol{v})$ and $\widetilde{\mathbf{\Pi}}_{\psi}(\boldsymbol{v})$, the first two terms in $\widetilde{\mathbf{\Pi}}\boldsymbol{v}$. Its last term is exactly the last term of $\widetilde{\mathbf{\Pi}}\boldsymbol{v}$. By triangle inequality and (2.14), (2.15), we have

$$\left\|\mathbf{\Pi}\boldsymbol{v} - \widetilde{\mathbf{\Pi}}\boldsymbol{v}\right\|_2 \leq 2\epsilon\|\boldsymbol{v}\|_2 + (1+\epsilon)\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} + (1+\epsilon)\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)}.$$

□

In the following sections, we will use $\textsc{Solve}(\mathbf{L}, \boldsymbol{b})$ to denote a high accuracy Laplacian solver that returns $\boldsymbol{x}$ such that $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ and runs in nearly linear time. We will overload notation to extend any dimension of a matrix from a subset of V to V, or from a subset of E to E by padding zeroes.

2.6.2 Dynamic Potential Maintanence

In this section, we show how to maintain the vector $\widetilde{\boldsymbol{\phi}}$ (Lemma 2.6.3) that approximates the potential vector $\boldsymbol{\phi}$. This data structure can also be used for $\boldsymbol{\psi}$ (Corollary 2.6.4).

Lemma 2.6.6 (Dynamic Potential). *For a graph $G = (V, E)$ with dynamic edge conductances $\overline{\boldsymbol{w}} \in \mathbb{R}_{\geq 0}^{E(G)}$ and a dynamic vector $\overline{\boldsymbol{v}} \in \mathbb{R}^{E(G)}$ for some constant C, there is a data structure (Algorithm 5) that supports the following operations against an oblivious adversary for parameters $\beta < \epsilon^2 < 1$.*

- *$\textsc{Initialize}(G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \beta, \epsilon)$. Initializes the data structure in time $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$ with an empty set $Z \leftarrow \emptyset$ of marked edges. Initialize $\overline{\boldsymbol{w}}$ as $\boldsymbol{w}$ and $\overline{\boldsymbol{v}}$ as $\boldsymbol{v}^{(\mathrm{init})}$.*
- *$\textsc{Update}V(e, \overline{\boldsymbol{v}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{v}}_e \leftarrow \overline{\boldsymbol{v}}^{(\mathrm{new})}$ in $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *$\textsc{Update}W(e, \overline{\boldsymbol{w}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{w}}_e \leftarrow \overline{\boldsymbol{w}}^{(\mathrm{new})}$ in $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*

- *QueryPotential(). For $C \subseteq V$ with $|C| = O(\beta m)$ and $Z \subseteq E(C)$, returns in $\widetilde{O}(\beta m \epsilon^{-2})$ time a vector $\widetilde{\boldsymbol{\phi}}$ satisfying $\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \le \epsilon \|\overline{\boldsymbol{v}}\|_2$ where $\boldsymbol{\phi} = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathcal{H}_C^\top \mathbf{B}\overline{\mathbf{W}}^{1/2}\overline{\boldsymbol{v}}$.*

Runtimes and output correctness hold w.h.p. if there are at most $O(\beta m)$ calls to UpdateV and UpdateW in total.

Proof. The pseudocode for the proof of Lemma 2.6.6 is in Algorithm 5. At a high-level, it simply maintains $\widetilde{\boldsymbol{\phi}}$ as in Lemma 2.6.3. The one difference is that it handles changes to $\boldsymbol{v}^{(\mathrm{init})}$ directly because both endpoints of all edges changed in $\boldsymbol{v}^{(\mathrm{init})}$ are in the marked set Z. We start by analyzing the correctness of the algorithm, then move the runtime.

Correctness. We only need to analyze the QueryPotential() operation. In this proof, we show the weaker bound $\left\|\widetilde{\boldsymbol{\phi}} - \boldsymbol{\phi}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \le 4\epsilon \left(\|\overline{\boldsymbol{v}}\|_2 + \|\boldsymbol{v}^{(\mathrm{init})}\|_2\right)$. However, this suffices because we can build $\widetilde{O}(1)$ copies of the data structure. For $-\widetilde{O}(1) \le j \le \widetilde{O}(1)$, the j-th instance initializes and answers queries only when $\|\overline{\boldsymbol{v}}\|_2 \in (2^j, 2^{j+1}]$. Updates are passed to all instances. When the number of updates exceeds $O(\beta m)$ for an instance but it cannot be initialized because $\|\overline{\boldsymbol{v}}\|_2$ does not fall in its range, it simply ignore following updates until it can be initialized. This only increases the runtime by $\widetilde{O}(1)$ factors.

Let $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ be the value of $D^{(\mathrm{sc})}$.InitialSC() returned in line 28 of Algorithm 5. It satisfies condition (2.11) by the guarantees of Theorem 2.4.10. Also, by inspection of the procedure QueryPotential() in Algorithm 5, the returned vector $\widetilde{\boldsymbol{\phi}}$ is defined as

$$\begin{aligned}
\widetilde{\boldsymbol{\phi}} &= \widetilde{\mathbf{SC}}^\dagger \begin{bmatrix} \widetilde{\mathbf{SC}}_{\mathcal{H}} & 0 \end{bmatrix} \boldsymbol{d}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}\mathbf{W}^{1/2}(\overline{\boldsymbol{v}}_Z - \boldsymbol{v}_Z^{(\mathrm{init})}) + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\overline{\boldsymbol{v}}_Z \\
&= \widetilde{\mathbf{SC}}^\dagger \begin{bmatrix} \widetilde{\mathbf{SC}}_{\mathcal{H}} & 0 \end{bmatrix} \mathbf{L}(\boldsymbol{w})^\dagger \mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}\mathbf{W}^{1/2}(\overline{\boldsymbol{v}}_Z - \boldsymbol{v}_Z^{(\mathrm{init})}) + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\overline{\boldsymbol{v}}_Z \\
&= \widetilde{\mathbf{SC}}^\dagger \widetilde{\mathcal{H}}^\top \mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\boldsymbol{v}^{(\mathrm{init})} + \widetilde{\mathbf{SC}}^\dagger \mathbf{B}\overline{\mathbf{W}}^{1/2}(\overline{\boldsymbol{v}}_Z - \boldsymbol{v}_Z^{(\mathrm{init})}).
\end{aligned}$$

Additionally, because $\boldsymbol{v}^{(\mathrm{init})} - \overline{\boldsymbol{v}}$ is supported on Z, the true $\boldsymbol{\phi}$ can be written as

$$\begin{aligned}
\boldsymbol{\phi} &= \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathcal{H}_C^\top \mathbf{B}\overline{\mathbf{W}}^{1/2}\overline{\boldsymbol{v}} \\
&= \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathcal{H}_C^\top \mathbf{B}\overline{\mathbf{W}}^{1/2}\boldsymbol{v}^{(\mathrm{init})} + \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^\dagger \mathbf{B}\overline{\mathbf{W}}^{1/2}(\overline{\boldsymbol{v}} - \boldsymbol{v}^{(\mathrm{init})}).
\end{aligned}$$

Hence we get that

$$\begin{aligned}
&\left\|\widetilde{\boldsymbol{\phi}}-\boldsymbol{\phi}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \\
\leq\;& \left\|\widetilde{\mathbf{SC}}^{\dagger}\widetilde{\mathcal{H}}_C\mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}^{(\mathrm{init})}+\widetilde{\mathbf{SC}}^{\dagger}\mathbf{B}(\overline{\mathbf{W}}^{1/2}-\mathbf{W}^{1/2})\boldsymbol{v}^{(\mathrm{init})}-\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}\overline{\mathbf{W}}^{1/2}\boldsymbol{v}^{(\mathrm{init})}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \\
+\;& \left\|\widetilde{\mathbf{SC}}^{\dagger}\mathbf{B}\overline{\mathbf{W}}^{1/2}(\overline{\boldsymbol{v}}_Z-\boldsymbol{v}_Z^{(\mathrm{init})})-\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)^{\dagger}\mathbf{B}\overline{\mathbf{W}}^{1/2}(\overline{\boldsymbol{v}}-\boldsymbol{v}^{(\mathrm{init})})\right\| \\
\overset{(i)}{\leq}\;& 3\epsilon\|\boldsymbol{v}^{(\mathrm{init})}\|_2+\epsilon\|\overline{\boldsymbol{v}}-\boldsymbol{v}^{(\mathrm{init})}\|_2\leq 4\epsilon\left(\|\overline{\boldsymbol{v}}\|_2+\|\boldsymbol{v}^{(\mathrm{init})}\|_2\right),
\end{aligned}$$

where (i) follows from Lemma 2.6.3 for the first term, and $\widetilde{\mathbf{SC}}\approx_{\epsilon}\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)$ from the guarantee of Theorem 2.4.10 for the second term. This suffices because Algorithm 5 set $\epsilon\leftarrow\epsilon/12$ in line 3.

Runtime. The runtimes of UpdateV, UpdateW are trivially the same as the runtime of Mark. The runtimes of Mark and Initialize follows from the AddTerminal and Initialize operations respectively of Theorem 2.4.10. The runtime of QueryPotential is $\widetilde{O}(\beta m\epsilon^{-2})$ by the runtime guarantees of SC and InitialSC of Theorem 2.4.10, and the fact that $\widetilde{\mathbf{SC}}$ and $\widetilde{\mathbf{SC}}_{\mathcal{H}}$ all have $\widetilde{O}(\beta m\epsilon^{-2})$ edges, and hence solving or multiplying by them costs $\widetilde{O}(\beta m\epsilon^{-2})$ time. $\square$

2.6.3 Dynamic Evaluator

Theorem 2.6.7 (Dynamic Evaluator)**.** *For a graph $G=(V,E)$ with dynamic edge conductances $\overline{\boldsymbol{w}}\in\mathbb{R}_{\geq 0}^{E(G)}$ and a dynamic vector $\overline{\boldsymbol{v}}\in\mathbb{R}^{E(G)}$, there is a data structure* Evaluator *that supports the following operations against an oblivious adversary for parameters $\beta<\epsilon^2<1$.*

- *Initialize$(G,\boldsymbol{w},\boldsymbol{v}^{(\mathrm{init})},\beta,\epsilon)$. Initializes the data structure in time $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$ with an empty set $Z\leftarrow\emptyset$ of marked edges. Initializes $\overline{\boldsymbol{w}}$ as $\boldsymbol{w}$, $\overline{\boldsymbol{v}}$ as $\boldsymbol{v}^{(\mathrm{init})}$.*
- *UpdateV$(e,\overline{\boldsymbol{v}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{v}}_e\leftarrow\overline{\boldsymbol{v}}^{(\mathrm{new})}$ in $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *UpdateW$(e,\overline{\boldsymbol{w}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{w}}_e\leftarrow\overline{\boldsymbol{w}}^{(\mathrm{new})}$ in $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *Query$()$. Returns a vector $\boldsymbol{u}\in\mathbb{R}^Z$ satisfying $\|\boldsymbol{u}-[\mathbf{P}_{\overline{\boldsymbol{w}}}\overline{\boldsymbol{v}}]_Z\|_2\leq\epsilon\|\overline{\boldsymbol{v}}\|_2+\epsilon\|\boldsymbol{v}\|_2$ in time $\widetilde{O}(\beta m\epsilon^{-2})$.*

Algorithm 5 Dynamic Potential

```
1:                                                  ▷ This implementation assumes
   that $\|\overline{\boldsymbol{v}}\|_2 \approx \|\boldsymbol{v}^{(\mathrm{init})}\|_2$ always. However, this can be achieved by duplicating the data structure
   $\widetilde{O}(1)$ times, one handling each range $\|\overline{\boldsymbol{v}}\|_2 \in [2^j, 2^{j+1}]$ for $-\widetilde{O}(1) \le j \le \widetilde{O}(1)$.
2: procedure INITIALIZE($G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \beta, \epsilon$)
3:     $\epsilon \leftarrow \epsilon/12$.
4:     Let $D^{(\mathrm{sc})}$ be a instance of the dynamic Schur complement data structure of Theorem 2.4.10.
5:     $D^{(\mathrm{sc})}$.INITIALIZE($G, \boldsymbol{w}, \epsilon, \beta$).
6:     $\overline{\boldsymbol{v}} \leftarrow \boldsymbol{v}^{(\mathrm{init})}$.    ▷ $\boldsymbol{v}^{(\mathrm{init})}$ is the initial vector and $\overline{\boldsymbol{v}}$ to denote the current vector $\boldsymbol{v}$ throughout
   the algorithm.
7:     $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w}$.    ▷ We use $\boldsymbol{w}$ to denote the initial vector and $\overline{\boldsymbol{w}}$ to denote the current vector $\boldsymbol{w}$
   throughout the algorithm.
8:     $\boldsymbol{d}^{(\mathrm{init})} \leftarrow$ SOLVE($\mathbf{L}(\boldsymbol{w}), \mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}^{(\mathrm{init})}$).
9:     $Z \leftarrow \emptyset$.                                  ▷ Marked edges.
10: end procedure
11: procedure MARK($e$)
12:     $D^{(\mathrm{sc})}$.ADDTERMINAL($u$).
13:     $D^{(\mathrm{sc})}$.ADDTERMINAL($v$).
14:     $Z \leftarrow Z \cup \{e\}$.
15:     $D^{(\mathrm{sc})}$.UPDATE($e, \boldsymbol{w}_e$).             ▷ Make sure the $D^{(\mathrm{sc})}$ puts edge $e$ in $Z$.
16: end procedure
17: procedure UPDATEV($e, \boldsymbol{v}^{(\mathrm{new})}$)
18:     MARK($e$).
19:     $\overline{\boldsymbol{v}}_e \leftarrow \boldsymbol{v}^{(\mathrm{new})}$.
20: end procedure
21: procedure UPDATEW($e, \boldsymbol{w}^{(\mathrm{new})}$)
22:     MARK($e$).
23:     $D^{(\mathrm{sc})}$.UPDATE($e, \boldsymbol{w}^{(\mathrm{new})}$).
24:     $\overline{\boldsymbol{w}}_e \leftarrow \boldsymbol{w}^{(\mathrm{new})}$.
25: end procedure
26: procedure QUERYPOTENTIAL
27:     $\widetilde{\mathbf{SC}} \leftarrow D^{(\mathrm{sc})}$.SC().
28:     $\widetilde{\boldsymbol{\phi}} \leftarrow$ SOLVE $\left(\widetilde{\mathbf{SC}}, \begin{bmatrix} D^{(\mathrm{sc})}.\mathrm{INITIALSC}() & 0 \end{bmatrix} \boldsymbol{d}^{(\mathrm{init})}\right)$.
29:     $\widetilde{\boldsymbol{\phi}} \leftarrow \widetilde{\boldsymbol{\phi}} +$ SOLVE($\widetilde{\mathbf{SC}}, \mathbf{B}\mathbf{W}^{1/2}(\overline{\boldsymbol{v}}_Z - \boldsymbol{v}_Z^{(\mathrm{init})})$).
30:     $\widetilde{\boldsymbol{\phi}} \leftarrow \widetilde{\boldsymbol{\phi}} +$ SOLVE($\widetilde{\mathbf{SC}}, \mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\overline{\boldsymbol{v}}_Z$).
31:     return $\widetilde{\boldsymbol{\phi}}$.
32: end procedure
```

Runtimes and output correctness hold w.h.p. if there are at most $O(\beta m)$ calls to UPDATEV, UPDATEW *in total.*

Proof. We decompose $\mathbf{P}_{\overline{\boldsymbol{w}}}\overline{\boldsymbol{v}}$ by (2.8). The $\boldsymbol{v}$ in (2.8) is the current vector $\overline{\boldsymbol{v}}$ and the $\widehat{\boldsymbol{v}}$ in (2.8) is the initial vector $\boldsymbol{v}$ here. We create two instances of Algorithm 5 $D^{(\phi)}$ and $D^{(\psi)}$ maintaining

$$\widetilde{\boldsymbol{\phi}} = \widetilde{\mathbf{SC}}^{\dagger}\mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\overline{\boldsymbol{v}} + \widetilde{\mathbf{SC}}^{\dagger}\widetilde{\mathcal{H}}^{\top}\mathbf{B}\mathbf{W}^{1/2}\overline{\boldsymbol{v}}$$

(Lemma 2.6.3) and

$$\widetilde{\boldsymbol{\psi}} = \widetilde{\mathbf{SC}}(\mathbf{L}(\boldsymbol{w}), C)^{\dagger}\widetilde{\mathcal{H}}^{\top}\mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}$$

(Corollary 2.6.4) respspectively. Initialize, Mark are forwarded to both $D^{(\phi)}$ and $D^{(\psi)}$. The operations UpdateV, UpdateW are forwarded only to $D^{(\phi)}$ as $D^{(\psi)}$ maintains $\widetilde{\boldsymbol{\psi}}$ where $\boldsymbol{w}$ and $\boldsymbol{v}$ do not change. We also compute the exact value of the last term $\mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}$ of $\mathbf{P}_{\overline{\boldsymbol{w}}}\overline{\boldsymbol{v}}$ by

$$\boldsymbol{x} = \mathbf{W}^{1/2}\mathbf{B}\textsc{Solve}(\mathbf{L}(\boldsymbol{w}), \mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}).$$

For Query(), let $\widetilde{\boldsymbol{\phi}} = D^{(\phi)}.\textsc{QueryPotential}()$, $\widetilde{\boldsymbol{\psi}} = D^{(\psi)}.\textsc{QueryPotential}()$. The Evaluator returns

$$\boldsymbol{u} = \left[\overline{\mathbf{W}}^{1/2}\mathbf{B}\widetilde{\boldsymbol{\phi}}\right]_Z + \left[\mathbf{W}^{1/2}\mathbf{B}\widetilde{\boldsymbol{\psi}}\right]_Z + \boldsymbol{x}_Z.$$

Clearly all runtimes transfer exactly from Lemma 2.6.6. It suffices to show the correctness of Query().

For the true potentials $\boldsymbol{\phi} = \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}\overline{\mathbf{W}}^{1/2}\overline{\boldsymbol{v}}$ and $\boldsymbol{\psi} = \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)^{\dagger}\mathcal{H}_C^{\top}\mathbf{B}\mathbf{W}^{1/2}\boldsymbol{v}$ we have $[\mathbf{P}_{\overline{\boldsymbol{w}}}\overline{\boldsymbol{v}}]_Z = \left[\overline{\mathbf{W}}^{1/2}\mathbf{B}\boldsymbol{\phi} + \mathbf{W}^{1/2}\mathbf{B}\boldsymbol{\psi}\right]_Z + \boldsymbol{x}_Z$. Thus,

$$\begin{aligned}
\|\boldsymbol{u} - [\mathbf{P}_{\overline{\boldsymbol{w}}}\overline{\boldsymbol{v}}]_Z\|_2 &= \left\|\overline{\mathbf{W}}_Z^{1/2}\mathbf{B}_Z\left(\widetilde{\boldsymbol{\phi}} - \boldsymbol{\phi}\right) + \mathbf{W}^{1/2}\mathbf{B}_Z\left(\widetilde{\boldsymbol{\psi}} - \boldsymbol{\psi}\right)\right\|_2 \\
&\le \left\|\overline{\mathbf{W}}_Z^{1/2}\mathbf{B}_Z\left(\widetilde{\boldsymbol{\phi}} - \boldsymbol{\phi}\right)\right\|_2 + \left\|\mathbf{W}^{1/2}\mathbf{B}_Z\left(\widetilde{\boldsymbol{\psi}} - \boldsymbol{\psi}\right)\right\|_2 \\
&= \left\|\widetilde{\boldsymbol{\phi}} - \boldsymbol{\phi}\right\|_{\mathbf{B}_Z^{\top}\overline{\mathbf{W}}_Z\mathbf{B}_Z} + \left\|\widetilde{\boldsymbol{\psi}} - \boldsymbol{\psi}\right\|_{\mathbf{B}_Z^{\top}\mathbf{W}_Z\mathbf{B}_Z} \\
&\overset{(i)}{\le} \left\|\widetilde{\boldsymbol{\phi}} - \boldsymbol{\phi}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} + \left\|\widetilde{\boldsymbol{\psi}} - \boldsymbol{\psi}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)} \\
&\overset{(ii)}{\le} \epsilon\|\overline{\boldsymbol{v}}\|_2 + \epsilon\|\boldsymbol{v}\|_2
\end{aligned}$$

where (i) follows from the fact that $\mathbf{L}(\overline{\boldsymbol{w}}_Z) \preceq \mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}), C)$ (and $\mathbf{L}(\boldsymbol{w}_Z) \preceq \mathbf{SC}(\mathbf{L}(\boldsymbol{w}), C)$) as Z is completely inside C, and (ii) follows from the guarantee of QUERYPOTENTIAL() of Lemma 2.6.6. This completes the proof. $\square$

2.6.4 Dynamic Locator

Theorem 2.6.8 (Dynamic Locator)**.** *For a graph $G = (V, E)$ with dynamic edge conductances $\overline{\boldsymbol{w}} \in \mathbb{R}_{\geq 0}^{E(G)}$ and a dynamic vector $\overline{\boldsymbol{v}} \in \mathbb{R}^{E(G)}$, there is a data structure LOCATOR (given in Algorithm 6) that supports the following operations against an oblivious adversary for parameters $\beta < \epsilon^2 < 1$.*

- *INITIALIZE$(G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \beta, \epsilon)$. Initializes the data structure in time $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$ and sets $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w}$ and $\overline{\boldsymbol{v}} \leftarrow \boldsymbol{v}^{(\mathrm{init})}$.*
- *UPDATEV$(e, \overline{\boldsymbol{v}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{v}}_e \leftarrow \overline{\boldsymbol{v}}^{(\mathrm{new})}$ in $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *UPDATEW$(e, \overline{\boldsymbol{w}}^{(\mathrm{new})})$. Updates $\overline{\boldsymbol{w}}_e \leftarrow \overline{\boldsymbol{w}}^{(\mathrm{new})}$ in $\widetilde{O}(\beta^{-2}\epsilon^{-2})$ time.*
- *LOCATE(). Returns in time $\widetilde{O}(\beta m \epsilon^{-2})$ a set $S \subseteq E(G)$ with $|S| \leq O(\epsilon^{-2})$ containing all edges e with $|[\mathbf{P}_{\boldsymbol{w}}\overline{\boldsymbol{v}}]_e| \geq \epsilon \|\overline{\boldsymbol{v}}\|_2$ whp.*

Runtimes and output correctness hold w.h.p. if there are at most $O(\beta m)$ calls to UPDATEV, UPDATEW in total.

The following lemma is implicit in [111] and allows us to recover the large entries of $\boldsymbol{x}$ by a low-dimensional projection of it.

Lemma 2.6.9 (ℓ_2-heavy hitter, [111])**.** *There exists a function SKETCH(ϵ, n) that given $\epsilon > 0$ explicitly returns a random matrix $\mathbf{Q} \in \mathbb{R}^{N \times m}$ with $N = O(\epsilon^{-2} \log^3 m)$ and column sparsity $c = O(\log^3 m)$ in $\widetilde{O}(N + m)$ time, and uses $\widetilde{O}(N + m)$ spaces to store the matrix $\mathbf{Q}$. There further exists a function RECOVER$(\mathbf{Q}\boldsymbol{x})$ that in time $O(\epsilon^{-2} \log^3 m)$ reports a list $S \subset [m]$ of size $O(\epsilon^{-2})$. For any fixed $\boldsymbol{x}$, the list includes all i with $|\boldsymbol{x}_i| \geq \epsilon \|\boldsymbol{x}\|_2$ with high probability over the randomness of Q.*

Proof of Theorem 2.6.8. At a high level, Algorithm 6 simply maintains the formula given by Lemma 2.6.5 for $\widetilde{\boldsymbol{\phi}}$ and $\widetilde{\boldsymbol{\psi}}$ given by the output of the dynamic potential maintenance data structure in Lemma 2.6.6.

Algorithm 6 Dynamic Locator

```
1: procedure INITIALIZE($G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \beta, \epsilon$)
2:     $\epsilon \leftarrow \epsilon/10$.
3:     Let $D^{(\phi)}$ be an instance of the dynamic potential data structure of Lemma 2.6.6.
4:     Let $D^{(\psi)}$ be an instance of the dynamic potential data structure of Lemma 2.6.6.
5:     Let $D^{(\mathrm{sc})}$ be an instance of the dynamic Schur complement data structure of Theorem 2.4.10.
6:     $D^{(\phi)}$.INITIALIZE($G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \epsilon, \beta$).
7:     $D^{(\psi)}$.INITIALIZE($G, \boldsymbol{w}, \boldsymbol{v}^{(\mathrm{init})}, \epsilon, \beta$).
8:     $D^{(\mathrm{sc})}$.INITIALIZE($G, \boldsymbol{w}, \epsilon, \beta$).
9:     Initialize an $N = O(\epsilon^{-2}\log^3 m)$ by $m$ matrix $\mathbf{Q}$ with rows $\boldsymbol{q}^{(1)}, \boldsymbol{q}^{(2)}, \ldots, \boldsymbol{q}^{(N)} \in \mathbb{R}^m$ by Lemma 2.6.9.
10:    for $i \in [N]$ do
11:        $\boldsymbol{\gamma}^{(i)} \leftarrow$ SOLVE($\mathbf{L}(\boldsymbol{w}), \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{q}^{(i)}$).    ▷ $\boldsymbol{\gamma}^{(i)}$ are rows of $\boldsymbol{\Gamma} \overset{\mathrm{def}}{=} \mathbf{Q}\mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^\dagger$.
12:    end for
13:    $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w}, \overline{\boldsymbol{v}} \leftarrow \boldsymbol{v}$.
14:    $\boldsymbol{y} \leftarrow \mathbf{B}^\top \mathbf{W} \boldsymbol{v}$.
15: end procedure
16: procedure UPDATEV($e, \boldsymbol{v}^{(\mathrm{new})}$)
17:    $D^{(\phi)}$.UPDATEV($e, \boldsymbol{v}^{(\mathrm{new})}$).
18:    $D^{(\psi)}$.UPDATEV($e, \boldsymbol{v}^{(\mathrm{new})}$).
19:    $D^{(\mathrm{sc})}$.UPDATEV($e, \boldsymbol{v}^{(\mathrm{new})}$).
20:    $\overline{\boldsymbol{v}}_e \leftarrow \boldsymbol{v}^{(\mathrm{new})}$.
21:    $\boldsymbol{y} \leftarrow \boldsymbol{y} + \mathbf{B}^\top \mathbf{W}(\boldsymbol{v}^{(\mathrm{new})} - \boldsymbol{v}_e)$.
22: end procedure
23: procedure UPDATEW($e, \boldsymbol{w}^{(\mathrm{new})}$)
24:    $D^{(\phi)}$.UPDATEW($e, \boldsymbol{v}^{(\mathrm{new})}$).    ▷ $D^{(\psi)}$ does not update $\overline{\boldsymbol{w}}_e$ to $\boldsymbol{w}^{(\mathrm{new})}$.
25:    $D^{(\mathrm{sc})}$.UPDATEW($e, \boldsymbol{w}^{(\mathrm{new})}$).
26:    $\overline{\boldsymbol{w}}_e \leftarrow \boldsymbol{w}^{(\mathrm{new})}$.
27: end procedure
28: procedure LOCATE
29:    $\widetilde{\boldsymbol{\phi}} \leftarrow D^{(\phi)}$.QUERYPOTENTIAL().    ▷ $\widetilde{\boldsymbol{\phi}}$ is padded with zeroes for computing $\mathbf{B}\widetilde{\boldsymbol{\phi}}$
30:    $\widetilde{\boldsymbol{\psi}} \leftarrow D^{(\psi)}$.QUERYPOTENTIAL().
31:    $\boldsymbol{p} \leftarrow \mathbf{Q}\left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \boldsymbol{\Gamma}\begin{bmatrix} 0 \\ D^{(\mathrm{sc})}.\textsc{InitialSC}() \end{bmatrix}\widetilde{\boldsymbol{\phi}} + \boldsymbol{\Gamma}\begin{bmatrix} 0 \\ D^{(\mathrm{sc})}.\textsc{InitialSC}() \end{bmatrix}\widetilde{\boldsymbol{\psi}} + \boldsymbol{\Gamma}\boldsymbol{y}$.
32:    Return the set $S$ returned by calling RECOVER($\boldsymbol{p}$) of Lemma 2.6.9.
33: end procedure
```

Let us first show correctness and then analyze runtime. We only have to check correctness of LOCATE(). We follow the decomposition (2.8) with both $\boldsymbol{v}$ and $\widehat{\boldsymbol{v}}$ being the current vector $\overline{\boldsymbol{v}}$ here. The $\widetilde{\boldsymbol{\phi}}$ and $\widetilde{\boldsymbol{\psi}}$ maintained by Algorithm 6 satisfy

$$\widetilde{\boldsymbol{\phi}} = \widetilde{\mathbf{SC}}^{\dagger}\mathbf{B}(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2})\boldsymbol{v} + \widetilde{\mathbf{SC}}^{\dagger}\widetilde{\mathcal{H}}^{\top}\mathbf{B}\mathbf{W}^{1/2}\overline{\boldsymbol{v}}$$

and

$$\widetilde{\boldsymbol{\psi}} = \widetilde{\mathbf{SC}}^{\dagger}\widetilde{\mathcal{H}}^{\top}\mathbf{B}\mathbf{W}^{1/2}\overline{\boldsymbol{v}}.$$

(Note that $\widetilde{\boldsymbol{\psi}}$ is defined differently from Theorem 2.6.7.)

Thus, for $\widetilde{\mathbf{SC}}_{\mathcal{H}} = D^{(\mathrm{sc})}.\textsc{InitialSC}()$, $\boldsymbol{p}$ as defined in $\textsc{Locate}()$ of Algorithm 6 satisfies

$$\begin{aligned}
\boldsymbol{p} &= \mathbf{Q}\left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \boldsymbol{\Gamma}\begin{bmatrix} 0 \\ D^{(\mathrm{sc})}.\textsc{InitialSC}() \end{bmatrix}\widetilde{\boldsymbol{\phi}} + \boldsymbol{\Gamma}\begin{bmatrix} 0 \\ D^{(\mathrm{sc})}.\textsc{InitialSC}() \end{bmatrix}\widetilde{\boldsymbol{\psi}} + \boldsymbol{\Gamma}\boldsymbol{y} \\
&= \mathbf{Q}\left(\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}\right)\mathbf{B}\widetilde{\boldsymbol{\phi}} + \begin{bmatrix} 0 \\ \mathbf{Q}\mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\widetilde{\mathbf{SC}}_{\mathcal{H}} \end{bmatrix}\widetilde{\boldsymbol{\phi}} + \begin{bmatrix} 0 \\ \mathbf{Q}\mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\widetilde{\mathbf{SC}}_{\mathcal{H}} \end{bmatrix}\widetilde{\boldsymbol{\psi}} \\
&\quad + \mathbf{Q}\mathbf{W}^{1/2}\mathbf{B}\mathbf{L}(\boldsymbol{w})^{\dagger}\mathbf{B}^{\top}\mathbf{W}^{1/2}\overline{\boldsymbol{v}} \\
&= \mathbf{Q}\widetilde{\boldsymbol{\Pi}}\overline{\boldsymbol{v}}
\end{aligned}$$

for $\widetilde{\boldsymbol{\Pi}}\overline{\boldsymbol{v}}$ as defined in Lemma 2.6.5. Note that $\widetilde{\boldsymbol{\phi}}$ is padded with zeroes for computing $\mathbf{B}\widetilde{\boldsymbol{\phi}}$ as in Algorithm 6. Because

$$\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} \leq \epsilon\|\overline{\boldsymbol{v}}\|_2 \tag{2.16}$$

and

$$\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)} \leq \epsilon\|\overline{\boldsymbol{v}}\|_2, \tag{2.17}$$

by the guarantee of Lemma 2.6.6 we have that

$$\begin{aligned}\left\|\widetilde{\mathbf{\Pi}}\overline{\boldsymbol{v}}\right\|_2 &\le \|\mathbf{\Pi}\overline{\boldsymbol{v}}\|_2 + \left\|\widetilde{\mathbf{\Pi}}\overline{\boldsymbol{v}} - \mathbf{\Pi}\overline{\boldsymbol{v}}\right\|_2 \\ &\le \|\overline{\boldsymbol{v}}\|_2 + 2\epsilon\|\overline{\boldsymbol{v}}\|_2 + (1+\epsilon)\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} + (1+\epsilon)\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)} \\ &\le 2\|\overline{\boldsymbol{v}}\|_2.\end{aligned}$$

By Lemma 2.6.9, the set $S \leftarrow$ Recover($\boldsymbol{p}$) contains all e such that $\left|\left[\widetilde{\mathbf{\Pi}}\overline{\boldsymbol{v}}\right]_e\right|$ is at least

$$\epsilon\left\|\widetilde{\mathbf{\Pi}}\overline{\boldsymbol{v}}\right\|_2 \le 2\epsilon\|\overline{\boldsymbol{v}}\|_2.$$

Finally, if e satisfies $|[\mathbf{\Pi}\overline{\boldsymbol{v}}]_e| \ge 10\epsilon\|\overline{\boldsymbol{v}}\|_2$ then

$$\begin{aligned}\left|\left[\widetilde{\mathbf{\Pi}}\overline{\boldsymbol{v}}\right]_e\right| &\ge |[\mathbf{\Pi}\overline{\boldsymbol{v}}]_e| - \left\|\widetilde{\mathbf{\Pi}}\overline{\boldsymbol{v}} - \mathbf{\Pi}\overline{\boldsymbol{v}}\right\|_2 \\ &\ge 10\epsilon\|\overline{\boldsymbol{v}}\|_2 - 2\epsilon\|\overline{\boldsymbol{v}}\|_2 - (1+\epsilon)\left\|\boldsymbol{\phi} - \widetilde{\boldsymbol{\phi}}\right\|_{\mathbf{SC}(\mathbf{L}(\overline{\boldsymbol{w}}),C)} - (1+\epsilon)\left\|\boldsymbol{\psi} - \widetilde{\boldsymbol{\psi}}\right\|_{\mathbf{SC}(\mathbf{L}(\boldsymbol{w}),C)} \ge 2\epsilon\|\overline{\boldsymbol{v}}\|_2\end{aligned}$$

where the final step follows from Lemma 2.6.5 with (2.16). Thus $e \in S$ as desired.

Now we bound the runtimes. The runtimes of UpdateV and UpdateW follow directly from Theorem 2.4.10 and Lemma 2.6.6. The cost of Initialize is the cost of Initialize in Theorem 2.4.10 and Lemma 2.6.6 plus the cost of computing $\mathbf{\Gamma}$. This involves solving N Laplacian systems, which costs $\widetilde{O}(Nm) = \widetilde{O}(m\epsilon^{-2})$ time. This is dominated by $\widetilde{O}(m\beta^{-2}\epsilon^{-2})$. Finally, the cost of Locate() is $\widetilde{O}(\beta m\epsilon^{-2})$ for computing $\widetilde{\boldsymbol{\phi}}, \widetilde{\boldsymbol{\psi}}$ by Lemma 2.6.6, and the cost of computing $\boldsymbol{p}$ in line 31 of Algorithm 6. The first term in line 31 can be computed in time $O(N\beta m) = O(\beta m\epsilon^{-2})$ as $\overline{\mathbf{W}}^{1/2} - \mathbf{W}^{1/2}$ is supported on $O(\beta m)$ entries and $\mathbf{Q}$ has N rows. The second and third terms in line 31 can be computed by first multiplying $D^{(\mathrm{sc})}$.InitialSC() times $\widetilde{\boldsymbol{\phi}}$ (or $\widetilde{\boldsymbol{\psi}}$) in time $\widetilde{O}(\beta m\epsilon^{-2})$, as $D^{(\mathrm{sc})}$.InitialSC() has $\widetilde{O}(\beta m\epsilon^{-2})$ edges, and then multiplying by $\mathbf{\Gamma}$ which is a N-by-$O(\beta m)$ size matrix in time $O(N\beta m) = \widetilde{O}(\beta m\epsilon^{-2})$ time. Thus the total runtime of Locate() is $\widetilde{O}(\beta m\epsilon^{-2})$ as desired. □

2.7 Reducing Adaptive to Oblivious Adversaries

In this section we show a blackbox reduction that is able to transform any dynamic algorithm that maintains some sequence of vectors $(\boldsymbol{v}^t)_{t\geq 1}$ against oblivious adversaries to one that can maintain the vectors against adaptive adversaries. We formalize the requirements of the dynamic algorithm via Definition 2.7.1. Roughly, Definition 2.7.1 states that the dynamic algorithm must support two operations: (i) find the entries of the current vector $\boldsymbol{v}^t$ with large absolute value, and (ii) query some set of the entries approximately.

Definition 2.7.1. We call a dynamic algorithm an *ϵ-approximate (L, S)-locator* for an online[5] sequence of vectors $(\boldsymbol{v}^t)_{t\geq 1}$, if in each iteration $t \geq 1$ the dynamic algorithm returns a set $I \subset [n]$ of size at most S containing all i with $|\boldsymbol{v}^t_i| > \epsilon$ in L time.

We call a dynamic algorithm an *ϵ-approximate C-evaluator*, if it supports a query operation that, given some $I \subset [n]$, returns all $\overline{\boldsymbol{v}}_i$ for $i \in I$ in $C(|I|)$ time for some $\overline{\boldsymbol{v}} \in \mathbb{R}^n$ with $\|\overline{\boldsymbol{v}} - \boldsymbol{v}^t\|_2 \leq \epsilon$.

We show that, given a locator (a dynamic algorithm that can tell us the large entries), and locators (dynamic algorithms that tells us the entries of the vectors) with different accuracies, we can combine these dynamic algorithms to work against an adaptive adversary. The more accurate locators will be used less frequently, resulting in an expected time complexity faster than the most accurate locator.

Theorem 2.7.2. *Assume we have ϵ-accurate (L, S)-locator and $(\epsilon/2^i)$-accurate C_i-evaluators for $i = 0, ..., K$ for an online sequence of n-dimensional vectors $(\boldsymbol{v}^t)_{t\geq 1}$. Both dynamic algorithms hold against an* oblivious *adversary. Also assume there is an $\epsilon/2^K$-accurate T-evaluator against an* adaptive *adversary.*

Then there exists a dynamic algorithm against an adaptive *adversary that in each iteration returns whp. some $\overline{\boldsymbol{v}}^t$ with $\|\overline{\boldsymbol{v}}^t - \boldsymbol{v}^t\|_\infty \leq O(\epsilon \log^2 n)$. Each iteration takes expected time*

$$O\left(SK + \frac{T(S)}{2^K} + L + \sum_{i=0}^{K} \frac{C_i(S)}{2^i}\right).$$

Note that the T-evaluator against an adaptive adversary could just be a method to compute

[5]The sequence is may depend on outputs of the data structures.

the exact solution statically. Alternatively, one could run several dynamic algorithms against an oblivious adversary in parallel, but use each data structure only once to answer a query.

In the overview of Section 2.2.3 we outlined how Theorem 2.7.2 is obtained. We here give a quick recap. Let w be the result of the T-evaluator and $\boldsymbol{w}'$ be the result of one of the other evaluators against an oblivious adversary. We want to construct an output $\overline{\boldsymbol{w}}$ whose distribution is similar to $\mathcal{N}(\boldsymbol{w}, \sigma^2)$ for some variance $\sigma = O(\epsilon \log n)$. Note that w.h.p $\|\overline{\boldsymbol{w}} - \boldsymbol{v}^t\|_\infty \leq O(\epsilon \log^2 n)$ because of the random Gaussian noise we added. We wish to improve upon the naive time of explicitly computing $\overline{\boldsymbol{w}}$ by directly adding Gaussian noise to $\boldsymbol{w}$. We achieve this by performing this sampling in a different way which guarantees that we compute $\boldsymbol{w}$ explicitly only with some small probability.

Let d be the density function of $\mathcal{N}(\boldsymbol{w}, \sigma^2)$ and d' be the density function of $\mathcal{N}(\boldsymbol{w}', \sigma^2)$. Then there is some small $\alpha > 0$ and very unlikely event D with $d'(\boldsymbol{x}) \leq \exp(\alpha) d(\boldsymbol{x})$ for all $\boldsymbol{x} \notin D$. For example, Figure 2.2 shows density function $d(\boldsymbol{x})$ and the scaled density function $\exp(-\alpha) \cdot d'(\boldsymbol{x})$ for the 1-dimensional case. If one were to pick uniformly at random a point below the top curve in Figure 2.2 and return its $\boldsymbol{x}$-coordinate, then this corresponds to sampling from $\mathcal{N}(\boldsymbol{w}, \sigma^2)$. The same distribution can be obtained by first flipping an unbalanced coin, and with probability $\exp(-\alpha)$ we sample from the area below the bottom curve $\exp(-\alpha)d'(\boldsymbol{x})$ in Figure 2.2 (i.e. sample according to $\mathcal{N}(\boldsymbol{w}', \sigma^2)$). Otherwise, with probability $1 - \exp(-\alpha)$, we sample from the area between the two curves. This way we are able to sample from $\mathcal{N}(\boldsymbol{w}, \sigma^2)$ more efficiently because only with probability $1 - \exp(-\alpha)$ must we compute $\boldsymbol{w}$. As computing $\boldsymbol{w}'$ is faster than computing $\boldsymbol{w}$, the expected time complexity improves.

This scheme is proven formally in Section 2.7.1 for the general case where the vectors are n-dimensional. The scheme can be extended recursively: note that in order to sample from $\mathcal{N}(\boldsymbol{w}', \sigma^2)$, we can use the same scheme again via some $\boldsymbol{w}''$, i.e. when sampling from $\mathcal{N}(\boldsymbol{w}', \sigma^2)$ we can sample from $\mathcal{N}(\boldsymbol{w}'', \sigma^2)$ instead with probability $\exp(-\alpha)$. This is why Theorem 2.7.2 has K many different evaluators with increasing accuracy. The evaluators with higher accuracy are used with smaller probability, thus the expected time complexity improves. This recursive scheme is proven in Section 2.7.2 and we use it in Section 2.7.3 to prove Theorem 2.7.2.

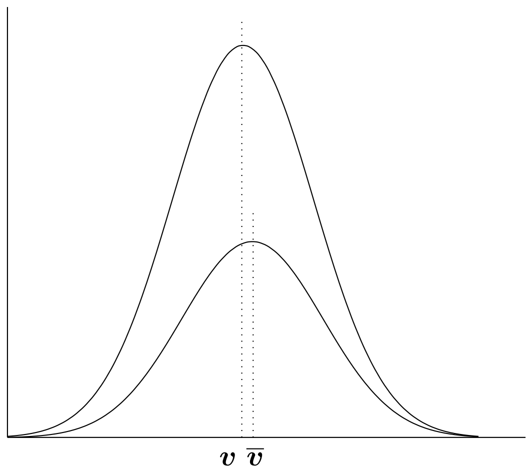

Figure 2.2: Density function d of $\mathcal{N}(\boldsymbol{v}, \sigma^2)$, and density function $\overline{d}$ of $\mathcal{N}(\overline{\boldsymbol{v}}, \sigma^2)$ scaled by some $\exp(-\alpha)$, $\alpha > 0$ so that $\overline{d}(\boldsymbol{x})\exp(-\alpha) \leq d(\boldsymbol{x})$.

Algorithm 7 Basic Simulation Algorithm

```
procedure SIMULATE($\boldsymbol{v} \in \mathbb{R}^n, \boldsymbol{u} \in \mathbb{R}^n, \alpha \geq 0, \sigma > 0$)    ▷ Simulates $\boldsymbol{v} + \boldsymbol{x}$ for $\boldsymbol{x} \sim \mathcal{N}(0, \sigma^2)$.
    with probability $\exp(-\alpha)$ do
        Sample $\boldsymbol{x} \sim \mathcal{N}(0, \sigma^2)$
        return $\boldsymbol{u} + \boldsymbol{x}$
    end
    while true do
        Sample $\boldsymbol{x} \sim \mathcal{N}(0, \sigma^2)$ conditioned on $\frac{|\|\boldsymbol{x}\|^2 - \|\boldsymbol{x} - \boldsymbol{u} + \boldsymbol{v}\|^2|}{2\sigma^2} \leq \alpha$

        with probability $1 - \exp\left(\frac{\|\boldsymbol{x}\|^2 - \|\boldsymbol{x} - \boldsymbol{u} + \boldsymbol{v}\|^2}{2\sigma^2} - \alpha\right)$ do
            return $\boldsymbol{v} + \boldsymbol{x}$
        end
    end while
end procedure
```

2.7.1 Simulating Gaussian Error

Here we prove the algorithm outlined in the previous subsection. We want to construct a variable with distribution $\mathcal{N}(\boldsymbol{v}, \sigma^2)$. This is done by flipping a biased coin: with probability $\exp(-\alpha)$ we return a vector according to $\mathcal{N}(\boldsymbol{u}, \sigma^2)$. Alternatively, with probability $1 - \exp(-\alpha)$ we must return a random vector whose distribution we pick in such a way, that the result of our algorithm has distribution $\mathcal{N}(\boldsymbol{v}, \sigma^2)$. The exact algorithm is given in Algorithm 7 and Lemma 2.7.3 stated the guarantees of that algorithm.

Lemma 2.7.3. *Let $\boldsymbol{z}$ be the result of a call to* Simulate$(\boldsymbol{v}, \boldsymbol{u}, \alpha, \sigma)$ *(Algorithm 7) with $\sigma \geq 2\ln(1.25/\delta)\epsilon/\alpha$ for any $\delta > 0$ and $\epsilon \geq \|\boldsymbol{v} - \boldsymbol{u}\|_2$. Then the distribution of $\boldsymbol{z}$ has total variation distance at most δ compared to $\mathcal{N}(\boldsymbol{v}, \sigma^2)$. Further, the expected time complexity is bounded by $O(n)$.*

To prove Lemma 2.7.3, we first consider the distribution of the result returned by Line 6 to Line 10 of Algorithm 7.

Lemma 2.7.4. *Consider executing Line 6 to Line 12 of Algorithm 7 and let $\boldsymbol{z}$ be the returned vector, i.e. $\boldsymbol{z}$ is the output of Algorithm 7 conditioned on being returned in Line 10. Then the distribution of $\boldsymbol{z}$ under this condition has density function*

$$d(\boldsymbol{z}) = \frac{\exp(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}) - \exp(\frac{-\|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2} - \alpha)}{\sqrt{2\pi}\sigma\mathbb{P}\left[|\|\boldsymbol{x}\|^2 - \|\boldsymbol{x} - \boldsymbol{u} + \boldsymbol{v}\|^2| \leq 2\sigma^2\alpha\right](1 - \exp(-\alpha))}$$

Proof. Up to normalization, the density function of the distribution of $\boldsymbol{z}$ is

$$\begin{aligned} &\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}\right)\left(1 - \exp\left(\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2 - \|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2} - \alpha\right)\right) \\ =\ &\frac{1}{\sqrt{2\pi}\sigma}\left(\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}\right) - \exp\left(\frac{-\|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2} - \alpha\right)\right) \end{aligned}$$

We now compute the normalization factor. For that note the set of possibly returned vectors is $S = \{\boldsymbol{z} \mid |\|\boldsymbol{z}-\boldsymbol{v}\|_2 - \|\boldsymbol{z}-\boldsymbol{u}\|_2| \leq 2\sigma^2\alpha\}$ and

$$\begin{aligned} \int_{\boldsymbol{z}\in S}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2}\right)\mathrm{d}\boldsymbol{z} &= \int_{\boldsymbol{z}\in S}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}\right)\mathrm{d}\boldsymbol{z} \\ &= \mathbb{P}[|\|\boldsymbol{x}\|^2 - \|\boldsymbol{x}-\boldsymbol{u}+\boldsymbol{v}\|^2| \leq 2\sigma^2\alpha] \end{aligned}$$

for some $\boldsymbol{x} \sim \mathcal{N}(0, \sigma^2)$. So the normalization factor is

$$\mathbb{P}\left[|\|\boldsymbol{x}\|^2 - \|\boldsymbol{x}-\boldsymbol{u}+\boldsymbol{v}\|^2| \leq 2\sigma^2\alpha\right](1 - \exp(-\alpha))$$

and the density function is

$$\frac{\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}\right)-\exp\left(\frac{-\|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2}-\alpha\right)}{\sqrt{2\pi}\sigma\mathbb{P}\left[\|\boldsymbol{x}\|^2,\|\boldsymbol{x}-\boldsymbol{u}+\boldsymbol{v}\|^2\leq 2\sigma^2\alpha\right](1-\exp(-\alpha))}$$

□

Our algorithm relies on the fact that the density function of $\mathcal{N}(\boldsymbol{u},\sigma^2)$ is smaller than the density function of $\mathcal{N}(\boldsymbol{v},\sigma^2)$ when scaled by $\exp(\alpha)$. This is generally not true, unless we restrict the two density function on to some event $E\subset\mathbb{R}^n$. Using the following result from differential privacy, we show that this event occurs with high probability, if the variance σ^2 of the added noise and the scaling-parameter α are sufficiently large.

Lemma 2.7.5 ([85, Appendix A])**.** *Let $\boldsymbol{u},\boldsymbol{v}\in\mathbb{R}^n$, $\epsilon\geq\|\boldsymbol{u}-\boldsymbol{v}\|$, $c^2>2\ln(1.25/\delta)$, $\sigma\geq c\epsilon/\alpha$ and $\boldsymbol{x}\sim\mathcal{N}(0,\sigma^2)$. Then $\mathbb{P}[|\|\boldsymbol{x}\|^2-\|\boldsymbol{x}-\boldsymbol{u}+\boldsymbol{v}\|^2|>2\alpha\sigma^2]\leq\delta$.*

We now have all tools available to prove Lemma 2.7.3.

Proof of Lemma 2.7.3. The density function of z conditioned on $|\|\boldsymbol{z}-\boldsymbol{v}\|^2-\|\boldsymbol{z}-\boldsymbol{u}\|^2|\leq 2\sigma^2\alpha$ is

$$\begin{aligned}&\exp(-\alpha)\cdot\frac{\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2}\right)}{\sqrt{2\pi}\sigma\mathbb{P}\left[\left|\|\boldsymbol{x}\|^2-\|\boldsymbol{x}-(\boldsymbol{u}-\boldsymbol{v})\|^2\right|\leq 2\sigma^2\alpha\right]}\\&+(1-\exp(-\alpha))\cdot\frac{\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}\right)-\exp\left(\frac{-\|\boldsymbol{z}-\boldsymbol{u}\|^2}{2\sigma^2}-\alpha\right)}{\sqrt{2\pi}\sigma\mathbb{P}\left[\left|\|\boldsymbol{x}\|^2-\|\boldsymbol{x}-(\boldsymbol{u}-\boldsymbol{v})\|^2\right|\leq 2\sigma^2\alpha\right](1-\exp(-\alpha))}\\=&\frac{\exp\left(-\frac{\|\boldsymbol{z}-\boldsymbol{v}\|^2}{2\sigma^2}\right)}{\sqrt{2\pi}\sigma\mathbb{P}\left[\left|\|\boldsymbol{x}\|^2-\|\boldsymbol{x}-(\boldsymbol{u}-\boldsymbol{v})\|^2\right|\leq 2\sigma^2\alpha\right]}\end{aligned}$$

which is also the density function of $\boldsymbol{x}\sim\mathcal{N}(0,\sigma^2)$ conditioned on $|\|\boldsymbol{x}\|^2-\|\boldsymbol{x}-(\boldsymbol{u}-\boldsymbol{v})\|^2|\leq 2\sigma^2\alpha$. Thus the total variation distance is bounded by δ via Lemma 2.7.5.

For the time complexity, note that the probability of returning the vector during an iteration of Line 6 is $1-\exp(-\alpha)$. Consequently, if we reach Line 6, it is invoked $(1-\exp(-\alpha))^{-1}$ times in expectation. The probability of reaching Line 6 is $1-\exp(-\alpha)$, so Line 6 is invoked 1 time in

Algorithm 8 Recursive Simulation Algorithm

```
procedure Simulate($\boldsymbol{v}_1, ..., \boldsymbol{v}_k \in \mathbb{R}^n, \alpha \geq 0, \sigma > 0$)    ▷ If $k = 2$ we call Algorithm 7 instead.
    with probability $\exp(-\alpha)$ do
        return Simulate($\boldsymbol{v}_2, ..., \boldsymbol{v}_k, 2\alpha, \sigma$)    ▷ Call Alg. 7 if $k = 2$.
    end
    while true do
        Sample $\boldsymbol{x} \sim \mathcal{N}(0, \sigma^2)$ conditioned on $\frac{|\|\boldsymbol{x}\|^2 - \|\boldsymbol{x} - \boldsymbol{v}_2 + \boldsymbol{v}_1\|^2|}{2\sigma^2} \leq \alpha$
        with probability $1 - \exp(\frac{\|\boldsymbol{x}\|^2 - \|\boldsymbol{x} - \boldsymbol{v}_2 + v_1\|^2}{2\sigma^2} - \alpha)$ do
            return $\boldsymbol{v}_1 + \boldsymbol{x}$
        end
    end while
end procedure
```

expectation. As each iteration needs $O(n)$ time to process the n-dimensional vectors, the expected runtime of the procedure is $O(n)$. □

2.7.2 Recursive Simulation

In this subsection we provide and analyze a recursive variant of Algorithm 7. This variant replaces Line 3 of Algorithm 7, which samples some $\mathcal{N}(\boldsymbol{u}, \sigma^2)$, by a recursive invocation of Algorithm 7. We first analyze the distribution of the returned vector in Lemma 2.7.6 and then bound the expected time complexity in Lemma 2.7.7.

Lemma 2.7.6. *Consider a call to Algorithm 8 with inputs $\boldsymbol{v}_1, ..., \boldsymbol{v}_k \in \mathbb{R}^n$, $k \geq 2$, $\epsilon \geq \|\boldsymbol{v}_i - \boldsymbol{v}_{i+1}\|_2/2^{i-1}$ for all $i = 1, ..., k-1$, $\sigma \geq 2\ln(1.25/\delta)\epsilon/\alpha$, Then the returned value has total variation distance at most $(k-1)\delta$ compared to $\mathcal{N}(\boldsymbol{v}_1, \sigma^2)$.*

Proof. We prove this by induction over k, the number of vectors.

Base Case For $k = 2$ we call Algorithm 7 instead, so the claim is true by Lemma 2.7.3.

Induction Now assume Lemma 2.7.6 holds for some $k-1$ and consider a call to Simulate with vectors $\boldsymbol{v}_1, ...\boldsymbol{v}_k$. Let $\boldsymbol{v}'_1, ..., \boldsymbol{v}'_{k-1}$, $\alpha' = 2\alpha$ be the parameters of the recursive call in Line 3 and let

$\epsilon' = 2\epsilon$. Then

$$\epsilon' = \epsilon/2 \geq \|\boldsymbol{v}_i - \boldsymbol{v}_{i+1}\|_2/2^i = \|\boldsymbol{v}_{i+1} - \boldsymbol{v}_i\|_2/2^{i-1} = \|\boldsymbol{v}'_i - \boldsymbol{v}'_{i+1}\|_2/2^{i-1} \text{ and}$$
$$\sigma \geq 2\ln(1.25/\delta)\epsilon/\alpha = 2\ln(1.25/\delta)\epsilon'/\alpha'$$

so the conditions to apply the induction hypothesis are satisfied. Thus, the vector returned in Line 3 has the same distribution as $\mathcal{N}(v'_1, \sigma^2) = \mathcal{N}(\boldsymbol{v}_2, \sigma^2)$ up to total variation distance $(k-2)\delta$.

Note that Algorithm 8 is the same as Algorithm 7, except for Line 3, so by the same proof as in Lemma 2.7.3 we return a vector that is distributed like $\mathcal{N}(\boldsymbol{v}_1, \sigma^2)$ up to total variation distance $(k-2)\delta + \delta = (k-1)\delta$. □

For computational efficiency, note that Algorithm 8 does not need to read vector $\boldsymbol{v}_1$ when performing the branch of Line 3. The following lemma bounds the probability of accessing any $\boldsymbol{v}_i$ for $i < k$.

Lemma 2.7.7. *Consider a call to Algorithm 8 with inputs $\boldsymbol{v}_1, ..., \boldsymbol{v}_k \in \mathbb{R}^n, \alpha \geq 0$. The probability that vector $\boldsymbol{v}_i$ is accessed is at most $2^i\alpha$ for all $i < k$. Further, the expected time complexity (ignoring the time for accessing any $\boldsymbol{v}_i$) is bounded by $O(kn)$.*

Proof. Vector $\boldsymbol{v}_1$ is accessed with probability $1 - \exp(-\alpha) \leq \alpha \leq \alpha 2^1$. Vector $\boldsymbol{v}_i$ for $i > 1$ is accessed with probability $1 - \exp(-\alpha 2^{i-2})$ when calling $\textsc{Simulate}(\boldsymbol{v}_{i-1}, ..., \boldsymbol{v}_k, \alpha, \sigma)$, or with probability $1 - \exp(-\alpha 2^{i-1})$ when calling $\textsc{Simulate}(\boldsymbol{v}_i, ..., \boldsymbol{v}_k, \alpha, \sigma)$. Thus the overall probability after a call $\textsc{Simulate}(\boldsymbol{v}_1, ..., \boldsymbol{v}_k, \alpha, \sigma)$ is

$$\underbrace{\prod_{t=1}^{i-1} \exp\left(-\alpha 2^{t-1}\right)}_{\substack{\text{Probability of recursing}\\ \text{to } \textsc{Simulate}(\boldsymbol{v}_i, ..., \boldsymbol{v}_k, \alpha, \sigma)}} \left(1 - \exp\left(-\alpha 2^{i-1}\right)\right) + \underbrace{\prod_{t=1}^{i-2} \exp\left(-\alpha 2^{t-1}\right)}_{\substack{\text{Probability of recursing}\\ \text{to } \textsc{Simulate}(\boldsymbol{v}_{i-1}, ..., \boldsymbol{v}_k, \alpha, \sigma)}} \left(1 - \exp\left(-\alpha 2^{i-2}\right)\right)$$

$$\begin{aligned}
&= \exp\left(-\alpha \sum_{t=0}^{i-2} 2^t\right)\left(1 - \exp\left(-\alpha 2^{i-1}\right)\right) + \exp\left(-\alpha \sum_{t=0}^{i-3} 2^t\right)\left(1 - \exp\left(-\alpha 2^{i-2}\right)\right)\\
&= \exp\left(-\alpha(2^{i-1}-1)\right)\left(1 - \exp\left(-\alpha 2^{i-1}\right)\right) + \exp\left(-\alpha(2^{i-2}-1)\right)\left(1 - \exp\left(-\alpha 2^{i-2}\right)\right)\\
&= \exp\left(-\alpha(2^{i-1}-1)\right) - \exp\left(-\alpha(2^{i}-1)\right) + \exp\left(-\alpha(2^{i-2}-1)\right) - \exp\left(-\alpha(2^{i-1}-1)\right)\\
&= \exp\left(-\alpha(2^{i-2}-1)\right) - \exp\left(-\alpha(2^{i}-1)\right)\\
&= \left(1 - \exp\left(-\alpha(2^{i} - 2^{i-2})\right)\right) \cdot \exp\left(-\alpha(2^{i-2}-1)\right)\\
&\le 1 - \exp\left(-\alpha(2^{i} - 2^{i-2})\right)\\
&\le \alpha(2^i - 2^{i-2}) \le \alpha 2^i
\end{aligned}$$

The time expected time complexity is at most $O(kn)$ because each recursion has expected time $O(n)$ by Lemma 2.7.3. □

2.7.3 Proof of Theorem 2.7.2

We can now prove Theorem 2.7.2 by applying Algorithm 8 to the vectors returned by the evaluator data structures.

Proof of Theorem 2.7.2. Given the locator and evaluators, we construct a new dynamic algorithm $\mathcal{A}$ against an adaptive adversary. The construction is done in a paragraph further below. For now, we claim that the output of the new dynamic algorithm $\mathcal{A}$ has the following distribution.

Let $\boldsymbol{w}^1$ be the output of the $\epsilon/2^K$-accurate oracle against an adaptive adversary. Then sample $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{w}^1, (c_1\epsilon \log n)^2)$ for some sufficiently large constant c_1. At last, set all entries of $\boldsymbol{x}$ with absolute value smaller than $c_2\epsilon\log^2 n$ to 0. Call the resulting vector $\boldsymbol{u}$. We claim the dynamic algorithm $\mathcal{A}$ will have the distribution of this vector $\boldsymbol{z}$.

Vector $\boldsymbol{z}$ satisfies w.h.p. $\|\boldsymbol{z} - \boldsymbol{v}^t\|_\infty = O(\epsilon \log^2 n)$, so returning $\boldsymbol{z}$ would satisfy the promised approximation guarantees of Theorem 2.7.2 and the algorithm would work against an adaptive adversary because the output does not depend on any of the oracles that use the oblivious adversary assumption.

We now describe the new dynamic algorithm $\mathcal{A}$ and how it constructs this vector $\boldsymbol{z}$ more efficiently than the procedure described above.

Algorithm Let I be the set returned by the ϵ-accurate locator. For $i > 1$ let $\boldsymbol{w}_I^i$ be result of the $\epsilon/2^{K-i+2}$-accurate oracles against oblivious adversaries when querying only entries from I. Let $\boldsymbol{x}'_I = \textsc{Simulate}(\boldsymbol{w}_S^1, ..., \boldsymbol{w}_S^{k+1}, 2^{-K}, c_1 \epsilon \log n)$ (Algorithm 8). Then set all entries of $\boldsymbol{x}$ with absolute value smaller than $c_2 \epsilon \log^2 n$ to 0 and let $\boldsymbol{z}'$ be the resulting vector. Here $c_2 > c_1$ is picked such that w.h.p. $|\boldsymbol{x}_i - \boldsymbol{w}_i^1| < c_2/2 \cdot \epsilon \log^2 n$. Our algorithm returns this vector $\boldsymbol{z}'$.

Correctness We claim $\boldsymbol{z}'$ has the same distribution as $\boldsymbol{z}$ up to total variation distance $1/\text{poly}(n)$. For $\epsilon' = 2\epsilon/2^K$ we have $\|\boldsymbol{w}^i - \boldsymbol{w}^{i+1}\|_2 \le \epsilon/2^{K-i} \le \epsilon' 2^{i-1}$. So x'_I has distribution $\mathcal{N}(\boldsymbol{w}_I^1, (c_1 \sigma \log n)^2)$ up to total variation distance $1/\text{poly}(n)$ by Lemma 2.7.6 for some large enough constant c_1. Thus if I only contained indices i where w.h.p. $\boldsymbol{z}_i$ would be 0 anyway, then $\boldsymbol{z}'$ has same distribution as z up to total variation distance $1/\text{poly}(n)$.

Note that by $\|\boldsymbol{w}^1 - \boldsymbol{v}^t\|_2 < \epsilon$ we have that I (which by definition contains all indices with $|\boldsymbol{v}_i^t| > \epsilon$) also contains all indices i with $|\boldsymbol{w}_i^1| > 2\epsilon$. Further, w.h.p. we have $\|\boldsymbol{w}^1 - \boldsymbol{x}\|_\infty < c_2/2\epsilon \log^2 n$ by choice of $c_2 > c_1$. So $i \in I$ this would imply $|\boldsymbol{x}_i| \le |\boldsymbol{w}_i^1| + |\boldsymbol{w}_i^1 - \boldsymbol{x}_i| \le 2\epsilon + c_2/2\epsilon \log^2 n < c_2 \epsilon \log^2 n$ so w.h.p. $\boldsymbol{z}_i$ will be set to 0. Thus the total variation distance of $\boldsymbol{z}$ and $\boldsymbol{z}'$ is at most some $1/\text{poly}(n)$.

Complexity By Lemma 2.7.7, we use each $\boldsymbol{w}_i$ with probability at most $2^i/2^K = 2^{i-K}$ for $i \le K$ and running $\textsc{Simulate}$ on $K+1$ many $|I|$-dimensional vectors needs $O(|I|K) = O(SK)$ time. We can delay the query to $\boldsymbol{w}^i$ until the vectors actually need to be used. As $\boldsymbol{w}_i$ is obtained from evaluator with complexity C_{K-i+2} for $i > 1$, we obtain time complexity

$$O(SK + L + C_0(S) + T(S)/2^K + \sum_{i=1}^{K} \frac{C_i(S)}{2^i}).$$

$\square$

2.8 Interior Point Method

In this section we provide the machinery we use to reduce minimum cost flow to dynamic graph data structure problems. First, in Section 2.8.1 we provide the general IPM framework for linear programming from [84] that we use. Then, in Section 2.8.2 we introduce the data structures, subroutines, and bounds that we develop in this paper to implement this framework efficiently and in Section 2.8.3 we combine these pieces to give the efficient IPM. The proofs for the tools we introduce are provided in Section 2.8.4, Section 2.8.5, and Section 2.9 (for the runtime bound for the graph solution maintainer (Definition 2.8.3) in Theorem 2).

2.8.1 Robust IPM Framework

Here we provide the the linear programming setup that we use to model minimum cost flow and the IPM framework provided by [84] for solving them. In particular, throughout the section, we consider the general linear programming problem. Given $\mathbf{B} \in \mathbb{R}^{m\times n}$, $\boldsymbol{c}, \boldsymbol{\ell}, \boldsymbol{u} \in \mathbb{R}^m$, and $\boldsymbol{d} \in \mathbb{R}^n$ where $\boldsymbol{\ell} < \boldsymbol{u}$ entrywise, we wish to solve.

$$\min_{\boldsymbol{x}\in\mathbb{R}^m \,|\, \mathbf{B}^\top \boldsymbol{x}=\boldsymbol{d} \text{ and } \boldsymbol{\ell}\leq\boldsymbol{x}\leq\boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{x}\,. \tag{2.18}$$

In the special case where $\mathbf{B}$ is the incidence matrix of graph and $\boldsymbol{\ell} = \mathbf{0}$, this problem directly corresponds to the minimum cost flow problem. Many of the reductions we provide in this section apply to this general linear program and we will explicitly state in which cases we instead assume that $\mathbf{B}$ is the incidence matrix of graph.

To solve (2.18) we leverage the general robust IPM framework of [84]. This method crudely follows a central path by maintaining centered points defined as follows.

Definition 2.8.1 (Centered Point). For $\mathcal{X} \overset{\text{def}}{=} \{\boldsymbol{x} \in \mathbb{R}^m \,|\, \boldsymbol{x}_i \in (\boldsymbol{\ell}_i, \boldsymbol{u}_i)\}$ we say $(\boldsymbol{x}, \boldsymbol{s}) \in \mathbb{R}^m \times \mathbb{R}^m$ is *μ-feasible* for $\mu > 0$ if $\boldsymbol{x} \in \mathcal{X}$, $\mathbf{B}^\top \boldsymbol{x} = \boldsymbol{d}$, $\mathbf{B}\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c}/\mu$ for some $\boldsymbol{y} \in \mathbb{R}^n$. We say $(\boldsymbol{x}, \boldsymbol{s})$ is μ-

centered[6] if $(\boldsymbol{x}, \boldsymbol{s})$ is μ-feasible and $\left\|\nabla^2\phi(\boldsymbol{x})^{-1/2}(\boldsymbol{s} + \nabla\phi(\boldsymbol{x}))\right\|_\infty \leq \frac{1}{64}$ where $\phi(\boldsymbol{x}) \overset{\text{def}}{=} \sum_{i\in[m]} \phi_i(\boldsymbol{x})$ with $\phi_i(\boldsymbol{x}) \overset{\text{def}}{=} -\log(\boldsymbol{u}_i - \boldsymbol{x}_i) - \log(\boldsymbol{x}_i - \boldsymbol{\ell}_i)$ for $\boldsymbol{x} \in \mathcal{X}$.

This definition is motivated for the fact that, *μ-central path point*, defined as

$$\boldsymbol{x}_\mu \overset{\text{def}}{=} \underset{\boldsymbol{x}\in\mathcal{X}\,|\,\mathbf{B}^\top\boldsymbol{x}=\boldsymbol{d}}{\arg\min} \quad \mu \cdot \boldsymbol{c}^\top\boldsymbol{x} + \phi(\boldsymbol{x})$$

is the unique μ-centered point with $\|\nabla^2\phi(\boldsymbol{x})^{-1/2}(\boldsymbol{s} + \nabla\phi(\boldsymbol{x}))\|_\infty = 0$. To see this, note that ϕ is convex on $\mathcal{X}$ and the optimality conditions for $\boldsymbol{x}_\mu$ are that

$$\boldsymbol{x} \in \mathcal{X}\ ,\ \mathbf{B}^\top\boldsymbol{x}_\mu = \boldsymbol{d} \text{ and } \boldsymbol{c} + \mu\nabla\phi(\boldsymbol{x}_\mu) \perp \text{Kernel}(()\mathbf{B}^\top)\,.$$

However, $\boldsymbol{c} + \mu\nabla\phi(\boldsymbol{x}_t) \perp \text{Kernel}(()\mathbf{B}^\top)$ if and only if $\boldsymbol{c} + \mu\nabla\phi(\boldsymbol{x}_\mu) \in \text{im}(\mathbf{B})$ which we can write equivalently as $\boldsymbol{c} + \mu\nabla\phi(\boldsymbol{x}_\mu) = \mu\mathbf{B}\boldsymbol{y}_\mu$ for some $\boldsymbol{y}_\mu$. Finally, the condition $\boldsymbol{c} + \mu\nabla\phi(\boldsymbol{x}_\mu) = \mu\mathbf{B}\boldsymbol{y}_\mu$ is equivalent to $\mathbf{B}\boldsymbol{y}_\mu + \boldsymbol{s}_\mu = \boldsymbol{c}/\mu$ for $\boldsymbol{s}_\mu = -\nabla\phi(\boldsymbol{x}_\mu)$, i.e. $\|\nabla^2\phi(\boldsymbol{x}_\mu)^{-1/2}(\boldsymbol{s}_\mu + \nabla\phi(\boldsymbol{x}_\mu))\|_\infty = 0$ as $\nabla^2\phi(\boldsymbol{x}_\mu)$ is positive definite. Consequently, a μ-centered point is a point which maintains an approximate notion of the optimality of $\boldsymbol{x}_\mu$.

The IPM framework works by maintaining μ-centered points by controlling centrality measures as potentials. The definition of these quantities (Definition 2.8.2), the framework (Algorithm 9), and the result from [84] that we use about this framework (Theorem 1) are all given below.

Definition 2.8.2 (Centrality). For μ-feasible $(\boldsymbol{x}, \boldsymbol{s})$ we define *centrality measure* $\gamma(\boldsymbol{x}, \boldsymbol{s}) \in \mathbb{R}^m$ where $\gamma_i(\boldsymbol{x}, \boldsymbol{s}) \overset{\text{def}}{=} \phi_i''(\boldsymbol{x})^{-1/2}(\boldsymbol{s}_i + \phi_i'(\boldsymbol{x}))$ and $\phi_i'(\boldsymbol{x}) \overset{\text{def}}{=} [\nabla\phi(\boldsymbol{x})]_i$ and $\phi_i''(\boldsymbol{x}) \overset{\text{def}}{=} [\nabla^2\phi(\boldsymbol{x})]_{ii}$. Further, we define *centrality potential* $\Psi(\boldsymbol{x}, \boldsymbol{s}) \overset{\text{def}}{=} \sum_{i\in[m]} \cosh(\lambda \cdot \gamma_i(\boldsymbol{x}, \boldsymbol{s}))$ where $\lambda \overset{\text{def}}{=} 128\log(16m)$ and $\cosh(z) \overset{\text{def}}{=} \frac{1}{2}[\exp(z) + \exp(-z)]$ for all $z \in \mathbb{R}$.

Theorem 1 (Theorem A.16 in [84])**.** *Using the notation in Algorithm 9, let $(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)})$ be the value of $(\boldsymbol{x}, \boldsymbol{s})$ before the step (Line 10) and let $(\boldsymbol{x}^{(1)}, \boldsymbol{s}^{(1)})$ be the value $(\boldsymbol{x}, \boldsymbol{s})$ after the step. If $(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)})$*

[6]In [84], the condition is $\mathbf{B}\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c}$ and $\left\|\nabla^2\phi(\boldsymbol{x})^{-1/2}(\boldsymbol{s}/\mu + \nabla\phi(\boldsymbol{x}))\right\|_\infty \leq \frac{1}{64}$ instead. We do the replacement from $\boldsymbol{s}/\mu$ to $\boldsymbol{s}$ to simplify the algorithm description and notations in the data structures. Further, the choice of variable names is different in the two papers with variable names chosen here for the application of minimum cost flow.

Algorithm 9 A robust interior point method in [84]

```
procedure CENTERING($\mathbf{B}, \boldsymbol{x}, \boldsymbol{s}, \boldsymbol{\ell}, \boldsymbol{u}, \mu_{\text{start}}, \mu_{\text{end}}$)
        ▷ Invariant: $(\boldsymbol{x}, \boldsymbol{s})$ is $\mu$-centered with $\Psi(\boldsymbol{x}, \boldsymbol{s}) \leq \cosh(\lambda/64)$    ▷ (See Definitions 2.8.1 and 2.8.2)
    Define step size $\alpha \overset{\text{def}}{=} \frac{1}{2^{15}\lambda}$.
    $\overline{\mu} = \mu = \mu_{\text{start}}$, $\overline{\boldsymbol{x}} = \boldsymbol{x}$, $\overline{\boldsymbol{s}} = \boldsymbol{s}$
    while $\mu \geq \mu_{\text{end}}$ do
        Set weight matrix $\mathbf{W} \leftarrow \nabla^2\phi(\overline{\boldsymbol{x}})^{-1}$.
        Set iterate approximation $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}) \in \mathbb{R}^m \times \mathbb{R}^m$ such that
            $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{x}} - \boldsymbol{x})\|_\infty \leq \alpha$ and $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \leq \alpha$.
        Set step direction $\boldsymbol{v} \in \mathbb{R}^m$ where $\boldsymbol{v}_i \leftarrow \sinh(\lambda\gamma_i(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}))$ for all $i \in [m]$ and $\sinh(z) \overset{\text{def}}{=} \frac{1}{2}(\exp(z) - \exp(-z))$.
        Set step size $h \leftarrow -\alpha/\|\cosh(\lambda\gamma(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}))\|_2$.
        Set $\boldsymbol{v}^{\|}$, $\boldsymbol{v}^{\perp}$ such that $\mathbf{W}^{-1/2}\boldsymbol{v}^{\|} \in \text{Im}\mathbf{B}$, $\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}^{\perp} = 0$, and
            $\|\boldsymbol{v}^{\|} - \mathbf{P}_{\mathbf{W}}v\|_2 \leq \alpha\|\boldsymbol{v}\|_2$ and $\|\boldsymbol{v}^{\perp} - (\mathbf{I} - \mathbf{P}_{\mathbf{W}})v\|_2 \leq \alpha\|\boldsymbol{v}\|_2$
        where $\mathbf{P} \overset{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}^{1/2}$
        Set $\boldsymbol{x} \leftarrow \boldsymbol{x} + h\mathbf{W}^{1/2}\boldsymbol{v}^{\perp}$, $\boldsymbol{s} \leftarrow \boldsymbol{s} + h\mathbf{W}^{-1/2}\boldsymbol{v}^{\|}$, $\mu \leftarrow \max\{(1 - \frac{\alpha}{64\sqrt{m}})\mu, \mu_{\text{end}}\}$
        If $|\overline{\mu} - \mu| \geq \alpha\overline{\mu}$, then $\boldsymbol{s} \leftarrow \frac{\mu}{\overline{\mu}}\boldsymbol{s}$, $\overline{\mu} \leftarrow \mu$
    end while
    Return $(\boldsymbol{x}, \boldsymbol{s})$
end procedure
```

is $\overline{\mu}$-feasible and $\Psi(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)}) \leq \cosh(\lambda/64)$, then $(\boldsymbol{x}^{(1)}, \boldsymbol{s}^{(1)})$ is $\overline{\mu}$-feasible and

$$\Psi(\boldsymbol{x}^{(1)}, \boldsymbol{s}^{(1)}) \leq \left(1 - \frac{\alpha\lambda}{8\sqrt{m}}\right)\Psi(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)}) + \alpha\lambda\sqrt{m} \leq \cosh(\lambda/64).$$

Proof. The proof of [84, Theorem A.16] shows $\Psi^{\mu'}(\boldsymbol{x}^{(1)}, \boldsymbol{s}^{(1)}) \leq \left(1 - \frac{\alpha\lambda}{8\sqrt{m}}\right)\Psi^{\mu'}(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)}) + \alpha\lambda\sqrt{m}$ for any $|\mu' - \mu| \leq \alpha\mu$ where $\Psi^{\mu}(\boldsymbol{x}, \boldsymbol{s}) \overset{\text{def}}{=} \left\|\nabla^2\phi(\boldsymbol{x})^{-1/2}(\boldsymbol{s}/\mu + \nabla\phi(\boldsymbol{x}))\right\|_\infty$. We picked $\mu' = \overline{\mu}$ and replaced $\boldsymbol{s}/\overline{\mu}$ by $\boldsymbol{s}$. □

2.8.2 Robust IPM Tools

Here we discuss the key tools we develop in this paper to efficiently implement the robust IPM (Algorithm 9) of [84] discussed in the previous Section 2.8.1.

First, as discussed in Section 2.1.1, a key advance of this paper is efficient procedures for approximately maintaining the iterates of Algorithm 9, i.e. approximating the result of approximate projection steps. We formalize this maintenance problem as a data structure problem defined below.

Definition 2.8.3 (Solution Maintainer). We call a data structure a $(\mathcal{T}_{\text{init}}, \mathcal{T}_{\text{phase}})$-*solution maintainer* if it supports the following operations against an adaptive adversary with high probability:

- $\textsc{Initialize}(\mathbf{B} \in \mathbb{R}^{m\times n}, \boldsymbol{w}^{(0)} \in \mathbb{R}^m_+, \boldsymbol{x}^{(0)} \in \mathbb{R}^m, \boldsymbol{s}^{(0)} \in \mathbb{R}^m, \alpha, C_r, k, C_z)$: Given input constraint matrix $\mathbf{B}$, weight vector $\boldsymbol{w}^{(0)}$, iterate $(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)})$, accuracy parameter α, weight range r, phase length k, sparsity of changes z, initialize the data structure with $\boldsymbol{w} := \boldsymbol{w}^{(0)}$, $\boldsymbol{x} := \boldsymbol{x}^{(0)}$, and $\boldsymbol{s} := \boldsymbol{s}^{(0)}$ in time $O(\mathcal{T}_{\text{init}})$ with $\mathcal{T}_{\text{init}} = \Omega(m)$.

- $\textsc{StartPhase}(\widetilde{\boldsymbol{x}} \in \mathbb{R}^m, \widetilde{\boldsymbol{s}} \in \mathbb{R}^m)$: Given input iterate $(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{s}})$ with $\|\mathbf{W}^{-1/2}(\widetilde{\boldsymbol{x}} - \boldsymbol{x})\|_2 \le 1$ and $\|\mathbf{W}^{1/2}(\widetilde{\boldsymbol{s}} - \boldsymbol{s})\|_2 \le 1$, update $\boldsymbol{x} \leftarrow \widetilde{\boldsymbol{x}}$ and $\boldsymbol{s} \leftarrow \widetilde{\boldsymbol{s}}$ in amortized $O(\mathcal{T}_{\text{phase}})$ time with $\mathcal{T}_{\text{phase}} = \Omega(m)$.

- $\textsc{Move}(\boldsymbol{w}^{(j)} \in \mathbb{R}^m_+, \boldsymbol{v}^{(j)} \in \mathbb{R}^m, h^{(j)} \in \mathbb{R}) \to \mathbb{R}^m \times \mathbb{R}^m$: In the j-th call to $\textsc{Move}$, given input weights $\boldsymbol{w}^{(j)}$, direction $\boldsymbol{v}^{(j)}$, and step size $h^{(j)}$ with $h^{(j)}\|\boldsymbol{v}^{(j)}\|_2 \le 1$, $\textsc{Move}$ updates $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(j)}$,

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + h^{(j)}\mathbf{W}_j^{1/2}(\mathbf{I} - \mathbf{P}_j)\boldsymbol{v}^{(j)}, \text{ and } \boldsymbol{s} \leftarrow \boldsymbol{s} + h^{(j)}\mathbf{W}_j^{-1/2}\mathbf{P}_j\boldsymbol{v}^{(j)},$$

  where $\mathbf{W}_j \overset{\text{def}}{=} \mathbf{diag}(\boldsymbol{w}^{(j)})$ and $\mathbf{P}_j \overset{\text{def}}{=} \mathbf{W}_j^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}_j\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}_j^{1/2}$ and $\textsc{Move}$ outputs $(\overline{\boldsymbol{x}}^{(j)}, \overline{\boldsymbol{s}}^{(j)}) \in \mathbb{R}^m \times \mathbb{R}^m$ with $\|(\mathbf{W}^{(j)})^{-1/2}(\overline{\boldsymbol{x}}^{(j)} - \boldsymbol{x})\|_\infty \le \alpha$, $\|(\mathbf{W}^{(j)})^{1/2}(\overline{\boldsymbol{s}}^{(j)} - \boldsymbol{s})\|_\infty \le \alpha$, and the number of coordinates changed from the previous output bounded by $O(2^{2\ell_{j+1}}\alpha^{-2}\log^3 m + S_j)$ where

$$S_j \overset{\text{def}}{=} \left|\left\{i \in [m] : \boldsymbol{w}_i^{(j)} \ne \boldsymbol{w}_i^{(j-1)}, \overline{\boldsymbol{x}}_i^{(j-1)} = \overline{\boldsymbol{x}}_i^{(j-2)}, \text{ and } \overline{\boldsymbol{s}}_i^{(j-1)} = \overline{\boldsymbol{s}}_i^{(j-2)}\right\}\right| .$$

  The input $\boldsymbol{w}^{(j)}$ and $\boldsymbol{v}^{(j)}$ and output $(\overline{\boldsymbol{x}}^{(j)}, \overline{\boldsymbol{s}}^{(j)})$ are given implicitly as a list of changes to the previous input and output of $\textsc{Move}$.

Furthermore, the above operations need only be supported under the following assumptions:

1. *Phase length*: $\textsc{StartPhase}$ is called at least every k calls to $\textsc{Move}$ and at most twice in a row.

2. *Number of changes*: for all $j \geq 1$ there are at most $\min\{C_z 2^{2\ell_j}, m\}$ coordinates changed in $\boldsymbol{w}^{(j)}$, $\boldsymbol{v}^{(j)}$ from $\boldsymbol{w}^{(j-1)}, \boldsymbol{v}^{(j-1)}$ where ℓ_j is the largest integer with ℓ with $j \equiv 0 \pmod{2^\ell}$.

3. *Magnitude of changes*: for any $|j_2 - j_1| \leq L$, we have $\sqrt{\boldsymbol{w}_i^{(j_2)}/\boldsymbol{w}_i^{(j_1)}} \leq C_r L^2$ for all $i \in [m]$.

Our algorithm actually always has $S_j = 0$, but we state Definition 2.8.3 with possibly nonzero S_j for more generality.

In the particular case of graphs, one of the key results of this paper is the following efficient solution maintenance data structure in the particular case of graphs (shown in Section 2.9.1).

Theorem 2.8.4 (Graph Solution Maintenance)**.** *In the special case that* $\mathbf{B}$ *is the incidence matrix of a m-edge, n-node graph, if* $C_r, C_z = \widetilde{O}(1)$ *and* $\alpha = \widetilde{\Omega}(1)$*, there is a* $(\mathcal{T}_{\text{init}}, \mathcal{T}_{\text{phase}})$*-solution maintainer (Definition 2.8.3) with* $\mathcal{T}_{\text{init}} = \widetilde{O}(m)$ *and* $\mathcal{T}_{\text{phase}} = \widetilde{O}(m + m^{15/16}k^{29/8})$*.*

Note that in the solution maintenance data structure problem it is required that StartPhase be called at least every k calls to Move. Consequently, to apply this data structure to implement the robust IPM framework the input $\widehat{x}$ and $\widehat{s}$ to StartPhase, i.e. weighted ℓ_2 approximations to (x, s), need to be computed efficiently. We formalize this problem below.

Definition 2.8.5 (Solution Approximation). We call a procedure $\mathcal{T}_{\text{approx}}$-*approximator* if given μ-feasible $(\boldsymbol{x}, \boldsymbol{s})$, weights $\boldsymbol{w}^{(1)}, \cdots, \boldsymbol{w}^{(k)} \in \mathbb{R}_+^m$, directions $\boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(k)} \in \mathbb{R}^m$, and step sizes $h^{(1)}, \cdots, h^{(k)}$ such that

- $h^{(i)} \|\boldsymbol{v}^{(i)}\|_2 \leq 1$,
- all the changes in $\boldsymbol{w}$ and $\boldsymbol{v}$ are supported on z many edges and the input is given as these changes,
- $\frac{1}{r} \leq \sqrt{\boldsymbol{w}_\ell^{(i)}/\boldsymbol{w}_\ell^{(j)}} \leq r$ for all $i, j \in [k]$ and $\ell \in [m]$,

with high probability, we can compute μ-feasible $(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{s}})$ such that

$$\left\| \widetilde{\boldsymbol{x}} - \boldsymbol{x} - \sum_{i \in [k]} h^{(i)} \mathbf{W}_i^{1/2} (\mathbf{I} - \mathbf{P}_{\mathbf{W}_i}) \boldsymbol{v}^{(i)} \right\|_{\mathbf{W}_k^{-1}} \leq \epsilon \text{ and } \left\| \widetilde{\boldsymbol{s}} - \boldsymbol{s} - \sum_{i \in [k]} h^{(i)} \mathbf{W}_i^{-1/2} \mathbf{P}_{\mathbf{W}_i} v^{(i)} \right\|_{\mathbf{W}_k} \leq \epsilon$$

in $O(\mathcal{T}_{\text{approx}})$ time where $\mathbf{W}_i \stackrel{\text{def}}{=} \mathbf{diag}(\boldsymbol{w}^{(i)})$.

In the particular case of graphs, in Section 2.8.4 we provide the following theorem on efficient solution approximation.

Theorem 2. *In the special case that* $\mathbf{B}$ *is the incidence matrix of a* m*-edge,* n*-node graph there is* $\mathcal{T}_{\text{approx}}$-approximator *with* $\mathcal{T}_{\text{approx}} = \widetilde{O}(m + zk^3r^2\epsilon^{-2})$.

Finally, to apply these results, we need to prove that the weights which in turn are induced by $\nabla^2\phi(\boldsymbol{x})$ do not change by too much. For this, in Section 2.8.5 we prove the following. The statement is similar to the bound in [25, Lemma 6.5] generalized to our setting. Note that this bound applies to the IPM framework regardless of whether or not $\mathbf{B}$ is the incidence matrix of a graph.

Lemma 2.8.6. *For* $\mu^{(0)}$*-centered* $(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)})$ *and* $\mu^{(1)}$*-centered* $(\boldsymbol{x}^{(1)}, \boldsymbol{s}^{(1)})$ *with* $\mu^{(0)} \approx_{1/32} \mu^{(1)}$*, if* $\boldsymbol{\eta}^{(j)} \stackrel{\text{def}}{=} \boldsymbol{s}^{(j)} + \nabla\phi(\boldsymbol{x}^{(j)})$ *for* $j \in \{0, 1\}$ *it follows that*

$$\sum_{\phi_i''(\boldsymbol{x}^{(0)})^{1/2} \geq 3\phi_i''(\boldsymbol{x}^{(1)})^{1/2}} \sqrt{\frac{\phi_i''(\boldsymbol{x}^{(0)})}{\phi_i''(\boldsymbol{x}^{(1)})}} \leq 2^{10} \sum_{i\in[m]} \frac{(\boldsymbol{\eta}_i^{(1)} - \boldsymbol{\eta}_i^{(0)})^2}{\phi_i''(\boldsymbol{x}^{(0)}) + \phi_i''(\boldsymbol{x}^{(1)})} + 2^4 m \left(\frac{\mu^{(1)} - \mu^{(0)}}{\mu^{(0)}}\right)^2 .$$

2.8.3 Robust IPM Implementation

Here we show how to use the tools of Section 2.8.2 to efficiently implement the Algorithm 9. The algorithm, Algorithm 9, and its analysis, Lemma 3, are given below.

Theorem 3. *For any* $k \geq 1$*,* $\mu_{\text{end}} \leq \mu_{\text{start}}$*, and* μ_{start}*-centered* $(\boldsymbol{x}, \boldsymbol{s})$ *with* $\Psi(\boldsymbol{x}, \boldsymbol{s}) \leq \cosh(\lambda/64)$*, Algorithm 10 outputs a* μ_{end}*-centered* $(\boldsymbol{x}', \boldsymbol{s}')$ *with* $\Psi(\boldsymbol{x}', \boldsymbol{s}') \leq \cosh(\lambda/64)$ *in time* $\widetilde{O}((\mathcal{T}_{\text{init}} + \frac{\sqrt{m}}{k}(\mathcal{T}_{\text{phase}} + \mathcal{T}_{\text{approx}})) \log(\mu_{\text{start}}/\mu_{\text{end}}))$.

Proof. First, we verify that the conditions of the solution maintenance data structure (Definition 2.8.3) are satisfied with C_r, C_z defined as in the Algorithm 9.

- C_z: Note that both $\boldsymbol{w}, \boldsymbol{v}$ are entrywise functions of $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$ and Definition 2.8.3 promises that $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$ changes in at most $O(2^{2\ell_{j+1}}\epsilon^{-2}\log^3 m + S_j)$ coordinates. Since we only change $\boldsymbol{w}$ when $\overline{\boldsymbol{x}}$ or $\overline{\boldsymbol{s}}$ changes, we have $S_j = 0$. Using the parameter choice $\epsilon = \Theta(1/\log m)$, the number of changes is bounded by $O(2^{2\ell_{j+1}}\log^5 m)$. This verifies the condition $C_z = O(\log^5 m)$.

Algorithm 10 Algorithm 9 Implementation with Solution Maintenance and Estimation

```
procedure CENTERINGIMPL($\mathbf{B}, \boldsymbol{x}, \boldsymbol{s}, \boldsymbol{\ell}, \boldsymbol{u}, \mu_{\text{start}}, \mu_{\text{end}}, k$)
    Define step size $\alpha \overset{\text{def}}{=} \frac{1}{2^{15}\lambda}$, weight range $C_r = \Theta(1)$, and sparsity parameter $C_z = \Theta(\log^5 m)$
    Set $\overline{\mu} = \mu = \mu_{\text{start}}$, $\overline{\boldsymbol{x}} = \boldsymbol{x}$, $\overline{\boldsymbol{s}} = \boldsymbol{s}$, $\boldsymbol{w} = \text{diag}(\nabla^2\phi(\overline{\boldsymbol{x}})^{-1})$, $j = 0$
    SOL.INITIALIZE($\mathbf{B}, \boldsymbol{w}, \boldsymbol{x}, \boldsymbol{s}, \alpha, C_r, k, C_z$) where SOL is a $(\mathcal{T}_{\text{init}}, \mathcal{T}_{\text{phase}})$-solution maintainer (Definition 2.8.3)
    while $\mu \geq \mu_{\text{end}}$ do
        if ( then                                  ▷ *[h]Reset every $k$ iterations)$k$ divides $j$ or $\mu = \mu_{\text{end}}$
            Let $(\boldsymbol{x}, \boldsymbol{s})$ be the solution SOL implicitly maintained.
            Find $\overline{\mu}$-feasible $(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{s}})$ with $\|\mathbf{W}^{-1/2}(\boldsymbol{x} - \widetilde{\boldsymbol{x}})\|_2 \leq \frac{\alpha}{100}$ and $\|\mathbf{W}^{1/2}(\boldsymbol{s} - \widetilde{\boldsymbol{s}})\|_2 \leq \frac{\alpha}{100}$ by using a $\mathcal{T}_{\text{approx}}$-approximator (Definition 2.8.5) with $k = k, r = O(k^4), z = \widetilde{O}(k^2)$.
            if $|\overline{\mu} - \mu| \geq \alpha\overline{\mu}$ then
                SOL.INITIALIZE($\mathbf{B}, \boldsymbol{w}, \boldsymbol{x}, \boldsymbol{s}, \alpha, r, k, z$)
                $\widetilde{\boldsymbol{s}} \leftarrow \frac{\mu}{\overline{\mu}}\widetilde{\boldsymbol{s}}$, $\overline{\mu} \leftarrow \mu$              ▷ *Reinitialize. All coordinates may have changed
            end if
            SOL.STARTPHASE($\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{s}}$)
        end if

                                   ▷ Step: $\boldsymbol{x} \leftarrow \boldsymbol{x} + h\mathbf{W}^{1/2}(\mathbf{I} - \mathbf{P_W})\boldsymbol{v}, \boldsymbol{s} \leftarrow \boldsymbol{s} + h\mathbf{W}^{-1/2}\mathbf{P_W}\boldsymbol{v}$
        Set the direction $\boldsymbol{v}_i = \sinh(\lambda\gamma_i(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}))$ and the step size $h = -\alpha/\|\cosh(\lambda\gamma(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}))\|_2$.
        $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}) \leftarrow$ SOL.MOVE($\boldsymbol{w}, \boldsymbol{v}, h$)
        $\mu \leftarrow \max((1 - \frac{\alpha}{64\sqrt{m}})\mu, \mu_{\text{end}})$, $\boldsymbol{w} \leftarrow \text{diag}(\nabla^2\phi(\overline{\boldsymbol{x}})^{-1})$, and $j \leftarrow j + 1$
    end while
    Return $(\boldsymbol{x}, \boldsymbol{s})$
end procedure
```

- C_r: For any two iterations $\boldsymbol{x}^{(j_1)}$ and $\boldsymbol{x}^{(j_2)}$ associated with path parameters $\mu^{(j_1)}$ and $\mu^{(j_2)}$, Lemma 2.8.6 shows that

$$\sqrt{\frac{\phi_i''(\boldsymbol{x}^{(j_1)})}{\phi_i''(\boldsymbol{x}^{(j_2)})}} \leq 3 + 2^{10}\sum_{i\in[m]}\frac{(\boldsymbol{\eta}_i^{(j_2)} - \boldsymbol{\eta}_i^{(j_1)})^2}{\phi_i''(\boldsymbol{x}^{(j_2)}) + \phi_i''(\boldsymbol{x}^{(j_1)})} + 2^4 m\left(\frac{\mu^{(j_2)} - \mu^{(j_1)}}{\mu^{(j_1)}}\right)^2$$

where $\boldsymbol{\eta}^{(j)} \overset{\text{def}}{=} \boldsymbol{s}^{(j)} + \nabla\phi(\boldsymbol{x}^{(j)})$. Note that to apply this lemma, we used that all iterations are centered (which we will show later) and that $\mu^{(j_1)} \approx_{1/32} \mu^{(j_2)}$ (since we reinitialize the data structure every $\widetilde{\Theta}(\sqrt{m})$ steps). For $L = |j_1 - j_2|$, we have $|\mu^{(j_2)} - \mu^{(j_1)}| \leq \frac{\alpha L}{32\sqrt{m}}\mu^{(j_1)}$, so

$$\sqrt{\frac{\phi_i''(\boldsymbol{x}^{(j_1)})}{\phi_i''(\boldsymbol{x}^{(j_2)})}} \leq 3 + 2^{10}\sum_{i\in[m]}\frac{(\boldsymbol{\eta}_i^{(j_2)} - \boldsymbol{\eta}_i^{(j_1)})^2}{\phi_i''(\boldsymbol{x}^{(j_2)}) + \phi_i''(\boldsymbol{x}^{(j_1)})} + O(\alpha^2 L^2).$$

To bound the first term, we note that every term in the summation is bounded by $O(1)$. We split the sum into two cases. The first case is when $\phi_i''(\boldsymbol{x})$ does not change by more than a $O(1)$ factor. In this case, one can prove that $\|\boldsymbol{\eta}^{(j+1)} - \boldsymbol{\eta}^{(j)}\|_{\phi''(\boldsymbol{x}^{(j)})} = O(\alpha)$ because $\|\boldsymbol{x}^{(j+1)} - \boldsymbol{x}^{(j)}\|_{(\mathbf{W}^{(j)})^{-1}} \leq \alpha$, $\|\boldsymbol{s}^{(j+1)} - \boldsymbol{s}^{(j)}\|_{\mathbf{W}^{(j)}} \leq \alpha$ and $\mathbf{W}^{(j)} \approx_{O(1)} \nabla^2\phi(\boldsymbol{x}^{(j)})^{-1}$. Therefore, after L steps, the sum for the first case is bounded by $O(\alpha^2 L^2) = O(L^2)$. For the second case, we can use $\|\boldsymbol{x}^{(j+1)} - \boldsymbol{x}^{(j)}\|_{(\mathbf{W}^{(j)})^{-1}} \leq \alpha$ to show that there are at most $O(L^2\alpha^2)$ coordinates where ϕ'' changes by more than a constant multiplicative factor. Hence, this shows that

$$\sqrt{\frac{\phi_i''(\boldsymbol{x}^{(j_1)})}{\phi_i''(\boldsymbol{x}^{(j_2)})}} = O(L^2). \tag{2.19}$$

This verifies the condition $C_r = O(1)$.

Now, we bound the potential. Theorem 1 shows that

$$\Psi(\boldsymbol{x}^{\text{new}}, \boldsymbol{x}^{\text{new}}) \leq \left(1 - \frac{\alpha\lambda}{8\sqrt{m}}\right)\Psi(\boldsymbol{x}, \boldsymbol{x}) + \alpha\lambda\sqrt{m}$$

for every step (excluding the effect of StartPhase). For StartPhase, we have that $\|\mathbf{W}^{-1/2}(\boldsymbol{x} - \widetilde{\boldsymbol{x}})\|_2 \leq \frac{\alpha}{100}$ and $\|\mathbf{W}^{1/2}(\boldsymbol{s} - \widetilde{\boldsymbol{s}})\|_2 \leq \frac{\alpha}{100}$. This increases Ψ by at most $\frac{\alpha\lambda}{16\sqrt{m}}\Psi(\boldsymbol{x}, \boldsymbol{s})$ additively. Finally, for the change of $\overline{\mu}$, it would increase Ψ by at most $2\alpha\lambda\Psi(\boldsymbol{x}, \boldsymbol{s})$, but this happens every $32\sqrt{m}$ steps. Therefore Ψ is decreasing on average and stays polynomially bounded.

Next, we discuss the parameters for the $\mathcal{T}_{\text{approx}}$-approximator. The number of terms is exactly given by k. For the number of coordinate changes z, Definition 2.8.3 promised that $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$ changes by $\widetilde{O}(2^{2\ell_{j+1}})$ coordinates at the j-th step. Since we restart every k iterations by calling StartPhase, by aligning our steps numbers appropriately, we have that $\sum_{j \text{ in a phase}} \widetilde{O}(2^{2\ell_j}) = \widetilde{O}(\max_{j \text{ in a phase}} 2^{2\ell_j}) = \widetilde{O}(k^2)$. Finally, the weight ratio is due to (2.19) with $L = k$.

Finally, for the runtime, note that there are $\widetilde{O}(\sqrt{m}\log(\mu_{\text{start}}/\mu_{\text{end}}))$ steps. For every k steps, we use a $\mathcal{T}_{\text{approx}}$-approximator and call Sol.StartPhase and they cost $\mathcal{T}_{\text{phase}}$ and $\mathcal{T}_{\text{approx}}$ respectively. All other costs are linear in the output size of the data structure and are not bottlenecks. Therefore, the total cost is $\widetilde{O}((\mathcal{T}_{\text{init}} + \frac{\sqrt{m}}{k}(\mathcal{T}_{\text{phase}} + \mathcal{T}_{\text{approx}}))\log(\mu_{\text{start}}/\mu_{\text{end}}))$. $\square$

2.8.4 Efficient Solution Approximation

In this section, we prove Theorem 2. Our algorithms leverage two powerful tools from algorithmic graph theory, in particular nearly linear time algorithms for subspace sparsification [112].

Proof of Theorem 2. Our algorithm for approximating $\widetilde{\boldsymbol{x}}$ involves two steps, we first find a $\boldsymbol{x}'$ such that it is close to the true vector $\boldsymbol{x}^* \stackrel{\mathrm{def}}{=} \boldsymbol{x} + \sum_{i\in[k]} h^{(i)}\mathbf{W}_i^{1/2}(\mathbf{I} - \mathbf{P}_{\mathbf{W}_i})\boldsymbol{v}^{(i)}$, but may not satisfies $\mathbf{B}^\top \boldsymbol{x}' = \boldsymbol{d}$. Then, we show how to use $\boldsymbol{x}'$ to find $\widetilde{\boldsymbol{x}}$ that is close to $\boldsymbol{x}^*$ and satisfies $\mathbf{B}^\top \boldsymbol{x}' = \boldsymbol{d}$.

Let $S \subseteq [m]$ be the set of at most z coordinates of $\boldsymbol{w}$ and $\boldsymbol{v}$ that change and let $C \subseteq [n]$ be an arbitrary subset (that we set later) such that every edge in S has both endpoints in C. Further, let $\Delta_1 \stackrel{\mathrm{def}}{=} \sum_{i\in[k]} h^{(i)}\mathbf{W}_i^{1/2}\mathbf{P}_{\mathbf{W}_i}\boldsymbol{v}^{(i)}$ and $\Delta_2 \stackrel{\mathrm{def}}{=} \sum_{i\in[k]} h^{(i)}\mathbf{W}_i^{1/2}\boldsymbol{v}^{(i)}$ so that $\sum_{i\in[k]} h^{(i)}\mathbf{W}_i^{1/2}(\mathbf{I} - \mathbf{P}_{\mathbf{W}_i})\boldsymbol{v}^{(i)} = \Delta_2 - \Delta_1$. Note that Δ_2 can be computed in $O(m + zk)$ time by first computing $\sum_{i\in[k]} h^{(i)}$ and with this computing $[\Delta_2]_j$ for $j \notin S$ in $O(1)$ time and for $j \in S$ in $O(k)$ time. Consequently, to compute $\tilde{x}$ in the given time bound, it suffices to approximately compute Δ_1.

Next, let $\mathbf{L}^{(i)} \stackrel{\mathrm{def}}{=} \mathbf{B}^\top\mathbf{W}_i\mathbf{B}$ and $F \stackrel{\mathrm{def}}{=} V \setminus C$ and note that

$$\Delta_1 = \sum_{i\in[k]} h^{(i)}\mathbf{W}_i\mathbf{B}\begin{bmatrix} \mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}\begin{bmatrix} \mathbf{L}_{FF}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{SC}(\mathbf{L}_i, C)^\dagger \end{bmatrix}\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix}\mathbf{B}^\top\mathbf{W}_i^{1/2}\boldsymbol{v}^{(i)}$$

since $\mathbf{L}_{FF} = [\mathbf{L}^{(i)}]_{FF}$ and $\mathbf{L}_{FC} = [\mathbf{L}^{(i)}]_{FC}$ for all i by the definition of C. Further, let $\mathbf{B}_C$ be the incidence matrix of edges with both endpoints in C and $\mathbf{B}_{-C}$ be the incidence matrix of the remaining edges so that $\mathbf{B}^\top\mathbf{W}_i^{1/2}\boldsymbol{v}^{(i)} = \mathbf{B}_C^\top\mathbf{W}_i^{1/2}\boldsymbol{v}^{(i)} + \mathbf{B}_{-C}^\top\mathbf{W}^{1/2}\boldsymbol{v}$ for all $i \in [k]$. Combining yields that, $\Delta_1 = \boldsymbol{a}_1 + \boldsymbol{a}_2 + \boldsymbol{a}_3 + \boldsymbol{a}_4$ where

$$\boldsymbol{a}_1 \overset{\text{def}}{=} \sum_{i \in [k]} h^{(i)} \mathbf{W}_i \mathbf{B}_C \mathbf{SC}(\mathbf{L}_i, C)^{\dagger} \mathbf{B}_C^{\top} \mathbf{W}_i^{1/2} \boldsymbol{v}^{(i)} \tag{2.20}$$

$$\boldsymbol{a}_2 \overset{\text{def}}{=} \sum_{i \in [k]} h^{(i)} \mathbf{W} \mathbf{B}_{-C} \begin{bmatrix} -\mathbf{L}_{FF}^{-1} \mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix} \mathbf{SC}(\mathbf{L}_i, C)^{-1} \mathbf{B}_C^{\top} \mathbf{W}_i^{1/2} \boldsymbol{v}^{(i)}$$

$$\boldsymbol{a}_3 \overset{\text{def}}{=} \sum_{i \in [k]} h^{(i)} \mathbf{W}_i \mathbf{B}_C \mathbf{SC}(\mathbf{L}_i, C)^{\dagger} \begin{bmatrix} -\mathbf{L}_{CF} \mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix} \mathbf{B}_{-C}^{\top} \mathbf{W}^{1/2} \boldsymbol{v}$$

$$\boldsymbol{a}_4 \overset{\text{def}}{=} \sum_{i \in [k]} h^{(i)} \mathbf{W} \mathbf{B}_{-C} \begin{bmatrix} \mathbf{I} & -\mathbf{L}_{FF}^{-1} \mathbf{L}_{FC} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{FF}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{SC}(\mathbf{L}_i, C)^{\dagger} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{L}_{CF} \mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix} \mathbf{B}_{-C}^{\top} \mathbf{W}^{1/2} \boldsymbol{v} \, . \tag{2.21}$$

Our algorithm simply computes Δ_1 through the above formula where every instance of $\mathbf{SC}(\mathbf{L}_i, C)$ is replaced with some efficiently computed $\widetilde{\mathbf{SC}}_i \approx_\delta \mathbf{SC}(\mathbf{L}_i, C)$ for δ we set later.

To compute the $\widetilde{\mathbf{SC}}_i$, first for each i, we define $\mathbf{L}_i(S) \overset{\text{def}}{=} \mathbf{B}_S^{\top} \mathbf{W}_i \mathbf{B}_S$ where $\mathbf{B}_S$ is the incidence matrix of edges S and let $\mathbf{L}_{\text{ext}} \overset{\text{def}}{=} \mathbf{L}^{(i)} - \mathbf{L}_i(S)$ for any $i \in [k]$. Note that this definition does not depend on i by the definition of S. Using [112, Theorem 1.3], we can compute $C \subseteq V$ such that every edge in S has both endpoints in C and a Laplacian $\widetilde{\mathbf{SC}} \in \mathbb{R}^{C \times C}$ such that $\widetilde{\mathbf{SC}} \approx_\delta \mathbf{SC}(\mathbf{L}_{\text{ext}}, C)$ and $|C| \le \text{nnz}(\widetilde{\mathbf{SC}}) = \tilde{O}(|S|\delta^{-2}) = \tilde{O}(z\delta^{-2})$ in $\tilde{O}(m)$ time with high probability. We use this procedure to determine C and compute $\widetilde{\mathbf{SC}}$. Further, we define $\widetilde{\mathbf{SC}}_i = \widetilde{\mathbf{SC}} + \mathbf{L}_i(S)$ and note that $\widetilde{\mathbf{SC}}_i \approx_\delta \mathbf{SC}(\mathbf{L}_i, C)$ for all $i \in [k]$ and $\text{nnz}(\widetilde{\mathbf{SC}}_i) = \text{nnz}(\widetilde{\mathbf{SC}}_i) + |S| = \tilde{O}(z\delta^{-2})$.

Now, we let $\tilde{\boldsymbol{a}}_1, \tilde{\boldsymbol{a}}_2, \tilde{\boldsymbol{a}}_3, \tilde{\boldsymbol{a}}_4$ be the result of computing $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3, \boldsymbol{a}_4$ respectively where each $\mathbf{SC}(\mathbf{L}_i, C)$ is replaced with $\widetilde{\mathbf{SC}}_i$ and each matrix inversion is computed to high precision using nearly linear time SDD-solvers for $\mathbf{L}_{FF}^{-1}$ and $\mathbf{SC}(\mathbf{L}_i, C)^{-1}$ (Theorem 2.3.1). Further, we let $\boldsymbol{x}' = \boldsymbol{x} + \sum_{i \in [4]} \tilde{\boldsymbol{a}}_i + \Delta_2$. Note that $h^{(i)} \mathbf{B}_C^{\top} \mathbf{W}_i^{1/2} \boldsymbol{v}^{(i)}$ can be computed explicitly for all $i \in [k]$ in $\tilde{O}(m + kz\delta^{-2})$ time by simply iterating through the changes in $\boldsymbol{w}$ and $\boldsymbol{v}$ and noting that each change only effects the resulting $\tilde{O}(z\delta^{-2})$ coordinate vector in 2 coordinates. Further, this implies that $\boldsymbol{d}_i \overset{\text{def}}{=} h^{(i)} \mathbf{SC}(\mathbf{L}_i, C)^{\dagger} \mathbf{B}_C^{\top} \mathbf{W}_i^{1/2} \boldsymbol{v}^{(i)}$ can be computed to high precision in $\tilde{O}(kz\delta^{-2})$ time by using a nearly linear time Laplacian system solver too apply $\mathbf{SC}(\mathbf{L}_i, C)^{\dagger}$. Next, to compute

$\tilde{\boldsymbol{a}}_1 \stackrel{\text{def}}{=} \sum_{i\in[k]} \mathbf{W}_i \mathbf{B}_C d_i$ note that the contribution of each row of $\mathbf{B}_C$ for $e \in S$ can be computed $O(k)$ and the contribution of all the remaining rows can be computed in $O(m)$; thus, $\tilde{\boldsymbol{a}}_1$ can be computed from the $\boldsymbol{d}_i$ in $O(m + kz)$. Further, given the $\boldsymbol{d}_i$ by using a nearly linear time SDD solver to apply $\mathbf{L}_{FF}^{-1}$ to a vector we see that $\tilde{\boldsymbol{a}}_2$ can be computed in $\tilde{O}(m)$. Similarly, all the $\boldsymbol{e}_i \stackrel{\text{def}}{=} \mathbf{SC}(\mathbf{L}_i, C)^\dagger \begin{bmatrix} -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix} \mathbf{B}_{-C}^\top \mathbf{W}^{1/2}\boldsymbol{v}$ can be computed in $\tilde{O}(m + zk\delta^{-2})$ and from these $\tilde{\boldsymbol{a}}_3$ can be computed in an additional $O(m + kz)$ time (analogous to computing $\tilde{\boldsymbol{a}}_1$). Further, $\tilde{\boldsymbol{a}}_4$ can be computed $\tilde{O}(m + zk\delta^{-2})$ since summation can be moved to the $\mathbf{SC}(\mathbf{L}_i, C)^\dagger$. Putting these pieces together shows that $\tilde{\boldsymbol{x}}$ can be computed in $\tilde{O}(m + zk\delta^{-2})$.

Next, to determine what to set δ to. Note that

$$\begin{aligned}
&\|\boldsymbol{x}^* - \widetilde{\boldsymbol{x}}\|_{\mathbf{W}_k^{-1}} \\
=&\left\| \sum_{i\in[k]} \mathbf{W}_i \mathbf{B} \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix} (\mathbf{SC}(\mathbf{L}_i, C)^\dagger - \widetilde{\mathbf{SC}}_i^\dagger) \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix}^\top \mathbf{B}^\top \mathbf{W}_i^{1/2} (h^{(i)}\boldsymbol{v}^{(i)}) \right\|_{\mathbf{W}_k^{-1}} \\
\leq& r \sum_{i\in[k]} \left\| \mathbf{W}_i \mathbf{B} \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix} (\mathbf{SC}(\mathbf{L}_i, C)^\dagger - \widetilde{\mathbf{SC}}_i^\dagger) \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix}^\top \mathbf{B}^\top \mathbf{W}_i^{1/2} (h^{(i)}\boldsymbol{v}^{(i)}) \right\|_{\mathbf{W}_i^{-1}} \\
\leq& r \sum_{i\in[k]} \left\| \mathbf{W}_i^{-1/2} \mathbf{B} \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix} (\mathbf{SC}(\mathbf{L}_i, C)^\dagger - \widetilde{\mathbf{SC}}_i^\dagger) \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix}^\top \mathbf{B}^\top \mathbf{W}_i^{1/2} \right\|_2 \\
=& r \sum_{i\in[k]} \left\| \mathbf{SC}(\mathbf{L}_i, C)^{1/2} (\mathbf{SC}(\mathbf{L}_i, C)^\dagger - \widetilde{\mathbf{SC}}_i^\dagger) \mathbf{SC}(\mathbf{L}_i, C)^{1/2} \right\|_2 = O(rk\delta)
\end{aligned}$$

where in the third line we used that assumption $\frac{1}{r} \leq \sqrt{(\boldsymbol{w}_i)_l / (\boldsymbol{w}_j)_l} \leq r$, in the fourth we used that $\left\| h\boldsymbol{v}^{(i)} \right\|_2 \leq 1$, and in the fifth we used that

$$\begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix}^\top \mathbf{B}^\top \mathbf{W}_i \mathbf{B} \begin{bmatrix} -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{I} \end{bmatrix} = \mathbf{SC}(\mathbf{L}_i, C),$$

and that $\|\mathbf{M}_1\mathbf{M}_2\mathbf{M}_3\|_2 = \left\| (\mathbf{M}_1^\top \mathbf{M}_1)^{1/2} \mathbf{M}_2 (\mathbf{M}_3\mathbf{M}_3^\top)^{1/2} \right\|_2$ for matrices $\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3$ of appropriate dimension and that $\widetilde{\mathbf{SC}}_i \approx_\delta \mathbf{SC}(\mathbf{L}_i, C)$. Consequently, it suffices to set $\delta = \Theta(\epsilon/(rk))$ and this gives the result for computing $\widetilde{\boldsymbol{x}}$.

Now, we show how to find a feasible $\widetilde{\boldsymbol{x}}$ using $\boldsymbol{x}'$. From the first part, we can find $\boldsymbol{x}'$ such that $\|\boldsymbol{x}' - \boldsymbol{x}^*\|_{\mathbf{W}_k^{-1/2}} \leq \frac{\epsilon}{2}$ in $\widetilde{O}(m + zr^2k^3/\epsilon^2)$ time. Note that $\mathbf{B}^\top \boldsymbol{x}^* = \mathbf{B}^\top \boldsymbol{x} = \boldsymbol{d}$. However, we may not have $\mathbf{B}^\top \boldsymbol{x}' = \boldsymbol{d}$. To fix this, we define

$$\widetilde{\boldsymbol{x}} \stackrel{\text{def}}{=} \boldsymbol{x}' + \mathbf{W}_k \mathbf{B}(\mathbf{B}^\top \mathbf{W}_k \mathbf{B})^{-1}(\boldsymbol{d} - \mathbf{B}^\top \boldsymbol{x}').$$

This can be found in an extra $\widetilde{O}(m)$ time (Theorem 2.3.1). Furthermore, we have

$$\begin{aligned} \|\widetilde{\boldsymbol{x}} - \boldsymbol{x}'\|_{\mathbf{W}_k^{-1}} &= \|\mathbf{W}_k^{1/2}\mathbf{B}(\mathbf{B}^\top \mathbf{W}_k \mathbf{B})^{-1}(\mathbf{B}^\top \boldsymbol{x}^* - \mathbf{B}^\top \boldsymbol{x}')\|_2 \\ &\leq \|\mathbf{W}_k^{-1/2}(\boldsymbol{x}^* - \boldsymbol{x}')\|_2 \leq \frac{\epsilon}{2}. \end{aligned}$$

Hence, we have $\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{W}_k^{-1}} \leq \epsilon$ and $\mathbf{B}^\top \widetilde{\boldsymbol{x}} = \boldsymbol{d}$.

The algorithm and analysis for computing $\boldsymbol{s}'$ is analogous with Δ_2 set to 0 and the signs of the exponents of some $\mathbf{W}_i$ flipped. The main difference is that $\boldsymbol{s}'$ is automatically feasible and hence we simply set $\widetilde{\boldsymbol{s}} = \boldsymbol{s}'$. To see this, we note that the new $\Delta_1^{(\boldsymbol{s})}$ is given by

$$\Delta_1^{(\boldsymbol{s})} = \sum_{i \in [k]} h^{(i)} \mathbf{B} \begin{bmatrix} \mathbf{I} & -\mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{FF}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{SC}(\mathbf{L}_i, C)^\dagger \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{bmatrix} \mathbf{B}^\top \mathbf{W}_i^{1/2} \boldsymbol{v}^{(i)}.$$

Note that after we replacing $\mathbf{SC}(\mathbf{L}_i, C)^\dagger$ by its approximation, the vector above is still in the image of $\mathbf{B}$. Hence, $\boldsymbol{s}' - \boldsymbol{s}^*$ is in the image of $\mathbf{B}$.

□

2.8.5 Robust IPM Stability Bound

In this section we prove Lemma 2.8.6 which bounds the relative change in ϕ in each iteration of the robust IPM method (See Section 2.8). We first provide helper Lemma 2.8.7 and Lemma 2.8.8 and then use it to prove Lemma 2.8.6. The first lemma is a statement about 1-dimensional log-barrier problems.

Lemma 2.8.7. *Let $\boldsymbol{\ell}, \boldsymbol{u} \in \mathbb{R}^m$ with $\boldsymbol{\ell}_i < \boldsymbol{u}_i$ for all $i \in [m]$, $c \in \mathbb{R}$, $\boldsymbol{w}_\ell^{(0)}, \boldsymbol{w}_u^{(0)}, \boldsymbol{w}_\ell^{(1)}, \boldsymbol{w}_u^{(1)} \in [\frac{7}{8}, \frac{8}{7}]^m$, and for $j \in \{0,1\}$ let*

$$x^{(j)} \stackrel{\text{def}}{=} \operatorname*{arg\,min}_{\max_{i\in[m]} \boldsymbol{\ell}_i \le x \le \min_{i\in[m]} \boldsymbol{u}_i} c \cdot x - \sum_{i\in[m]} \boldsymbol{w}_{\ell,i}^{(j)} \log(x - \boldsymbol{\ell}_i) - \sum_{i\in[m]} \boldsymbol{w}_{u,i}^{(j)} \log(\boldsymbol{u}_i - x)$$

Then, for $r(a) \stackrel{\text{def}}{=} \max\{a-3, a^{-1}-3, 0\}$ we have

$$\sum_{i\in[m]} r\left(\frac{x^{(1)} - \boldsymbol{\ell}_i}{x^{(0)} - \boldsymbol{\ell}_i}\right) + \sum_{i\in[m]} r\left(\frac{\boldsymbol{u}_i - x^{(1)}}{\boldsymbol{u}_i - x^{(0)}}\right) \le 16\left[\left\|\boldsymbol{w}_\ell^{(0)} - \boldsymbol{w}_\ell^{(1)}\right\|_2^2 + \left\|\boldsymbol{w}_u^{(0)} - \boldsymbol{w}_u^{(1)}\right\|_2^2\right]$$

Remark. Note that $r\left(\frac{x^{(1)}-\boldsymbol{\ell}_i}{x^{(0)}-\boldsymbol{\ell}_i}\right)$ large implies either $x^{(1)}$ or $x^{(0)}$ is much closer to $\boldsymbol{\ell}_i$ compared to the another one. The inequality above shows that if the weights do not change too much, then $x^{(1)}$ cannot be too much closer to $\boldsymbol{\ell}_i$ compared to $x^{(0)}$.

Proof. Without loss of generality, we assume $x^{(0)} < x^{(1)}$. By the optimality condition of $x^{(j)}$, we have that

$$c - \sum_{i\in[m]} \frac{\boldsymbol{w}_{\ell,i}^{(j)}}{x^{(j)} - \boldsymbol{\ell}_i} + \sum_{i\in[m]} \frac{\boldsymbol{w}_{u,i}^{(j)}}{\boldsymbol{u}_i - x^{(j)}} = 0.$$

Subtracting this equation for $j = 0$ and 1, we have

$$\sum_{i\in[m]} \left(\frac{\boldsymbol{w}_{\ell,i}^{(0)}}{x^{(0)} - \boldsymbol{\ell}_i} - \frac{\boldsymbol{w}_{\ell,i}^{(1)}}{x^{(1)} - \boldsymbol{\ell}_i}\right) = \sum_{i\in[m]} \left(\frac{\boldsymbol{w}_{u,i}^{(0)}}{\boldsymbol{u}_i - x^{(0)}} - \frac{\boldsymbol{w}_{u,i}^{(1)}}{\boldsymbol{u}_i - x^{(1)}}\right). \tag{2.22}$$

We bound the left and right hand side above separately.

To lower bound the left hand side of (2.22), we let $\boldsymbol{\alpha}_i = \frac{x^{(1)} - x^{(0)}}{x^{(1)} - \boldsymbol{\ell}_i} \in (0,1)$ (since $x^{(1)} > x^{(0)} > \boldsymbol{\ell}_i$). Note that

$$\frac{\boldsymbol{w}_{\ell,i}^{(0)}}{x^{(0)} - \boldsymbol{\ell}_i} - \frac{\boldsymbol{w}_{\ell,i}^{(1)}}{x^{(1)} - \boldsymbol{\ell}_i} = \frac{\boldsymbol{\alpha}_i}{x^{(1)} - x^{(0)}} \left(\frac{\boldsymbol{w}_{\ell,i}^{(0)}}{1 - \alpha_i} - \boldsymbol{w}_{\ell,i}^{(1)}\right).$$

If $\boldsymbol{\alpha}_i \le 2(\boldsymbol{w}_{\ell,i}^{(1)} - \boldsymbol{w}_{\ell,i}^{(0)})$ then

$$\frac{\boldsymbol{w}_{\ell,i}^{(0)}}{x^{(0)} - \boldsymbol{\ell}_i} - \frac{\boldsymbol{w}_{\ell,i}^{(1)}}{x^{(1)} - \boldsymbol{\ell}_i} \ge \frac{\boldsymbol{\alpha}_i(\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)})}{x^{(1)} - x^{(0)}} \ge -\frac{2(\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)})^2}{x^{(1)} - x^{(0)}}$$

where we used $\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)} \leq 0$ in the last inequality. Otherwise, we have $\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)} > -\boldsymbol{\alpha}_i/2$ and hence

$$\frac{\boldsymbol{w}_{\ell,i}^{(0)}}{1-\boldsymbol{\alpha}_i} - \boldsymbol{w}_{\ell,i}^{(1)} = \frac{\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)} + \boldsymbol{\alpha}_i \boldsymbol{w}_{\ell,i}^{(1)}}{1-\boldsymbol{\alpha}_i} \geq \frac{\boldsymbol{\alpha}_i(\boldsymbol{w}_{\ell,i}^{(1)} - 1/2)}{1-\boldsymbol{\alpha}_i} \geq \frac{1}{3}\frac{\boldsymbol{\alpha}_i}{1-\boldsymbol{\alpha}_i} \geq 0$$

where we used that $\boldsymbol{w}_{\ell,i}^{(1)} \geq \frac{7}{8}$ and $\frac{7}{8} - \frac{1}{2} \geq \frac{1}{3}$. Combining both cases, we have

$$\begin{aligned}
&(x^{(1)} - x^{(0)}) \cdot \sum_{i\in[m]} \left(\frac{\boldsymbol{w}_{\ell,i}^{(0)}}{x^{(0)} - \boldsymbol{\ell}_i} - \frac{\boldsymbol{w}_{\ell,i}^{(1)}}{x^{(1)} - \boldsymbol{\ell}_i} \right) \\
\geq &- 2 \sum_{\boldsymbol{\alpha}_i \leq 2(\boldsymbol{w}_{\ell,i}^{(1)} - \boldsymbol{w}_{\ell,i}^{(0)})} (\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)})^2 + \sum_{\boldsymbol{\alpha}_i > 2(\boldsymbol{w}_{\ell,i}^{(1)} - \boldsymbol{w}_{\ell,i}^{(0)})} \frac{1}{3}\frac{\boldsymbol{\alpha}_i^2}{1-\boldsymbol{\alpha}_i} \\
\geq &- 2 \sum_{i\in[m]} (\boldsymbol{w}_{\ell,i}^{(0)} - \boldsymbol{w}_{\ell,i}^{(1)})^2 + \frac{1}{8} \sum_{\boldsymbol{\alpha}_i \geq \frac{2}{3}} \frac{1}{1-\boldsymbol{\alpha}_i}
\end{aligned} \tag{2.23}$$

where we used that $\boldsymbol{\alpha}_i \geq \frac{2}{3}$ implies $\boldsymbol{\alpha}_i > 2(\boldsymbol{w}_{\ell,i}^{(1)} - \boldsymbol{w}_{\ell,i}^{(0)})$ at the end.

To upper bound the right hand side of (2.22), we let $\boldsymbol{\beta}_i = \frac{x^{(1)} - x^{(0)}}{\boldsymbol{u}_i - x^{(0)}} \in (0, 1)$. Note that

$$\frac{\boldsymbol{w}_{u,i}^{(0)}}{\boldsymbol{u}_i - x^{(0)}} - \frac{\boldsymbol{w}_{u,i}^{(1)}}{\boldsymbol{u}_i - x^{(1)}} = \frac{\boldsymbol{\beta}_i}{x^{(1)} - x^{(0)}} \left(\boldsymbol{w}_{u,i}^{(0)} - \frac{\boldsymbol{w}_{u,i}^{(1)}}{1-\boldsymbol{\beta}_i} \right).$$

If $\boldsymbol{\beta}_i \leq 2(\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)})$, we have that

$$\frac{\boldsymbol{w}_{u,i}^{(0)}}{\boldsymbol{u}_i - x^{(0)}} - \frac{\boldsymbol{w}_{u,i}^{(1)}}{\boldsymbol{u}_i - x^{(1)}} \leq \frac{\boldsymbol{\beta}_i(\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)})}{x^{(1)} - x^{(0)}} \leq \frac{2(\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)})^2}{x^{(1)} - x^{(0)}}$$

where we used $\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)} \geq 0$ at the last inequality. Otherwise, we have $\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)} \leq \boldsymbol{\beta}_i/2$ and hence

$$\boldsymbol{w}_{u,i}^{(0)} - \frac{\boldsymbol{w}_{u,i}^{(1)}}{1-\boldsymbol{\beta}_i} = \frac{\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)} - \boldsymbol{\beta}_i \boldsymbol{w}_{u,i}^{(0)}}{1-\boldsymbol{\beta}_i} \leq \frac{\boldsymbol{\beta}_i(\frac{1}{2} - \boldsymbol{w}_{u,i}^{(0)})}{1-\boldsymbol{\beta}_i} \leq -\frac{1}{3}\frac{\boldsymbol{\beta}_i}{1-\boldsymbol{\beta}_i} \leq 0.$$

Combining both cases and using $\boldsymbol{\beta}_i \geq \frac{2}{3}$ implies $\boldsymbol{\beta}_i > 2(\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)})$, we have

$$(x^{(1)} - x^{(0)}) \cdot \sum_{i\in[m]} \left(\frac{\boldsymbol{w}_{u,i}^{(0)}}{\boldsymbol{u}_i - x^{(0)}} - \frac{\boldsymbol{w}_{u,i}^{(1)}}{\boldsymbol{u}_i - x^{(1)}} \right) \leq 2 \sum_{i\in[m]} (\boldsymbol{w}_{u,i}^{(0)} - \boldsymbol{w}_{u,i}^{(1)})^2 - \frac{1}{8} \sum_{\boldsymbol{\beta}_i \geq \frac{2}{3}} \frac{1}{1-\boldsymbol{\beta}_i} \tag{2.24}$$

Combining (2.23) and (2.24) with (2.22), we have

$$-2\sum_{i\in[m]}(\boldsymbol{w}_{\ell,i}^{(0)}-\boldsymbol{w}_{\ell,i}^{(1)})^2+\frac{1}{8}\sum_{\boldsymbol{\alpha}_i\geq\frac{2}{3}}\frac{1}{1-\boldsymbol{\alpha}_i}\leq 2\sum_{i\in[m]}(\boldsymbol{w}_{u,i}^{(0)}-\boldsymbol{w}_{u,i}^{(1)})^2-\frac{1}{8}\sum_{\boldsymbol{\beta}_i\geq\frac{2}{3}}\frac{1}{1-\boldsymbol{\beta}_i}.$$

Using this, $x^{(0)}\leq x^{(1)}$ and the formula of r, we have

$$\begin{aligned}\sum_{i\in[m]}r\left(\frac{x^{(1)}-\boldsymbol{\ell}_i}{x^{(0)}-\boldsymbol{\ell}_i}\right)+r\left(\frac{\boldsymbol{u}_i-x^{(1)}}{\boldsymbol{u}_i-x^{(0)}}\right)&\leq\sum_{i\in[m]}\max\left\{\frac{x^{(1)}-\boldsymbol{\ell}_i}{x^{(0)}-\boldsymbol{\ell}_i}-3,0\right\}+\sum_{i\in[m]}\max\left\{\frac{\boldsymbol{u}_i-x^{(0)}}{\boldsymbol{u}_i-x^{(1)}}-3,0\right\}\\&=\sum_{i\in[m]}\max\left\{\frac{1}{1-\boldsymbol{\alpha}_i}-3,0\right\}+\sum_{i\in[m]}\max\left\{\frac{1}{1-\boldsymbol{\beta}_i}-3,0\right\}\\&\leq\sum_{\boldsymbol{\alpha}_i\geq\frac{2}{3}}\frac{1}{1-\boldsymbol{\alpha}_i}+\sum_{\boldsymbol{\beta}_i\geq\frac{2}{3}}\frac{1}{1-\boldsymbol{\beta}_i}\\&\leq 16\left[\left\|\boldsymbol{w}_\ell^{(0)}-\boldsymbol{w}_\ell^{(1)}\right\|_2^2+\left\|\boldsymbol{w}_u^{(0)}-\boldsymbol{w}_u^{(1)}\right\|_2^2\right].\end{aligned}$$

□

We leverage this lemma to generalize to higher dimensions in the following lemma.

Lemma 2.8.8. *In the setting of* (2.18)*, given weights* $\boldsymbol{w}_\ell^{(0)},\boldsymbol{w}_u^{(0)},\boldsymbol{w}_\ell^{(1)},\boldsymbol{w}_u^{(1)}\in[\frac{7}{8},\frac{8}{7}]^m$ *let*

$$\boldsymbol{x}_j\overset{\text{def}}{=}\operatorname*{arg\,min}_{\boldsymbol{x}\in\mathcal{X}\,|\,\mathbf{B}^\top\boldsymbol{x}=\boldsymbol{d}}\boldsymbol{c}^\top\boldsymbol{x}-\sum_{i\in[m]}\boldsymbol{w}_{\ell,i}^{(j)}\log(\boldsymbol{x}-\boldsymbol{\ell}_i)-\sum_{i\in[m]}\boldsymbol{w}_{u,i}^{(j)}\log(\boldsymbol{u}_i-\boldsymbol{x})$$

for $j\in\{0,1\}$*. Then for* $r(a)\overset{\text{def}}{=}\max\{a-3,a^{-1}-3,0\}$ *we have*

$$\sum_{i\in[m]}r\left(\frac{\boldsymbol{x}_i^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}_i^{(0)}-\boldsymbol{\ell}_i}\right)+\sum_{i\in[m]}r\left(\frac{\boldsymbol{u}_i-\boldsymbol{x}^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}^{(0)}}\right)\leq 16\left[\left\|\boldsymbol{w}_\ell^{(0)}-\boldsymbol{w}_\ell^{(1)}\right\|_2^2+\left\|\boldsymbol{w}_u^{(0)}-\boldsymbol{w}_u^{(1)}\right\|_2^2\right].$$

Proof. For all $t\in\mathbb{R}$ let $\boldsymbol{x}^{(t)}\overset{\text{def}}{=}\boldsymbol{x}^{(0)}+t\boldsymbol{\delta}_x$ with $\boldsymbol{\delta}_x=\boldsymbol{x}^{(1)}-\boldsymbol{x}^{(0)}$. By the definition of $\boldsymbol{x}^{(j)}$, we have that for $j \in \{0,1\}$

$$\begin{aligned}
\boldsymbol{x}^{(j)} &= \operatorname*{arg\,min}_{t\in\mathbb{R}|\boldsymbol{x}^{(t)}\in\mathcal{X}} \boldsymbol{c}^\top \boldsymbol{x}^{(t)} - \sum_{i\in[m]} \boldsymbol{w}_{\ell,i}^{(j)} \log(\boldsymbol{x}_i^{(t)} - \boldsymbol{\ell}_i) - \sum_{i\in[m]} \boldsymbol{w}_{u,i}^{(j)} \log(\boldsymbol{u}_i - \boldsymbol{x}_i^{(t)}) \\
&= \operatorname*{arg\,min}_{t\in\mathbb{R}|\boldsymbol{x}^{(t)}\in\mathcal{X}} t\cdot \boldsymbol{c}^\top \boldsymbol{\delta}_x - \sum_{i\in[m]} \boldsymbol{w}_{\ell,i}^{(j)} \log(t\boldsymbol{\delta}_{x,i} - (\boldsymbol{\ell}_i - \boldsymbol{x}_i^{(0)})) - \sum_{i\in[m]} \boldsymbol{w}_{u,i}^{(j)} \log((\boldsymbol{u}_i - \boldsymbol{x}_i^{(0)}) - t\boldsymbol{\delta}_{x,i}) \\
&= \operatorname*{arg\,min}_{t\in\mathbb{R}|\boldsymbol{x}^{(t)}\in\mathcal{X}} t\cdot \tilde{c} - \sum_{i\in[m]\,:\,\boldsymbol{x}_i^{(1)}\neq\boldsymbol{x}_i^{(0)}} \tilde{\boldsymbol{w}}_{\ell,i}^{(j)} \log(t - \tilde{\boldsymbol{\ell}}_i) - \sum_{i\in[m]} \tilde{\boldsymbol{w}}_{u,i}^{(j)} \log(\tilde{\boldsymbol{u}}_i - t) \qquad (2.25)
\end{aligned}$$

where $\tilde{c} \overset{\text{def}}{=} \boldsymbol{c}^\top \boldsymbol{\delta}_x$ and

$$(\tilde{\boldsymbol{\ell}}_i, \tilde{\boldsymbol{u}}_i, \tilde{\boldsymbol{w}}_{\ell,i}^{(j)}, \tilde{\boldsymbol{w}}_{u,i}^{(j)}) \overset{\text{def}}{=} \begin{cases} (\frac{\boldsymbol{\ell}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(1)}-\boldsymbol{x}_i^{(0)}}, \frac{\boldsymbol{u}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(1)}-\boldsymbol{x}_i^{(0)}}, \boldsymbol{w}_{\ell,i}^{(j)}, \boldsymbol{w}_{u,i}^{(j)}) & \text{if } \boldsymbol{\delta}_{x,i} > 0 \\ (\frac{\boldsymbol{u}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(0)}-\boldsymbol{x}_i^{(1)}}, \frac{\boldsymbol{\ell}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(0)}-\boldsymbol{x}_i^{(1)}}, \boldsymbol{w}_{u,i}^{(j)}, \boldsymbol{w}_{\ell,i}^{(j)}) & \text{if } \boldsymbol{\delta}_{x,i} < 0 \end{cases}.$$

Applying Lemma 2.8.7 to (2.25) then yields the result as for all $i \in [m]$ with $\boldsymbol{\delta}_{x,i} > 0$, we have that

$$\frac{1-\tilde{\boldsymbol{\ell}}_i}{0-\tilde{\boldsymbol{\ell}}_i} = \frac{1-\frac{\boldsymbol{\ell}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(1)}-\boldsymbol{x}_i^{(0)}}}{\frac{\boldsymbol{x}_i^{(0)}-\boldsymbol{\ell}_i}{\boldsymbol{x}_i^{(1)}-\boldsymbol{x}_i^{(0)}}} = \frac{\boldsymbol{x}_i^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}_i^{(0)}-\boldsymbol{\ell}_i} \text{ and } \frac{1-\tilde{\boldsymbol{u}}_i}{0-\tilde{\boldsymbol{u}}_i} = \frac{1-\frac{\boldsymbol{u}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(1)}-\boldsymbol{x}_i^{(0)}}}{\frac{\boldsymbol{x}_i^{(0)}-\boldsymbol{u}_i}{\boldsymbol{x}_i^{(1)}-\boldsymbol{x}_i^{(0)}}} = \frac{\boldsymbol{u}_i-\boldsymbol{x}_i^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}_i^{(0)}}$$

and similarly when $\boldsymbol{\delta}_{x,i} < 0$, we have

$$\frac{1-\tilde{\boldsymbol{\ell}}_i}{0-\tilde{\boldsymbol{\ell}}_i} = \frac{1-\frac{\boldsymbol{u}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(0)}-\boldsymbol{x}_i^{(1)}}}{\frac{\boldsymbol{x}_i^{(0)}-\boldsymbol{u}_i}{\boldsymbol{x}_i^{(0)}-\boldsymbol{x}_i^{(1)}}} = \frac{\boldsymbol{u}_i-\boldsymbol{x}_i^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}_i^{(0)}} \text{ and } \frac{1-\tilde{\boldsymbol{u}}_i}{0-\tilde{\boldsymbol{u}}_i} = \frac{1-\frac{\boldsymbol{\ell}_i-\boldsymbol{x}_i^{(0)}}{\boldsymbol{x}_i^{(0)}-\boldsymbol{x}_i^{(1)}}}{\frac{\boldsymbol{x}_i^{(0)}-\boldsymbol{\ell}_i}{\boldsymbol{x}_i^{(0)}-\boldsymbol{x}_i^{(1)}}} = \frac{\boldsymbol{x}_i^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}_i^{(0)}-\boldsymbol{\ell}_i}.$$

□

Leveraging Lemma 2.8.8, we prove the main result of this section, Lemma 2.8.6.

Proof of Lemma 2.8.6. To apply Lemma 2.8.8 for $j \in \{0,1\}$ we define $\overline{\mu} \overset{\text{def}}{=} \frac{1}{2}[\mu^{(0)} + \mu^{(1)}]$, $\overline{\boldsymbol{y}}^{(j)} =$

$\frac{\mu^{(j)}}{\overline{\mu}}\boldsymbol{y}^{(j)}$, $\overline{\boldsymbol{\eta}}^{(j)} = \frac{\mu^{(j)}}{\overline{\mu}}\boldsymbol{\eta}^{(j)} + \overline{\boldsymbol{c}} - \boldsymbol{c}/\overline{\mu}$ and

$$\overline{\boldsymbol{c}}_i = \begin{cases} \boldsymbol{c}_i/\overline{\mu} - \frac{\mu^{(0)}}{\overline{\mu}}\boldsymbol{\eta}_i^{(0)} & \text{if } \phi_i''(\boldsymbol{x}^{(0)}) < \phi_i''(\boldsymbol{x}^{(1)}) \\ \boldsymbol{c}_i/\overline{\mu} - \frac{\mu^{(1)}}{\overline{\mu}}\boldsymbol{\eta}_i^{(1)} & \text{else} \end{cases}.$$

With these definitions, we note that $\mathbf{B}\overline{\boldsymbol{y}}^{(j)} + \overline{\boldsymbol{c}} + \nabla\phi(\boldsymbol{x}^{(j)}) = \overline{\boldsymbol{\eta}}^{(j)}$. Since Lemma 2.8.7 considers only exact minimizers of weighted log barriers, we remove the term $\overline{\beta}^{(j)}$ using the weights as follows, we define $\phi_{\ell,i}(\boldsymbol{x}) \stackrel{\text{def}}{=} -\sum_{i\in[m]}\log(\boldsymbol{x}_i - \boldsymbol{\ell}_i)$, $\phi_{u,i}(\boldsymbol{x}) \stackrel{\text{def}}{=} -\sum_{i\in[m]}\log(\boldsymbol{u}_i - \boldsymbol{x}_i)$, $\phi_\ell(\boldsymbol{x}) = \sum_{i\in[m]}\phi_{\ell,i}(\boldsymbol{x})$, and $\phi_u(\boldsymbol{x}) = \sum_{i\in[m]}\phi_{u,i}(\boldsymbol{x})$. Then, we define the weights

$$\boldsymbol{\alpha}_i^{(j)} = 1 - \frac{\operatorname{sign}([\nabla\phi_{\ell,i}(\boldsymbol{x}^{(j)})]_i)}{|[\nabla\phi_{\ell,i}(\boldsymbol{x}^{(j)})]_i| + |[\nabla\phi_{u,i}(\boldsymbol{x}^{(j)})]_i|}\overline{\boldsymbol{\eta}}_i^{(j)} \text{ and } \boldsymbol{\beta}_i^{(j)} = 1 - \frac{\operatorname{sign}([\nabla\phi_{u,i}(\boldsymbol{x}^{(j)})]_i)}{|[\nabla\phi_{\ell,i}(\boldsymbol{x}^{(j)})]_i| + |[\nabla\phi_{u,i}(\boldsymbol{x}^{(j)})]_i|}\overline{\boldsymbol{\eta}}_i^{(j)}$$

so that $\sum_{i\in[m]}(\boldsymbol{\alpha}_i^{(j)}\nabla\phi_{\ell,i}(\boldsymbol{x}^{(j)}) + \boldsymbol{\beta}_i^{(j)}\nabla\phi_{u,i}(\boldsymbol{x}^{(j)})) = \nabla\phi(\boldsymbol{x}^{(j)}) - \overline{\boldsymbol{\eta}}$ and

$$\mathbf{B}\overline{\boldsymbol{y}}^{(j)} + \overline{\boldsymbol{c}} + \sum_{i\in[m]}(\boldsymbol{\alpha}_i^{(j)}\nabla\phi_{l,i}(\boldsymbol{x}^{(j)}) + \boldsymbol{\beta}_i^{(j)}\nabla\phi_{u,i}(\boldsymbol{x}^{(j)})) = 0.$$

Consequently,

$$\boldsymbol{x}^{(j)} = \operatorname*{arg\,min}_{\boldsymbol{x}\in\mathcal{X}\,|\,\mathbf{B}^\top\boldsymbol{x}=\boldsymbol{d}} \overline{\boldsymbol{c}}^\top\boldsymbol{x} + \sum_{i\in[m]}\boldsymbol{\alpha}_i^{(j)}\phi_{\ell,i}(\boldsymbol{x}) + \sum_{i\in[m]}\boldsymbol{\beta}_i^{(j)}\phi_{u,i}(\boldsymbol{x}).$$

Hence, we can apply Lemma 2.8.7 with $\boldsymbol{w}_{\ell,i}^{(j)} = \boldsymbol{\alpha}_i^{(j)}$ and $\boldsymbol{w}_{u,i}^{(j)} = \boldsymbol{\beta}_i^{(j)}$ provided $\frac{7}{8} \le \boldsymbol{w}_{\ell,i}^{(j)} \le \frac{8}{7}$ and $\frac{7}{8} \le \boldsymbol{w}_{u,i}^{(j)} \le \frac{8}{7}$ for all $i \in [m]$ and $j \in \{1, 2\}$. To show this, note that

$$\left|\boldsymbol{\alpha}_i^{(j)} - 1\right| \le \frac{|\overline{\boldsymbol{\eta}}_i^{(j)}|}{|\phi_{l,i}'(\boldsymbol{x}^{(j)})| + |\phi_{u,i}'(\boldsymbol{x}^{(j)})|} = \frac{|\overline{\boldsymbol{\eta}}_i^{(j)}|}{\sqrt{\phi_{u,i}''(\boldsymbol{x}^{(j)})} + \sqrt{\phi_{u,i}''(\boldsymbol{x}^{(j)})}} \le \frac{|\overline{\boldsymbol{\eta}}_i^{(j)}|}{\sqrt{\phi_i''(\boldsymbol{x}^{(j)})}} \tag{2.26}$$

where we used the definition of ϕ_u and ϕ_l in the equality. Now, using the definition of $\overline{\boldsymbol{\eta}}_i^{(j)}$, we have

$$|\overline{\boldsymbol{\eta}}_i^{(j)}| = \begin{cases} \overline{\mu}^{-1} \left|\boldsymbol{\eta}_i^{(j)}\mu^{(j)} - \boldsymbol{\eta}_i^{(0)}\mu^{(0)}\right| & \text{if } \phi_i''(\boldsymbol{x}^{(0)}) < \phi_i''(\boldsymbol{x}^{(1)}) \\ \overline{\mu}^{-1} \left|\boldsymbol{\eta}_i^{(j)}\mu^{(j)} - \boldsymbol{\eta}_i^{(1)}\mu^{(1)}\right| & \text{otherwise} \end{cases}$$
$$= \begin{cases} 0 & \text{if } \phi_i''(\boldsymbol{x}^{(j)}) < \phi_i''(\boldsymbol{x}^{(1-j)}) \\ \overline{\mu}^{-1} \left|\boldsymbol{\eta}_i^{(1)}\mu^{(1)} - \boldsymbol{\eta}_i^{(0)}\mu^{(0)}\right| & \text{otherwise} \end{cases}.$$

Hence, we have

$$\frac{|\overline{\boldsymbol{\eta}}_i^{(j)}|}{\sqrt{\phi_i''(\boldsymbol{x}^{(j)})}} \leq \frac{\left|\boldsymbol{\eta}_i^{(1)}\mu^{(1)} - \boldsymbol{\eta}_i^{(0)}\mu^{(0)}\right| / \overline{\mu}}{\max(\sqrt{\phi_i''(\boldsymbol{x}^{(0)})}, \sqrt{\phi_i''(\boldsymbol{x}^{(1)})})}. \tag{2.27}$$

Using that $\mu^{(0)} \approx_{1/32} \mu^{(1)}$ and $\left\|\eta^{(j)}/\sqrt{\phi''(\boldsymbol{x}^{(j)})}\right\|_\infty \leq \frac{1}{32}$, we have that $\frac{|\overline{\boldsymbol{\eta}}_i^{(j)}|}{\sqrt{\phi_i''(\boldsymbol{x}^{(j)})}} \leq \frac{1}{8}$. Hence, (2.26) shows that $|\boldsymbol{w}_{\ell,i}^{(j)} - 1| \leq 1/8$ for all $i \leq m$ and j. The same proof gives the bound of $\boldsymbol{\beta}_i^{(j)}$.

Consequently, Lemma 2.8.7 shows that

$$\sum_{i\in[m]} r\left(\frac{\boldsymbol{x}_i^{(1)} - \boldsymbol{\ell}_i}{\boldsymbol{x}_i^{(0)} - \boldsymbol{\ell}_i}\right) + \sum_{i\in[m]} r\left(\frac{\boldsymbol{u}_i - \boldsymbol{x}_i^{(1)}}{\boldsymbol{u}_i - \boldsymbol{x}_i^{(0)}}\right) \leq 16\left[\left\|\boldsymbol{w}_\ell^{(0)} - \boldsymbol{w}_\ell^{(1)}\right\|_2^2 + \left\|\boldsymbol{w}_u^{(0)} - \boldsymbol{w}_u^{(1)}\right\|_2^2\right].$$

Using (2.26) and (2.27), we have that

$$\begin{aligned} \left\|\boldsymbol{w}_\ell^{(0)} - \boldsymbol{w}_\ell^{(1)}\right\|_2^2 + \left\|\boldsymbol{w}_u^{(0)} - \boldsymbol{w}_u^{(1)}\right\|_2^2 &\leq 8\sum_{i\in[m]} \frac{\overline{\mu}^{-2}(\boldsymbol{\eta}_i^{(1)}\mu^{(1)} - \boldsymbol{\eta}_i^{(0)}\mu^{(0)})^2}{\max(\phi_i''(\boldsymbol{x}^{(0)}), \phi_i''(\boldsymbol{x}^{(1)}))} \\ &\leq 32\overline{\mu}^{-2}\sum_{i\in[m]} \frac{(\boldsymbol{\eta}_i^{(1)}\mu^{(1)} - \boldsymbol{\eta}_i^{(0)}\mu^{(1)})^2 + (\boldsymbol{\eta}_i^{(0)}\mu^{(1)} - \boldsymbol{\eta}_i^{(0)}\mu^{(0)})^2}{\phi_i''(\boldsymbol{x}^{(0)}) + \phi_i''(\boldsymbol{x}^{(1)})} \\ &\leq 64\sum_{i\in[m]} \frac{(\boldsymbol{\eta}_i^{(1)} - \boldsymbol{\eta}_i^{(0)})^2}{\phi_i''(\boldsymbol{x}^{(0)}) + \phi_i''(\boldsymbol{x}^{(1)})} + \frac{1}{2}\overline{\mu}^{-2}m(\mu^{(1)} - \mu^{(0)})^2 \\ &\leq 64\sum_{i\in[m]} \frac{(\boldsymbol{\eta}_i^{(1)} - \boldsymbol{\eta}_i^{(0)})^2}{\phi_i''(\boldsymbol{x}^{(0)}) + \phi_i''(\boldsymbol{x}^{(1)})} + m\left(\frac{\mu^{(1)} - \mu^{(0)}}{\mu^{(0)}}\right)^2. \end{aligned} \tag{2.28}$$

Finally, we note that $\phi_i''(\boldsymbol{x}^{(0)}) \le \max(\frac{\boldsymbol{x}^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}^{(0)}-\boldsymbol{\ell}_i}, \frac{\boldsymbol{u}_i-\boldsymbol{x}^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}^{(0)}})^2 \cdot \phi_i''(\boldsymbol{x}^{(1)})$. Hence, we have

$$\begin{aligned}
\sum_{\phi_i''(\boldsymbol{x}^{(0)})^{1/2} \ge 3\phi_i''(\boldsymbol{x}^{(1)})^{1/2}} \sqrt{\frac{\phi_i''(\boldsymbol{x}^{(0)})}{\phi_i''(\boldsymbol{x}^{(1)})}} &\le \sum_{\max(\frac{\boldsymbol{x}^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}^{(0)}-\boldsymbol{\ell}_i}, \frac{\boldsymbol{u}_i-\boldsymbol{x}^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}^{(0)}}) \ge 3} \max\left\{\frac{\boldsymbol{x}^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}^{(0)}-\boldsymbol{\ell}_i}, \frac{\boldsymbol{u}_i-\boldsymbol{x}^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}^{(0)}}\right\} \\
&\le \sum_{i\in[m]} r\left(\frac{\boldsymbol{x}^{(1)}-\boldsymbol{\ell}_i}{\boldsymbol{x}^{(0)}-\boldsymbol{\ell}_i}\right) + \sum_{i\in[m]} r\left(\frac{\boldsymbol{u}_i-\boldsymbol{x}^{(1)}}{\boldsymbol{u}_i-\boldsymbol{x}^{(0)}}\right) \\
&\le 16\left[\left\|\boldsymbol{w}_\ell^{(0)} - \boldsymbol{w}_\ell^{(1)}\right\|_2^2 + \left\|\boldsymbol{w}_u^{(0)} - \boldsymbol{w}_u^{(1)}\right\|_2^2\right]. \qquad (2.29)
\end{aligned}$$

The result then follows from (2.28) and (2.29). □

2.9 Final Runtime Bound

In this section we show Theorem 2.8.4 which describes how efficiently the data structures we developed in Sections 2.4, 2.6 and 2.7 can implement an IPM step. Our final runtime is then achieved via Theorem 3. Finally, we cite previous work to explain how to get an initial point for the IPM, and how to get a mincost flow after running $\widetilde{O}(\sqrt{m})$ IPM iterations.

2.9.1 Efficient Solution Maintenance

Theorem 2.8.4 (Graph Solution Maintenance). *In the special case that* $\mathbf{B}$ *is the incidence matrix of a m-edge, n-node graph, if* $C_r, C_z = \widetilde{O}(1)$ *and* $\alpha = \widetilde{\Omega}(1)$*, there is a* $(\mathcal{T}_{\text{init}}, \mathcal{T}_{\text{phase}})$*-solution maintainer (Definition 2.8.3) with* $\mathcal{T}_{\text{init}} = \widetilde{O}(m)$ *and* $\mathcal{T}_{\text{phase}} = \widetilde{O}(m + m^{15/16}k^{29/8})$.

We first make several useful definitions. We let $\boldsymbol{x}^{(j)}, \boldsymbol{s}^{(j)}$ be the true iterates after the j-th call to MOVE. Our algorithm will explicitly approximate iterates $\widehat{\boldsymbol{x}}^{(j)}, \widehat{\boldsymbol{s}}^{(j)}$. Using these approximate iterates, the algorithm will output $\overline{\boldsymbol{x}}^{(j)}, \overline{\boldsymbol{s}}^{(j)}$ satisfying the desired update schedule, i.e. at most $\widetilde{O}(2^{2\ell_j}\alpha^{-2})$ coordinates are updated after the j-th call to MOVE. Additionally, call an index j corresponding to the j-th MOVE operation *special* if it occurs immediately following a STARTPHASE operation. The formal construction of $\widehat{\boldsymbol{x}}^{(j)}$ and $\widehat{\boldsymbol{s}}^{(j)}$ is given in the following definition.

Definition 2.9.1 (Approximate iterates). Say there have been j MOVE operations so far. If the next operation is STARTPHASE$(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{s}})$, then set $\widehat{\boldsymbol{x}}^{(j+1)} \leftarrow \tilde{\boldsymbol{x}}$ and $\widehat{\boldsymbol{s}}^{(j+1)} \leftarrow \tilde{\boldsymbol{s}}$. If the next operation is

MOVE operation $(j+1)$, then let $\boldsymbol{\delta_x}, \boldsymbol{\delta_s}$ satisfy

$$\left\|\boldsymbol{\delta_x} - h^{(j+1)}(\mathbf{I} - \mathbf{P}_{j+1})\boldsymbol{v}^{(j+1)}\right\|_\infty \le \epsilon \quad \text{and} \quad \left\|\boldsymbol{\delta_s} - h^{(j+1)}\mathbf{P}_{j+1}\boldsymbol{v}^{(j+1)}\right\|_\infty \le \epsilon, \tag{2.30}$$

with $\boldsymbol{\delta_x}, \boldsymbol{\delta_s}$ supported on $O(\epsilon^{-2})$ coordinates. If the previous operation was Move, define and let $\widehat{\boldsymbol{x}}^{(j+1)} \leftarrow \widehat{\boldsymbol{x}}^{(j)} + \mathbf{W}_{j+1}^{1/2}\boldsymbol{\delta_x}$ and $\widehat{\boldsymbol{s}}^{(j+1)} \leftarrow \widehat{\boldsymbol{s}}^{(j)} + \mathbf{W}_{j+1}^{-1/2}\boldsymbol{\delta_s}$. Otherwise, if the previous operation was STARTPHASE define $\widehat{\boldsymbol{x}}^{(j+1)} \leftarrow \widehat{\boldsymbol{x}}^{(j+1)} + \mathbf{W}_{j+1}^{1/2}\boldsymbol{\delta_x}$ and $\widehat{\boldsymbol{s}}^{(j+1)} \leftarrow \widehat{\boldsymbol{s}}^{(j+1)} + \mathbf{W}_{j+1}^{-1/2}\boldsymbol{\delta_s}$ (so that we redefine $\widehat{\boldsymbol{x}}^{(j+1)}, \widehat{\boldsymbol{s}}^{(j+1)}$).

We show that the $\widehat{\boldsymbol{x}}^{(j)}$ are slowly changing, except potentially at special indices. This is because $\|\boldsymbol{\delta_x}\|_2 = O(1)$ as it is supported on $\widetilde{O}(\epsilon^{-2})$ nonzeros and $h^{(j+1)}\|\boldsymbol{v}^{(j+1)}\|_2 \le 1$.

We now argue that $\boldsymbol{\delta_x}$ and $\boldsymbol{\delta_s}$ can be computed efficiently.

Lemma 2.9.2 (Computation of $\boldsymbol{\delta_x}, \boldsymbol{\delta_s}$)**.** *In the context of Theorem 2.8.4, there is an operation that computes $\boldsymbol{\delta_x}, \boldsymbol{\delta_s}$ satisfying* (2.30) *in average amortized time $\widetilde{O}(m^{15/16}\epsilon^{-7/8})$ and succeeds with high probability against adaptive adversaries.*

Proof. We first write $h^{(j+1)}(\mathbf{I} - \mathbf{P}_{j+1})\boldsymbol{v}^{(j+1)} = h^{(j+1)}\boldsymbol{v}^{(j+1)} - h^{(j+1)}\mathbf{P}_{j+1}\boldsymbol{v}^{(j+1)}$, and handle both parts separately up to error $\epsilon/2$. The first part can be trivially handled, as it can be explicitly maintained in time proportional to the number of changes in $\boldsymbol{v}^{(j)}$, and $\|h^{(j+1)}\boldsymbol{v}^{(j+1)}\|_2 \le 1$. For the second part, we first call the dynamic LOCATOR (Theorem 2.6.8) to get a set S of size $O(\epsilon^{-2})$. Then we call the dynamic EVALUATORS (Theorem 2.6.7) wrapped inside Theorem 2.7.2 with $\epsilon \leftarrow \epsilon/(C\log^2 n)$ on S by calling QUERY() on S. The algorithm for $\boldsymbol{\delta_s}$ follows exactly as the second term. Also, $\boldsymbol{\delta_x}, \boldsymbol{\delta_s}$ are supported on $|S| = O(\epsilon^{-2})$ coordinates by Theorem 2.7.2.

Correctness follows directly from the guarantees of Theorems 2.6.7, 2.6.8 and 2.7.2. It suffices to analyze the amortized runtime. We focus on the cost of applying Theorem 2.6.7 inside Theorem 2.7.2, as the cost of Theorem 2.6.8 is less. Let $\delta_i \stackrel{\text{def}}{=} 2^{-i}$ so that the i-th EVALUATOR is run with accuracy $\epsilon_i \stackrel{\text{def}}{=} \delta_i\epsilon$ in Theorem 2.7.2. Let β_i be the terminal size parameter for the i-th EVALUATOR.

There are two possible ways to run the i-th EVALUATOR. Either it pays $\widetilde{O}(m)$ time per call to solve a Laplacian exactly (while this algorithm is randomized, we can hide randomness by adding polynomially small noise that is larger than the error we solve the Laplacian to [88]) or applies Theorem 2.6.7. Let us calculate the runtime of the latter approach. After $\beta_i m$ edge updates or marking, the data structure must re-initialize. Thus, after T MOVE updates, because $C_z = \widetilde{O}(1)$, there are at most $\widetilde{O}(T^2 + T\epsilon^{-2})$ total edges we have queried or updated: $\widetilde{O}(T^2)$ from updates, and $\widetilde{O}(T\epsilon^{-2})$ from the set S returned by LOCATOR. We assume for now that the $\widetilde{O}(T^2)$ term dominates – thus the data structure must reinitialize every $\sqrt{\beta_i m}$ iterations, where each initialization costs $\widetilde{O}(m\beta_i^{-2}\epsilon_i^{-2})$ time. Thus the amortized reinitialization time per Move is

$$\widetilde{O}(m\beta_i^{-2}\epsilon_i^{-2}/\sqrt{\beta_i m}) = \widetilde{O}(\sqrt{m}\beta_i^{-5/2}\epsilon^{-2}\delta_i^{-2}).$$

By Theorem 2.7.2, the i-th EVALUATOR is queried with probability $O(\delta_i)$, hence the expected query time is $\widetilde{O}(\beta_i m\epsilon_i^{-2}\delta_i) = \widetilde{O}(\beta_i m\epsilon^{-2}\delta_i^{-1})$ by Theorem 2.6.7 QUERY(), or $\widetilde{O}(\delta_i m)$ if EVALUATOR simply solves a Laplacian every iteration. Thus, the amortized runtime for the i-th EVALUATOR is

$$\widetilde{O}\left(\min\left\{\delta_i m, \beta_i m\epsilon^{-2}\delta_i^{-1} + \sqrt{m}\beta_i^{-5/2}\epsilon^{-2}\delta_i^{-2}\right\}\right).$$

For the choice $\beta_i = m^{-1/7}\delta_i^{-2/7}$, this becomes

$$\widetilde{O}\left(\min\left\{\delta_i m, m^{6/7}\epsilon^{-2}\delta_i^{-9/7}\right\}\right).$$

This is maximized when the two expressions are equal at $\delta_i = m^{-1/16}\epsilon^{-7/8}$, yielding a runtime of $\widetilde{O}(m^{15/16}\epsilon^{-7/8})$ as desired. Finally, note that this means that $\epsilon \geq m^{-1/14}$ or the previous runtime is trivial. All $\beta_i \geq m^{-1/7}$, so $T\epsilon^{-2} \leq T^2$ for the choice $T = \sqrt{\beta_i m} \geq m^{3/7} > \epsilon^{-2}$, so the $\widetilde{O}(T^2)$ term dominated earlier, as desired. □

We now show that $\widehat{\boldsymbol{x}}^{(j)}$ and $\widehat{\boldsymbol{s}}^{(j)}$ are close to $\boldsymbol{x}^{(j)}, \boldsymbol{s}^{(j)}$.

Lemma 2.9.3. *For $\epsilon = \frac{\alpha}{10C_r k^3}$ and $\widehat{\boldsymbol{x}}^{(j)}, \widehat{\boldsymbol{s}}^{(j)}$ defined in Definition 2.9.1, $\left\|\mathbf{W}_j^{-1/2}\left(\widehat{\boldsymbol{x}}^{(j)} - \boldsymbol{x}^{(j)}\right)\right\|_\infty \leq \alpha/10$ and $\left\|\mathbf{W}_j^{1/2}\left(\widehat{\boldsymbol{s}}^{(j)} - \boldsymbol{s}^{(j)}\right)\right\|_\infty \leq \alpha/10$.*

Proof. It suffices to analyze j between STARTPHASEs, as $\widehat{\boldsymbol{x}}^{(j)}, \widehat{\boldsymbol{s}}^{(j)}$ and $\boldsymbol{x}^{(j)}, \boldsymbol{s}^{(j)}$ are both set to $\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{s}}$ during a STARTPHASE. Over L steps between STARTPHASEs (from j_1 to $j_2 = j_1 + L$), we have that $\left\|\mathbf{W}_{j_2}^{-1/2}\left(\widehat{\boldsymbol{x}}^{(j_2)}-\boldsymbol{x}^{(j_2)}\right)\right\|_\infty$ is at most

$$\begin{aligned}
&\left\|\mathbf{W}_{j_2}^{-1/2}\sum_{j\in[j_1,j_2)}\mathbf{W}_j^{1/2}\left(\boldsymbol{\delta}_{\boldsymbol{x}}^{(j)}-h^{(j+1)}(\mathbf{I}-\mathbf{P}_{j+1})\boldsymbol{v}^{(j+1)}\right)\right\|_\infty\\
\le\;& \max_{j\in[j_1,j_2]}\sqrt{\|\boldsymbol{w}_j/\boldsymbol{w}_{j_2}\|_\infty}\left\|\sum_{j\in[j_1,j_2)}\boldsymbol{\delta}_{\boldsymbol{x}}^{(j)}-h^{(j+1)}(\mathbf{I}-\mathbf{P}_{j+1})\boldsymbol{v}^{(j+1)}\right\|_\infty\\
\overset{(i)}{\le}\;& C_rL^2\epsilon L=\epsilon C_rL^3\le\alpha/10,
\end{aligned}$$

where (i) follows from the guarantee that of Definition 2.8.3 that $\sqrt{\left\|\boldsymbol{w}_{(j_2)}/\boldsymbol{w}_{j_1}\right\|_\infty}\le rL^2$ and (2.30). The bound on the error for $\boldsymbol{s}^{(j)}$ follows similarly. $\square$

Proof of Theorem 2.8.4. We show this by carefully defining $\overline{\boldsymbol{x}}^{(j)},\overline{\boldsymbol{s}}^{(j)}$ given $\widehat{\boldsymbol{x}}^{(j)},\widehat{\boldsymbol{s}}^{(j)}$. We mimic the approach based on binary expansions given in previous works on robust IPMs, for example [92, Theorem 8]. Precisely, we first calculate $\sum_{j'=j-2^{\ell_j}}^{j}\boldsymbol{\delta}_{\boldsymbol{x}}^{(j')}$, i.e. the sum of errors in the last 2^{ℓ_j} steps. If this exceeds $\frac{\alpha}{100\log n}$, then we set $\overline{\boldsymbol{x}}^{(j)}\leftarrow\widehat{\boldsymbol{x}}^{(j)}$, otherwise we set $\overline{\boldsymbol{x}}^{(j)}\leftarrow\overline{\boldsymbol{x}}^{(j-1)}$ (no change). We do the same for $\overline{\boldsymbol{s}}^{(j)}$. Now, the bounds on number of changes follows from the bounds $\|\boldsymbol{\delta}_{\boldsymbol{x}}\|_2\le O(1)$ and that $\|\mathbf{W}_j^{-1/2}(\boldsymbol{x}-\tilde{\boldsymbol{x}})\|_2\le 1$ in StartPhase. More precisely, over 2^{ℓ_j} steps, only $O(2^{2\ell_j}\alpha^{-2}\log^2 n)$ could change by $\frac{\alpha}{100\log n}$, because every step satisfies $\|\boldsymbol{\delta}_{\boldsymbol{x}}\|_2\le 1$ or $\|\mathbf{W}_j^{-1/2}(\boldsymbol{x}-\tilde{\boldsymbol{x}})\|_2\le 1$. This completes the proof of the number of changes.

We now claim that $\left\|\mathbf{W}_j^{-1/2}\left(\overline{\boldsymbol{x}}^{(j)}-\widehat{\boldsymbol{x}}^{(j)}\right)\right\|_\infty\le\alpha/10$, so $\left\|\mathbf{W}_j^{-1/2}\left(\overline{\boldsymbol{x}}^{(j)}-\boldsymbol{x}^{(j)}\right)\right\|_\infty\le\alpha$ by combining with Lemma 2.9.3 for $\epsilon=\frac{\alpha}{10C_rk^3}=\tilde{\Theta}(1/k^3)$ as $C_r=\widetilde{O}(1)$. This claim follows from the same argument as [92, Theorem 8]: each interval $[j_1,j_2]$ can be split into $\log n$ intervals contained in intervals $[j-2^{\ell_j},j]$ for $j\le j_2$. Each of these has at most $\alpha/(100\log n)$ error, so the total error is at most $\alpha/(100\log n)\cdot\log n\le\alpha/10$.

Finally, we must calculate the runtime of $\mathcal{T}_{\text{phase}}$. The first cost is $\widetilde{O}(m)$ (eg. for reading $\tilde{\boldsymbol{x}},\tilde{\boldsymbol{s}}$). The second cost is calling Lemma 2.9.2 k times (as there are at most k Move operations between StartPhase). For our choice $\epsilon=\tilde{\Theta}(1/k^3)$, the total time for this is $\widetilde{O}(m^{15/16}\epsilon^{-7/8}k)=\widetilde{O}(m^{15/16}k^{29/8})$ as desired. $\square$

2.9.2 Initial Point, Final Point, and Proof of Main Theorem

It is standard to get an initial μ-centered feasible pair $(\boldsymbol{x}, \boldsymbol{s})$ for path parameter $\mu = \mu_{\text{start}} \geq (mU)^{O(1)}$. Additionally, given a μ-centered feasible pair $(\boldsymbol{x}, \boldsymbol{s})$ for path parameter $\mu = \mu_{\text{end}} \leq (mU)^{-O(1)}$ we can recover a high-accuracy mincost flow (and hence round to an exact solution).

Lemma 2.9.4 ([24, Lemma 7.5, Lemma 7.8])**.** *Given a graph $G = (V, E)$ and mincost flow instance with demand $\boldsymbol{d} \in [-U, \ldots, U]^V$, costs $\boldsymbol{c} \in [-U, \ldots, U]^E$, and capacities $\boldsymbol{\ell}, \boldsymbol{u} \in [-U, \ldots, U]^E$, we can build a mincost flow instance on a graph G' with at most $O(m)$ edges with demands, costs, and capacities bounded by $\operatorname{poly}(mU)$. Additionally, we can construct a μ_{start}-centered pair $(\boldsymbol{f}, \boldsymbol{s})$ on G' for $\mu_{\text{start}} = \operatorname{poly}(mU)$. Additionally, given a $1/\operatorname{poly}(mU)$-accurate mincost flow on G' we can recover an exact mincost flow on G in time $\widetilde{O}(m \log U)$.*

Proof of Theorem 2.1.1. We apply Lemma 2.9.4 to get an initial point for Algorithm 10. Then, we run Algorithm 10 and round to an exact mincost flow using Lemma 2.9.4. This succeeds by Theorem 3 in time

$$\widetilde{O}\left(\left(\mathcal{T}_{\text{init}} + \frac{\sqrt{m}}{k}(\mathcal{T}_{\text{phase}} + \mathcal{T}_{\text{approx}})\right) \log\left(\frac{\mu_{\text{start}}}{\mu_{\text{end}}}\right)\right) = \widetilde{O}\left(\left(\mathcal{T}_{\text{init}} + \frac{\sqrt{m}}{k}(\mathcal{T}_{\text{phase}} + \mathcal{T}_{\text{approx}})\right) \log U\right). \tag{2.31}$$

It suffices to plug in the values of $\mathcal{T}_{\text{init}}, \mathcal{T}_{\text{phase}}$ from Theorem 2.8.4 and $\mathcal{T}_{\text{approx}}$ from Theorem 2.

We take $k = m^{1/58}$ so $\mathcal{T}_{\text{phase}} = \widetilde{O}(m^{15/16}k^{29/8} + m) = \widetilde{O}(m)$ by Theorem 2.8.4. Also by Theorem 2, $\mathcal{T}_{\text{approx}} = \widetilde{O}(m + zr^2k^3/\epsilon^2)$ for $z = \widetilde{O}(k^2), r = \widetilde{O}(k^4), \epsilon = \widetilde{\Omega}(1)$, so $\mathcal{T}_{\text{approx}} = \widetilde{O}(m + zr^2k^3/\epsilon^2) = \widetilde{O}(m + k^{13}) = \widetilde{O}(m)$ for $k = m^{1/58}$. Thus, the expression in (2.31) evaluates to $\widetilde{O}(m^{3/2-1/58} \log U)$ as desired. □

CHAPTER 3
NESTED DISSECTION MEETS IPMS: PLANAR MIN-COST FLOW IN NEARLY-LINEAR TIME

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 is based on interior point methods (IPMs) and works for general sparse graphs in $O(n^{1.5}\text{poly}(\log n))$ time [Daitch-Spielman, STOC'08].[1]

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 $O(n \cdot \text{poly}(\log n))$. Using parallel Laplacian solvers in the data structure, the algorithm has $\widetilde{O}(\sqrt{m})$ depth.

Our results immediately extend to all families of separable graphs.

3.1 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 [124] 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. 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 [125, 126], planar graphs by Borradaile-Klein [127], and finally bounded genus graphs by Chambers-Erickson-Nayyeri [128]. However, for the more general

[1]A preliminary version of this work appeared in SODA 2022.

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 3.1.1 (Main result)**.** *Let $G = (V, E)$ be a directed planar graph with n vertices and m edges. 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 $\widetilde{O}(n \log M)$* [2] *expected time.*

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 the input graph 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 α-*separable* if there are constants $0 < b < 1$ and $c > 0$ such that every graph in $\mathcal{C}$ with n vertices and m edges has a balanced vertex separator with at most cm^{α} vertices, and both components obtained after removing the separator have at most bm edges. Then, our algorithm generalizes as follows:

Corollary 3.1.2 (Separable min-cost flow)**.** *Let $\mathcal{C}$ be an 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 convex function s. Given a graph $G \in \mathcal{C}$ with n vertices and m edges, 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 $\widetilde{O}((m + m^{1/2+\alpha}) \log M + s(m))$ 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 [132], and using geometric

[2]Throughout the paper, we use $\widetilde{O}(f(n))$ to denote $O(f(n) \log^{O(1)} f(n))$.

structures to find faster solvers for structured linear systems provided foundations of Laplacian algorithms as well as combinatorial scientific computing [133, 134]. 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 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.

3.1.1 Previous work

The min-cost flow problem is well studied in both structured graphs and general graphs. Table 3.1 summarizes the best algorithms for different settings prior to this work.

Table 3.1: 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, and M is the maximum of edge capacity and cost value. After the preliminary version of this work was published at SODA 2022, the best weakly polytime algorithm was improved to $\widetilde{O}(m^{1+o(1)} \log^2 M)$ by [141].

Min-cost flow	Time bound	Reference
Strongly polytime	$O(m^2 \log n + mn \log^2 n)$	[135]
Weakly polytime	$\widetilde{O}((m + n^{3/2}) \log^2 M)$	[24]
Unit-capacity	$m^{\frac{4}{3}+o(1)} \log M$	[73]
Planar graph	$\widetilde{O}(n \log M)$	**this paper**
Unit-capacity planar graph	$O(n^{4/3} \log M)$	[136]
Graph with treewidth τ	$\widetilde{O}(n\tau^2 \log M)$	[84]
Outerplanar graph	$O(n \log^2 n)$	[137]
Unidirectional, bidirectional cycle	$O(n)$, $O(n \log n)$	[138]

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 [139, 140]. For the approximate max-flow problem, we refer the reader to the recent papers [19, 20, 75, 77, 78, 106].

To understand the recent progress, we view the max-flow problem as finding a unit s, t-flow with minimum ℓ_∞-norm, and the shortest path problem as finding a 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 problems, (shortest path) since the latter can be solved efficiently. This changed with the celebrated work of Spielman and Teng, which showed how to find electrical flows (ℓ_2-minimizing unit s, t-flow) in nearly-linear time [13]. Since the ℓ_2-norm is closer to ℓ_∞ than ℓ_1, this gives a more powerful primitive for the max-flow problem. In 2008, Daitch and Spielman demonstrated that one could apply interior point methods (IPMs) to reduce min-cost flow to roughly $\sqrt{m}$ electrical flow computations. This follows from the fact that IPMs take $\widetilde{O}(\sqrt{m})$ iterations and each iteration requires solving an electrical flow problem, which can now be solved in $\widetilde{O}(m)$ time due to the work of Spielman and Teng. Consequently, they obtained an algorithm with a $\widetilde{O}(m^{3/2} \log M)$ runtime

[9]. 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 [21]. It was later improved and generalized to $O(m^{4/3+o(1)} \log M)$ [73] for the min-cost flow problem. Kathuria et al. [22] gave a similar runtime of $O(m^{4/3+o(1)}U^{1/3})$ 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 [70].

For general capacities, the runtime has recently been improved to $\widetilde{O}((m+n^{3/2}) \log^2 M)$ for min-cost flow on dense graphs [24], and $\widetilde{O}(m^{\frac{3}{2}-\frac{1}{328}} \log M)$ for max-flow on sparse graphs [25]. These algorithms focus on decreasing the per-iteration cost of IPMs by dynamically maintaining electrical flows. After the preliminary version of this work was accepted to SODA 2022, [26] gave a runtime of $\widetilde{O}(m^{\frac{3}{2}-\frac{1}{58}} \log^2 M)$ for general min-cost flow following the dynamic electrical flow framework. Most recently, [141] improved the runtime for general min-cost flow to $\widetilde{O}(m^{1+o(1)} \log^2 M)$ by solving a sequence of approximate undirected minimum-ratio cycles.

Max-flow on planar graphs. The planar max-flow problem has an equally long history. We refer the reader to the thesis [142] for a detailed exposition. In the seminal work of Ford and Fulkerson 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)[124]. 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 [143]. In this setting, Hassin also showed that the max-flow can be computed using shortest-path distances in the planar dual in $O(n \log n)$ time [144]. Building on Hassin's work, the current best runtime is $O(n)$ by Henzinger, Klein, Rao, and Subramanian [145].

For undirected planar graphs, Reif first gave an $O(n \log^2 n)$ time algorithm for finding the max-flow value [125]. Hassin and Johnson then showed how to compute the flow in the same runtime [126]. The current best runtime is $O(n \log \log n)$ by Italiano, Nussbaum, Sankowski, and Wulff-Nilsen [146].

For general planar graphs, Weihe gave the first $O(n \log n)$ time algorithm, assuming the graph satisfies certain connectivity conditions [147]. Later, Borradaile and Klein gave an $O(n \log n)$ time algorithm for any planar graph [127].

The multiple-source multiple-sink version of max-flow is considered much harder 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 [148]. This was then improved to $O(n \log^3 n)$ in [149].

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 [128]. Miller and Peng gave an $\widetilde{O}(n^{6/5})$-time algorithm for approximating undirected max-flow for the class of $O(\sqrt{n})$-separable graphs [150], although this is superseded by the previously mentioned works for general graphs [20, 75].

Min-cost flow on planar graphs. Imai and Iwano gave a $O(n^{1.594} \log M)$ time algorithm for min-cost flow for the more general class of $O(\sqrt{n})$-separable graphs [151]. 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, [152, 153] 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 [136]. Currently, bounded treewidth graphs is the only graph family we know that admits min-cost flow algorithms that run in nearly-linear time [84].

3.1.2 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 [154]. 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 [9]. Intuitively, $\Omega(n^{3/2})$ is a natural runtime barrier for IPM-based methods, since they require $\Omega(\sqrt{n})$ iterations, each computing 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 [19] 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 [**AdilKPS19**, 70, 155, 156], 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 [8, 21, 22, 73, 157], 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 [**van2020deterministic**, 89, 92, 94, 158]. 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 edges with 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 (robust) 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 $\widetilde{O}(n)\boldsymbol{l}\widetilde{O}(m)$ time cost per step [92]. 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 [25, 26]. Unfortunately, this still requires $m^{1-\frac{1}{58}}$ time per iteration.

Finally, we note that [84] 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 vertex separator set. 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 $\Omega(n)$ time per step.

Our paper follows the approach in [84] 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 [22, 73], resulting in super-linear time and polynomial dependence on capacities, rather than logarithmic.

3.1.3 Our approaches

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

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

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 min-cost circulation problem and the dual is a variant of the min-cut problem. Our algorithm for min-cost flow is composed of a novel application of IPM (Section 3.2.1) and new data structures (Section 3.2.3). 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 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 [89]). 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 [84] in a black-box manner. In Section 3.2.1, 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 [133, 159]. By recursively partitioning the graph into edge-disjoint subgraphs (i.e. regions) using balanced vertex separators, we can construct a hierarchical decomposition of a planar graph

G which is called a *separator tree* [160]. 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, an $O(\sqrt{|H|})$-sized 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 [133, 159] essentially replace each region by a graph involving only its boundary vertices, in a bottom-up manner. For planar 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 [160].

Technique 1: Approximate Nested Dissection and Lazy Propagation Our representation of the Laplacian inverse, and in turn the projection matrix, 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 [150]. However, that work pre-dated (and in some sense motivated) subsequent works on nearly-linear time approximations of Schur complements on general graphs [118, 119, 161]. Re-incorporating these sparsified algorithms gives runtime 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 size 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 approximate Schur complements and the slack and flow projection matrices all utilize this idea. For example, to maintain the Schur complement of a region H onto its boundary (which is required in implementating the IPM 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 H. Thus, to update an edge, the path in the separator tree from the leaf node containing the edge to the root is visited. To update multiple edges in a batch, each node in the union of the tree paths is visited. The runtime is nearly linear in the total number of boundary vertices of all nodes (regions) in the union. For K edges being updated, the runtime is bounded by $\widetilde{O}(\sqrt{mK})$. Step i of our IPM algorithm takes $\widetilde{O}(\sqrt{mK_i})$ time, where K_i is the number of coordinates changed in $\mathbf{W}$ and $\boldsymbol{v}$ in the step. Such a recursive approximate Schur complement structure was used in [162], 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 $\mathbf{W}$ and $\boldsymbol{v}$ is bounded by $O(t^2)$. This directly gives $\sum_{i=1}^{\widetilde{O}(\sqrt{m})} K_i = m$ and thus a total runtime 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 runtime, 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 $l \le \ell_i$, 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 runtime of the data structures, this gives an $\widetilde{O}(m)$ overall runtime.

Technique 3: Maintaining feasibility via two projections. A major difficulty in the IPM 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 $\boldsymbol{v}$ 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 distinct 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.

3.2 Overview

In this section, we give formal statements of the main theorems proved in the paper, along with the proof for our main result. We provide a high-level explanation of the algorithm, sometimes using a simplified setup.

The main components of this paper are: the IPM from [84] (Section 3.2.1); a data structure to maintain a collection of Schur complements via nested dissection of the graph (Section 3.2.2); abstract data structures to maintain the solutions $\boldsymbol{s}, \boldsymbol{f}$ implicitly, notably using an abstract tree operator (Section 3.2.3); a sketching-based data structure to maintain the approximations $\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{f}}$ needed in the IPM (Section 3.2.4); and finally, the definition of the tree operators for slack and flow corresponding to the IPM projection matrices onto their respective feasible subspaces, along with the complete IPM data structure for slack and flow (Sections 3.2.5 and 3.2.6).

We extend our result to α-separable graphs in Section 3.9.

3.2.1 Robust interior point method

In this subsection, we explain the robust interior point method developed in [84], which is a refinement of the methods in [**van2020deterministic**, 89]. Although there are many other robust interior point methods, we simply refer to this method as RIPM. Consider a linear program of the form[3]

$$\min_{\boldsymbol{f}\in\mathcal{F}} \boldsymbol{c}^\top \boldsymbol{f} \quad \text{where} \quad \mathcal{F} = \{\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{b},\ \boldsymbol{\ell} \le \boldsymbol{f} \le \boldsymbol{u}\} \tag{3.1}$$

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 ($t\boldsymbol{g}0$) to the optimal solution ($t = 0$):

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

[3]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.

where the term ϕ controls how close the flow $\boldsymbol{f}_i$ can be to the capacity constraints $\boldsymbol{u}_i$ and $\boldsymbol{\ell}_i$. 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

$$\begin{aligned} \mu^t(\boldsymbol{f}, \boldsymbol{s}) &\stackrel{\text{def}}{=} \boldsymbol{s}/t + \nabla\phi(\boldsymbol{f}) = \boldsymbol{0} \\ (\boldsymbol{f}, \boldsymbol{s}) &\in \mathcal{F} \times \mathcal{S} \end{aligned} \tag{3.2}$$

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

$$\|\gamma^t(\boldsymbol{f}, \boldsymbol{s})\|_\infty \le \frac{1}{C \log m} \text{ where } \gamma^t(\boldsymbol{f}, \boldsymbol{s})_i = \frac{\mu^t(\boldsymbol{f}, \boldsymbol{s})_i}{(\nabla^2\phi(\boldsymbol{f}))_{ii}^{1/2}}, \tag{3.3}$$

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 $\boldsymbol{\ell}$ 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. [84] 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 11 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

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

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

Algorithm 11 Robust Interior Point Method from [84]

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

   ▷ Modify the linear program and obtain an initial $(x, s)$ for modified linear program
4:     Let $t = 2^{21} m^5 \cdot \frac{LR}{128} \cdot \frac{R}{r}$
5:     Compute $\boldsymbol{f}_c = \arg\min_{\boldsymbol{\ell}\le\boldsymbol{f}\le\boldsymbol{u}} \boldsymbol{c}^\top\boldsymbol{f} + t\phi(\boldsymbol{f})$ and $\boldsymbol{f}_\circ = \arg\min_{\mathbf{B}^\top\boldsymbol{f}=\boldsymbol{b}} \|\boldsymbol{f} - \boldsymbol{f}_c\|_2$
6:     Let $\boldsymbol{f} = (\boldsymbol{f}_c, 3R + \boldsymbol{f}_\circ - \boldsymbol{f}_c, 3R)$ and $\boldsymbol{s} = (-t\nabla\phi(\boldsymbol{f}_c), \frac{t}{3R+\boldsymbol{f}_\circ-\boldsymbol{f}_c}, \frac{t}{3R})$
7:     Let the new matrix $\mathbf{B}^{\text{new}} \overset{\text{def}}{=} [\mathbf{B}; \mathbf{B}; -\mathbf{B}]$, 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
8:     $((\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \boldsymbol{f}^{(3)}), (\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \boldsymbol{s}^{(3)})) \leftarrow$ CENTERING($\mathbf{B}^{\text{new}}, \phi^{\text{new}}, \boldsymbol{f}, \boldsymbol{s}, t, LR$)
9:     $(\boldsymbol{f}, \boldsymbol{s}) \leftarrow (\boldsymbol{f}^{(1)} + \boldsymbol{f}^{(2)} - \boldsymbol{f}^{(3)}, \boldsymbol{s}^{(1)})$

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

13: procedure CENTERING($\mathbf{B}, \phi, \boldsymbol{f}, \boldsymbol{s}, t_{\text{start}}, t_{\text{end}}$)
14:    Let $\alpha = \frac{1}{2^{20}\lambda}$ and $\lambda = 64\log(256m^2)$ where $m$ is the number of rows in $\mathbf{B}$
15:    Let $t \leftarrow t_{\text{start}}$, $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}, \overline{\boldsymbol{s}} \leftarrow \boldsymbol{s}, \bar{t} \leftarrow t$
16:    while $t \ge t_{\text{end}}$ do
17:        Set $t \leftarrow \max((1 - \frac{\alpha}{\sqrt{m}})t, t_{\text{end}})$
18:        Update $h = -\alpha/\|\cosh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}))\|_2$ where $\gamma$ is defined in (3.2)
19:        Update the diagonal weight matrix $\mathbf{W} = \nabla^2\phi(\overline{\boldsymbol{f}})^{-1}$
20:        Update the direction $\boldsymbol{v}$ where $\boldsymbol{v}_i = \sinh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}})_i)$
21:        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
               $\|\boldsymbol{v}^{\|} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\|_2 \le \alpha\|\boldsymbol{v}\|_2,$
               $\|\boldsymbol{v}^{\perp} - (\mathbf{I} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}\|_2 \le \alpha\|\boldsymbol{v}\|_2$    $(\mathbf{P}_{\boldsymbol{w}} \overset{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}^{1/2})$
22:        Implicitly update $\boldsymbol{f} \leftarrow \boldsymbol{f} + h\mathbf{W}^{1/2}\boldsymbol{v}^{\perp}$, $\boldsymbol{s} \leftarrow \boldsymbol{s} + \bar{t}h\mathbf{W}^{-1/2}\boldsymbol{v}^{\|}$
23:        Explicitly maintain $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \le \alpha$ and $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \le \bar{t}\alpha$
24:        Update $\bar{t} \leftarrow t$ if $|\bar{t} - t| \ge \alpha\bar{t}$
25:    end while
26:    return $(\boldsymbol{f}, \boldsymbol{s})$
27: end procedure
```

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 26.

Theorem 3.2.1. *Consider the linear program*

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

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* $\boldsymbol{\ell} + r \leq \boldsymbol{f}_\circ \leq \boldsymbol{u} - r$.[4] *Let* $L = \|\boldsymbol{c}\|_2$ *and* $R = \|\boldsymbol{u} - \boldsymbol{\ell}\|_2$. *For any* $0 < \epsilon \leq 1/2$, *the algorithm RIPM (Algorithm 11) finds* $\boldsymbol{f}$ *such that* $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{b}$, $\boldsymbol{\ell} \leq \boldsymbol{f} \leq \boldsymbol{u}$ *and*

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

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* $\overline{t}$ *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 18 to Line 20 takes* $O(K)$ *time in total, where* K *is the total number of coordinate changes in* $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$.

[4]For any vector $\boldsymbol{v}$ and scalar x, we define $\boldsymbol{v} + x$ to be the vector obtained by adding x to each coordinate of $\boldsymbol{v}$. We define $\boldsymbol{v} - x$ to be the vector obtained by subtracting x from each coordinate of $\boldsymbol{v}$.

Proof. The number of steps follows from Theorem A.1 in [163], 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 18 to Line 20 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 involves the computation of projection matrices required for the RIPM. Recall from the definition of $\mathbf{P}_{\boldsymbol{w}}$ in Algorithm 11, 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

$$\mathbf{P}_{\boldsymbol{w}} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\mathbf{B}^\top\mathbf{W}^{1/2}. \tag{3.4}$$

Lemma 3.2.2. *To implement Line 21 in Algorithm 11, it suffices to find an approximate slack projection matrix $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ satisfying $\left\|\left(\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\right)\boldsymbol{v}\right\|_2 \leq \alpha\|\boldsymbol{v}\|_2$ and $\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} \in \mathrm{Range}(\mathbf{B})$; and an approximation flow projection matrix $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}$ satisfying $\left\|\left(\widetilde{\mathbf{P}}'_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\right)\boldsymbol{v}\right\|_2 \leq \alpha\|\boldsymbol{v}\|_2$ and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$.*

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

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

3.2.2 Nested dissection and approximate Schur complements

In this subsection, we discuss nested dissection and the corresponding Schur complements, and explain how it relates to our goal of finding the approximate projection matrices for Lemma 3.2.2.

As we will discuss later in the main proof, our LP formulation for the IPM uses a *modified planar* *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. We may add these two vertices to any relevant sets defined in nested dissection without changing the overall complexity. As such, we can apply nested dissection as we would for planar graphs.

We first illustrate the key ideas using a two-layer nested dissection scheme. By the well-known planar separator theorem [164], a planar graph G can be decomposed into two edge-disjoint (not vertex-disjoint) subgraphs H_1 and H_2 called *regions*, such that each subgraph has at most $2n/3$ vertices. 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 $F_{H_i} = V(H_i) \setminus \partial H_i$ denote the remaining interior vertices *eliminated* at region H_i.

Let $C = \partial H_1 \cup \partial H_2$ denote the union of the boundaries, and let $F = F_{H_1} \cup F_{H_2}$ be the disjoint union of the two interior sets. Note that C is a *balanced vertex separator* of G, with size

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

Furthermore, F and C give a natural partition of the vertices of G. Using block Cholesky decomposition, we can now write[5]

$$\mathbf{L}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{L}_{F,C}{\mathbf{L}_{F,F}}^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} {\mathbf{L}_{F,F}}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C)^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{L}_{F,C}{\mathbf{L}_{F,F}}^{-1} & \mathbf{I} \end{bmatrix}, \tag{3.5}$$

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

The IPM in Algorithm 11 involves 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 $\Omega(|C|^2) = \Omega(n)$ time, which is already too expensive. Our key idea here is to maintain an approximate Schur complement, which is of a smaller size based on the graph decomposition, and can be maintained in amortized $\sqrt{n}$ time per

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

step throughout the IPM.

Let $\mathbf{L}[H_i]$ denote the weighted Laplacian of the region H_i for $i = 1, 2$. Since these regions are edge-disjoint, we can write the Laplacian $\mathbf{L}$ as the sum

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

Based on the graph decomposition, we have the Schur complement decomposition

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

This decomposition allows us to localize edge weight updates. Namely, if the weight of edge e is updated, and e is contained in region H_i, we only need to recompute the single Schur complement term for H_i, rather than both terms in the sum.

For the appropriate projection matrices in the IPM, it further suffices to maintain a sparse *approximate Schur complement* $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C) \approx \mathbf{Sc}(\mathbf{L}[H_i], C)$ for each region H_i rather than the exact. Then, the approximate Schur complement of $\mathbf{L}$ on C is given by

$$\widetilde{\mathbf{Sc}}(\mathbf{L}, C) \stackrel{\text{def}}{=} \widetilde{\mathbf{Sc}}(\mathbf{L}[H_1], C) + \widetilde{\mathbf{Sc}}(\mathbf{L}[H_2], C). \tag{3.6}$$

Each term $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)$ can be computed in time nearly-linear in the size of H_i. Furthermore, $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)$ is supported only on the vertex set ∂H_i, which is of size $O(\sqrt{n})$. Hence, any *sparse* approximate Schur complement has only $\widetilde{O}(\sqrt{n})$ edges. When we need to compute $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)^{-1}\boldsymbol{x}$ for some vector $\boldsymbol{x}$, we use a generic SDD-solver which runs in $\widetilde{O}(|C|)$ time; this is crucial in bounding the overall runtime.

To extend the two-level scheme to more layers, we apply nested dissection recursively to each region H_i, until the regions are of constant size. This recursive procedure naturally gives rise to a *separator tree* $\mathcal{T}$ of the input graph G, which we discuss in detail in Section 3.4.2. Each node of $\mathcal{T}$ correspond to a region of G, and can be obtained by taking the edge-disjoint union of the regions of its two children. Taking the union over all leaf regions gives the original graph

G. The separator tree $\mathcal{T}$ allows us to define a set F_H of *eliminated vertices* and a set $\partial(H)$ of *boundary vertices* for each node H, analogous to what was shown in the two-layer dissection. Moreover, if we let F_i denote the disjoint union of sets F_H over all nodes H at level i, and C_i denote the union of sets $\partial(H)$, then we essentially generalize the set C from the two-layer dissection to $V(G) = C_{-1} \supset C_0 \supset \cdots \supset C_{\eta-1} \supset C_\eta = \emptyset$, where each C_i is some vertex separator of $G \setminus C_{i-1}$, and generalize the set F to $F_0, \ldots, F_\eta$ partitioning $V(G)$, where $F_i \stackrel{\text{def}}{=} C_{i-1} \setminus C_i$. With a height-$\eta$ separator tree, we can write

$$\mathbf{L}^{-1} = \boldsymbol{\mu}^{(0)\top} \cdots \boldsymbol{\mu}^{(\eta-1)\top} \begin{bmatrix} \mathbf{Sc}(\mathbf{L}, C_{-1})_{F_0,F_0}{}^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C_{\eta-1})_{F_\eta,F_\eta}{}^{-1} \end{bmatrix} \boldsymbol{\mu}^{(\eta-1)} \cdots \boldsymbol{\mu}^{(0)}, \quad (3.7)$$

for some explicit upper triangular matrices $\boldsymbol{\mu}^{(i)}$. Here, $\mathbf{Sc}(\mathbf{L}, C_i)_{F_{i+1},F_{i+1}}$ denotes the $F_{i+1} \times F_{i+1}$ submatrix of $\mathbf{Sc}(\mathbf{L}, C_i)$.

In the expression (3.7), the Schur complement term $\mathbf{Sc}(\mathbf{L}, C_i)$ at level i can further be decomposed at according to the nodes at the level. Then, we can obtain an approximation to $\mathbf{L}^{-1}$ by using approximate Schur complements as follows:

Theorem 3.2.3 ($\mathbf{L}^{-1}$ approximation)**.** *Suppose for each $H \in \mathcal{T}$, we have a Laplacian $\mathbf{L}^{(H)}$ satisfying*

$$\mathbf{L}^{(H)} \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H).$$

Then, we have

$$\mathbf{L}^{-1} \approx_{\eta\delta} \mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(\eta-1)\top} \widetilde{\mathbf{\Gamma}} \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}, \quad (3.8)$$

where

$$\widetilde{\mathbf{\Gamma}} = \begin{bmatrix} \sum_{H \in \mathcal{T}(0)} \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \sum_{H \in \mathcal{T}(\eta)} \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1} \end{bmatrix}.$$

and

$$\mathbf{\Pi}^{(i)} = \mathbf{I} - \sum_{H \in \mathcal{T}(i)} \mathbf{L}^{(H)}_{\partial(H),F_H} \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1},$$

where $\mathcal{T}(i)$ denotes the set of nodes at level i of $\mathcal{T}$, and $\mathbf{I}$ is the $n \times n$ identity matrix.

Compared to (3.7), we see that $\widetilde{\mathbf{\Gamma}}$ approximates the middle block-diagonal matrix, and $\mathbf{\Pi}^{(i)}$ approximates $\boldsymbol{\mu}^{(i)}$.

To compute and maintain the necessary $\mathbf{L}^{(H)}$'s as the edge weights undergo updates throughout the IPM, we have the following data structure:

Theorem 3.2.4 (Schur complements maintenance)**.** *Given a modified planar graph G with m edges and its separator tree $\mathcal{T}$ with height $\eta = O(\log m)$, the deterministic data structure* DynamicSC *(Algorithm 13) maintains the edge weights $\boldsymbol{w}$ from the IPM, and at every node $H \in \mathcal{T}$, maintains two vertex sets F_H and $\partial(H)$, and two Laplacians $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H \cup F_H)$ dependent on $\boldsymbol{w}$. It supports the following procedures:*

- Initialize$(G, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \delta > 0)$*: Given a graph G, initial weights $\boldsymbol{w}$, projection matrix approximation accuracy δ, preprocess in $\widetilde{O}(\delta^{-2}m)$ time.*
- Reweight$(\boldsymbol{w} \in \mathbb{R}^m_{>0},$ *given implicitly as a set of changed coordinates*$)$*: Update the weights to $\boldsymbol{w}$, and update the relevant Schur complements in $\widetilde{O}(\delta^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

  If $\mathcal{H}$ is the set of leaf nodes in $\mathcal{T}$ that contain an edge whose weight is updated, then $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ are updated only for nodes $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$.
- *Access to Laplacian $\mathbf{L}^{(H)}$ at any node $H \in \mathcal{T}$ in time $\widetilde{O}\left(\delta^{-2}|\partial H \cup F_H|\right)$.*
- *Access to Laplacian $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ at any node $H \in \mathcal{T}$ in time $\widetilde{O}\left(\delta^{-2}|\partial H|\right)$.*

Furthermore, the $\mathbf{L}^{(H)}$'s maintained by the data structure satisfy

$$\mathbf{L}^{(H)} \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H), \tag{3.9}$$

for all $H \in \mathcal{T}$ with high probability. The $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$'s maintained satisfy

$$\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H)) \tag{3.10}$$

for all $H \in \mathcal{T}$ with high probability.

3.2.3 Implicit representations using tree operator

In this section, we outline the data structures for maintaining the flow and slack solutions $\boldsymbol{f}, \boldsymbol{s}$ as needed in Algorithm 11, Line 22. Recall from Lemma 3.2.2, at IPM step k with step direction $\boldsymbol{v}^{(k)}$, we want to update

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \bar{t}h\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)},$$
$$\boldsymbol{f} \leftarrow \boldsymbol{f} + h\mathbf{W}^{1/2}\boldsymbol{v}^{(k)} - h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)},$$

for some approximate projection matrices $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ and $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}$ satisfying $\mathrm{Range}(\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}) \subseteq \mathrm{Range}(\mathbf{B})$ and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}} = \mathbf{B}^\top\mathbf{W}^{1/2}$. The first term for the flow update is straightforward to maintain. For this overview, we therefore focus on maintaining the second term

$$\boldsymbol{f}^\perp \leftarrow \boldsymbol{f}^\perp + h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)}.$$

Computing $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ and $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ respectively is too costly to do at every IPM step. Instead, we maintain vectors $\boldsymbol{s}_0, \boldsymbol{f}_0^\perp, \boldsymbol{z}$, and implicitly maintain two linear operators $\mathbf{M}^{(\text{slack})}, \mathbf{M}^{(\text{flow})}$ which depend on the weights $\boldsymbol{w}$, so at the end of every IPM step, the correct current solutions $\boldsymbol{s}, \boldsymbol{f}^\perp$ are recoverable via the identity

$$\boldsymbol{s} = \boldsymbol{s}_0 + \mathbf{M}^{(\text{slack})}\boldsymbol{z}$$
$$\boldsymbol{f}^\perp = \boldsymbol{f}_0^\perp + \mathbf{M}^{(\text{flow})}\boldsymbol{z}.$$

In this subsection, we abstract away the difference between slack and flow, and give a general data structure MAINTAINREP to maintain $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ for $\mathbf{M}$ with a special tree structure.

At a high level, MAINTAINREP implements the IPM operations MOVE and REWEIGHT as follows: To move in step k with direction $\boldsymbol{v}^{(k)}$ and step size $\alpha^{(k)}$, the data structure first computes $\boldsymbol{z}^{(k)}$ as a function of $\boldsymbol{v}^{(k)}$, then updates $\boldsymbol{z} \leftarrow \boldsymbol{z} + \alpha^{(k)}\boldsymbol{z}^{(k)}$, which translates to the desired overall update in $\boldsymbol{x}$ of $\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathbf{M}(\alpha^{(k)}\boldsymbol{z}^{(k)})$. To reweight with new weights $\boldsymbol{w}^{(\text{new})}$ (which does not change the value of $\boldsymbol{x}$), the data structure first computes $\mathbf{M}^{(\text{new})}$ using $\boldsymbol{w}^{(\text{new})}$ and $\Delta\mathbf{M} \stackrel{\text{def}}{=} \mathbf{M}^{(\text{new})} - \mathbf{M}$, then updates $\mathbf{M} \leftarrow \mathbf{M}^{(\text{new})}$. This causes an increase in value in the $\mathbf{M}\boldsymbol{z}$ term by $\Delta\mathbf{M}\boldsymbol{z}$, which is then offset in the $\boldsymbol{y}$ term with $\boldsymbol{y} \leftarrow \boldsymbol{y} - \Delta\mathbf{M}\boldsymbol{z}$.

In later sections, we will define $\mathbf{M}^{(\text{slack})}$ and $\mathbf{M}^{(\text{flow})}$ so that $\mathbf{M}^{(\text{slack})}\boldsymbol{z}^{(k)} = \mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ and $\mathbf{M}^{(\text{flow})}\boldsymbol{z}^{(k)} = \mathbf{W}^{-1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ for the desired approximate projection matrices. With these operators appropriately defined, observed that MAINTAINREP correctly captures the updates to $\boldsymbol{s}$ and $\boldsymbol{f}^{\perp}$ at every IPM step.

Let us now discuss the definition of $\boldsymbol{z}$, which is common to both slack and flow: Recall the DYNAMICSC data structure from the previous section maintains some Laplacian $\mathbf{L}^{(H)}$ for every node H in the separator tree $\mathcal{T}$, so that at each IPM step, we can implicitly represent the matrices $\mathbf{\Pi}^{(0)}, \cdots, \mathbf{\Pi}^{(\eta-1)}, \widetilde{\mathbf{\Gamma}}$ based on the current weights $\boldsymbol{w}$, which together give an $\eta\delta$-approximation of $\mathbf{L}^{-1}$. MAINTAINREP will contain a DYNAMICSC data structure, so we can use these Laplacians in the definition of $\boldsymbol{z}$:

At step k, let

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

where $\widetilde{\mathbf{\Gamma}}$, the $\mathbf{\Pi}^{(i)}$'s, and $\mathbf{W}$ are based on the state of the data structure at the end of step k. $\boldsymbol{z}$ is defined to be the accumulation of $\alpha^{(i)}\boldsymbol{z}^{(i)}$'s up to the current step; that is, at the end of step k,

$$\boldsymbol{z} = \sum_{i=1}^{k} \alpha^{(i)}\boldsymbol{z}^{(i)}.$$

Rather than naively maintaining $\boldsymbol{z}$, we decompose $\boldsymbol{z}$ and explicitly maintaining c, $\boldsymbol{z}^{(\text{prev})}$, and $\boldsymbol{z}^{(\text{sum})}$, such that

$$\boldsymbol{z} \stackrel{\text{def}}{=} c \cdot \boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})},$$

where we have the additional guarantee that at the end of IPM step k,

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

The other term, $\boldsymbol{z}^{(\text{sum})}$, is some remaining accumulation so that the overall representation is correct.

The purpose of this decomposition of $\boldsymbol{z}$ is to facilitate sparse updates to $\boldsymbol{v}$ between IPM steps: Suppose $\boldsymbol{v}^{(k)}$ differ from $\boldsymbol{v}^{(k-1)}$ on K coordinates, then we can update $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ with runtime as a function of K, while producing the correct overall update in $\boldsymbol{z}$. Specifically, we decompose $\boldsymbol{v}^{(k)} = \boldsymbol{v}^{(k-1)} + \Delta\boldsymbol{v}$. We compute $\Delta\boldsymbol{z}^{(\text{prev})} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\mathbf{B}^\top \mathbf{W}^{1/2}\Delta\boldsymbol{v}$, and then set

$$\boldsymbol{z}^{(\text{prev})} \leftarrow \boldsymbol{z}^{(\text{prev})} + \Delta\boldsymbol{z}^{(\text{prev})},\ \boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c \cdot \Delta\boldsymbol{z}^{(\text{prev})}, c \leftarrow c + \alpha,$$

which can be performed in $O(nnz(\Delta\boldsymbol{z}^{(\text{prev})}))$ time.

Let us briefly discuss how to compute $\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\boldsymbol{d}$ for some vector $\boldsymbol{d}$. We use the two-layer nested dissection setup from Section 3.2.2 for intuition, so

$$\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(0)}\boldsymbol{d} = \begin{bmatrix} \mathbf{L}_{F,F}^{-1} & \mathbf{0} \\ \mathbf{0} & \widetilde{\mathbf{Sc}}(\mathbf{L}, C)^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & 0 \\ -\mathbf{L}_{C,F}\mathbf{L}_{F,F}^{-1} & \mathbf{I} \end{bmatrix} \boldsymbol{d}.$$

The only difficult part for the next left matrix multiplication is $-\mathbf{L}_{C,F}\mathbf{L}_{F,F}^{-1}$. However, we note that $\mathbf{L}_{F,F}$ 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 $\mathbf{L}_{C,F}\mathbf{L}_{F,F}^{-1}\boldsymbol{d}$ 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 runtime. 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 runtime of these Laplacian solves by the size of the corresponding regions for the desired overall runtime.

On the other hand, to work with $\mathbf{M}$ efficiently, we define the notion of a *tree operator* $\mathbf{M}$ supported on a tree. In our setting, we use the separator tree $\mathcal{T}$. Informally, our tree operator is a linear operator mapping $\mathbb{R}^{V(G)}$ to $\mathbb{R}^{E(G)}$. It is constructed from the concatenation of a collection of *edge operators* and *leaf operators* defined on the edges and leaves of $\mathcal{T}$. If H is a node in $\mathcal{T}$ with parent P, then the edge operator for edge (H, P) will map vectors supported on $\partial(P) \cup F_P$ to vectors supported on $\partial(H) \cup F_H$. If H is a leaf node, the leaf operator for H will map vectors on $\partial(H) \cup F_H$ to vectors on $E(H)$. In this way, we take advantage of the recursive partitioning of G via $\mathcal{T}$ to map a vector supported on $V(G)$ recursive to be supported on smaller vertex subsets and finally to the edges. Furthermore, we will show that when edge weights update, the change to $\mathbf{M}$ can be localized to a small collection of edge and leaf operators along some tree paths, thus allowing for an efficient implementation. We postpone the formal definition of the operator until Section 3.5.2.

Theorem 3.2.5 (Implicit representation maintenance)**.** *Given a modified planar graph G with n vertices and m edges, and its separator tree $\mathcal{T}$ with height η, the deterministic data structure* MaintainRep *(Algorithm 16) maintains the following variables correctly at the end of every IPM step:*

- *the dynamic edge weights $\boldsymbol{w}$ and step direction $\boldsymbol{v}$ from the current IPM step,*
- *a* DynamicSC *data structure on $\mathcal{T}$ based on the current edge weights $\boldsymbol{w}$,*
- *an implicitly represented tree operator $\mathbf{M}$ supported on $\mathcal{T}$ with complexity $T(K)$,* computable using information from DynamicSC*,*
- *scalar c and vectors $\boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})}$, which together represent $\boldsymbol{z} = c\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})}$, such that at the end of step k,*
$$\boldsymbol{z} = \sum_{i=1}^{k} \alpha^{(i)} \boldsymbol{z}^{(i)},$$
where $\alpha^{(i)}$ is the step size α given in Move *for step i,*
- *$\boldsymbol{z}^{(\text{prev})}$ satisfies $\boldsymbol{z}^{(\text{prev})} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)} \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}$,*

- *an offset vector* $\boldsymbol{y}$ *which together with* $\mathbf{M}, \boldsymbol{z}$ *represent* $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$*, such that after step* k*,*

  $$\boldsymbol{x} = \boldsymbol{x}^{(\text{init})} + \sum_{i=1}^{k} \mathbf{M}^{(i)}(\alpha^{(i)}\boldsymbol{z}^{(i)}),$$

  where $\boldsymbol{x}^{(\text{init})}$ *is an initial value from* Initialize*, and* $\mathbf{M}^{(i)}$ *is the state of* $\mathbf{M}$ *after step* i*.*

The data structure supports the following procedures:

- Initialize$(G, \mathcal{T}, \mathbf{M}, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \boldsymbol{x}^{(\text{init})} \in \mathbb{R}^m, \epsilon_{\mathbf{P}} > 0)$*: Given a graph* G*, its separator tree* $\mathcal{T}$*, a tree operator* $\mathbf{M}$ *supported on* $\mathcal{T}$ *with complexity* T*, initial step direction* $\boldsymbol{v}$*, initial weights* $\boldsymbol{w}$*, initial vector* $\boldsymbol{x}^{(\text{init})}$*, and target projection matrix accuracy* $\epsilon_{\mathbf{P}}$*, preprocess in* $\widetilde{O}(\delta^{-2}m + T(m))$ *time and set* $\boldsymbol{x} \leftarrow \boldsymbol{x}^{(\text{init})}$*.*

- Reweight$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$ *given implicitly as a set of changed coordinates*$)$*: Update the weights to* $\boldsymbol{w}$*. Update the implicit representation of* $\boldsymbol{x}$ *without changing its value, so that all the variables in the data structure are based on the new weights.*

  The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK} + T(K))$ *total time, where* K *is an upper bound on the number of coordinates changed in* $\boldsymbol{w}$ *and the number of leaf or edge operators changed in* $\mathbf{M}$*. There are most* $\widetilde{O}(K)$ *nodes* $H \in \mathcal{T}$ *for which* $\boldsymbol{z}^{(\text{prev})}|_{F_H}$ *and* $\boldsymbol{z}^{(\text{sum})}|_{F_H}$ *are updated.*

- Move$(\alpha \in \mathbb{R},\ \boldsymbol{v} \in \mathbb{R}^n$ *given implicitly as a set of changed coordinates*$)$*: Update the current direction to* $\boldsymbol{v}$*, and then* $\boldsymbol{z}^{(\text{prev})}$ *to maintain the claimed invariant. Update the implicit representation of* $\boldsymbol{x}$ *to reflect the following change in value:*

  $$\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathbf{M}(\alpha\boldsymbol{z}^{(\text{prev})}).$$

  The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ *time, where* K *is the number of coordinates changed in* $\boldsymbol{v}$ *compared to the previous IPM step.*

- Exact$()$*: Output the current exact value of* $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ *in* $\widetilde{O}(T(m))$ *time.*

3.2.4 Solution 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 IPM. Since we have implicit representations of the solutions of the form $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$, we now show how to maintain $\overline{\boldsymbol{x}}$ close to $\boldsymbol{x}$. To accomplish this, we will give a meta data structure that solves this in a more general setting. The data structure involves three steps; the first two steps are similar to [84] and the key contribution is the last step:

1. We maintain an approximate vector by detecting coordinates of the exact vector $\boldsymbol{x}$ with large changes. In step k of the IPM, for every ℓ such that $2^{\ell} | k$, we consider all coordinates of the approximate vector $\overline{\boldsymbol{x}}$ 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 $\boldsymbol{x}$, it is updated. We can prove that each coordinate of $\overline{\boldsymbol{x}}$ has additive error at most δ compared to $\boldsymbol{x}$. The number of updates to $\overline{\boldsymbol{x}}$ 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 throughout the IPM algorithm.

2. We detect coordinates with large changes in $\boldsymbol{x}$ via a random sketch and sampling using the separator tree. We can sample a coordinate with probability exactly proportional to the magnitude of its change, when given access to the approximate sum of probabilities in each region of the separator tree and to the exact value of any single coordinate of $\boldsymbol{x}$.

3. We show how to maintain random sketches for vectors of the form $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$, where $\mathbf{M}$ is an implicit tree operator supported on a tree $\mathcal{T}$. Specifically, to maintain sketches of $\mathbf{M}\boldsymbol{z}$, we store intermediate sketches for every complete subtree of $\mathcal{T}$ at their roots. When an edge operator of $\mathbf{M}$ or a coordinate of $\boldsymbol{z}$ is modified, we only need to update the sketches along a path in $\mathcal{T}$ from a node to the root. For our use case, the cost of updating the sketches at a node H will be proportional to its separator size, so that a K-sparse update takes $\widetilde{O}(\sqrt{mK})$ time.

While the data structure is randomized, it is guaranteed to work against an adaptive adversary that is allowed to see the entire internal state of the data structure, including the random bits.

Theorem 3.2.6 (Approximate vector maintenance with tree operator)**.** *Given a constant degree tree $\mathcal{T}$ with height η that supports tree operator $\mathbf{M}$ with complexity T, there is a randomized data structure* MaintainApprox *that takes as input the dynamic variables $\mathbf{M}, c, \boldsymbol{z}^{(\mathrm{prev})}, \boldsymbol{z}^{(\mathrm{sum})}, \boldsymbol{y}, \mathbf{D}$ at every IPM step, and maintains the approximation $\overline{\boldsymbol{x}}$ to $\boldsymbol{x} \overset{\mathrm{def}}{=} \boldsymbol{y} + \mathbf{M}\boldsymbol{z} = \boldsymbol{y} + \mathbf{M}(c \cdot \boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})})$ satisfying $\left\|\mathbf{D}^{1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\right\|_\infty \le \delta$. It supports the following procedures:*

- *Initialize(tree $\mathcal{T}$, tree operator $\mathbf{M}, c \in \mathbb{R}, \boldsymbol{z}^{(\mathrm{prev})} \in \mathbb{R}^n, \boldsymbol{z}^{(\mathrm{sum})} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m, \mathbf{D} \in \mathbb{R}^{n \times n}, \rho > 0, \delta > 0$): Initialize the data structure with initial vector $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}(c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})})$, diagonal scaling matrix $\mathbf{D}$, target approximation accuracy δ, success probability $1 - \rho$, in $O(m\eta^2 \log m \log(\frac{m}{\rho}))$ time. Initialize $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}$.*

- *Approximate($\mathbf{M}, c, \boldsymbol{z}^{(\mathrm{prev})}, \boldsymbol{z}^{(\mathrm{sum})}, \boldsymbol{y}, \mathbf{D}$): Update the internal variables to their new iterations as given. Then output a vector $\overline{\boldsymbol{x}}$ such that $\|\mathbf{D}^{1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\|_\infty \le \delta$ for the current vector $\boldsymbol{x}$ and the current diagonal scaling $\mathbf{D}$.*

Suppose $\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_{\mathbf{D}^{(k+1)}} \le \beta$ for all *k, where $\mathbf{D}^{(k)}$ and $\boldsymbol{x}^{(k)}$ are the $\mathbf{D}$ and $\boldsymbol{x}$ at the k-th call to* Approximate*. Then, for the k-th call to* Approximate*, we have*

- *the data structure first updates $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k-1)}$ for the coordinates i with $\mathbf{D}_{ii}^{(k)} \neq \mathbf{D}_{ii}^{(k-1)}$, then updates $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k)}$ for $O(N_k \overset{\mathrm{def}}{=} 2^{2\ell_k}(\beta/\delta)^2 \log^2 m)$ coordinates, where ℓ_k is the largest integer ℓ with $k = 0 \bmod 2^\ell$.*

- *The amortized time cost of* Approximate *is*

$$\Theta(\eta^2 \log(\frac{m}{\rho}) \log m) \cdot T(\eta \cdot (N_{k-2^{\ell_k}} + |\mathcal{S}|)),$$

  where $\mathcal{S}$ is the set of nodes H where either $\mathbf{M}_{(H,P)}$, $\mathbf{J}_H$, $\boldsymbol{z}^{(\mathrm{prev})}|_{F_H}$, or $\boldsymbol{z}^{(\mathrm{sum})}|_{F_H}$ changed, or where $\boldsymbol{y}_e$ or $\mathbf{D}_{e,e}$ changed for some edge e in H, compared to the $(k-1)$-th step.

3.2.5 Slack projection

We want to use a MaintainRep data structure to implicitly maintain the slack solution $\boldsymbol{s}$ throughout the IPM, and use a MaintainApprox data structure to explicitly maintain the approximate slack solution $\overline{\boldsymbol{s}}$.

To use MAINTAINREP, it remains to define a suitable tree operator $\mathbf{M}^{(\text{slack})}$, so that at IPM step k, the update in MAINTAINREP is the correct IPM slack update; that is:

$$\mathbf{M}^{(\text{slack})}(\bar{t}h \cdot \boldsymbol{z}^{(\text{prev})}) = \bar{t}h\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}.$$

Let $\widetilde{\mathbf{L}}^{-1}$ denote the approximation of $\mathbf{L}^{-1}$ from (3.8), maintained and computable with a DYNAMICSC data structure. We define

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

then $\widetilde{\mathbf{P}}_{\boldsymbol{w}} \approx_{\eta\delta} \mathbf{P}_{\boldsymbol{w}}$, and $\text{Range}(\widetilde{\mathbf{P}}_{\boldsymbol{w}}) = \text{Range}(\mathbf{P}_{\boldsymbol{w}})$ by definition. Hence, this suffices as our approximate slack projection matrix.

Using Section 3.2.3, we can write

$$\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{\Pi}^{(0)\top}\cdots\mathbf{\Pi}^{(\eta-1)\top}\boldsymbol{z}^{(\text{prev})}, \tag{3.11}$$

where $\boldsymbol{z}^{(\text{prev})} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}^{(k)}$ at the end of IPM step k, as defined in the previous section. The remaining matrix multiplication on the left in (3.11) can indeed be represented by a tree operator $\mathbf{M}$ on the tree $\mathcal{T}$. Intuitively, observe that each $\mathbf{\Pi}^{(i)}$ operates on level i of $\mathcal{T}$ and can be decomposed to be written in terms of the nodes at level i. Furthermore, the $\mathbf{\Pi}^{(i)}$'s are applied in order of descending level in $\mathcal{T}$. Finally, at the leaf level, $\mathbf{W}^{1/2}\mathbf{B}$ maps vectors on vertices to vectors on edges. In Section 3.7, we present the exact tree operator and its correctness proof. With it, we have

$$\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \mathbf{W}^{1/2}\mathbf{M}\boldsymbol{z}^{(\text{prev})}.$$

We set $\mathbf{M}^{(\text{slack})}$ to be $\mathbf{W}^{1/2}\mathbf{M}$, which is also a valid tree operator.

Now, we state the full data structure for maintaining slack.

Theorem 3.2.7 (Slack maintenance)**.** *Given a modified planar graph G with m edges and its separator tree $\mathcal{T}$ with height η, the randomized data structure* MAINTAINSLACK *(Algorithm 19)* *implicitly maintains the slack solution $\boldsymbol{s}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{s}}$, and supports the following procedures with high probability against an adaptive adversary:*

- *Initialize*$(G, \boldsymbol{s}^{(\mathrm{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0)$*: Given a graph G, initial solution $\boldsymbol{s}^{(\mathrm{init})}$, initial direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$ and target approximation accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time, and set the representations $\boldsymbol{s} \leftarrow \boldsymbol{s}^{(\mathrm{init})}$ and $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{s}$.*

- *Reweight*$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$*, given implicitly as a set of changed weights): Set the current weights to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

- *Move*$(\alpha \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ *given implicitly as a set of changed coordinates): Implicitly update $\boldsymbol{s} \leftarrow \boldsymbol{s} + \alpha \mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}$ for some $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ with $\|(\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}\|_2 \leq \eta\delta \left\|\boldsymbol{v}\right\|_2$, and $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} \in \mathrm{Range}(\mathbf{B})$. The total runtime is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ where K is the number of coordinates changed in $\boldsymbol{v}$.*

- *Approximate*$() \to \mathbb{R}^m$*: Return the vector $\overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \leq \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{s}$.*

- *Exact*$() \to \mathbb{R}^m$*: Output the current vector $\boldsymbol{s}$ in $\widetilde{O}(m\delta^{-2})$ time.*

Suppose $\alpha\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to Move*. Suppose in each step,* Reweight*,* Move *and* Approximate *are called in order. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $(k-1)$-th and k-th* Reweight *and* Move *calls. Then at the k-th* Approximate *call,*

- *the data structure first sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last* Reweight*, then sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k)}$ for $O(N_k \overset{\mathrm{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*

- *The amortized time for the k-th* Approximate *call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})})$.*

3.2.6 Flow projection

Similar to slack, we want to use a MaintainRep data structure to implicitly maintain the flow solution $\boldsymbol{f}$ throughout the IPM, and use a MaintainApprox data structure to explicitly maintain the approximate flow solution $\overline{\boldsymbol{f}}$. For the overview, we focus on the non-trivial part of the flow update at every step given by

$$\boldsymbol{f}^{\perp} \leftarrow \boldsymbol{f}^{\perp} + h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}.$$

To use MaintainRep, it remains to define a suitable tree operator $\mathbf{M}^{(\text{flow})}$ so that at IPM step k, the update in MaintainRep is the correct IPM flow update; that is:

$$\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{M}^{(\text{flow})}\boldsymbol{z}^{(\text{prev})}.$$

Rather than finding an explicit $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}$ as we did for slack, observe it suffices to find some *weighted flow* $\widetilde{\boldsymbol{f}} \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$ satisfying $\mathbf{B}^{\top}\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}} = \mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}$. (We use the term "weighted flow" to mean it is obtained by multiplying the edge weights $\mathbf{W}$ to some valid flow.) Then the IPM update becomes

$$h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} = h\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}}.$$

Hence, our goal is to write $\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}} = \mathbf{M}^{(\text{flow})}\boldsymbol{z}^{(\text{prev})}$ for an appropriate weighted flow $\widetilde{\boldsymbol{f}}$.

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 condition of $\widetilde{\boldsymbol{f}} \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$ is actually $\widetilde{\boldsymbol{f}} \approx \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$. The second condition says $\widetilde{\boldsymbol{f}}$ is a weighted flow routing demand $\boldsymbol{d}$. Suppose we had $\widetilde{\boldsymbol{f}} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$ exactly, then we see immediately that the second condition is satisfied with $\mathbf{B}^{\top}\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}} = \mathbf{B}^{\top}\mathbf{W}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d} = \boldsymbol{d}$. To realize the approximation, we make use of the approximation of $\mathbf{L}^{-1}$ from (3.8). Hence, one important fact about our construction is that when the Schur complements are exact, our flow $\widetilde{\boldsymbol{f}}$ agrees with the true electrical flow routing the demand.

In constructing $\widetilde{\boldsymbol{f}}$ to route the demand $\boldsymbol{d}$, we show that $\widetilde{\boldsymbol{f}}$ can be written as $\mathbf{M}\boldsymbol{z}^{(\text{prev})}$, where $\mathbf{M}$ is a tree operator on the tree $\mathcal{T}$, and $\boldsymbol{z}^{(\text{prev})}$ is from MaintainRep, and in fact correspond to electric potentials. Here we explain what $\mathbf{M}$ captures intuitively. For simplicity, let $\boldsymbol{z}$ denote $\boldsymbol{z}^{(\text{prev})}$.

The first step is recognizing a decomposition of $\boldsymbol{d}$ using the separator tree, such that we have a demand term $\boldsymbol{d}^{(H)}$ for each node $H \in \mathcal{T}$. Furthermore, $\boldsymbol{d}^{(H)} = \mathbf{L}^{(H)}\boldsymbol{z}|_{F_H}$, for the Laplacian $\mathbf{L}^{(H)}$

supported on the region H maintained by DYNAMICSC. This decomposition 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 runtime. To show that the resulting flow $\widetilde{\boldsymbol{f}}$ indeed is close to the electric flow, one key insight is that the decomposed demands are orthogonal (Lemma 3.8.10). Hence, routing them separately by electrical flows gives a good approximation to the true electrical flow of the whole demand (Theorem 3.8.3).

Let us illustrate this partially using the two-layer decomposition scheme from Section 3.2.2: Suppose we have a demand term $\boldsymbol{d}$ *that is non-zero only on vertices of* C. Then, observe that

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

Looking at the sub-vector indexed by C on both sides, we have that

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

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 (3.6), we have

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

This gives a decomposition of the demand $\boldsymbol{d}$ into demand terms $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$ for $i = 1, 2$. Crucially, each demand $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$ is supported on the vertices of the region H_i, and we can route the flow on the corresponding region only. In a $O(\log n)$-level decomposition, we recursively decompose the demand further based on the sub-regions according to the separator tree $\mathcal{T}$. This guarantees that $\widetilde{\boldsymbol{f}}_i$ is the electric flow on the subgraph H_i that satisfies the demand $\widetilde{\mathbf{Sc}}(\mathbf{L}[H_i], C)\boldsymbol{z}$. Finally, we will let the output be $\widetilde{\boldsymbol{f}} = \sum \widetilde{\boldsymbol{f}}_i$. By construction, this $\widetilde{\boldsymbol{f}}$ satisfies $\mathbf{B}^\top \mathbf{W}^{1/2} \widetilde{\boldsymbol{f}} = \boldsymbol{d} = \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}$.

In Section 3.8, we show that this recursive operation can be realized using a tree operator. We then present the full proof for Theorem 3.2.8 below, and implement the data structure.

Theorem 3.2.8 (Flow maintenance)**.** *Given a modified planar graph G with m edges and its separator tree $\mathcal{T}$ with height η, the randomized data structure* MAINTAINFLOW *(Algorithm 20) implicitly* *maintains the flow solution $\boldsymbol{f}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{f}}$, and supports the following procedures with high probability against an adaptive adversary:*

- Initialize$(G, \boldsymbol{f}^{(\mathrm{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0)$*: Given a graph G, initial solution $\boldsymbol{f}^{(\mathrm{init})}$, initial direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$, and target approximation accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time and set the internal representation $\boldsymbol{f} \leftarrow \boldsymbol{f}^{(\mathrm{init})}$ and $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}$.*

- Reweight$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$ *given implicitly as a set of changed weights): Set the current weights to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

- Move$(\alpha \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ *given implicitly as a set of changed coordinates): Implicitly update $\boldsymbol{f} \leftarrow \boldsymbol{f} + \alpha \mathbf{W}^{1/2}\boldsymbol{v} - \alpha \mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$ for some $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$, where $\|\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\|_2 \leq O(\eta\delta)\|\boldsymbol{v}\|_2$ and $\mathbf{B}^\top \mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}$. The runtime is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$, where K is the number of coordinates changed in $\boldsymbol{v}$.*

- Approximate$() \to \mathbb{R}^m$*: Output the vector $\overline{\boldsymbol{f}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \leq \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{f}$.*

- Exact$() \to \mathbb{R}^m$*: Output the current vector $\boldsymbol{f}$ in $\widetilde{O}(m\delta^{-2})$ time.*

Suppose $\alpha\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to Move*. Suppose in each step,* Reweight*,* Move *and* Approximate *are called in order. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $(k-1)$-th and k-th* Reweight *and* Move *calls. Then at the k-th* Approximate *call,*

- *the data structure first sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last* Reweight*, then sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k)}$ for $O(N_k \stackrel{\mathrm{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*

- *The amortized time for the k-th* Approximate *call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})})$.*

3.2.7 Main proof

We are now ready to prove our main result. Algorithm 12 presents the implementation of RIPM Algorithm 11 using our data structures.

Algorithm 12 Implementation of Robust Interior Point Method

```
1: procedure CenteringImpl($\mathbf{B}, \phi, \boldsymbol{f}, \boldsymbol{s}, t_{\text{start}}, t_{\text{end}}$)
2:     $G$: graph on $n$ vertices and $m$ edges with incidence matrix $\mathbf{B}$
3:     $\mathcal{S}, \mathcal{F}$: data structures for slack and flow maintenance          ▷ Theorems 3.2.7 and 3.2.8
4:     $\alpha \overset{\text{def}}{=} \frac{1}{2^{20}\lambda}, \lambda \overset{\text{def}}{=} 64\log(256m^2)$
5:     $t \leftarrow t_{\text{start}}$, $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}, \overline{\boldsymbol{s}} \leftarrow \boldsymbol{s}, \overline{t} \leftarrow t$, $\mathbf{W} \leftarrow \nabla^2\phi(\overline{\boldsymbol{f}})^{-1}$          ▷ variable initialization
6:     $\boldsymbol{v}_i \leftarrow \sinh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}})_i)$ for all $i \in [n]$
                                                                      ▷ data structure initialization
7:     $\mathcal{F}$.Initalize($G, \boldsymbol{f}, \boldsymbol{v}, \mathbf{W}, \delta = O(\alpha/\log m), \overline{\epsilon} = \alpha$)     ▷ choose $\delta$ so $\eta\delta \le \alpha$ in Theorem 3.2.7
8:     $\mathcal{S}$.Initalize($G, \overline{t}^{-1}\boldsymbol{s}, \boldsymbol{v}, \mathbf{W}, \delta = O(\alpha/\log m), \overline{\epsilon} = \alpha$)     ▷ and $O(\eta\delta) \le \alpha$ in Theorem 3.2.8
9:     while $t \ge t_{\text{end}}$ do
10:        $t \leftarrow \max\{(1 - \frac{\alpha}{\sqrt{m}})t, t_{\text{end}}\}$
11:        Update $h = -\alpha/\|\cosh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}))\|_2$          ▷ $\gamma$ as defined in (3.2)
12:        Update the diagonal weight matrix $\mathbf{W} = \nabla^2\phi(\overline{\boldsymbol{f}})^{-1}$
13:        $\mathcal{F}$.Reweight($\mathbf{W}$)          ▷ update the implicit representation of $\boldsymbol{f}$ with new weights
14:        $\mathcal{S}$.Reweight($\mathbf{W}$)          ▷ update the implicit representation of $\boldsymbol{s}$ with new weights
15:        $\boldsymbol{v}_i \leftarrow \sinh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}})_i)$ for all $i$ where $\overline{\boldsymbol{f}}_i$ or $\overline{\boldsymbol{s}}_i$ has changed          ▷ update direction $\boldsymbol{v}$
16:                                                  ▷ $\mathbf{P}_{\boldsymbol{w}} \overset{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}^{1/2}$
17:        $\mathcal{F}$.Move($h, \boldsymbol{v}$)          ▷ Update $\boldsymbol{f} \leftarrow \boldsymbol{f} + h\mathbf{W}^{1/2}\boldsymbol{v} - h\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}}$ with $\widetilde{\boldsymbol{f}} \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$
18:        $\mathcal{S}$.Move($h, \boldsymbol{v}$)          ▷ Update $\boldsymbol{s} \leftarrow \boldsymbol{s} + \overline{t}h\mathbf{W}^{-1/2}\widetilde{\boldsymbol{s}}$ with $\widetilde{\boldsymbol{s}} \approx \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$
19:        $\overline{\boldsymbol{f}} \leftarrow \mathcal{F}$.Approximate()          ▷ Maintain $\overline{\boldsymbol{f}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \le \alpha$
20:        $\overline{\boldsymbol{s}} \leftarrow \overline{t}\mathcal{S}$.Approximate()          ▷ Maintain $\overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \le \overline{t}\alpha$
21:        if $|\overline{t} - t| \ge \alpha\overline{t}$ then
22:            $\boldsymbol{s} \leftarrow \overline{t}\mathcal{S}$.Exact()
23:            $\overline{t} \leftarrow t$
24:            $\mathcal{S}$.Initalize($G, \overline{t}^{-1}\boldsymbol{s}, \boldsymbol{v}, \mathbf{W}, \delta = O(\alpha/\log m), \overline{\epsilon} = \alpha$)
25:        end if
26:    end while
27:    return ($\mathcal{F}$.Exact(), $\overline{t}\mathcal{S}$.Exact())
28: end procedure
```

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

Lemma 3.2.9. *When updating $\boldsymbol{w}$ and $\boldsymbol{v}$ at the $(k+1)$-th step of the* CenteringImpl *algorithm, $\boldsymbol{w}$ and $\boldsymbol{v}$ change in $O(2^{2\ell_{k-1}}\log^2 m + 2^{2\ell_k}\log^2 m)$ coordinates, where ℓ_k is the largest integer ℓ with*

$k \equiv 0 \mod 2^\ell$.

Proof. Since both $\boldsymbol{w}$ and $\boldsymbol{v}$ are an entry-wise function of $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$ and $\overline{t}$, we need to examine these variables. First, $\overline{t}$ changes every $\widetilde{O}(\sqrt{m})$ steps, and when $\overline{t}$ changes, every coordinate of $\boldsymbol{w}$ and $\boldsymbol{v}$ changes. Over the entire CenteringImpl run, $\overline{t}$ changes $\widetilde{O}(1)$ number of times, so we may incur an additive $\widetilde{O}(m)$ term overall, and assume $\overline{t}$ does not change for the rest of the analysis.

By Theorem 3.2.1, we have $h\|\boldsymbol{v}\|_2 = O(\frac{1}{\log m})$ at all steps. So we apply Theorem 3.2.7 and Theorem 3.2.8 both with parameters $\beta = O(\frac{1}{\log m})$ and $\overline{\epsilon} = \alpha = \Theta(\frac{1}{\log m})$. We use their conclusions in the following argument. Let the superscript ${}^{(k)}$ denote the variable at the end of the k-th step.

By definition, $\boldsymbol{w}^{(k+1)}$ is an entry-wise function of $\overline{\boldsymbol{f}}^{(k)}$, and recursively, $\overline{\boldsymbol{f}}^{(k)}$ is an entry-wise function of $\boldsymbol{w}^{(k)}$. We first prove inductively that at step k, $O(2^{2\ell_k} \log^2 m)$ coordinates of $\overline{\boldsymbol{f}}$ change to $\boldsymbol{f}^{(k)}$ where $\boldsymbol{f}^{(k)}$ is the exact solution, and there are no other changes. This allows us to conclude that $\boldsymbol{w}^{(k+1)}$ differ from $\boldsymbol{w}^{(k)}$ on $O(2^{2\ell_k} \log^2 m)$ coordinates.

In the base case at step $k = 1$, because $\boldsymbol{w}^{(1)}$ is equal to the initial weights $\boldsymbol{w}^{(0)}$, only $O(2^{2\ell_1} \log^2 m)$ coordinates $\overline{\boldsymbol{f}}_e$ change to $\boldsymbol{f}_e^{(1)}$. Suppose at step k, a set S of $O(2^{2\ell_k} \log^2 m)$ coordinates of $\overline{\boldsymbol{f}}$ change; that is, $\overline{\boldsymbol{f}}|_S$ is updated to $\boldsymbol{f}^{(k)}|_S$, and there are no other changes. Then at step $k+1$, by definition, $\boldsymbol{w}^{(k+1)}$ differ from $\boldsymbol{w}^{(k)}$ exactly on S, and in turn, $\overline{\boldsymbol{f}}^{(k+1)}|_S$ is set to $\boldsymbol{f}^{(k)}|_S$ again (Line 20 of Algorithm 17). In other words, there is no change from this operation. Then, $O(2^{2\ell_{k+1}} \log^2 m)$ additional coordinates $\overline{\boldsymbol{f}}_e$ change to $\boldsymbol{f}_e^{(k+1)}$.

Now, we bound the change in $\overline{\boldsymbol{s}}$: Theorem 3.2.7 guarantees that in the k-th step, there are $O(2^{2\ell_k} \log^2 m) + D$ coordinates in $\overline{\boldsymbol{s}}$ that change, where D is the number of changes between $\boldsymbol{w}^{(k-1)}$ and $\boldsymbol{w}^{(k)}$ and is equal to $O(2^{2\ell_{k-1}} \log^2 m)$ as shown above.

Finally, $\boldsymbol{v}^{(k+1)}$ is an entry-wise function of $\overline{\boldsymbol{f}}^{(k)}$ and $\overline{\boldsymbol{s}}^{(k)}$, so we conclude that $\boldsymbol{v}^{(k+1)}$ and $\boldsymbol{v}^{(k)}$ differ on at most $O(2^{2\ell_k} \log^2 m) + 2 \cdot O(2^{2\ell_{k-1}} \log^2 m)$ coordinates. $\square$

Theorem 3.2.10 (Main result)**.** *Let $G = (V, E)$ be a directed planar graph with n vertices and m edges. Assume that the demands $\mathbf{d}$, edge capacities $\mathbf{u}$ and costs $\mathbf{c}$ 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 $\widetilde{O}(n \log M)$ expected time.*

Proof. The proof is structured as follows. We first write the minimum cost flow problem as a linear program of the form (3.1). We prove the linear program has an interior point and is bounded, so to satisfy the assumptions in Theorem 3.2.1. Then, we implement the IPM algorithm using the data structures from Sections 3.2.3 to 3.2.6. 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 (3.1), we add extra vertices s and t. Let $\boldsymbol{d}$ be the demand vector of the min-cost flow problem. For every vertex v with $\boldsymbol{d}_v < 0$, we add a directed edge from s to v with capacity $-\boldsymbol{d}_v$ and cost 0. For every vertex v with $\boldsymbol{d}_v > 0$, 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 $4nM$ and cost $-4nM$. The modified graph is no longer planar but it has only two extra vertices s and t.

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 3.2.1 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

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

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 $\boldsymbol{\ell}^{\text{new}}, \boldsymbol{u}^{\text{new}}$ are the new capacity constraints. If an edge e has only one direction, we set $\boldsymbol{\ell}_e^{\text{new}} = 0$ and $\boldsymbol{u}_e^{(\text{new})} = \boldsymbol{u}_e$, otherwise, we orient the edge arbitrarily and set $-\boldsymbol{\ell}_e^{\text{new}} = \boldsymbol{u}_e^{\text{new}} = \boldsymbol{u}_e$.

Now, we bound the parameters L, R, r in Theorem 3.2.1. Clearly, $L = \|\boldsymbol{c}^{\text{new}}\|_2 = O(Mm)$ and $R = \|\boldsymbol{u}^{\text{new}} - \boldsymbol{\ell}^{\text{new}}\|_2 = O(Mm)$. 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 $r \geq \frac{1}{4m}$.

The RIPM in Theorem 3.2.1 runs the subroutine Centering twice. In the first run, the constraint matrix is the incidence matrix of a new underlying graph, constructed by making three copies of each edge in the original graph G. Since copying edges does not affect planarity, and our data structures allow for duplicate edges, we use the implementation given in CenteringImpl (Algorithm 12) for both runs.

By the guarantees of Theorem 3.2.7 and Theorem 3.2.8, we correctly maintain $\boldsymbol{s}$ and $\boldsymbol{f}$ at every step in CenteringImpl, and the requirements on $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$ for the RIPM are satisfied. Hence, Theorem 3.2.1 shows that we can find a circulation $\boldsymbol{f}$ such that $(\boldsymbol{c}^{\text{new}})^{\top}\boldsymbol{f} \leq \text{OPT} - \frac{1}{2}$ by setting $\epsilon = \frac{1}{CM^2m^2}$ for some large constant C in Algorithm 11. 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 $\leq -knM$. 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 [**kang2015flow**]. 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 runtime of one call to CenteringImpl. We initialize the data structures for flow and slack by Initialize. Here, the data structures are given the first IPM step direction $\boldsymbol{v}$ for preprocessing; the actual step is taken in the first iteration of the main while-loop. At each step of CenteringImpl, we perform the implicit update of $\boldsymbol{f}$ and $\boldsymbol{s}$ using Move; we update $\mathbf{W}$ in the data structures using Reweight; and we construct the explicit approximations $\overline{\boldsymbol{f}}$ and $\overline{\boldsymbol{s}}$ using Approximate; each in the respective flow and slack data structures. We return the true $(\boldsymbol{f}, \boldsymbol{s})$ by Exact. The total cost of CenteringImpl 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, it suffices to bound the runtime for APPROXIMATE only. Theorem 3.2.1 guarantees that there are $T = O(\sqrt{m}\log n \log(nM))$ total APPROXIMATE calls. Lemma 3.2.9 shows that at the k-th call, the number of coordinates changed in $\boldsymbol{w}$ and $\boldsymbol{v}$ is bounded by $K \overset{\text{def}}{=} O(2^{2\ell_{k-1}}\log^2 m + 2^{2\ell_{k-2}}\log^2 m)$, where ℓ_k is the largest integer ℓ with $k \equiv 0 \mod 2^\ell$, or equivalently, the number of trailing zeros in the binary representation of k. Theorem 3.2.1 further guarantees we can apply Theorem 3.2.7 and Theorem 3.2.8 with parameter $\beta = O(1/\log m)$, which in turn shows the amortized time for the k-th call is

$$\widetilde{O}(\delta^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})}).$$

where $N_k \overset{\text{def}}{=} 2^{2\ell_k}(\beta/\alpha)^2\log^2 m = O(2^{2\ell_k}\log^2 m)$, where $\alpha = O(1/\log m)$ and $\epsilon_{\mathbf{P}} = O(1/\log m)$ are defined in CENTERINGIMPL.

Observe that $K + N_{k-2^{\ell_k}} = O(N_{k-2^{\ell_k}})$. Now, summing over all T calls, the total time is

$$\begin{aligned} O(\sqrt{m}\log m)\sum_{k=1}^{T}\sqrt{N_{k-2^{\ell_k}}} &= O(\sqrt{m}\log^2 m)\sum_{k=1}^{T}2^{\ell_{(k-2^{\ell_k})}} \\ &= O(\sqrt{m}\log^2 m)\sum_{k'=1}^{T}2^{\ell_{k'}}\sum_{k=1}^{T}[k - 2^{\ell_k} = k'], \end{aligned}$$

where we use $[\cdot]$ for the indicator function, i.e., $[k - 2^{\ell_k} = k'] = 1$ if $k - 2^{\ell_k} = k'$ is true and 0 otherwise. As there are only $\log T$ different powers of 2 in $[1, T]$, the count $\sum_{1\le k\le T}[k - 2^{\ell_k} = k']$ is bounded by $O(\log T)$ for any $k' \in \{1, \dots, T\}$. Then the above expression is

$$= O(\sqrt{m}\log^2 m \log T)\sum_{k'=1}^{T}2^{\ell_{k'}}.$$

Since ℓ_k is the number of trailing zeros on k, it can be at most $\log T$ for $k \le T$. We again rearrange the summation by possible values of $\ell_{k'}$, and note that there are at most $T/2^{i+1}$ numbers between 1 and T with i trailing zeros, so

$$\sum_{k'=1}^{T}2^{\ell_{k'}} = \sum_{i=0}^{\log T}2^i \cdot T/2^{i+1} = O(T\log T).$$

So the overall runtime is $O(\sqrt{m}T\log m\log^2 T)$. Combined with Theorem 3.2.1's guarantee of

$T = O(\sqrt{m} \log n \log(nM))$, we conclude the overall runtime is $\widetilde{O}(m \log M)$. $\square$

3.3 Preliminaries

We assume all matrices and vectors in an expression have matching dimensions. That is, we will trivially pad matrices and vectors with zeros when necessary. This abuse of notation is unfortunately unavoidable as we will be considering lots of submatrices and subvectors.

General Notations. An event holds with high probability if it holds with probability at least $1 - n^c$ for arbitrarily large constant c. The choice of c affects guarantees by constant factors.

We use boldface lowercase variables to denote vectors, and boldface uppercase variables to denote matrices. We use $\|\boldsymbol{v}\|_2$ to denote the 2-norm of vector $\boldsymbol{v}$ and $\|\boldsymbol{v}\|_{\mathbf{M}}$ to denote $\boldsymbol{v}^\top \mathbf{M} \boldsymbol{v}$. For any vector $\boldsymbol{v}$ and scalar x, we define $\boldsymbol{v} + x$ to be the vector obtained by adding x to each coordinate of $\boldsymbol{v}$ and similarly $\boldsymbol{v} - x$ to be the vector obtained by subtracting x from each coordinate of $\boldsymbol{v}$. We use $\mathbf{0}$ for all-zero vectors and matrices where dimensions are determined by context. We use $\mathbf{1}_A$ for the vector with value 1 on coordinates in A and 0 everywhere else. We use $\mathbf{I}$ for the identity matrix and $\mathbf{I}_S$ for the identity matrix in $\mathbb{R}^{S \times S}$. For any vector $\boldsymbol{x} \in \mathbb{R}^S$, $\boldsymbol{x}|_C$ denotes the sub-vector of $\boldsymbol{x}$ supported on $C \subseteq S$; *more specifically,* $\boldsymbol{x}|_C \in \mathbb{R}^S$*, where* $\boldsymbol{x}_i = 0$ *for all* $i \notin C$*.*

For any matrix $\mathbf{M} \in \mathbb{R}^{A \times B}$, we use the convention that $\mathbf{M}_{C,D}$ denotes the sub-matrix of $\mathbf{M}$ supported on $C \times D$ where $C \subseteq A$ and $D \subseteq B$. When $\mathbf{M}$ is not symmetric and only one subscript is specified, as in $\mathbf{M}_D$, this denotes the sub-matrix of $\mathbf{M}$ supported on $A \times D$. To keep notations simple, $\mathbf{M}^{-1}$ will denote the inverse of $\mathbf{M}$ if it is an invertible matrix and the Moore-Penrose pseudo-inverse otherwise.

For two positive semi-definite matrices $\mathbf{L}_1$ and $\mathbf{L}_2$, we write $\mathbf{L}_1 \approx_t \mathbf{L}_2$ if $e^{-t}\mathbf{L}_1 \preceq \mathbf{L}_2 \preceq e^t \mathbf{L}_1$, where $\mathbf{A} \preceq \mathbf{B}$ means $\mathbf{B} - \mathbf{A}$ is positive semi-definite. Similarly we define $\geq_t$ and $\leq_t$ for scalars, that is, $x \leq_t y$ if $e^{-t}x \leq y \leq e^t x$.

Graphs and Trees. We define *modified planar graph* to mean a graph obtained from a planar graph by adding 2 new vertices s, t and any number of edges incident to the new vertices. We allow distinguishable parallel edges in our graphs. We assume the input graph is connected.

We use n for the number of vertices and m for the number of edges in the input graph. We will use $\boldsymbol{w}$ for the vector of edge weights in a graph. We define $\mathbf{W}$ as the diagonal matrix $\operatorname{diag}(\boldsymbol{w})$.

We define $\mathbf{L} = \mathbf{B}^\top \mathbf{W} \mathbf{B}$ be the Laplacian matrix associated with an undirected graph G with non-negative edge weights $\mathbf{W}$. We at times use a graph and its Laplacian interchangeably. For a subgraph $H \subseteq G$, we use $\mathbf{L}[H]$ to denote the weighted Laplacian on H, and $\mathbf{B}[H]$ to denote the incidence matrix of H.

For a tree $\mathcal{T}$, we write $H \in \mathcal{T}$ to mean H is a node in $\mathcal{T}$. We write $\mathcal{T}_H$ to mean the complete subtree of $\mathcal{T}$ rooted at H. We say a node A is an ancestor of H if H is in the subtree rooted at A, and $H \neq A$.

The *level* of a node in a tree is defined so that leaf nodes have level 0, and the root has level η, where η is the height of the tree. For interior nodes, the level is the length of the longest path from the node to a leaf. By this definition, note that the level of a node and its child can differ by more than 1.

For binary tree data structures, we assume there is constant time access to each node.

IPM data structures. When we discuss the data structures in the context of the IPM, step 0 means the initialization step. For $k > 0$, step k means the k-th iteration of the while-loop in Centering (Algorithms 11 and 12); that is, it is the k-th time we update the current solutions. For any vector or matrix $\boldsymbol{x}$ used in the IPM, we use $\boldsymbol{x}^{(k)}$ to denote the value of $\boldsymbol{x}$ at the end of the k-th step.

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. This can be implemented by checking memory for changes.

We use smallCaps to denote function names and data structure classes, and `typewriterFont` to denote an instantiation of a data structure.

We say a data structure B *extends* A in the object-oriented sense. Inside data structure B, we directly access functions and variables of A when the context is clear, or use the keyword `super`.

In the data structure 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 3.3.1 ([**JambulapatiS21**, 13])**.** *There is a randomized algorithm which is an ϵ-approximate Laplacian system solver for the any input n-vertex m-edge graph and $\epsilon \in (0,1)$ and has the following runtime $O(m\mathrm{poly}(\log\log n)\log(1/\epsilon))$.*

3.4 Nested dissection and approximate Schur complements

This section lays the foundation for a recursive decomposition of the input graph. Our goal is to set up the machinery necessary for approximating $\mathbf{P}_{\boldsymbol{w}} \stackrel{\mathrm{def}}{=} \mathbf{W}^{1/2}\mathbf{B}(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{W}^{1/2}$ as needed in the robust IPM. In particular, we are interested in the weighted Laplacian matrix $\mathbf{L} \stackrel{\mathrm{def}}{=} \mathbf{B}^\top\mathbf{W}\mathbf{B}$.

We begin with a discussion of nested dissection and the associated Schur complements.

3.4.1 Cholesky decomposition and Schur complement

Let G be a weighted graph. Consider the partition of vertices in G into two subsets C and $F = V(G) \setminus C$ called *boundary* and *interior* vertices. This partitions $\mathbf{L}$ into four blocks:

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{F,F} & \mathbf{L}_{F,C} \\ \mathbf{L}_{C,F} & \mathbf{L}_{C,C} \end{bmatrix}.$$

Definition 3.4.1 (Block Cholesky decomposition)*.* The *block Cholesky decomposition* of a symmetric $\mathbf{L}$ with blocks indexed by F and C defined as above is:

$$\mathbf{L} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{L}_{C,F}(\mathbf{L}_{F,F})^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{F,F} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L},C) \end{bmatrix} \begin{bmatrix} \mathbf{I} & (\mathbf{L}_{F,F})^{-1}\mathbf{L}_{F,C} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}. \tag{3.12}$$

The middle matrix in the decomposition is a block-diagonal matrix with blocks indexed by F and C, with the lower-right block being:

Definition 3.4.2 (Schur complement)*.* The *Schur complement* $\mathbf{Sc}(\mathbf{L},C)$ of $\mathbf{L}$ onto C is the Laplacian matrix resulting from a partial symmetric Gaussian elimination on $\mathbf{L}$. Formally,

$$\mathbf{Sc}(\mathbf{L},C) = \mathbf{L}_{C,C} - \mathbf{L}_{C,F}\mathbf{L}_{F,F}^{-1}\mathbf{L}_{F,C}.$$

It is known that $\mathbf{Sc}(\mathbf{L}, C)$ is the Laplacian of another graph with vertex set C. We further use the convention that if H is a subgraph of G and $V(H) \subset C$, then $\mathbf{Sc}(H, C)$ simply means $\mathbf{Sc}(H, C \cap V(H))$. Graph theoretically, the Schur complement has the following interpretation:

Lemma 3.4.3. *Let $V(G) = \{v_1, \dots, v_n\}$. Let $C = V(G) - v_1$. Let $\boldsymbol{w}_{ij}$ denote the weight of edge v_iv_j. Then*

$$\mathbf{Sc}(\mathbf{L}, C) = G[C] + H,$$

where $G[C]$ is the subgraph of G induced on the vertex set S, and H is the graph on S with edges v_iv_j where $i, j \in N(v_1)$, and $\boldsymbol{w}_{ij} = \boldsymbol{w}_{1i}\boldsymbol{w}_{1j}/\boldsymbol{w}_1$, where $\boldsymbol{w}_1$ is the total weight of edges incident to v_1 in G. Note that on the right hand side, we use a graph to mean its Laplacian. □

Taking Schur complement is an associative operation. Furthermore, it commutes with edge deletion, and more generally, edge weight deletion. Finally, for our purposes, it can be decomposed under certain special circumstances.

Lemma 3.4.4. *If $X \subseteq Y \subseteq V(G)$, then $\mathbf{Sc}(\mathbf{Sc}(\mathbf{L}, Y), X) = \mathbf{Sc}(\mathbf{L}, X)$.* □

Lemma 3.4.5. *Let $\boldsymbol{w}_e$ denote the weight of edge e in G. Suppose $C \subseteq V(G)$, and H is a subgraph of G on the vertex set C with edge weights $\boldsymbol{w}'_e \leq \boldsymbol{w}_e$ for all edges in $G[C]$. Let $\mathbf{L}'$ denote the Laplacian of H. Then, $\mathbf{Sc}(\mathbf{L} - \mathbf{L}', C) = \mathbf{Sc}(\mathbf{L}, C) - \mathbf{L}'$.* □

Lemma 3.4.6. *Let $\mathbf{L}$ be the Laplacian of graph G with the decomposition $\mathbf{L} = \mathbf{L}_1 + \mathbf{L}_2$, where $\mathbf{L}_1$ is a Laplacian supported on the vertex set V_1 and $\mathbf{L}_2$ on V_2. Furthermore, suppose $V_1 \cap V_2 \subseteq C$ for some vertex set $C \subseteq V(G)$. Then*

$$\mathbf{Sc}(\mathbf{L}, C) = \mathbf{Sc}(\mathbf{L}_1, C \cap V_1) + \mathbf{Sc}(\mathbf{L}_2, C \cap V_2).$$

Proof. We have

$$
\begin{aligned}
\mathbf{Sc}(\mathbf{L}, C) &= \mathbf{Sc}(\mathbf{L}_1 + \mathbf{L}_2, C) \\
&= \mathbf{Sc}(\mathbf{Sc}(\mathbf{L}_1 + \mathbf{L}_2, C \cup V_2), C) \\
&= \mathbf{Sc}(\mathbf{Sc}(\mathbf{L}_1, C \cup V_2) + \mathbf{L}_2, C) && \text{(by Lemma 3.4.5)} \\
&= \mathbf{Sc}(\mathbf{Sc}(\mathbf{L}_1, C) + \mathbf{L}_2, C) && \text{(since } (C \cup V_2) \cap V_1 \subseteq C\text{)} \\
&= \mathbf{Sc}(\mathbf{L}_1, C) + \mathbf{Sc}(\mathbf{L}_2, C), && \text{(by Lemma 3.4.5)} \\
&= \mathbf{Sc}(\mathbf{L}_1, C \cap V_1) + \mathbf{Sc}(\mathbf{L}_2, C \cap V_2) && \text{(since } \mathbf{L}_i \text{ is supported on } V_i \text{ for } i = 1, 2\text{)}
\end{aligned}
$$

as desired. □

3.4.2 Separator tree

In the overview, we briefly gave the intuition for a 2-level partition of the input graph; here we extend it to a recursive partitioning scheme with $O(\log n)$-levels. We begin with the formal definitions.

Definition 3.4.7 (Separable graph). A graph $G = (V, E)$ is α-separable if there exists two constants $c > 0$ and $b \in (0, 1)$ such that every nonempty subgraph $H = (V(H) \subseteq V, E(H) \subseteq E)$ with $|E(H)| \geq 2$ of G can be partitioned into H_1 and H_2 such that

- $E(H_1) \cup E(H_2) = E(H)$, $E(H_1) \cap E(H_2) = \emptyset$,
- $|V(H_1) \cap V(H_2)| \leq c\lceil |E(H)|^{\alpha} \rceil$,
- $|E(H_i)| \leq b|E(H)|$, for $i = 1, 2$.

We call $S(H) \overset{\text{def}}{=} V(H_1) \cap V(H_2)$ the *balanced vertex separator* of H.

It is known that any planar graph is 1/2-separable.

Remark 3.4.8. As we discussed in Section 3.2.7, our LP formulation for the IPM uses a *modified planar graph* which is the original planar graph with two additional vertices and $O(n)$ additional edges incident to them. By adding two vertices and edges incident to them to a planar graph, the modified graph is also 1/2-separable with the constant c in Definition 3.4.7 increased by 2.

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:

Definition 3.4.9 (Separator tree $\mathcal{T}$). Let G be a modified planar graph. A separator tree $\mathcal{T}$ is a binary tree whose nodes represent subgraphs of G such that the children of each node H form a balanced partition of H.

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 $\partial(H)$ 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 in Definition 3.4.7, and their vertex sets intersect on the balanced separator $S(H)$ of H. D_1 and D_2 does not have any isolated vertex. Define $\partial(D_1) = (\partial(H) \cup S(H)) \cap V(D_1)$, and similarly $\partial(D_2) = (\partial(H) \cup S(H)) \cap V(D_2)$. Define $F_H = S(H) \setminus \partial(H)$.
3. If a region H contains a constant number of edges, 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.

Let $\eta(H)$ denote the height of node H which is defined as the maximum number of edges on a tree path from H to one of its descendants. $\eta(H) = 0$ if H is a leaf. Note that the height difference between a parent and child node could be greater than one. Let η denote the height of $\mathcal{T}$ which is defined as the maximum height of nodes in $\mathcal{T}$. We say H is at *level* i if $\eta(H) = i$.

Observation 3.4.10. *Using the above definition, $\{F_H : H \in \mathcal{T}\}$ partitions the vertex set $V(G)$.*

Observation 3.4.11. *Suppose H is a node in $\mathcal{T}$ with children D_1 and D_2. We have $\partial D_1 \cup \partial D_2 = \partial H \cup F_H$.*

Observation 3.4.12. *Suppose H is a node in $\mathcal{T}$. Then $\partial(H) \subseteq \cup_{ancestor\ A\ of\ H} F_A$.*

Fakcharoenphol and Rao [160] gave an algorithm that computes the separator tree for any planar graph.

Theorem 3.4.13 (Separator tree construction [160])**.** *Given a 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.*

For computing the separator tree $\mathcal{T}$ of a modified planar graph, we may apply their method to the original planar graph to get the separator $\mathcal{T}'$, and add the two new vertices s, t to F_G at the root node G, and to the boundary sets $\partial(H)$ at every non-root node H. The additional edges incident to s, t can be recursively partitioned from a node to its children, which increases the height of $\mathcal{T}$ by $O(\log n)$. Thus, we have the following corollary:

Corollary 3.4.14 (Separator tree construction for modified planar graph)**.** *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.*

To discuss the structures in the separator tree, we define the following terms:

Definition 3.4.15. Let $\mathcal{T}(i)$ be the subset of nodes in $\mathcal{T}$ at level i. For a node H, let $\mathcal{T}_H$ be the subtree of $\mathcal{T}$ rooted at H. 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 nodes $\mathcal{H} = \{H : H \in \mathcal{T}\}$, define

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

Finally, 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 $\mathcal{T}$.

Fakcharoenphol and Rao [160, Section 3.5] showed that for a set $\mathcal{H}$ of K nodes in T, the total number of boundary vertices from the nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ is $O(\sqrt{mK})$. However, their claim is not stated as a result we can cite here. We provide a simple, self-contained proof in Section 3.10 of a slightly weaker bound that in addition requires bounding the number of separator vertices.

Lemma 3.4.16. *Let G be a modified planar graph with separator tree $\mathcal{T}$. Let $\mathcal{H}$ be a set of K nodes in $\mathcal{T}$. Then*

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

3.4.3 Approximating $\mathbf{L}^{-1}$ using the separator tree

For a height-η separator tree, we generalize the sets C and F from the block Cholesky decomposition ((3.12)) to a sequence of sets $C_0, \ldots, C_\eta$, and $F_0, \ldots, F_\eta$ based on $\mathcal{T}$.

Definition 3.4.17 (C_i, F_i). Let $\mathcal{T}$ be the separator tree from Corollary 3.4.14. For all $0 \leq i \leq \eta$, we define $F_i = \bigcup_{H \in \mathcal{T}(i)} F_H$ to be the vertices eliminated at level i. For all $0 \leq i \leq \eta$, we define $C_i = \bigcup_{H \in \mathcal{T}(i)} \partial(H)$ to be the vertices remaining after eliminating vertices in F_i. We define C_{-1} to be $V(G)$.

By Observation 3.4.10, F_i is the disjoint union of F_H over all nodes H at level i in the separator tree. $F_0, \ldots, F_\eta$ partitions $V(G)$. By the definition of $\partial(H)$ and F_H, we know $F_i = C_{i-1} \setminus C_i$ for all $0 \leq i \leq \eta$. It follows that $V(G) = C_{-1} \supset C_0 \supset \cdots \supset C_{\eta-1} \supset C_\eta = \emptyset$ and $C_i = \cup_{j>i} F_j$.

Now, the decomposition from (3.12) can be extended and inverted as follows:

$$\mathbf{L}^{-1} = \boldsymbol{\mu}^{(0)\top} \cdots \boldsymbol{\mu}^{(\eta-1)\top} \begin{bmatrix} \mathbf{Sc}(\mathbf{L}, C_{-1})_{F_0,F_0}{}^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C_{\eta-1})_{F_\eta,F_\eta}{}^{-1} \end{bmatrix} \boldsymbol{\mu}^{(\eta-1)} \cdots \boldsymbol{\mu}^{(0)}, \quad (3.13)$$

where the $\boldsymbol{\mu}^{(i)}$'s are upper triangular matrices with

$$\boldsymbol{\mu}^{(i)} = \mathbf{I} - \mathbf{Sc}(\mathbf{L}, C_{i-1})_{C_i,F_i} \left(\mathbf{Sc}(\mathbf{L}, C_{i-1})_{F_i,F_i}\right)^{-1},$$

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 in (3.13).

Definition 3.4.18 (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 ϵ-*approximate Schur complement* of $\mathbf{L}$ onto C if $\widetilde{\mathbf{Sc}}(\mathbf{L}, C) \approx_\epsilon \mathbf{Sc}(\mathbf{L}, C)$, where we use $\approx_\epsilon$ to mean an e^ϵ-spectral approximation.

Definition 3.4.19 ($\mathbf{L}^{(H)}$). Let $\delta > 0$. For each $H \in \mathcal{T}$, let $\mathbf{L}^{(H)}$ be a Laplacian on the vertex set $F_H \cup \partial H$ such that

$$\mathbf{L}^{(H)} \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H).$$

We show how to compute and maintain $\mathbf{L}^{(H)}$ in the next subsection.

Here, we define the necessary approximate matrices and show how to approximate $\mathbf{L}^{-1}$.

Definition 3.4.20 ($\mathbf{\Pi}^{(i)}, \mathbf{X}^{(H)}, \widetilde{\mathbf{\Gamma}}$). To approximate $\boldsymbol{\mu}^{(i)}$, we define

$$\mathbf{\Pi}^{(i)} = \mathbf{I} - \sum_{H \in \mathcal{T}(i)} \mathbf{X}^{(H)}, \tag{3.14}$$

where

$$\mathbf{X}^{(H)} = \mathbf{L}^{(H)}_{\partial(H), F_H} \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \tag{3.15}$$

for each $H \in \mathcal{T}$.

To approximate the block diagonal matrix in (3.13), we define

$$\widetilde{\mathbf{\Gamma}} = \begin{bmatrix} \sum_{H \in \mathcal{T}(0)} \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \sum_{H \in \mathcal{T}(\eta)} \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \end{bmatrix}.$$

Theorem 3.4.21 ($\mathbf{L}^{-1}$ approximation)**.** *Suppose for each $H \in \mathcal{T}$, we have a Laplacian $\mathbf{L}^{(H)}$ satisfying*

$$\mathbf{L}^{(H)} \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H).$$

Then, we have

$$\mathbf{L}^{-1} \approx_{\eta\delta} \mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(\eta-1)\top} \widetilde{\mathbf{\Gamma}} \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}. \tag{3.16}$$

Proof. Let C_i, F_i be defined for each i according to Definition 3.4.17. Let $\mathbf{L}^{(i)} \stackrel{\text{def}}{=} \sum_{H \in \mathcal{T}(i)} \mathbf{L}^{(H)}$.

Note that $\mathbf{L}^{(i)}_{F_i, F_i} \stackrel{\text{def}}{=} \sum_{H \in \mathcal{T}(i)} \mathbf{L}^{(H)}_{F_H, F_H}$ is a block-diagonal matrix with blocks indexed by $H \in$

$\mathcal{T}(i)$, since F_i is a disjoint union over F_H for $H \in \mathcal{T}(i)$, and only $\mathbf{L}^{(H)}$ is supported on F_H. Hence, ${\mathbf{L}^{(i)}_{F_i,F_i}}^{-1} = \sum_{H\in\mathcal{T}(i)} \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1}$.

Recall that the regions in $\mathcal{T}(i)$ partition the graph G. Furthermore, the intersection of $H, H' \in \mathcal{T}(i)$ is on their boundary, which is contained in $C_i \subseteq C_{i-1}$. Thus, we apply Lemma 3.4.6 to get

$$\begin{aligned}\mathbf{Sc}(\mathbf{L}, C_{i-1}) &= \sum_{H\in\mathcal{T}(i)} \mathbf{Sc}(\mathbf{L}[H], C_{i-1} \cap V(H)) \\ &\approx_\delta \sum_{H\in\mathcal{T}(i)} \widetilde{\mathbf{Sc}}(\mathbf{L}[H], \partial H \cup F_H) = \sum_{H\in\mathcal{T}(i)} \mathbf{L}^{(H)} = \mathbf{L}^{(i)}.\end{aligned} \tag{3.17}$$

Now, we prove inductively that

$$\mathbf{L}^{-1} \approx_{i\delta} \mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(i-1)\top} \begin{bmatrix} \left(\mathbf{L}^{(0)}_{F_0,F_0}\right)^{-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \left(\mathbf{L}^{(i-1)}_{F_{i-1},F_{i-1}}\right)^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \left(\mathbf{L}^{(i)}\right)^{-1} \end{bmatrix} \mathbf{\Pi}^{(i-1)} \cdots \mathbf{\Pi}^{(0)}, \tag{3.18}$$

When $i = 0$, we have the approximation trivially as $\mathbf{L}^{(0)} = \mathbf{L}$.

For general i, we factor $\mathbf{L}^{(i)}$ in (3.18) recursively using Cholesky decomposition. $\mathbf{L}^{(i)}$ is supported on C_{i-1}, and we can partition $C_{i-1} = F_i \cup C_i$. Then,

$$\mathbf{L}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{L}^{(i)}_{C_i,F_i}(\mathbf{L}^{(i)}_{F_i,F_i})^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}^{(i)}_{F_i,F_i} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L}^{(i)}, C_i) \end{bmatrix} \begin{bmatrix} \mathbf{I} & (\mathbf{L}^{(i)}_{F_i,F_i})^{-1}\mathbf{L}^{(i)}_{F_i,C_i} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}. \tag{3.19}$$

For the Schur complement term in the factorization, we have

$$\begin{aligned}\mathbf{Sc}(\mathbf{L}^{(i)}, C_i) &\approx_{i\delta} \mathbf{Sc}(\mathbf{Sc}(\mathbf{L}, C_{i-1}), C_i) && \text{(by (3.17))} \\ &= \mathbf{Sc}(\mathbf{L}, C_i) && \text{(by transitivity of Schur complements)} \\ &\approx_\delta \mathbf{L}^{(i)}. && \text{(by (3.17))}\end{aligned}$$

So we can use $\mathbf{L}^{(i)}$ in place of the Schur complement term, and the equality becomes an approximation with factor $(i+1)\delta$. Furthermore, in (3.19), we can rewrite

$$\mathbf{L}^{(i)}_{C_i,F_i} = \sum_{H\in\mathcal{T}(i)} \mathbf{L}^{(H)}_{C_i,F_i} = \sum_{H\in\mathcal{T}(i)} \mathbf{L}^{(H)}_{\partial H,F_H}.$$

Plugging the inverse of (3.19) into (3.18), we get the correct recursive approximation.

Finally, we note that at the η-th level, $\mathbf{L}^{(\eta)}_{F_\eta,F_\eta} = \mathbf{L}^{(\eta)}$ since $C_\eta = \emptyset$. So we have the overall expression. $\square$

3.4.4 Recursive Schur complements on separator tree

In this section, we prove Theorem 3.2.4 which maintains approximate Schur complements onto the boundary vertices of each node H in $\mathcal{T}$.

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

Lemma 3.4.22 (ApproxSchur procedure [122]). *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 error tolerance. Given approximation parameter $\epsilon \in (0,1/2)$, there is an algorithm ApproxSchur$(\mathbf{L}, C, \epsilon)$ that computes and outputs a ϵ-approximate Schur complement $\widetilde{\mathbf{Sc}}(\mathbf{L}, C)$ that satisfies the following properties with probability at least $1-\gamma$:*

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

First, we prove the correctness and runtime of ApproxSchurNode(H). We say ApproxSchurNode(H) runs correctly on a node H at level i in $\mathcal{T}$, if at the end of the procedure, the following properties are satisfied:

- $\mathbf{L}^{(H)}$ is the Laplacian of a graph on vertices $\partial(H) \cup F_H$ with $\widetilde{O}(\delta^{-2}|\partial(H)\cup F_H|)$ edges,
- $\mathbf{L}^{(H)} \approx_{(i-1)\delta} \mathbf{Sc}(\mathbf{L}[H], \partial(H)\cup F_H)$,
- $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \approx_{i\delta} \mathbf{Sc}(\mathbf{L}[H], \partial(H))$, and the graph is on $\partial(H)$ with $\widetilde{O}(\delta^{-2}|\partial(H)|)$ edges.

Algorithm 13 Data structure to maintain dynamic approximate Schur complements

```
data structure DynamicSC
private: member
    Graph $G$ with incidence matrix $\mathbf{B}$
    $\boldsymbol{w} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{m\times m}$: Weight vector and diagonal weight matrix, used interchangeably
    $\delta > 0$: Overall approximation factor
    $\delta > 0$: Fast Schur complement approximation factor
    $\mathcal{T}$: Separator tree of height $\eta$. Every node $H$ of $\mathcal{T}$ stores:
        $F_H$, $\partial(H)$: Interior and boundary vertices of region $H$
        $\mathbf{L}^{(H)} \in \mathbb{R}^{m\times m}$: Laplacian supported on $F_H \cup \partial(H)$
        $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \in \mathbb{R}^{m\times m}$: $\delta$-approximate Schur complement of $\mathbf{L}^{(H)}$

procedure Initialize($G$, $\boldsymbol{w} \in \mathbb{R}^m$, $\delta > 0$)
    $\mathbf{B} \leftarrow$ incidence matrix of $G$
    $\mathcal{T} \leftarrow$ separator tree of $G$ of height $\eta$ constructed by Theorem 3.4.13
    $\delta \leftarrow \delta/(\eta+1)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w}$
    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($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m$)
    $\mathcal{H} \leftarrow$ set of leaf nodes in $\mathcal{T}$ that contain each edge $e$ whose weight is updated
    $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})}$
    $\mathcal{P}_{\mathcal{T}}(\mathcal{H}) \leftarrow$ set of all ancestor nodes of $\mathcal{H}$ in $\mathcal{T}$ and $\mathcal{H}$
    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
        ▷ $\mathbf{B}[H]$ is the incidence matrix for the induced subgraph $H$ with edge set $E(H)$
        $\mathbf{L}^{(H)} \leftarrow (\mathbf{B}[H])^\top \mathbf{W}_{E(H)} \mathbf{B}[H]$
        $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \leftarrow$ ApproxSchur($\mathbf{L}^{(H)}, \partial(H), \delta$)        ▷ Lemma 3.4.22
    else
        Let $D_1, D_2$ be the children of $H$
        $\mathbf{L}^{(H)} \leftarrow \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial(D_1)) + \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial(D_2))$
        $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \leftarrow$ ApproxSchur($\mathbf{L}^{(H)}, \partial(H), \delta$)
    end if
end procedure
```

Lemma 3.4.23. *Suppose $\mathbf{L}^{(D)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D)}, \partial(D))$ are computed correctly for all descendants D of H, then* ApproxSchurNode*(H) runs correctly.*

Proof. When H is a leaf, the proof is trivial. $\mathbf{L}^{(H)}$ is set to the exact Laplacian matrix of the induced subgraph H of constant size. $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ δ-approximates $\mathbf{Sc}(\mathbf{L}^{(H)}, \partial(H)) = \mathbf{Sc}(\mathbf{L}[H], \partial(H))$ by Lemma 3.4.22.

Otherwise, suppose H is at level i with children D_1 and D_2. By construction of the separator tree and Observation 3.4.11, we have $\partial D_1 \cup \partial D_2 = \partial H \cup F_H$. For each $j = 1, 2$, we know inductively $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_j)}, \partial(D_j))$ has $\widetilde{O}(\delta^{-2}|\partial(D_j)|)$ edges. Since we define $\mathbf{L}^{(H)}$ to be the sum, it has $\widetilde{O}(\delta^{-2}(|\partial(D_1)| + |\partial(D_2)|)) = \widetilde{O}(\delta^{-2}|\partial(H) \cup F_H|)$ edges, and is supported on vertices $\partial H \cup F_H$, so we have the first correctness property.

Inductively, we know $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_j)}, \partial(D_j)) \approx_{(i-1)\delta} \mathbf{Sc}(\mathbf{L}[D_j], \partial(D_j))$ for both $j = 1, 2$. (The height of D_j may or may not equal to $i - 1$ but it is guaranteed to be no more than $i - 1$.) Then

$$\begin{aligned}
\mathbf{L}^{(H)} &= \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial(D_1)) + \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial(D_2)) \\
&\approx_{(i-1)\delta} \mathbf{Sc}(\mathbf{L}[D_1], \partial(D_1)) + \mathbf{Sc}(\mathbf{L}[D_2], \partial(D_2)) \\
&= \mathbf{Sc}(\mathbf{L}[D_1], (\partial(H) \cup F_H) \cap V(D_1)) + \mathbf{Sc}(\mathbf{L}[D_2], (\partial(H) \cup F_H) \cap V(D_2)) \\
&\qquad \text{(by construction of the separator tree, } \partial D_j = (\partial H \cup F_H) \cap V(D_j) \text{ for } j = 1, 2) \\
&= \mathbf{Sc}(\mathbf{L}[H], \partial H \cup F_H), \qquad \text{(by Lemma 3.4.6)}
\end{aligned}$$

so we have the second correctness property.

Line 43 returns $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ with $\widetilde{O}(\delta^{-2}|\partial(H)|)$ edges by Lemma 3.4.22. Also,

$$\begin{aligned}
\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) &\approx_{\delta} \mathbf{Sc}(\mathbf{L}^{(H)}, \partial(H)) \\
&\approx_{(i-1)\delta} \mathbf{Sc}(\mathbf{Sc}(\mathbf{L}[H], \partial H \cup F_H), \partial(H)) \\
&= \mathbf{Sc}(\mathbf{L}[H], \partial H), \qquad \text{(by Lemma 3.4.4)}
\end{aligned}$$

giving us the third correctness property. □

Lemma 3.4.24. *The runtime of* ApproxSchurNode*(H) is $\widetilde{O}(\delta^{-2}|\partial(H) \cup F_H|)$.*

Proof. When H is a leaf node, computing $\mathbf{L}^{(H)} = \mathbf{L}[H]$ takes time proportional to $|H| = \partial H \cup F_H$. Computing $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H)$ takes $\widetilde{O}(\delta^{-2}|H|)$ time by Lemma 3.4.22.

Otherwise, when H has children D_1, D_2, computing $\mathbf{L}^{(H)}$ requires accessing $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_j)}, \partial D_j)$ for $j = 1, 2$ and summing them together, in time $\widetilde{O}(|\partial D_1| + |\partial D_2|) = \widetilde{O}(|\partial H \cup F_H|)$. Then, computing $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H)$ take $\widetilde{O}(\delta^{-2}|\partial H \cup F_H|)$ by Lemma 3.4.22. □

Next, we prove the overall data structure correctness and runtime:

Theorem 3.2.4 (Schur complements maintenance)**.** *Given a modified planar graph G with m edges and its separator tree $\mathcal{T}$ with height $\eta = O(\log m)$, the deterministic data structure* DynamicSC *(Algorithm 13) maintains the edge weights $\boldsymbol{w}$ from the IPM, and at every node $H \in \mathcal{T}$, maintains two vertex sets F_H and $\partial(H)$, and two Laplacians $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H \cup F_H)$ dependent on $\boldsymbol{w}$. It supports the following procedures:*

- Initialize$(G, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \delta > 0)$*: Given a graph G, initial weights $\boldsymbol{w}$, projection matrix approximation accuracy δ, preprocess in $\widetilde{O}(\delta^{-2}m)$ time.*
- Reweight$(\boldsymbol{w} \in \mathbb{R}^m_{>0},$ *given implicitly as a set of changed coordinates): Update the weights to $\boldsymbol{w}$, and update the relevant Schur complements in $\widetilde{O}(\delta^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

  If $\mathcal{H}$ is the set of leaf nodes in $\mathcal{T}$ that contain an edge whose weight is updated, then $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ are updated only for nodes $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$.
- *Access to Laplacian $\mathbf{L}^{(H)}$ at any node $H \in \mathcal{T}$ in time $\widetilde{O}\left(\delta^{-2}|\partial H \cup F_H|\right)$.*
- *Access to Laplacian $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ at any node $H \in \mathcal{T}$ in time $\widetilde{O}\left(\delta^{-2}|\partial H|\right)$.*

Furthermore, the $\mathbf{L}^{(H)}$'s maintained by the data structure satisfy

$$\mathbf{L}^{(H)} \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H), \tag{3.9}$$

for all $H \in \mathcal{T}$ with high probability. The $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$'s maintained satisfy

$$\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \approx_\delta \mathbf{Sc}(\mathbf{L}[H], \partial(H)) \tag{3.10}$$

for all $H \in \mathcal{T}$ with high probability.

Proof of Theorem 3.2.4. Because we set $\delta \leftarrow \delta/(\eta+1)$ in INITIALIZE, combined with Lemma 3.4.23, we conclude that for each $H \in \mathcal{T}$,

$$\mathbf{L}^{(H)} \approx_{\delta} \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H)$$

and

$$\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H)) \approx_{\delta} \mathbf{Sc}(\mathbf{L}[H], \partial(H)).$$

We next prove the correctness and runtime of INITIALIZE. Computing the separator tree costs $O(n \log n)$ time by Theorem 3.4.13. Because APPROXSCHURNODE(H) is called in increasing order of level of H, each APPROXSCHURNODE(H) runs correctly and stores the initial value of $\mathbf{L}^{(H)}$ by Lemma 3.4.23. The runtime of INITIALIZE is bounded by running APPROXSCHURNODE on each node, i.e:

$$\widetilde{O}(\delta^{-2} \sum_{H \in \mathcal{T}} |\partial(H) \cup F_H|) = \widetilde{O}(\delta^{-2} m) = \widetilde{O}(\delta^{-2} m).$$

Where we bound the sum using Lemma 3.4.16 with $K = O(m)$, since $\mathcal{T}$ has $O(m)$ nodes in total.

The proof for REWEIGHT is similar to INITIALIZE. Let K be the number of coordinates changed in $\boldsymbol{w}$. Then $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ contains all the regions with an edge with weight update. For each node H not in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$, no edge in H has a modified weight, and in this case, we do not need to update $\mathbf{L}^{(H)}$. For the nodes that do require updates, since APPROXSCHURNODE(H) is called in increasing order of level of H, we can prove inductively that all APPROXSCHURNODE(H) for $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ run correctly. The time spent is bounded by $\widetilde{O}(\delta^{-2} \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})} |\partial(H) \cup F_H|)$. By Lemma 3.4.16, this is further bounded by $\widetilde{O}(\delta^{-2}\sqrt{mK})$.

For accessing $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$, we simply return the stored values. The time required is proportional to the size of $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial(H))$ respectively, by the correctness properties of these Laplacians, we get the correct size and therefore the runtime. ☐

3.5 Maintaining the implicit representation

In this section, we give a general data structure MaintainRep.

At a high level, MaintainRep implicitly maintains a vector $\boldsymbol{x}$ throughout the IPM, by explicitly maintaining vector $\boldsymbol{y}$, and implicitly maintaining a *tree operator* $\mathbf{M}$ and vector $\boldsymbol{z}$, with $\boldsymbol{x} \stackrel{\text{def}}{=} \boldsymbol{y}+\mathbf{M}\boldsymbol{z}$. MaintainRep supports the IPM operations Move and Reweight as follows: To move in step k with direction $\boldsymbol{v}^{(k)}$ and step size $\alpha^{(k)}$, the data structure computes some $\boldsymbol{z}^{(k)}$ from $\boldsymbol{v}^{(k)}$ and updates $\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathbf{M}(\alpha^{(k)}\boldsymbol{z}^{(k)})$. To reweight with new weights $\boldsymbol{w}^{(\text{new})}$ (which does not change the value of $\boldsymbol{x}$), the data structure computes $\mathbf{M}^{(\text{new})}$ using $\boldsymbol{w}^{(\text{new})}$, updates $\mathbf{M} \leftarrow \mathbf{M}^{(\text{new})}$, and updates $\boldsymbol{y}$ to offset the change in $\mathbf{M}\boldsymbol{z}$. In Section 3.5.1, we define $\boldsymbol{z}^{(k)}$ and show how to maintain $\boldsymbol{z} = \sum_{i=1}^{k} \boldsymbol{z}^{(i)}$ efficiently. In Section 3.5.2, we define tree operators. Finally in Section 3.5.3, we implement MaintainRep for a general tree operator $\mathbf{M}$.

Our goal is for this data structure to maintain the updates to the slack and flow solutions at every IPM step. Recall at step k, we want to update the slack solution by $\bar{t}h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ and the partial flow solution by $h\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$. In later sections, we define specific tree operators $\mathbf{M}^{(\text{slack})}$ and $\mathbf{M}^{(\text{flow})}$ so that the slack and flow updates can be written as $\mathbf{M}^{(\text{slack})}(\bar{t}h\boldsymbol{z}^{(k)})$ and $\mathbf{M}^{(\text{flow})}(h\boldsymbol{z}^{(k)})$ respectively. This then allows us to use two copies of MaintainRep to maintain the solutions throughout the IPM.

To start, recall the information stored in the DynamicSC data structure: at every node H we have Laplacian $\mathbf{L}^{(H)}$. In the previous section, we defined matrices $\widetilde{\mathbf{\Gamma}}$ and $\mathbf{\Pi}^{(i)}$'s as functions of the $\mathbf{L}^{(H)}$'s, in order to approximate $\mathbf{L}^{-1}$. MaintainRep will contain a copy of the DynamicSC data structure; therefore, the remainder of this section will freely refer to $\widetilde{\mathbf{\Gamma}}$ and $\mathbf{\Pi}^{(0)}, \cdots, \mathbf{\Pi}^{(\eta-1)}$.

3.5.1 Maintaining the intermediate vector $\boldsymbol{z}$

We define a partial computation at each step of the IPM, which will be shared by both the slack and flow solutions:

Definition 3.5.1 ($\boldsymbol{z}^{(k)}$). At the k-th step of the IPM, let $\boldsymbol{v}^{(k)}$ be the step direction. Let $\boldsymbol{d} \stackrel{\text{def}}{=}$

$\mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}^{(k)}$. Define $\boldsymbol{z}^{(k)}$ to be the partial computation

$$\boldsymbol{z}^{(k)} \stackrel{\text{def}}{=} \widetilde{\mathbf{\Gamma}} \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)} \boldsymbol{d}. \tag{3.20}$$

Observe that this is a partial projection: If we apply $\mathbf{W}^{1/2}\mathbf{B}\mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(\eta-1)\top}$ to $\boldsymbol{z}^{(k)}$, then by Theorem 3.4.21, the result is an approximation to $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$.

We first show how to multiply $\widetilde{\mathbf{\Gamma}} \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}$ to a vector efficiently. The main idea is to take advantage of the hierarchical structure of the separator tree $\mathcal{T}$ in a bottom-up fashion. If $\boldsymbol{d}$ is a sparse vector with only K non-zero entries, then we can apply the operator while avoiding exploring parts of $\mathcal{T}$ that are guaranteed to contain zero values.

Lemma 3.5.2. *Given a vector $\boldsymbol{d} \in \mathbb{R}^n$, let $\mathcal{H} \supseteq \{H \in \mathcal{T} : \boldsymbol{d}|_{F_H} \neq \mathbf{0}\}$ and suppose $|\mathcal{H}| = K$. Then the procedure* PartialProject$(\boldsymbol{d}, \mathcal{H})$ *in the* MaintainZ *data structure (Algorithm 14) returns the vector*

$$\boldsymbol{u} = \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(1)} \mathbf{\Pi}^{(0)} \boldsymbol{d},$$

where the $\mathbf{\Pi}^{(i)}$'s and δ are from the DynamicSC *data structure in* MaintainZ.

The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, and $\boldsymbol{u}|_{F_H}$ is non-zero for at most $\widetilde{O}(K)$ nodes $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$.

Proof. First, we consider the runtime. We remark that the creation of vector $\boldsymbol{u}$ is for readability; the procedure can in fact be computed using $\boldsymbol{d}$ in-place.

The bottleneck of PartialProject is Line 24. For each $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$, recall from Theorem 3.2.4 that $\mathbf{L}^{(H)}$ is supported on the vertex set $F_H \cup \partial(H)$ and has $\widetilde{O}(\delta^{-2}|F_H \cup \partial(H)|)$ edges. Hence, $(\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{u}|_{F_H}$ can be computed by an exact Laplacian solver in $\widetilde{O}(\delta^{-2}|F_H \cup \partial(H)|)$ time, and the subsequent left-multiplying by $\mathbf{L}^{(H)}_{\partial(H),F_H}$ also takes $\widetilde{O}(\delta^{-2}|F_H \cup \partial(H)|)$ time. Finally, we can add the resulting vector to $\boldsymbol{u}$ in time linear in the sparsity. Summing this over all $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$, we get that the total runtime is $\widetilde{O}(\delta^{-2}\sqrt{mK})$ by Lemma 3.4.16.

To show the correctness of PartialProject, we have the following claim:

Algorithm 14 Data structure to maintain the intermediate vector $\boldsymbol{z}$, Part 1

```
1: data structure MaintainZ
2: private: member
3:     $G$: input graph $G$ with incidence matrix $\mathbf{B}$
4:     $\mathcal{T}$: separator tree of $G$ of height $\eta$
5:     $c \in \mathbb{R}, \boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})} \in \mathbb{R}^n$: coefficient and vectors to be maintained
6:     $\boldsymbol{u} \in \mathbb{R}^n$: vector to be maintained such that $\boldsymbol{u} = \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\mathbf{B}^\top \mathbf{W}\boldsymbol{v}$
7:     $\boldsymbol{v} \in \mathbb{R}^m$: direction vector from the current iteration
8:     $\boldsymbol{w} \in \mathbb{R}^m$: weight vector                    ▷ we sometimes also use $\mathbf{W} \stackrel{\text{def}}{=} \text{diag}(\boldsymbol{w})$
9:     dynamicSC: an instance of DynamicSC struct         ▷ gives read access to $\mathbf{L}^{(H)}$ for $H \in \mathcal{T}$
10:
11: procedure Initialize($G, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \delta > 0$)
12:     $\boldsymbol{w} \leftarrow \boldsymbol{w}$, $\boldsymbol{v} \leftarrow \boldsymbol{v}$
13:     dynamicSC.Initialize($G, \boldsymbol{w}, \delta$)
14:     $\boldsymbol{u} \leftarrow$ PartialProject($\mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}$)
15:     $\boldsymbol{z}^{(\text{prev})} \leftarrow \widetilde{\mathbf{\Gamma}}\boldsymbol{u}$
16:     $\boldsymbol{z}^{(\text{sum})} \leftarrow \mathbf{0}$
17:     $c \leftarrow 0$
18: end procedure
19:
20: procedure PartialProject($\boldsymbol{d} \in \mathbb{R}^n, \mathcal{H} = \{H \in \mathcal{T} : \boldsymbol{d}|_{F_H} \neq \mathbf{0}\}$)
21:                     ▷ if $\mathcal{H}$ is not given in the argument, then it takes the default value above
22:     $\boldsymbol{u} \leftarrow \boldsymbol{d}$
23:     for $i$ from 0 to $\eta - 1$ do
24:         $\boldsymbol{u} \leftarrow \boldsymbol{u} - \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} \mathbf{L}^{(H)}_{\partial(H),F_H}(\mathbf{L}^{(H)}_{F_H,F_H})^{-1} \cdot \boldsymbol{u}|_{F_H}$
25:     end for
26:     return $\boldsymbol{u}$
27: end procedure
28:
29: procedure InversePartialProject($\boldsymbol{u} \in \mathbb{R}^n, \mathcal{H}$)
30:     for $i$ from $\eta - 1$ to 0 do
31:         $\boldsymbol{u} \leftarrow \boldsymbol{u} + \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} \mathbf{L}^{(H)}_{\partial(H),F_H}(\mathbf{L}^{(H)}_{F_H,F_H})^{-1} \cdot \boldsymbol{u}|_{F_H}$
32:     end for
33:     $\boldsymbol{d} \leftarrow \boldsymbol{u}$
34:     return $\boldsymbol{d}$
35: end procedure
```

Claim 3.5.3. *Let $\boldsymbol{u}^{(-1)} = \boldsymbol{d}$ be the value of $\boldsymbol{u}$ in* PartialProject$(\boldsymbol{d}, \mathcal{H})$ *before the first double for-loop. Let $\boldsymbol{u}^{(i)}$ be the value of $\boldsymbol{u}$ after iteration i of the outer loop (Line 23) for $0 \le i < \eta$. Then*

$$\boldsymbol{u}^{(i)} = \mathbf{\Pi}^{(i)} \cdots \mathbf{\Pi}^{(0)} \boldsymbol{d}.$$

Furthermore, $\boldsymbol{u}^{(i)}|_{F_H} \neq \mathbf{0}$ only if $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$.

Proof. We prove the claim by induction. For $i = -1$, we are given $\boldsymbol{u}^{(-1)}|_{F_H} = \boldsymbol{d}|_{F_H} \neq \mathbf{0}$ exactly for all $H \in \mathcal{H} \subseteq \mathcal{P}_{\mathcal{T}}(\mathcal{H})$.

For $i+1$, we have, by inductive hypothesis and definition of $\mathbf{\Pi}^{(i)}$,

$$\begin{aligned}\mathbf{\Pi}^{(i+1)}\mathbf{\Pi}^{(i)} \cdots \mathbf{\Pi}^{(0)} \boldsymbol{d} &= \mathbf{\Pi}^{(i+1)} \boldsymbol{u}^{(i)} \\ &= \left(\mathbf{I} - \sum_{H \in \mathcal{T}(i+1)} \mathbf{X}^{(H)} \right) \boldsymbol{u}^{(i)}.\end{aligned}$$

Since $\mathbf{X}^{(H)} \in \mathbb{R}^{\partial(H) \times F_H}$ and $\boldsymbol{u}^{(i)}|_{F_H} \neq \mathbf{0}$ only if $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$, the summation above can be taken over the smaller set $\mathcal{T}(i+1) \cap \mathcal{P}_{\mathcal{T}}(\mathcal{H}) \stackrel{\text{def}}{=} \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i+1)$, giving

$$= \boldsymbol{u}^{(i)} - \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i+1)} \mathbf{X}^{(H)} \boldsymbol{u}^{(i)}|_{F_H}.$$

This is exactly what is computed as $\boldsymbol{u}$ after iteration i of the outer loop at Line 23. Hence, this is equal to $\boldsymbol{u}^{(i+1)}$ by definition.

For the sparsity condition, we note that if $\boldsymbol{u}^{(i+1)}|_{F'_H}$ differs from $\boldsymbol{u}^{(i)}|_{F'_H}$ at a node H', then it was changed by a term in the summation above, and so we must have $F_{H'} \cap \partial H \neq \emptyset$ for some $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i+1)$. By construction of the separator tree, this occurs only if H' is an ancestor of H, which implies $H' \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$. Combined with the inductive hypothesis, we have that $\boldsymbol{u}^{(i+1)}|_{F_H} \neq \mathbf{0}$ only if $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$. $\square$

Setting $i = \eta - 1$ in the above claim immediately shows that at the end of the first double for-loop in PartialProject, we have $\boldsymbol{u} = \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(1)} \mathbf{\Pi}^{(0)} \boldsymbol{d}$.

Finally, to complete the sparsity argument, we have $|\mathcal{H}| = K$, and consequently $|\mathcal{P}_{\mathcal{T}}(\mathcal{H})| =$

$O(K \cdot \eta) = \widetilde{O}(K)$. Combined with the claim, we get the overall sparsity guarantee. $\square$

For the correctness of our data structure, we will need a more specific structural property of PartialProject:

Lemma 3.5.4. *Let $\mathcal{H}$ be any subset of nodes in $\mathcal{T}$. Let $H_1, \ldots, H_r$ be any permutation of all nodes from $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ such that if H_i is an ancestor of H_j, then $i < j$. Then*

$$\textsc{PartialProject}(\boldsymbol{d}, \mathcal{H}) = (\mathbf{I} - \mathbf{X}^{(H_1)}) \ldots (\mathbf{I} - \mathbf{X}^{(H_r)})\boldsymbol{d}.$$

Proof. First, we observe that $\mathbf{I} - \mathbf{X}^{(H_i)}$ and $\mathbf{I} - \mathbf{X}^{(H_j)}$ are commutative if H_i and H_j are not ancestor-descendants. The reason is that $\mathbf{X}^{(H_i)}\mathbf{X}^{(H_j)} = \mathbf{0}$, since $\mathbf{X}^{(H_i)} \in \mathbb{R}^{\partial(H_i) \times F_{H_i}}$, and $F_{H_i} \cap \partial(H_j) \neq \emptyset$ only if H_i is an ancestor of H_j.

From the proof of Claim 3.5.3, we observe that iteration i of the for-loop in PartialProject applies the operator

$$\mathbf{I} - \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i)} \mathbf{X}^{(H)} = \prod_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i)} (\mathbf{I} - \mathbf{X}^{(H)}),$$

where the equality follows from expanding the RHS and applying the property $\mathbf{X}^{(H_i)}\mathbf{X}^{(H_j)} = \mathbf{0}$. Thus, we have a stricter version of the claim:

$$\textsc{PartialProject}(\boldsymbol{d}, \mathcal{H}) = (\mathbf{I} - \mathbf{X}^{(H_1)}) \ldots (\mathbf{I} - \mathbf{X}^{(H_r)})\boldsymbol{d},$$

where $H_1, \ldots, H_r$ is any permutation of $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ such that nodes at lower levels come later. Then we apply commutativity to allow $H_1, \ldots, H_r$ to be any permutation such that if H_i is an ancestor of H_j then $i < j$. $\square$

Next, we show there is a procedure that reverses PartialProject using select nodes of $\mathcal{T}$.

Lemma 3.5.5. *Given a set of K nodes $\mathcal{H}$ in $\mathcal{T}$ and a vector $\boldsymbol{u}$, InversePartialProject$(\boldsymbol{u}, \mathcal{H})$ in the MaintainZ data structure (Algorithm 14) is a procedure that returns $\boldsymbol{d}$ such that*

$$\boldsymbol{d} = (\mathbf{I} + \mathbf{X}^{(H_r)}) \ldots (\mathbf{I} + \mathbf{X}^{(H_1)})\boldsymbol{u},$$

where $H_1, \ldots, H_r$ is any permutation of all nodes from $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ such that if H_i is an ancestor of H_j, then $i < j$. The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where $K = |\mathcal{H}|$.

Proof. Intuitively, observe that InversePartialProject is reversing all the operations in PartialProject. The runtime analysis is analogous to PartialProject. The proof of the equation is also analogous to PartialProject. We first observe that iteration i of the for-loop applies the operator

$$\mathbf{I} + \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i)} \mathbf{X}^{(H)} = \prod_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, i)} (\mathbf{I} + \mathbf{X}^{(H)}).$$

Then by commutativity as in Lemma 3.5.4, we have

$$\boldsymbol{d} = (\mathbf{I} + \mathbf{X}^{(H_r)}) \ldots (\mathbf{I} + \mathbf{X}^{(H_1)}) \boldsymbol{u}.$$

where $H_1, \ldots, H_r$ is any permutation of $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ such that nodes at lower levels come later. Then we apply commutativity to allow $H_1, \ldots, H_r$ to be any permutation such that if H_i is an ancestor of H_j then $i < j$. $\square$

Finally, we have the data structure for maintaining a vector $\boldsymbol{z}$ dependent on $\boldsymbol{v}$ throughout the IPM. For one IPM step, there is one call to Reweight followed by one call to Move.

Theorem 3.5.6 (Maintain intermediate vector $\boldsymbol{z}$)**.** *Given a modified planar graph G with n vertices and m edges and its separator tree $\mathcal{T}$ with height η, the deterministic data structure MaintainZ (Algorithm 14) maintains the following variables correctly at the end of each IPM step:*

- *the dynamic edge weights $\boldsymbol{w}$ is and current step direction $\boldsymbol{v}$ from the IPM*
- *a DynamicSC data structure on $\mathcal{T}$ based on the current edge weights $\boldsymbol{w}$*
- *scalar c and vectors $\boldsymbol{z}^{(\mathrm{prev})}, \boldsymbol{z}^{(\mathrm{sum})}$, which together represent $\boldsymbol{z} = c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})}$, such that at the end of IPM step k,*
$$\boldsymbol{z} = \sum_{i=1}^{k} \boldsymbol{z}^{(i)}. \tag{3.21}$$
- *$\boldsymbol{z}^{(\mathrm{prev})}$ satisfies $\boldsymbol{z}^{(\mathrm{prev})} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)} \mathbf{B}^{\top} \mathbf{W}^{1/2} \boldsymbol{v}$.*

Algorithm 14 Data structure to maintain the intermediate vector $\boldsymbol{z}$, Part 2

```
36: procedure Reweight($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m_{>0}$)
37:     $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})}$
38:     $\mathcal{H} \leftarrow$ set of leaf nodes in $\mathcal{T}$ that contain all the edges of $G$ whose weight has changed
39:     $\Delta\boldsymbol{u} \leftarrow$ PartialProject($\mathbf{B}^\top(\mathbf{W}^{(\text{new})1/2} - \mathbf{W}^{1/2})\boldsymbol{v}$)
40:     $\boldsymbol{u} \leftarrow \boldsymbol{u} + \Delta\boldsymbol{u}$
41:     $\boldsymbol{d} \leftarrow$ InversePartialProject($\boldsymbol{u}, \mathcal{H}$)          ▷ revert projection with old weights
42:     dynamicSC.Reweight($\boldsymbol{w}^{(\text{new})}$)          ▷ update $\mathbf{L}^{(H)}$'s to use the new weights
43:                    ▷ specifically, $\mathbf{L}^{(H)}$ changes for each $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$
44:     $\boldsymbol{u} \leftarrow$ PartialProject($\boldsymbol{d}, \mathcal{H}$)          ▷ apply projection with new weights
45:     $\boldsymbol{y} \leftarrow \boldsymbol{z}^{(\text{prev})}$          ▷ backup copy of $\boldsymbol{z}^{(\text{prev})}$
46:     for $H$ in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ do
47:         $\boldsymbol{z}^{(\text{prev})}|_{F_H} \leftarrow (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{u}|_{F_H}$
48:     end for
49:     $\boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c \cdot (\boldsymbol{z}^{(\text{prev})} - \boldsymbol{y})$          ▷ update $\boldsymbol{z}^{(\text{sum})}$ to maintain the invariant
50: end procedure
51:
52: procedure Move($\alpha \in \mathbb{R}, \boldsymbol{v}^{(\text{new})} \in \mathbb{R}^m$)
53:     $\Delta\boldsymbol{v} \leftarrow \boldsymbol{v}^{(\text{new})} - \boldsymbol{v}$
54:     $\boldsymbol{v} \leftarrow \boldsymbol{v}^{(\text{new})}$
55:     $\Delta\boldsymbol{u} \leftarrow$ PartialProject($\mathbf{B}^\top\mathbf{W}^{1/2}\Delta\boldsymbol{v}$)
56:     $\boldsymbol{u} \leftarrow \boldsymbol{u} + \Delta\boldsymbol{u}$
57:     $\boldsymbol{y} \leftarrow \boldsymbol{z}^{(\text{prev})}$          ▷ backup copy of $\boldsymbol{z}^{(\text{prev})}$
58:     for $H$ in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ do
59:         $\boldsymbol{z}^{(\text{prev})}|_{F_H} \leftarrow (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{u}|_{F_H}$
60:     end for
61:     $\boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c \cdot (\boldsymbol{z}^{(\text{prev})} - \boldsymbol{y})$
62:     $c \leftarrow c + \alpha$
63: end procedure
```

The data structure supports the following procedures:

- Initialize$(G,$ *separator tree* $\mathcal{T}, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0)$*: Given a graph* G*, its separator tree* $\mathcal{T}$*, initial step direction* $\boldsymbol{v}$*, initial weights* $\boldsymbol{w}$*, and target projection matrix accuracy* $\epsilon_{\mathbf{P}}$*, preprocess in* $\widetilde{O}(\delta^{-2}m)$ *time and initialize* $\boldsymbol{z} = \mathbf{0}$*.*

- Reweight$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$ *given implicitly as a set of changed coordinates): Update the current weight to* $\boldsymbol{w}$ *and update* DynamicSC*, and update the representation of* $\boldsymbol{z}$*. The procedure runs in* $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ *total time, where* K *is the number of coordinates updated in* $\boldsymbol{w}$*. There are most* $\widetilde{O}(K)$ *nodes* $H \in \mathcal{T}$ *for which* $\boldsymbol{z}^{(\mathrm{prev})}|_{F_H}$ *and* $\boldsymbol{z}^{(\mathrm{sum})}|_{F_H}$ *are updated.*

- Move$(\alpha \in \mathbb{R},$ $\boldsymbol{v} \in \mathbb{R}^n$ *given implicitly as a set of changed coordinates): Update the current direction to* $\boldsymbol{v}$*, and set* $\boldsymbol{z} \leftarrow \boldsymbol{z} + \alpha\widetilde{\boldsymbol{\Gamma}}\boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$ *with the correct representation. The procedure runs in* $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ *time, where* K *is the number of coordinates changed in* $\boldsymbol{v}$ *compared to the previous IPM step.*

Proof. If Move is implemented correctly, then by the definition of the update to $\boldsymbol{z}$, the invariant in (3.21) is correctly maintained.

For the runtime analysis, recall $\{F_H : H \in \mathcal{T}\}$ partition the vertex set of G. Therefore $\boldsymbol{v}$ has K non-zero entries, then $\boldsymbol{d} \overset{\mathrm{def}}{=} \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$ has $O(K)$ non-zero entries, and consequently $\boldsymbol{d}|_{F_H} \neq \mathbf{0}$ for $O(K)$ nodes H. There are $O(m)$ total nodes in the separator tree $\mathcal{T}$.

We maintain a vector $\boldsymbol{u}$ with the invariant $\boldsymbol{u} = \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$. We now prove the correctness and runtime of each procedure separately.

Initialize: By the guarantee of Lemma 3.5.2, at the end of Initialize, we have

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

and

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

Since c and $\boldsymbol{z}^{(\mathrm{sum})}$ are initialized to zero, we have $\boldsymbol{z} = c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})} = \mathbf{0}$.

We initialize the DYNAMICSC data structure in $\varnothing(\delta^{-2}m)$ time. There is no sparsity guarantee for $\boldsymbol{v}$, but the call to PARTIALPROJECT takes at most $O(\delta^{-2}m)$ time because of the size of $\mathcal{T}$. To calculate $\widetilde{\boldsymbol{\Gamma}}\boldsymbol{u}$, we solve a Laplacian system $(\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{u}|_{F_H}$ in time $\widetilde{O}(|\mathbf{L}^{(H)}|)$ for each node H. The total time is $\widetilde{O}(\delta^{-2}m)$ as well by $|\mathbf{L}^{(H)}| = \widetilde{O}\left(\delta^{-2}|F_H \cup \partial(H)|\right)$ from Theorem 3.2.4 and by Lemma 3.4.16.

Move: Let $\boldsymbol{v}, \boldsymbol{u}$ be the variables at the start of MOVE, and let $\boldsymbol{v}', \boldsymbol{u}'$ denote them at the end. Similarly, let $\boldsymbol{z} = c\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})}$ denote $\boldsymbol{z}$ and the respective variables at the start of MOVE, and let $\boldsymbol{z}' = c'\boldsymbol{z}^{(\text{prev})'} + \boldsymbol{z}^{(\text{sum})'}$ denote these variables at the end.

First, after Line 56, we have

$$\begin{aligned} \boldsymbol{u}' &= \boldsymbol{u} + \Delta\boldsymbol{u} \\ &= \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}(\boldsymbol{v} + \Delta\boldsymbol{v}) \\ &= \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}', \end{aligned}$$

where the second equality follows from the guarantee of PARTIALPROJECT and the guarantee from the previous IPM step. By Lemma 3.5.2, $\boldsymbol{u}'$ is updated only on F_H where $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$. Thus, to update $\boldsymbol{z}^{(\text{prev})'} = \widetilde{\boldsymbol{\Gamma}}\boldsymbol{u}'$, we only need to update $\boldsymbol{z}^{(\text{prev})'}|_{F_H}$ for $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$, which happens on Line 59. Observe that the update in value to $\boldsymbol{z}^{(\text{prev})}$ is cancelled out by the update in $\boldsymbol{z}^{(\text{sum})}$ at Line 61, so that the value of $\boldsymbol{z}$ does not change overall up to that point. But we have

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

Then in Line 62, incrementing c by α represents increasing the value of $\boldsymbol{z}$ by $\alpha\boldsymbol{z}^{(\text{prev})'}$, which is exactly the desired update.

For the runtime, first note $nnz(\Delta\boldsymbol{v}) = K$. So PARTIALPROJECT runs in $\widetilde{O}(\delta^{-2}\sqrt{mK})$ time by Lemma 3.5.2. Line 59 takes $\widetilde{O}(\delta^{-2}\sqrt{mK})$ time in total by Theorem 3.2.4 and Lemma 3.4.16. The remaining operations in the procedure are adding vectors with bounded sparsity.

Reweight: Let $\boldsymbol{w}^{(\text{old})}$ denote the weight vector immediately before this procedure is called, and $\boldsymbol{w}^{(\text{new})}$ is the new weight passed in as an argument.

Let $\widetilde{\boldsymbol{\Gamma}}$ and $\boldsymbol{\Pi}^{(i)}$ denote these matrices defined using the old weights, and let $\widetilde{\boldsymbol{\Gamma}}'$ and $\boldsymbol{\Pi}^{(i)'}$ denote the matrices using the new weights. Similarly let $\boldsymbol{u}$ be the state of the vector at the start of the procedure call and $\boldsymbol{u}'$ at the end.

In REWEIGHT, we do not change the value of $\boldsymbol{z}$, but rather update $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ so that at the end of the procedure,

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

so that we maintain the invariant claimed in the theorem statement.

To see that the value of $\boldsymbol{z}$ does not change at the end of the procedure, observe that we modify $\boldsymbol{z}^{(\text{prev})}$ during the procedure, and cancel all the changes to $\boldsymbol{z}^{(\text{prev})}$ by updating $\boldsymbol{z}^{(\text{sum})}$ appropriately at the last line (Line 49).

Immediately before Line 40, the algorithm invariant guarantees

$$\boldsymbol{u} = \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top \mathbf{W}^{(\text{old})1/2}\boldsymbol{v}.$$

By Lemma 3.5.2,

$$\Delta\boldsymbol{u} = \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top \left(\mathbf{W}^{(\text{new})1/2} - \mathbf{W}^{(\text{old})1/2}\right)\boldsymbol{v}.$$

Therefore, after executing Line 40, we have

$$\boldsymbol{u} \leftarrow \boldsymbol{u} + \Delta\boldsymbol{u} = \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}\mathbf{B}^\top \mathbf{W}^{(\text{new})1/2}\boldsymbol{v}.$$

Next, we need to update $\boldsymbol{u}$ to reflect the changes to $\widetilde{\boldsymbol{\Gamma}}, \boldsymbol{\Pi}^{(i)}$. Updating these matrices is done via `dynamicSC`. However, calling PARTIALPROJECT($\mathbf{B}^\top \mathbf{W}^{(\text{new})1/2}\boldsymbol{v}$) afterwards is too costly if done directly, since the argument is a dense vector. To circumvent this problem, we make the key observation that the change to $\boldsymbol{u}$ is restricted to a subcollection of nodes on $\mathcal{T}$ (in fact a connected subtree containing the root), and it suffices to partially reverse and reapply the operator $\widetilde{\boldsymbol{\Gamma}}\boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)}$.

Intuitively, InversePartialProject revert all computations in PartialProject that are related to the changes to $\mathbf{W}$.

Let $H_1, \dots, H_t$ be a permutation of all nodes in $\mathcal{T}$, such that the nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ is a prefix of the permutation, and it satisfies that for any node H_i with descendant H_j, $i < j$. Then by Lemma 3.5.4, after executing Line 40, we have

$$\begin{aligned} \boldsymbol{u} &= \textsc{PartialProject}(\mathbf{B}^\top \mathbf{W}^{(\text{new})1/2} \boldsymbol{v}, \mathcal{T}) \\ &= (\mathbf{I} - \mathbf{X}^{(H_1)}) \dots (\mathbf{I} - \mathbf{X}^{(H_t)}) \mathbf{B}^\top \mathbf{W}^{(\text{new})1/2} \boldsymbol{v}. \end{aligned} \tag{3.22}$$

Let $r = |\mathcal{P}_{\mathcal{T}}(\mathcal{H})|$. Then InversePartialProject$(\boldsymbol{u}, \mathcal{H})$ on Line 41 returns $\boldsymbol{d}$ by Lemma 3.5.5 satisfying

$$\boldsymbol{d} = (\mathbf{I} + \mathbf{X}^{(H_r)}) \dots (\mathbf{I} + \mathbf{X}^{(H_1)}) \boldsymbol{u}.$$

Plugging in $\boldsymbol{u}$ from (3.22), we have

$$\boldsymbol{d} = (\mathbf{I} + \mathbf{X}^{(H_r)}) \dots (\mathbf{I} + \mathbf{X}^{(H_1)})(\mathbf{I} - \mathbf{X}^{(H_1)}) \dots (\mathbf{I} - \mathbf{X}^{(H_t)}) \mathbf{B}^\top \mathbf{W}^{(\text{new})1/2} \boldsymbol{v}.$$

We use the fact that each $\mathbf{I} - \mathbf{X}^{(H_i)}$ is nonsingular and has inverse $\mathbf{I} + \mathbf{X}^{(H_i)}$ to get

$$\boldsymbol{d} = (\mathbf{I} - \mathbf{X}^{(H_{r+1})}) \dots (\mathbf{I} - \mathbf{X}^{(H_t)}) \mathbf{B}^\top \mathbf{W}^{(\text{new})1/2} \boldsymbol{v}.$$

We then call `dynamicSC`.Reweight, which updates $\mathbf{L}^{(H)}$ and in turn $\mathbf{X}^{(H_i)}$ for precisely all nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H}) = \{H_1, \dots, H_r\}$. Let $\mathbf{X}^{(H)'}$ denote the matrix after reweight. Next, we call PartialProject again. Let us denote it by PartialProject$^{(\text{new})}$ to emphasize that it runs with new weights. This gives

$$\begin{aligned}
\boldsymbol{u}' &= \textsc{PartialProject}^{(\text{new})}(\boldsymbol{d}, \mathcal{H}) \\
&= (\mathbf{I} - \mathbf{X}^{(H_1)'}) \dots (\mathbf{I} - \mathbf{X}^{(H_r)'})\boldsymbol{d} \\
&= (\mathbf{I} - \mathbf{X}^{(H_1)'}) \dots (\mathbf{I} - \mathbf{X}^{(H_r)'})(\mathbf{I} - \mathbf{X}^{(H_{r+1})}) \dots (\mathbf{I} - \mathbf{X}^{(H_t)})\mathbf{B}^\top \mathbf{W}^{(\text{new})1/2}\boldsymbol{v} \\
&= (\mathbf{I} - \mathbf{X}^{(H_1)'}) \dots (\mathbf{I} - \mathbf{X}^{(H_t)'})\mathbf{B}^\top \mathbf{W}^{(\text{new})1/2}\boldsymbol{v} \qquad (\text{since } \mathbf{X}^{(H_i)'} = \mathbf{X}^{(H_i)} \text{ for all } i > r) \\
&= \textsc{PartialProject}^{(\text{new})}(\mathbf{B}^\top \mathbf{W}^{(\text{new})1/2}\boldsymbol{v}, \mathcal{T}).
\end{aligned}$$

Because $\boldsymbol{u}'|_{F_H}$ is updated on $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$, and $\mathbf{L}^{(H)}$ is updated on $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ by Theorem 3.2.4, running Line 47 on $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ correctly sets $\boldsymbol{z}^{(\text{prev})'} = \widetilde{\mathbf{\Gamma}}'\boldsymbol{u}'$.

For the runtime, the first call to PartialProject has a vector with $O(K)$ sparsity as the argument, and therefore runs in $\widetilde{O}(\delta^{-2}\sqrt{mK})$. Next, we know $|\mathcal{H}| = O(K)$. The call to InverseP-artialProject and the subsequent call to PartialProject both have $\mathcal{H}$ as an argument, so they run in $\widetilde{O}(\delta^{-2}\sqrt{mK})$. The `DynamicSC`.Reweight call runs in $\widetilde{O}(\delta^{-2}\sqrt{mK})$. Updating $\boldsymbol{z}^{(\text{prev})}$ (Line 47) takes $\widetilde{O}(\delta^{-2}\sqrt{mK})$ time in total by Theorem 3.2.4 and Lemma 3.4.16. And finally we can update $\boldsymbol{z}^{(\text{sum})}$ in the same time.

We remark that although InversePartialProject returns a vector $\boldsymbol{d}$ that is not necessarily sparse, and we then assign $\boldsymbol{u} \leftarrow$ PartialProject$(\boldsymbol{d}, \mathcal{H})$, this is for readability. $\boldsymbol{d}$ is in fact an intermediate state of $\boldsymbol{u}$, on which we perform in-place operations. □

3.5.2 Tree operator

At IPM step k, our goal is to write the slack update $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ as $\mathbf{M}^{(\text{slack})}\boldsymbol{z}^{(k)}$, and similarly, write the partial flow update $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ approximately as $\mathbf{M}^{(\text{flow})}\boldsymbol{z}^{(k)}$, where $\boldsymbol{z}^{(k)}$ is defined in the previous subsection, and $\mathbf{M}^{(\text{slack})}$ and $\mathbf{M}^{(\text{flow})}$ are linear operators that are efficiently maintainable between IPM steps.

In this section, we define a general class of operators called *tree operators* and show how to efficiently compute and maintain them. In later sections, we show that $\mathbf{M}^{(\text{slack})}$ and $\mathbf{M}^{(\text{flow})}$ can be defined as tree operators.

We begin with the formal definitions. Recall for a tree $\mathcal{T}$ and node $H \in \mathcal{T}$, we use $\mathcal{T}_H$ to denote the subtree rooted at H.

Definition 3.5.7 (Tree operator). Suppose $\mathcal{T}$ is a rooted tree with constant degree. Let each node $H \in \mathcal{T}$ be associated with two sets $V(H)$ and $F_H \subseteq V(H)$. Let each leaf node $H \in \mathcal{T}$ be further associated with a non-empty set $E(H)$ of constant size, where *the $E(H)$'s are pairwise disjoint over all leaf nodes.* For a non-leaf node H, define $E(H) \overset{\text{def}}{=} \bigcup_{\text{leaf } D \in \mathcal{T}_H} E(D)$. Finally, define $E \overset{\text{def}}{=} E(G) \bigcup_{\text{leaf } H \in \mathcal{T}} E(H)$ and $V \overset{\text{def}}{=} V(G) = \bigcup_{H \in \mathcal{T}} V(H)$, where G is the root node of $\mathcal{T}$.

Let each node H with parent P be associated with a linear *edge operator* $\mathbf{M}_{(H,P)} : \mathbb{R}^{V(P)} \mapsto \mathbb{R}^{V(H)}$. In addition, let each leaf node H be associated with a *constant-time computable* linear *leaf operator* $\mathbf{J}_H : \mathbb{R}^{V(H)} \mapsto \mathbb{R}^{E(H)}$. We extend all these operators trivially to $\mathbb{R}^V$ and $\mathbb{R}^E$ respectively, in order to have matching dimensions overall. When a edge or leaf operator is not given, we assume it to be $\mathbf{0}$.

For a path $H_t \to H_1 \overset{\text{def}}{=} (H_t, \ldots, H_1)$, where each H_i is the parent of H_{i-1} and H_1 is a leaf node (call these *tree paths*), we define

$$\mathbf{M}_{H_1 \leftarrow H_t} = \mathbf{M}_{(H_1,H_2)} \mathbf{M}_{(H_2,H_3)} \cdots \mathbf{M}_{(H_{t-1},H_t)}.$$

If $t = 1$, then $\mathbf{M}_{H_1 \leftarrow H_t} \overset{\text{def}}{=} \mathbf{I}$.

We define *the tree operator* $\mathbf{M} : \mathbb{R}^V \mapsto \mathbb{R}^E$ *supported on* $\mathcal{T}$ to be

$$\mathbf{M} \overset{\text{def}}{=} \sum_{\text{leaf } H,\ \text{node } A \,:\, H \in \mathcal{T}_A} \mathbf{J}_H \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A}. \tag{3.23}$$

We always maintain a tree operator implicitly by maintaining

$$\{\mathbf{J}_H : \text{leaf } H\} \cup \{\mathbf{M}_{(H,P)} : \text{edge } (H,P)\} \cup \{F_H : \text{node } H\}.$$

Remark 3.5.8. Although we define the tree operator in general and hope it will find applications in other problems, we have used suggestive names in the definition to suit our min-cost flow setting. In particular, our tree operators will be supported on the separator tree $\mathcal{T}$. For each node H, the sets $V(H), F_H, E(H)$ associated with the tree operator are, respectively, $\partial H \cup F_H$ of region H, the eliminated vertices F_H of region H, and the edge set of region H, all from the separator tree construction.

To maintain $\mathbf{M}$ using the tree efficiently, we also need some partial operators:

Definition 3.5.9 ($\mathbf{M}^{(H)}, \overline{\mathbf{M}^{(H)}}$). For notational convenience, define $\mathcal{T}_H$ to be the subtree of $\mathcal{T}$ rooted at H.

We define the subtree operator $\mathbf{M}^{(H)} : V(H) \mapsto E(H)$ at each node H to be

$$\mathbf{M}^{(H)} \stackrel{\text{def}}{=} \sum_{\text{leaf } D \in \mathcal{T}_H} \mathbf{J}_D \mathbf{M}_{D \leftarrow H}. \tag{3.24}$$

We also define the partial sum

$$\overline{\mathbf{M}^{(H)}} \stackrel{\text{def}}{=} \sum_{D \in \mathcal{T}_H} \mathbf{M}^{(D)} \mathbf{I}_{F_D}. \tag{3.25}$$

We state a straightforward corollary based on the definitions without proof.

Corollary 3.5.10. *For any node $H \in \mathcal{T}$,*

$$\mathbf{M} = \sum_{H \in \mathcal{T}} \mathbf{M}^{(H)} \mathbf{I}_{F_H} = \overline{\mathbf{M}^{(G)}},$$

where G is the root node of $\mathcal{T}$.

Furthermore, if H has with children D_1, D_2, then

$$\mathbf{M}^{(H)} = \mathbf{M}^{(D_1)} \mathbf{M}_{(D_1,H)} + \mathbf{M}^{(D_2)} \mathbf{M}_{(D_2,H)}. \tag{3.26}$$

We define the complexity of a tree operator to be parameterized by the number of tree edges.

Definition 3.5.11 (Complexity of tree operator). Let $\mathbf{M}$ be a tree operator on tree $\mathcal{T}$. We say $\mathbf{M}$ has complexity function T, if for any $k > 0$, for any set S of k distinct edges in $\mathcal{T}$ and any families of vectors $\{\boldsymbol{u}_e : e \in S\}$ and $\{\boldsymbol{v}_e : e \in S\}$, the total cost of computing $\{\boldsymbol{u}_e^\top \mathbf{M}_e : e \in S\}$ and $\{\mathbf{M}_e \boldsymbol{v}_e : e \in S\}$ is bounded by $T(k)$.

Without loss of generality, we may assume $T(0) = 0$, $T(k) \geq k$, and T is concave.

We can show the structure of a tree operator by the procedure ComputeMz($\mathbf{M}, \boldsymbol{z}$) to compute $\mathbf{M}\boldsymbol{z}$. Intuitively, $\boldsymbol{z}$ is given as input to each node H. The edge operators are concatenated in the order of tree paths from H to a leaf, but we apply them level-wise in descending order.

Algorithm 15 Compute $\mathbf{M}\boldsymbol{z}$ for a tree operator $\mathbf{M}$

```
1: procedure ComputeMz($\mathbf{M}, \boldsymbol{z}$)
2:     $\mathcal{H} \leftarrow$ set of all nodes $H$ in $\mathcal{T}$ such that $\mathbf{M}_{(H,P)}$ or $\mathbf{J}_H$ is nonzero
3:     $\mathcal{P}_{\mathcal{T}}(\mathcal{H}) \leftarrow$ set of $\mathcal{H}$ and all ancestor nodes of $\mathcal{H}$ in $\mathcal{T}$
4:     $\boldsymbol{v}_H \leftarrow \mathbf{0}$ for each $H \in \mathcal{T}$          ▷ sparse vectors for intermediate computations
5:     for each node $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ do
6:         $\boldsymbol{v}_H \leftarrow \mathbf{I}_{F_H}\boldsymbol{z} = \boldsymbol{z}|_{F_H}$          ▷ apply the $\mathbf{I}_{F_H}$ part of the operator
7:     end for
8:     for each node $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ by decreasing level do
9:         Let $P$ be the parent of $H$
10:        $\boldsymbol{v}_H \leftarrow \boldsymbol{v}_H + \mathbf{M}_{(H,P)}\boldsymbol{v}_P$          ▷ apply $\mathbf{M}_{(H,P)}$ as we move from $P$ to $H$
11:    end for
12:    for each leaf node $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ do
13:        $\boldsymbol{x}|_{E(H)} \leftarrow \mathbf{J}_H\boldsymbol{v}_H$          ▷ apply the leaf operator
14:    end for
15:    return $\boldsymbol{x}$
16: end procedure
```

Corollary 3.5.12. *Suppose* $\mathbf{M} : \mathbb{R}^V \to \mathbb{R}^E$ *is a tree operator on tree* $\mathcal{T}$ *with complexity* T*, where* $|V| = n$ *and* $|E| = m$*. Then for* $\boldsymbol{z} \in \mathbb{R}^V$*,* Exact($\mathbf{M}, \boldsymbol{z}$) *outputs* $\mathbf{M}\boldsymbol{z}$ *in* $O(T(K)) = O(T(m))$ *time where* K *is the total number of non-zero edge and leaf operators in* $\mathbf{M}$*.*

Proof. Note only non-zero edge and leaf operators contribute to $\mathbf{M}\boldsymbol{z}$. We omit the proof of correctness as it is simply an application of the definition.

Since $E = \cup_{\text{leaf } D} E(D)$, and each $E(D)$ has constant size, we know there are at most $O(m)$ leaves in $\mathcal{T}$. Hence, there are $O(m)$ edges in $\mathcal{T}$, and $K = O(m)$. Since we define each leaf operator to be constant time computable, applying $\mathbf{J}_H$ for leaves in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ costs $O(K)$ time in total. The bottleneck of the procedure is to apply the edge operator $\mathbf{M}_e$ to some vector exactly once for each edge e in $\mathcal{T}$; the time cost is $O(T(K))$ by definition of the operator complexity. □

3.5.3 Proof of Theorem 3.2.5

Finally, we give the data structure for maintaining an implicit representation of the form $\boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ throughout the IPM. For an instantiation of this data structure, there is exactly one call to Initialize at the very beginning, and one call to Exact at the very end. Otherwise, each step of the IPM consists of one call to Reweight followed by one call to Move. Note that this data structure *extends* MaintainZ in the object-oriented programming sense.

Theorem 3.2.5 (Implicit representation maintenance)**.** *Given a modified planar graph G with n vertices and m edges, and its separator tree $\mathcal{T}$ with height η, the deterministic data structure MaintainRep (Algorithm 16) maintains the following variables correctly at the end of every IPM step:*

- *the dynamic edge weights $\boldsymbol{w}$ and step direction $\boldsymbol{v}$ from the current IPM step,*
- *a DynamicSC data structure on $\mathcal{T}$ based on the current edge weights $\boldsymbol{w}$,*
- *an implicitly represented tree operator $\mathbf{M}$ supported on $\mathcal{T}$ with complexity $T(K)$,* computable using information from DynamicSC*,*
- *scalar c and vectors $\boldsymbol{z}^{(\mathrm{prev})}, \boldsymbol{z}^{(\mathrm{sum})}$, which together represent $\boldsymbol{z} = c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})}$, such that at the end of step k,*
$$\boldsymbol{z} = \sum_{i=1}^{k} \alpha^{(i)} \boldsymbol{z}^{(i)},$$
  where $\alpha^{(i)}$ is the step size α given in Move for step i,
- *$\boldsymbol{z}^{(\mathrm{prev})}$ satisfies $\boldsymbol{z}^{(\mathrm{prev})} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}$,*
- *an offset vector $\boldsymbol{y}$ which together with $\mathbf{M}, \boldsymbol{z}$ represent $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$, such that after step k,*
$$\boldsymbol{x} = \boldsymbol{x}^{(\mathrm{init})} + \sum_{i=1}^{k} \mathbf{M}^{(i)}(\alpha^{(i)}\boldsymbol{z}^{(i)}),$$
  where $\boldsymbol{x}^{(\mathrm{init})}$ is an initial value from Initialize, and $\mathbf{M}^{(i)}$ is the state of $\mathbf{M}$ after step i.

The data structure supports the following procedures:

- *Initialize*$(G, \mathcal{T}, \mathbf{M}, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \boldsymbol{x}^{(\mathrm{init})} \in \mathbb{R}^m, \epsilon_{\mathbf{P}} > 0)$*: Given a graph* G*, its separator tree* $\mathcal{T}$*, a tree operator* $\mathbf{M}$ *supported on* $\mathcal{T}$ *with complexity* T*, initial step direction* $\boldsymbol{v}$*, initial weights* $\boldsymbol{w}$*, initial vector* $\boldsymbol{x}^{(\mathrm{init})}$*, and target projection matrix accuracy* $\epsilon_{\mathbf{P}}$*, preprocess in* $\widetilde{O}(\delta^{-2}m + T(m))$ *time and set* $\boldsymbol{x} \leftarrow \boldsymbol{x}^{(\mathrm{init})}$.

- *Reweight*$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$ *given implicitly as a set of changed coordinates): Update the weights to* $\boldsymbol{w}$*. Update the implicit representation of* $\boldsymbol{x}$ *without changing its value, so that all the variables in the data structure are based on the new weights.*

  The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK} + T(K))$ *total time, where* K *is an upper bound on the number of coordinates changed in* $\boldsymbol{w}$ *and the number of leaf or edge operators changed in* $\mathbf{M}$*. There are most* $\widetilde{O}(K)$ *nodes* $H \in \mathcal{T}$ *for which* $\boldsymbol{z}^{(\mathrm{prev})}|_{F_H}$ *and* $\boldsymbol{z}^{(\mathrm{sum})}|_{F_H}$ *are updated.*

- *Move*$(\alpha \in \mathbb{R},\ \boldsymbol{v} \in \mathbb{R}^n$ *given implicitly as a set of changed coordinates): Update the current direction to* $\boldsymbol{v}$*, and then* $\boldsymbol{z}^{(\mathrm{prev})}$ *to maintain the claimed invariant. Update the implicit representation of* $\boldsymbol{x}$ *to reflect the following change in value:*

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathbf{M}(\alpha \boldsymbol{z}^{(\mathrm{prev})}).$$

  The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ *time, where* K *is the number of coordinates changed in* $\boldsymbol{v}$ *compared to the previous IPM step.*

- *Exact*()*: Output the current exact value of* $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ *in* $\widetilde{O}(T(m))$ *time.*

Proof. First, we discuss how $\mathbf{M}$ is stored in the data structure: Recall $\mathbf{M}$ is represented implicitly by a collection of edge operators and leaf operators on the separator tree $\mathcal{T}$, so that each edge operator is stored at a corresponding node of $\mathcal{T}$, and each leaf operator is stored at a corresponding leaf node of $\mathcal{T}$. However, the data structure *does not* store any edge or leaf operator matrix explicitly. We make a key assumption that each edge and leaf operator is computable using $O(1)$-number of $\mathbf{L}^{(H)}$ matrices from DynamicSC. This will be true for the slack and flow operators we define. As a result, to store an edge or leaf operator at a node, we simply store *pointers to the matrices*

Algorithm 16 Implicit representation maintenance

```
1: data structure MaintainRep extends MaintainZ
2: private: member
3:     $\mathcal{T}$: separator tree
4:     $\boldsymbol{y} \in \mathbb{R}^m$: offset vector
5:     $\mathbf{M}$: instructions to compute the tree operator $\mathbf{M} \in \mathbb{R}^{m \times n}$
6: ▷ $\boldsymbol{z} = c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})}$ maintained by MaintainZ, accessable in this data structure
7: ▷ DynamicSC: an accessable instance of DynamicSC maintained by MaintainZ
8:
9: procedure Initialize($G, \mathcal{T}, \mathbf{M}, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \boldsymbol{x}^{(\mathrm{init})} \in \mathbb{R}^m, \delta > 0$)
10:     $\mathbf{M} \leftarrow \mathbf{M}$                          ▷ initialize the instructions to compute $\mathbf{M}$
11:     Super.Initialize($G, \mathcal{T}, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \delta > 0$)          ▷ initialize $\boldsymbol{z}$
12:     $\boldsymbol{y} \leftarrow \boldsymbol{x}^{(\mathrm{init})}$
13: end procedure
14:
15: procedure Reweight($\boldsymbol{w}^{(\mathrm{new})}$)
16:     Let $\mathbf{M}^{(\mathrm{old})}$ represent the current tree operator $\mathbf{M}$
17:     Super.Reweight($\boldsymbol{w}^{(\mathrm{new})}$)              ▷ update representation of $\boldsymbol{z}$ and DynamicSC
18:                                   ▷ $\mathbf{M}$ is updated as a result of reweight in DynamicSC
19:     $\Delta\mathbf{M} \leftarrow \mathbf{M} - \mathbf{M}^{(\mathrm{old})}$                   ▷ $\Delta\mathbf{M}$ is represented implicitly
20:     $\boldsymbol{y} \leftarrow \boldsymbol{y} -$ ComputeMz($\Delta\mathbf{M}, c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})}$)            ▷ Algorithm 15
21: end procedure
22:
23: procedure Move($\alpha, \boldsymbol{v}^{(\mathrm{new})}$)
24:     Super.Move($\alpha, \boldsymbol{v}^{(\mathrm{new})}$)
25: end procedure
26:
27: procedure Exact()
28:     return $\boldsymbol{y} +$ ComputeMz($\mathbf{M}, c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})}$)              ▷ Algorithm 15
29: end procedure
```

from $\textit{D}\textsc{ynamic}\textit{SC}$ required in the definition, and an $O(1)$-sized instruction for how to compute the operator. The computation time is proportional to the size of the matrices in the definitions, but crucially the instructions have only $O(1)$-size.

Now, we prove the correctness and runtime of each procedure separately. Observe that the invariants claimed in the theorem are maintained correctly if each procedure is implemented correctly.

Initialize: Line 12 sets $\boldsymbol{y} \leftarrow \boldsymbol{x}^{(\text{init})}$, and `Super`.Initialize sets $\boldsymbol{z} \leftarrow \mathbf{0}$. So we have $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ at the end of initialization. Furthermore, the initialization of $\boldsymbol{z}$ correctly sets $\boldsymbol{z}^{(\text{prev})}$ in terms of $\boldsymbol{v}$.

By Theorem 3.5.6, `Super`.Initialize takes $\widetilde{O}(\delta^{-2}m)$ time. Storing the implicit representation of $\mathbf{M}$ takes $O(m)$ time.

Reweight: By Theorem 3.5.6, `Super`.Reweight updates its current weight and `DynamicSC`, and updates $\boldsymbol{z}^{(\text{prev})}$ correspondingly to maintain the invariant, while not changing the value of $\boldsymbol{z}$. Because $\mathbf{M}$ is stored by instructions, no explicit update to $\mathbf{M}$ is required. Line 20 updates $\boldsymbol{y}$ to zero out the changes to $\mathbf{M}\boldsymbol{z}$.

The instructions for computing $\Delta\mathbf{M}$ require the Laplacians from `DynamicSC` before and after the update in Line 17. For this, we monitor the updates of `dynamicSC` and stores the old and new values. The runtime of this is bounded by the runtime of updating `dynamicSC`, which is in turn included in the runtime for `Super`.Reweight.

Let K upper bound the number of coordinates changed in $\boldsymbol{w}$ and the number of edge and leaf operators changed in $\mathbf{M}$. Then `Super`.Reweight takes $\widetilde{O}(\delta^{-2}\sqrt{mK})$ time, and Exact$(\Delta\mathbf{M}, \boldsymbol{z})$ takes $O(T(K))$ time. Thurs, the total runtime is $\widetilde{O}(\delta^{-2}\sqrt{mK} + T(m))$.

Move: The runtime and correctness follow from Theorem 3.5.6.

Exact: ComputeM$\boldsymbol{z}$ computes $\mathbf{M}\boldsymbol{z}$ correctly in $O(T(m))$ time by Corollary 3.5.12. Adding the result to $\boldsymbol{y}$ takes $O(m)$ time and gives the correct value of $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$. Thus, Exact returns $\boldsymbol{x}$ in $O(T(m))$ time. □

3.6 Maintaining vector approximation

Recall at every step of the IPM, we want to maintain approximate vectors $\overline{\boldsymbol{s}}, \overline{\boldsymbol{f}}$ so that

$$\left\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}}-\boldsymbol{f})\right\|_{\infty} \leq \delta \quad \text{and} \quad \left\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}}-\boldsymbol{s})\right\|_{\infty} \leq \delta'$$

for some additive error tolerances δ and δ'.

In the previous section, we showed how to maintain some vector $\boldsymbol{x}$ implicitly as $\boldsymbol{x} \overset{\text{def}}{=} \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ throughout the IPM, where $\boldsymbol{x}$ should represent $\boldsymbol{s}$ or part of $\boldsymbol{f}$. In this section, we give a data structure to efficiently maintain an approximate vector $\overline{\boldsymbol{x}}$ to the $\boldsymbol{x}$ from MaintainRep, so that at every IPM step,

$$\left\|\mathbf{D}^{1/2}\left(\overline{\boldsymbol{x}}-\boldsymbol{x}\right)\right\|_{\infty} \leq \delta,$$

where $\mathbf{D}$ is a dynamic diagonal scaling matrix. (It will be $\mathbf{W}^{-1}$ for the flow or $\mathbf{W}$ for the slack.)

In Section 3.6.1, we reduce the problem of maintaining $\overline{\boldsymbol{x}}$ to detecting coordinates in $\boldsymbol{x}$ with large changes. In Section 3.6.2, we detect coordinates of $\boldsymbol{x}$ with large changes using a sampling technique on a binary tree, where Johnson-Lindenstrauss sketches of subvectors of $\boldsymbol{x}$ are maintained at each node the tree. In Section 3.6.3, we show how to compute and maintain the necessary collection of JL-sketches on the separator tree $\mathcal{T}$; in particular, we do this efficiently with only an implicit representation of $\boldsymbol{x}$. Finally, we put the three parts together to prove Theorem 3.2.6.

We use the superscript $^{(k)}$ to denote the variable at the end of the k-th step of the IPM; that is, $\mathbf{D}^{(k)}$ and $\boldsymbol{x}^{(k)}$ are $\mathbf{D}$ and $\boldsymbol{x}$ at the end of the k-th step. Step 0 is the state of the data structure immediately after initialization.

3.6.1 Reduction to change detection

In this subsection, we show that in order to maintain an approximation $\overline{\boldsymbol{x}}$ to some vector $\boldsymbol{x}$, it suffices to detect coordinates of $\boldsymbol{x}$ that change a lot.

Here, we make use of dyadic intervals, and at step k of the IPM, for each ℓ such that $k = 0 \bmod 2^{\ell}$, we find the set $I_{\ell}^{(k)}$ that contains all coordinates i of $\boldsymbol{x}$ such that $\boldsymbol{x}_i^{(k)}$ changed significantly compared to $\boldsymbol{x}_i^{(k-2^\ell)}$, that is, compared to 2^ℓ steps ago. Formally:

Definition 3.6.1. At step k of the IPM, for each ℓ such that $k = 0 \bmod 2^\ell$, we define

$$I_\ell^{(k)} \overset{\text{def}}{=} \{i \in [n] : \sqrt{\mathbf{D}_{ii}^{(k)}} \cdot |\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)}| \geq \frac{\delta}{2\lceil \log m \rceil}$$
$$\text{and } \overline{\boldsymbol{x}}_i \text{ has not been updated since the } (k-2^\ell)\text{-th step}\}.$$

We say that $\overline{\boldsymbol{x}}_i$ has not been updated since the $(k-2^\ell)$-th step if $\overline{\boldsymbol{x}}_i^{(j)} = \overline{\boldsymbol{x}}_i$ and $\mathbf{D}_{ii}^{(j)} = \mathbf{D}_{ii}^{(k-2^\ell)}$ for $j \geq k - 2^\ell$, i.e. $\overline{\boldsymbol{x}}_i$ was not updated by Line 20 or Line 29 in the $(k-2^\ell+1), \ldots, (i-1)$-th steps.

We show how to find the sets $I_\ell^{(k)}$ with high probability in the next subsection. Assuming the correct implementation, we have the following data structure for maintaining the desired approximation $\overline{\boldsymbol{x}}$:

Lemma 3.6.2 (Approximate Vector Maintenance). *Suppose* FindLargeCoordinates(ℓ) *is a procedure in* AbstractMaintainApprox *that correctly computes the set $I_\ell^{(k)}$ at the k-th step. Then the deterministic data structure* AbstractMaintainApprox *in Algorithm 17 maintains an approximation $\overline{\boldsymbol{x}}$ of $\boldsymbol{x}$ with the following procedures:*

- Initialize($\mathcal{T}, \boldsymbol{x} \in \mathbb{R}^m$, $\mathbf{D} \in \mathbb{R}_{>0}^{m \times m}$, $\rho > 0$, $\delta > 0$)*: Initialize the data structure at step 0 with tree $\mathcal{T}$, initial vector $\boldsymbol{x}$, initial diagonal scaling matrix $\mathbf{D}$, target additive approximation error δ, and success probability $1 - \rho$.*

- Approximate($\boldsymbol{x}^{(\text{new})} \in \mathbb{R}^m$, $\mathbf{D}^{(\text{new})} \in \mathbb{R}_{>0}^{m \times m}$)*: Increment the step counter and update vector $\boldsymbol{x}$ and diagonal scaling matrix $\mathbf{D}$. Output a vector $\overline{\boldsymbol{x}}$ such that $\|\mathbf{D}^{1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\|_\infty \leq \delta$ for the latest $\boldsymbol{x}$ and $\mathbf{D}$.*

Furthermore, if $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\|_{\mathbf{D}^{(k)}} \leq \beta$ for all k, *then at the k-th step, the data structure first updates $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k-1)}$ for the coordinates i with $\mathbf{D}_{ii}^{(k)} \neq \mathbf{D}_{ii}^{(k-1)}$, then updates $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k)}$ for $O(2^{2\ell_k}(\beta/\delta)^2 \log^2 m)$ coordinates, where ℓ_k is the largest integer ℓ with $k \equiv 0 \bmod 2^\ell$.*

Remark 3.6.3. In our problem setting of maintaining approximate flows and slacks, we do not have full access to the exact vector. The algorithms in the next two subsections however will refer to

Algorithm 17 Data structure AbstractMaintainApprox, Part 1

```
data structure AbstractMaintainApprox
private : member
    $\mathcal{T}$: constant-degree rooted tree with height $\eta$ and $m$ leaves        ▷ leaf $i$ corresponds to $\boldsymbol{x}_i$
    $w \overset{\text{def}}{=} \Theta(\eta^2 \log(\frac{m}{\rho}))$: sketch dimension
    $\boldsymbol{\Phi} \sim \mathbf{N}(0, \frac{1}{w})^{w \times m}$: JL-sketch matrix
    $\delta > 0$: additive approximation error
    $k$: current IPM step
    $\overline{\boldsymbol{x}} \in \mathbb{R}^m$: current valid approximate vector
    $\{\boldsymbol{x}^{(j)} \in \mathbb{R}^m\}_{j=0}^{k}$: list of previous inputs
    $\{\mathbf{D}^{(j)} \in \mathbb{R}^{m \times m}\}_{j=0}^{k}$: list of previous diagonal scaling matrices

procedure Initialize($\mathcal{T}, \boldsymbol{x} \in \mathbb{R}^m, \mathbf{D} \in \mathbb{R}_{>0}^{m \times m}, \rho > 0, \delta > 0$)
    $\mathcal{T} \leftarrow \mathcal{T}$, $\delta \leftarrow \delta$, $k \leftarrow 0$
    $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}, \boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}, \mathbf{D}^{(0)} \leftarrow \mathbf{D}$
    sample $\boldsymbol{\Phi} \sim \mathbf{N}(0, \frac{1}{w})^{w \times m}$
end procedure

procedure Approximate($\boldsymbol{x}^{(\text{new})} \in \mathbb{R}^m, \mathbf{D}^{(\text{new})} \in \mathbb{R}_{>0}^{m \times m}$)
    $k \leftarrow k + 1$, $\boldsymbol{x}^{(k)} \leftarrow \boldsymbol{x}^{(\text{new})}$, $\mathbf{D}^{(k)} \leftarrow \mathbf{D}^{(\text{new})}$
    $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k-1)}$ for all $i$ such that $\mathbf{D}_{ii}^{(k)} \neq \mathbf{D}_{ii}^{(k-1)}$
    $I \leftarrow \emptyset$
    for all $0 \leq \ell < \lceil \log m \rceil$ such that $k \equiv 0 \bmod 2^\ell$ do
        $I_\ell^{(k)} \leftarrow$ FindLargeCoordinates($\ell$)
        $I \leftarrow I \cup I_\ell^{(k)}$
    end for
    if $k = 0 \bmod 2^{\lceil \log m \rceil}$ then
        $I \leftarrow [m]$        ▷ Update $\overline{\boldsymbol{x}}$ in full every $2^{\lceil \log m \rceil}$ steps
    end if
    $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k)}$ for all $i \in I$
    return $\overline{\boldsymbol{x}}$
end procedure
```

the exact vector $\boldsymbol{x}$ for readability and modularity. We observe that access to $\boldsymbol{x}$ is limited to two types: accessing the JL-sketches of specific subvectors, and accessing exact coordinates and other specific subvectors of sufficiently small size. In later sections, we show how to implement these oracle accesses to $\boldsymbol{x}$.

Proof of Lemma 3.6.2. We first prove the correctness of APPROXIMATE in ABSTRACTMAINTAINAPPROX. Fix some coordinate $i \in [m]$ and fix some IPM step k. Suppose the latest update to $\overline{\boldsymbol{x}}_i$ is $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k')}$. This may happen in Line 29 at step k' or in Line 20 at step $k'+1$. In both case, we have that $\mathbf{D}_{ii}^{(d)}$ is the same for all $k \geq d > k'$ and that i is not in the set $I_\ell^{(d)}$ returned by FINDLARGECOORDINATES for all $k \geq d > k'$. (In the former case, we further have $\mathbf{D}_{ii}^{(k'+1)} = \mathbf{D}_{ii}^{(k')}$ but this is not required in the proof.) Since we set $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}$ every $2^{\lceil \log m \rceil}$ steps by Line 27, we have $k - 2^{\lceil \log m \rceil} \leq k' < k$. Using dyadic intervals, we can write $k' = k_0 < k_1 < k_2 < \cdots < k_s = k$ such that $k_{j+1} - k_j$ is a power of 2, $k_{j+1} - k_j$ divides k_{j+1}, and $|s| \leq 2\lceil \log m \rceil$. Hence, we have that

$$\boldsymbol{x}_i^{(k)} - \overline{\boldsymbol{x}}_i^{(k)} = \boldsymbol{x}_i^{(k_s)} - \overline{\boldsymbol{x}}_i^{(k_0)} = \boldsymbol{x}_i^{(k_s)} - \boldsymbol{x}_i^{(k_0)} = \sum_{j=0}^{s-1} (\boldsymbol{x}_i^{(k_{j+1})} - \boldsymbol{x}_i^{(k_j)}).$$

We know that $\mathbf{D}_{ii}^{(d)}$ is the same for all $k \geq d > k'$. By the guarantees of FINDLARGECOORDINATES, we have

$$\sqrt{\mathbf{D}_{ii}^{(k)}} \cdot |\boldsymbol{x}_i^{(k_{j+1})} - \boldsymbol{x}_i^{(k_j)}| = \sqrt{\mathbf{D}_{ii}^{(k_{j+1})}} \cdot |\boldsymbol{x}_i^{(k_{j+1})} - \boldsymbol{x}_i^{(k_j)}| \leq \frac{\delta}{2\lceil \log m \rceil}$$

for all $0 \leq j < s$. (Summing over all $j = 0, 1, \ldots, s-1$ gives

$$\sqrt{\mathbf{D}_{ii}^{(k)}} \cdot |\boldsymbol{x}_i^{(k)} - \overline{\boldsymbol{x}}_i^{(k)}| \leq \delta.$$

Hence, we have $\|\mathbf{D}^{1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\|_\infty \leq \delta$.

Next, we bound the number of coordinates changed from $\overline{\boldsymbol{x}}^{(k-1)}$ to $\overline{\boldsymbol{x}}^{(k)}$. Fix some ℓ with $k = 0 \bmod 2^\ell$. For any $i \in I_\ell^{(k)}$, we know $\mathbf{D}_{ii}^{(j)} = \mathbf{D}_{ii}^{(k)}$ for all $j > k - 2^\ell$ because $\overline{\boldsymbol{x}}_i$ did not change in the meanwhile. By definition of $I_\ell^{(k)}$, we have

$$\sqrt{\mathbf{D}_{ii}^{(k)}} \cdot \sum_{j=k-2^\ell}^{k-1} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}| \geq \sqrt{\mathbf{D}_{ii}^{(k)}} \cdot |\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)}| \geq \frac{\delta}{2\lceil \log m \rceil}.$$

Using $\mathbf{D}_{ii}^{(j)} = \mathbf{D}_{ii}^{(k)}$ for all $j > k - 2^\ell$ again, the above inequality yields

$$\begin{aligned} \frac{\delta}{2\lceil \log m \rceil} &\leq \sum_{j=k-2^\ell}^{k-1} \sqrt{\mathbf{D}_{ii}^{(j+1)}} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}| \\ &\leq \sqrt{2^\ell \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|^2}. \qquad \text{(by Cauchy-Schwarz)} \end{aligned}$$

Squaring and summing over all $i \in I_\ell^{(k)}$ gives

$$\begin{aligned} \Omega\left(\frac{2^{-\ell}\delta^2}{\log^2 m}\right) |I_\ell^{(k)}| &\leq \sum_{i \in I_\ell^{(k)}} \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|^2 \\ &\leq \sum_{i=1}^{m} \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|^2 \\ &\leq 2^\ell \beta^2, \end{aligned}$$

where we use $\|\boldsymbol{x}^{(j+1)} - \boldsymbol{x}^{(j)}\|_{\mathbf{D}^{(j+1)}} \leq \beta$ at the end. Hence, we have

$$|I_\ell^{(k)}| = O(2^{2\ell}(\beta/\delta)^2 \log^2 m).$$

Recall this expression is for a fixed ℓ. At the k-th step, summing over all ℓ with $k = 0 \bmod 2^\ell$, we have that the total number of coordinates changed, excluding those induced by a change in $\mathbf{D}$, is

$$\sum_{\ell=0}^{\ell_k} |I_\ell^{(k)}| = O(2^{2\ell_k}(\beta/\delta)^2 \log^2 m).$$

$\square$

3.6.2 From change detection to sketch maintenance

Now we discuss the implementation of FindLargeCoordinates(ℓ) to find the set $I_\ell^{(k)}$ in Line 23 of Algorithm 17. We accomplish this by repeatedly sampling a coordinate i with probability proportional to $\mathbf{D}_{ii}^{(k)} \cdot |\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)}|^2$, among all coordinates i where $\overline{\boldsymbol{x}}_i$ has not been updated since 2^ℓ steps ago. With high probability, we can find all $i \in I_\ell^{(k)}$ in this way efficiently. To implement the sampling procedure, we make use of a data structure based on segment trees [17] along with sketching based on the Johnson-Lindenstrauss lemma.

Formally, we define the vector $\boldsymbol{q} \in \mathbb{R}^m$ where $\boldsymbol{q}_i \overset{\text{def}}{=} {\mathbf{D}_{ii}^{(k)}}^{1/2}(\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)})$ if $\overline{\boldsymbol{x}}_i$ has not been updated after the $k - 2^\ell$-th step, and $\boldsymbol{q}_i = 0$ otherwise. Our goal is precisely to find all large coordinates of $\boldsymbol{q}$.

Let $\mathcal{T}$ be a constant-degree rooted tree with m leaves, where leaf i represents coordinate $\boldsymbol{q}_i$. For each node $u \in \mathcal{T}$, we define $E(u) \subseteq [m]$ to be set of indices of leaves in the subtree rooted at u. We make a random descent down $\mathcal{T}$, in order to sample a coordinate i with probability proportional to $\boldsymbol{q}_i^2$. At a node u, for each child u' of u, the total probability of the leaves under u' is given precisely by $\left\|\boldsymbol{q}|_{E(u')}\right\|_2^2$. We can estimate this by the Johnson-Lindenstrauss lemma using a sketching matrix $\boldsymbol{\Phi}$. Then we randomly move from u down to child u' with probability proportional to the estimated value. To tolerate the estimation error, when reaching some leaf node representing coordinate i, we accept with probability proportional to the ratio between the exact probability of i and the estimated probability of i. If i is rejected, we repeat the process from the root again independently.

Lemma 3.6.4. *Assume that $\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_{\mathbf{D}^{(k+1)}} \leq \beta$ for all IPM steps k. Let $\rho < 1$ be any given failure probability, and let $N \overset{\text{def}}{=} \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho))$ be the number of samples Algorithm 17 takes. Then with probability $\geq 1 - \rho$, during the k-th call of* Approximate, *Algorithm 17 finds the set $I_\ell^{(k)}$ correctly. Furthermore, the while-loop in Line 40 happens only $O(1)$ times in expectation per sample.*

Proof. The proof is similar to Lemma 6.17 in [163]. We include it for completeness. For a set S of indices, let $\mathbf{I}_S$ be the $m \times m$ diagonal matrix that is one on S and zero otherwise.

We first prove that Line 47 breaks with probability at least $\frac{1}{4}$. By the choice of w, Johnson–

Algorithm 17 Data structure AbstractMaintainApprox, Part 2

```
32: procedure FindLargeCoordinates($\ell$)
33:                                              ▷ $\overline{\mathbf{D}}$ and $\boldsymbol{q}$ are symbolic definitions
34:     ▷ $\overline{\mathbf{D}}$: diagonal matrix such that
```

$$\overline{\mathbf{D}}_{ii} = \begin{cases} \mathbf{D}_{ii}^{(k)} & \text{if } \overline{\boldsymbol{x}}_i \text{ has not been updated after the } (k-2^\ell)\text{-th step} \\ 0 & \text{otherwise.} \end{cases}$$

```
35:     ▷ $\boldsymbol{q} \overset{\text{def}}{=} \overline{\mathbf{D}}^{1/2}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^\ell)})$       ▷ vector to sample coordinates from
36:
37:     $I \leftarrow \emptyset$                                  ▷ set of candidate coordinates
38:     for $N \overset{\text{def}}{=} \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho))$ iterations do
39:         ▷ Sample coordinate $i$ of $\boldsymbol{q}$ w.p. proportional to $\boldsymbol{q}_i^2$ by random descent down $\mathcal{T}$ to a leaf
40:         while true do
41:             $u \leftarrow \text{root}(\mathcal{T})$, $p_u \leftarrow 1$
42:             while $u$ is not a leaf node do
43:                 Sample a child $u'$ of $u$ with probability
```

$$\mathbf{P}(u \to u') \overset{\text{def}}{=} \frac{\|\mathbf{\Phi}_{E(u')}\boldsymbol{q}\|_2^2}{\sum_{\text{child } u'' \text{ of } u} \|\mathbf{\Phi}_{E(u'')}\boldsymbol{q}\|_2^2}$$

```
                                             ▷ let $\mathbf{\Phi}_{E(u)} \overset{\text{def}}{=} \mathbf{\Phi}\mathbf{I}_{E(u)}$ for each node $u$
44:                 $p_u \leftarrow p_u \cdot \mathbf{P}(u \to u')$
45:                 $u \leftarrow u'$
46:             end while
47:             break with probability $p_{\text{accept}} \overset{\text{def}}{=} \left\|\boldsymbol{q}|_{E(u)}\right\|^2 / (2 \cdot p_u \cdot \|\mathbf{\Phi}\boldsymbol{q}\|_2^2)$
48:         end while
49:         $I \leftarrow I \cup E(u)$
50:     end for
51:     return $\{i \in I \ : \ \boldsymbol{q}_i \geq \frac{\delta}{2\lceil \log m \rceil}\}$.
52: end procedure
```

Lindenstrauss lemma shows that $\|\boldsymbol{\Phi}_{E(u)}\boldsymbol{q}\|_2^2 = (1 \pm \frac{1}{9\eta})\|\mathbf{I}_{E(u)}\boldsymbol{q}\|_2^2$ for all $u \in \mathcal{T}$ with probability at least $1-\rho$. Therefore, the probability we move from a node u to its child node u' is given by

$$\mathbf{P}(u \to u') = \left(1 \pm \frac{1}{3\eta}\right)\frac{\|\mathbf{I}_{E(u')}\boldsymbol{q}\|_2^2}{\sum_{u'' \text{ is a child of } u}\|\mathbf{I}_{E(u'')}\boldsymbol{q}\|_2^2} = \left(1 \pm \frac{1}{3\eta}\right)\frac{\|\mathbf{I}_{E(u')}\boldsymbol{q}\|_2^2}{\|\mathbf{I}_{E(u)}\boldsymbol{q}\|_2^2}.$$

Hence, the probability the walk ends at a leaf $u \in \mathcal{T}$ is given by

$$p_u = \left(1 \pm \frac{1}{3\eta}\right)^{\eta}\frac{\|\mathbf{I}_u\boldsymbol{q}\|_2^2}{\|\boldsymbol{q}\|_2^2} = (1 \pm \frac{1}{3\eta})^{\eta}\frac{\left\|\boldsymbol{q}|_{E(u)}\right\|^2}{\|\boldsymbol{q}\|_2^2}.$$

Now, p_{accept} on Line 47 is at least

$$p_{\text{accept}} = \frac{\left\|\boldsymbol{q}|_{E(u)}\right\|^2}{2 \cdot p_u \cdot \|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2} \geq \frac{\left\|\boldsymbol{q}|_{E(u)}\right\|^2}{2 \cdot (1+\frac{1}{3\eta})^{\eta}\frac{\left\|\boldsymbol{q}|_{E(u)}\right\|^2}{\|\boldsymbol{q}\|_2^2} \cdot \|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2} \geq \frac{\|\boldsymbol{q}\|_2^2}{2 \cdot (1+\frac{1}{3\eta})^{\eta}\|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2} \geq \frac{1}{4}.$$

On the other hand, we have that $p_{\text{accept}} \leq \frac{\|\boldsymbol{q}\|_2^2}{2(1-\frac{1}{3\eta})^{\eta}\|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2} < 1$ and hence this is a valid probability.

Next, we note that u is accepted on Line 47 with probability

$$p_{\text{accept}}p_u = \frac{\left\|\boldsymbol{q}|_{E(u)}\right\|^2}{2 \cdot \|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2}.$$

Since $\|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2$ remains the same in all iterations, this probability is proportional to $\left\|\boldsymbol{q}|_{E(u)}\right\|^2$. Since the algorithm repeats when u is rejected, on Line 49, u is chosen with probability exactly $\left\|\boldsymbol{q}|_{E(u)}\right\|^2/\|\boldsymbol{q}\|^2$.

Now, we want to show the output set is exactly $\{i \in [n] : |\boldsymbol{q}_i| \geq \frac{\delta}{2\lceil\log m\rceil}\}$. Let S denote the set of indices where $\overline{\boldsymbol{x}}$ did not update between the $(k-2^\ell)$-th step and the current k-th step. Then

$$\begin{aligned}
\|\boldsymbol{q}\|_2 &= \|\mathbf{I}_S(\mathbf{D}^{(k)})^{1/2}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^\ell)})\|_2 \\
&\le \sum_{i=k-2^\ell}^{k-1} \|\mathbf{I}_S(\mathbf{D}^{(k)})^{1/2}(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})\|_2 \\
&= \sum_{i=k-2^\ell}^{k-1} \|\mathbf{I}_S(\mathbf{D}^{(i+1)})^{1/2}(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})\|_2 \\
&\le \sum_{i=k-2^\ell}^{k-1} \|(\mathbf{D}^{(i+1)})^{1/2}(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})\|_2 \\
&\le 2^\ell \beta,
\end{aligned}$$

where we used $\mathbf{I}_S\mathbf{D}^{(i+1)} = \mathbf{I}_S\mathbf{D}^{(k)}$, because $\overline{\boldsymbol{x}}_i$ changes whenever $\mathbf{D}_{ii}$ changes at a step. Hence, each leaf u is sampled with probability at least $\left\|\boldsymbol{q}|_{E(u)}\right\|^2/(2^\ell\beta)^2$. If $|\boldsymbol{q}_i| \ge \frac{\delta}{2\lceil\log m\rceil}$, and $i \in E(u)$ for a leaf node u, then the coordinate i is not in I with probability at most

$$\left(1 - \frac{\left\|\boldsymbol{q}|_{E(u)}\right\|^2}{(2^\ell\beta)^2}\right)^N \le \left(1 - \frac{1}{2^{2\ell+2}(\beta/\delta)^2\lceil\log m\rceil^2}\right)^N \le \frac{\rho}{m},$$

by our choice of N. Hence, all i with $|\boldsymbol{q}_i| \ge \frac{\delta}{2\lceil\log m\rceil}$ lies in I with probability at least $1-\rho$. This proves that the output set is exactly $I_\ell^{(k)}$ with probability at least $1-\rho$. □

Remark 3.6.5. In Algorithm 17, we only need to compute $\|\mathbf{\Phi}_{E(u)}\boldsymbol{q}\|_2^2$ for $O(N)$ many nodes $u \in \mathcal{T}$. Furthermore, the randomness of the sketch is not leaked and we can use the same random sketch $\mathbf{\Phi}$ throughout the algorithm. This allows us to efficiently maintain $\mathbf{\Phi}_{E(u)}\boldsymbol{q}$ for each $u \in \mathcal{T}$ throughout the IPM.

3.6.3 Sketch maintenance

In FindLargeCoordinates in the previous subsection, we assumed the existence of a constant degree tree $\mathcal{T}$, and for the dynamic vector $\boldsymbol{q}$, the ability to access $\mathbf{\Phi}_{E(u)}\boldsymbol{q}$ at each node $u \in \mathcal{T}$ and $\boldsymbol{q}|_{E(u)}$ at each leaf node $u \in \mathcal{T}(0)$.

In this section, we consider when the required tree is the separator tree $\mathcal{T}$ of the overall input graph, and the vector $\boldsymbol{q}$ is of the form $\boldsymbol{q} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$, where $\mathbf{M}$ is a tree operator supported on $\mathcal{T}$, and each of $\boldsymbol{y}, \mathbf{M}, \boldsymbol{z}$ undergo changes at every IPM step. We present a data structure that implements two features efficiently on $\mathcal{T}$:

- access $(\boldsymbol{y} + \mathbf{M}\boldsymbol{z})|_{E(H)}$ at every leaf node H, where $E(H) \stackrel{\text{def}}{=} \text{Range}(\mathbf{J}_H)$.
- access $\mathbf{\Phi}_{E(H)}(\boldsymbol{y} + \mathbf{M}\boldsymbol{z})$ at every node H, where $\mathbf{\Phi}_{E(H)}$ is $\mathbf{\Phi}$ restricted to columns given by $E(H) \stackrel{\text{def}}{=} \bigcup_{\text{leaf } D \in \mathcal{T}_H} E(D)$.

Remark 3.6.6. As seen in the pseudocode, sketches for $\boldsymbol{y}$ and $\mathbf{M}\boldsymbol{z}$ can be maintained separately. We collected them together to represent $\boldsymbol{x}$ as a whole for simplicity.

First, we present some lemmas about the structure of the expression $\mathbf{M}\boldsymbol{z}$ which will help us to implement the requirements above. For any node $H \in \mathcal{T}$, let $\mathcal{T}_H$ be the subtree of $\mathcal{T}$ rooted at H.

Lemma 3.6.7. *At any leaf node $H \in \mathcal{T}(0)$, we have*

$$(\mathbf{M}\boldsymbol{z})|_{E(H)} = \sum_{A : H \in \mathcal{T}_A} \mathbf{J}_H \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z} = \mathbf{J}_H \mathbf{I}_{F_H} \boldsymbol{z} + \sum_{\textit{ancestor } A \textit{ of } H} \mathbf{J}_H \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z}.$$

Proof. Recall from the definition of the tree operator that $\text{Range}(\mathbf{J}_H)$ are disjoint. So to get $(\mathbf{M}\boldsymbol{z})|_{E(H)}$, it suffices to only consider the terms corresponding to the leaf H in the expression (3.23) for $\mathbf{M}$; this gives the first equality. The second equality simply splits the sum into two parts. (We do not consider a node to be its own ancestor.) □

Lemma 3.6.8. *At any node $H \in \mathcal{T}$, we have*

$$\mathbf{\Phi}_{E(H)} \mathbf{M}\boldsymbol{z} = \mathbf{\Phi} \overline{\mathbf{M}^{(H)}} \boldsymbol{z} + \mathbf{\Phi} \mathbf{M}^{(H)} \sum_{\textit{ancestor } A \textit{ of } H} \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z}.$$

Intuitively, the lemma shows that the sketch of $\mathbf{M}\boldsymbol{z}$ restricted to $E(H)$ can be split into two parts. The first part involves some sum over all nodes in $\mathcal{T}_H$, ie. descendants of H and H itself, and the second part involves a sum over all ancestors of H.

Proof. First, note that since $\mathbf{\Phi}$ is restricted to $E(H)$, it suffices to consider the terms in the sum for $\mathbf{M}$ that map into to $E(H)$. In particular, this is the set of leaf nodes $\mathcal{T}_H$ in the subtree rooted at H.

$$\mathbf{\Phi}_{E(H)}\mathbf{M}\boldsymbol{z} = \mathbf{\Phi} \sum_{\text{leaf } D \in \mathcal{T}_H} \sum_{A : D \in \mathcal{T}_A} \mathbf{J}_D \mathbf{M}_{D \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z}.$$

The right hand side involves a sum over the set $\{(D, A) \ : \ D \in \mathcal{T}_H \text{ is a leaf node}, D \in \mathcal{T}_A\}$. Observe that (D, A) is in this set if and only if $A \in \mathcal{T}_H$ or A is an ancestor of H. Hence, the summation can be written as

$$\sum_{\text{leaf } D \in \mathcal{T}_H} \sum_{A \in \mathcal{T}_H} \mathbf{J}_D \mathbf{M}_{D \leftarrow H} \mathbf{I}_{F_H} \boldsymbol{z} + \sum_{\text{leaf } D \in \mathcal{T}_H} \sum_{\text{ancestor } A \text{ of } H} \mathbf{J}_D \mathbf{M}_{D \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z}.$$

The first term is precisely $\overline{\mathbf{M}^{(H)}}\boldsymbol{z}$. For the second term, we can use the fact that A is an ancestor of H to expand $\mathbf{M}_{D \leftarrow A} = \mathbf{M}_{D \leftarrow H} \mathbf{M}_{H \leftarrow A}$. Then, the second term is

$$\begin{aligned} &\sum_{\text{leaf } D \in \mathcal{T}_H} \sum_{\text{ancestor } A \text{ of } H} \mathbf{J}_D \mathbf{M}_{D \leftarrow H} \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z} \\ &= \sum_{\text{leaf } D \in \mathcal{T}_H} \mathbf{J}_D \mathbf{M}_{D \leftarrow H} \left(\sum_{\text{ancestor } A \text{ of } H} \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z} \right) \\ &= \mathbf{M}^{(H)} \left(\sum_{\text{ancestor } A \text{ of } H} \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z} \right), \end{aligned}$$

by definition of $\mathbf{M}^{(H)}$. □

Lemma 3.6.9. *Let $\mathcal{T}$ be a rooted tree with height η supporting tree operator $\mathbf{M}$ with complexity T. Let $w = \Theta(\eta^2 \log(\frac{m}{\rho}))$ be as defined in Algorithm 17, and let $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$ be a JL-sketch matrix. Then* MaintainSketch *(Algorithm 18) is a data structure that maintains $\mathbf{\Phi}(\boldsymbol{y} + \mathbf{M}\boldsymbol{z})$, as $\boldsymbol{y}$, $\mathbf{M}$ and $\boldsymbol{z}$ undergo changes in the IPM. The data structure supports the following procedures:*

- Initialize*(rooted tree $\mathcal{T}$, $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$, tree operator $\mathbf{M}^{(\text{init})} \in \mathbb{R}^{m \times n}$, $\boldsymbol{z}^{(\text{init})} \in \mathbb{R}^n$, $\boldsymbol{y}^{(\text{init})} \in \mathbb{R}^m$): Initialize the data structure with tree operator $\mathbf{M} \leftarrow \mathbf{M}^{(\text{init})}$, and vectors $\boldsymbol{z} \leftarrow \boldsymbol{z}^{(\text{init})}$, $\boldsymbol{y} \leftarrow \boldsymbol{y}^{(\text{init})}$, and compute the initial sketches in $O(w \cdot m)$ time.*
- Update*$(\mathbf{M}^{(\text{new})}, \boldsymbol{z}^{(\text{new})}, \boldsymbol{y}^{(\text{new})})$: Update $\mathbf{M} \leftarrow \mathbf{M}^{(\text{new})}$, $\boldsymbol{z} \leftarrow \boldsymbol{z}^{(\text{new})}$, $\boldsymbol{y} \leftarrow \boldsymbol{y}^{(\text{new})}$ and all the necessary sketches in $O(w \cdot T(\eta \cdot |\mathcal{S}|))$ time, where $\mathcal{S}$ is the set of all nodes H where one of*

Algorithm 18 Data structure for maintaining $\mathbf{\Phi}(\boldsymbol{y} + \mathbf{M}\boldsymbol{z})$, Part 1

```
1: data structure MaintainSketch
2: private : member
3:     $\mathcal{T}$ : rooted constant degree tree, where at every node $H$, there is
4:         $\boxed{\mathbf{\Phi}\mathbf{M}^{(H)}}$ : sketch of partial tree operator
5:         $\boxed{\mathbf{\Phi}\overline{\mathbf{M}^{(H)}}\boldsymbol{z}}$ : sketched vector        ▷ This gives $\mathbf{\Phi}\mathbf{M}\boldsymbol{z}$ at the root
6:         $\boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(H)}}$ : sketched subvector of $\boldsymbol{y}$
7:     $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$ : JL-sketch matrix
8:     $\mathbf{M}$ : tree operator on $\mathcal{T}$
9:     $\boldsymbol{z} \in \mathbb{R}^n$ : vector $\boldsymbol{z}$
10:    $\boldsymbol{y} \in \mathbb{R}^n$ : vector $\boldsymbol{y}$        ▷ $\mathbf{M}, \boldsymbol{z}, \boldsymbol{y}$ are pointers to read-only memory
11:
12: procedure Initialize(rooted tree $\mathcal{T}$, $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$, tree operator $\mathbf{M}$, $\boldsymbol{z}$, $\boldsymbol{y}$)
13:     $\mathbf{\Phi} \leftarrow \mathbf{\Phi}$, $\mathcal{T} \leftarrow \mathcal{T}$
14:     $\boxed{\mathbf{\Phi}\mathbf{M}^{(H)}} \leftarrow \mathbf{0}$, $\boxed{\mathbf{\Phi}\overline{\mathbf{M}^{(H)}}\boldsymbol{z}} \leftarrow \mathbf{0}$, $\boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(H)}} \leftarrow \mathbf{0}$ for all $H \in \mathcal{T}$
15:     Update($\mathbf{M}, \boldsymbol{z}, \boldsymbol{y}, V(\mathcal{T})$)
16: end procedure
17:
18: procedure Update($\mathbf{M}^{(\mathrm{new})}, \boldsymbol{z}^{(\mathrm{new})}, \boldsymbol{y}^{(\mathrm{new})}, \mathcal{S} \stackrel{\mathrm{def}}{=}$ set of nodes admitting changes)
19:     $\mathbf{M} \leftarrow \mathbf{M}^{(\mathrm{new})}$, $\boldsymbol{z} \leftarrow \boldsymbol{z}^{(\mathrm{new})}$, $\boldsymbol{y} \leftarrow \boldsymbol{y}^{(\mathrm{new})}$
20:     for $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{S})$ by increasing node level do
21:         if $H$ is a leaf then
22:             $\boxed{\mathbf{\Phi}\mathbf{M}^{(H)}} \leftarrow \mathbf{\Phi}\mathbf{J}_H$
23:             $\boxed{\mathbf{\Phi}\overline{\mathbf{M}^{(H)}}\boldsymbol{z}} \leftarrow \mathbf{\Phi}\mathbf{J}_H\boldsymbol{z}|_{F_H}$
24:             $\boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(H)}} \leftarrow \mathbf{\Phi}\boldsymbol{y}|_{E(H)}$
25:         else
26:             $\boxed{\mathbf{\Phi}\mathbf{M}^{(H)}} \leftarrow \sum_{\text{child } D \text{ of } H} \boxed{\mathbf{\Phi}\mathbf{M}^{(D)}}\mathbf{M}_{(D,H)}$
27:             $\boxed{\mathbf{\Phi}\overline{\mathbf{M}^{(H)}}\boldsymbol{z}} \leftarrow \boxed{\mathbf{\Phi}\mathbf{M}^{(H)}}\boldsymbol{z}|_{F_H} + \sum_{\text{child } D \text{ of } H} \boxed{\mathbf{\Phi}\overline{\mathbf{M}^{(D)}}\boldsymbol{z}}$
28:             $\boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(H)}} \leftarrow \sum_{\text{child } D \text{ of } H} \boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(D)}}$
29:         end if
30:     end for
31: end procedure
32:
33: procedure SumAncestors($H \in \mathcal{T}$)
34:     if Update has not been called since the last call to SumAncestors($H$) then
35:         return the result of the last SumAncestors($H$)
36:     end if
37:     if $H$ is the root then return $\mathbf{0}$
38:     end if
39:     return $\mathbf{M}_{(H,P)}(\boldsymbol{z}|_{F_P} + \text{SumAncestors}(P))$        ▷ $P$ is the parent of $H$
40: end procedure
```

Algorithm 18 Data structure for maintaining $\mathbf{\Phi}(\boldsymbol{y}+\mathbf{M}\boldsymbol{z})$, part 2

```
41: procedure Estimate($H \in \mathcal{T}$)
42:     Let $\boldsymbol{u}$ be the result of SumAncestors($H$)
43:     return $\boxed{\mathbf{\Phi}\mathbf{M}^{(H)}}\boldsymbol{u} + \boxed{\mathbf{\Phi}\overline{\mathbf{M}^{(H)}\boldsymbol{z}}} + \boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(H)}}$
44: end procedure
45:
46: procedure Query(leaf $H \in \mathcal{T}$)
47:     return $\boldsymbol{y}|_{E(H)} + \mathbf{J}_H(\boldsymbol{z}|_{F_H} + \text{SumAncestors}(H))$
48: end procedure
```

$\mathbf{M}_{(H,P)}, \mathbf{J}_H, \boldsymbol{z}|_{F_H}, \boldsymbol{y}|_{E(H)}$ *is updated.*

- *SumAncestors*$(H \in \mathcal{T})$*: Return* $\sum_{\text{ancestor } A \text{ of } H} \mathbf{M}_{H \leftarrow A}\mathbf{I}_{F_A}\boldsymbol{z}$.
- *Estimate*$(H \in \mathcal{T})$*: Return* $\mathbf{\Phi}_{E(H)}(\boldsymbol{y}+\mathbf{M}\boldsymbol{z})$.
- *Query*$(H \in \mathcal{T})$*: Return* $(\boldsymbol{y}+\mathbf{M}\boldsymbol{z})|_{E(H)}$.

If we call Query *on* N *nodes, the total runtime is* $O(w \cdot T(\eta N))$.

If we call Estimate *along a sampling path (by which we mean starting at the root, calling estimate at both children of a node, and then recursively descending to one child until reaching a leaf), and then we call* Query *on the resulting leaf, and we repeat this* N *times with no updates during the process, then the total runtime of these calls is* $O(w \cdot T(\eta N))$.

Proof. First, we note that each edge operator $\mathbf{M}_e$ should be stored implicitly. In particular, it suffices to only support the operation of computing $\boldsymbol{u}^\top \mathbf{M}_e$ and $\mathbf{M}_e\boldsymbol{x}$ for any vectors $\boldsymbol{u}$ and $\boldsymbol{x}$.

We prove the running time and correctness for each procedure.

Initialize: It sets the sketches to $\mathbf{0}$ in $O(w \cdot m)$ time. It then calls Update with the initial $\mathbf{M}$, $\boldsymbol{z}$, $\boldsymbol{y}$, and updates the sketches everywhere on $\mathcal{T}$. By the runtime and correctness of Update, this step is correct and runs in $\widetilde{O}(w \cdot T(m))$ time.

Update($\mathbf{M}^{(\text{new})}, \boldsymbol{z}^{(\text{new})}, \boldsymbol{y}^{(\text{new})}$): Let $\mathcal{S}$ denote the set of nodes admitting changes as defined in the theorem statement. If a node H is not in $\mathcal{S}$ and it has no descendants in $\mathcal{S}$, then by definition, $\mathbf{M}^{(H)}$ and $\overline{\mathbf{M}^{(H)}\boldsymbol{z}}$ are not affected by the updates in $\mathbf{M}$ and $\boldsymbol{z}$. Similarly, in this case, $\boldsymbol{y}|_{E(H)}$ is not affected by the updates to $\boldsymbol{y}$. Hence, it suffices to update the sketches only at all nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{S})$. We update the nodes from the bottom level of the tree upwards, so that when we're at a node H, all the sketches at its descendant nodes are correct. Hence, by definition, the sketch at H is also correct.

To compute the runtime, first note $|\mathcal{P}_{\mathcal{T}}(\mathcal{S})| = O(\eta|\mathcal{S}|)$, since for each node $H \in \mathcal{S}$, the set includes all the $O(\eta)$ nodes on the path from H to the root. For each leaf node $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{S})$, we can compute its sketches in constant time. For each non-leaf node $H \in \mathcal{S}$ with children D_1, D_2, Line 26 multiplies each row of $\boxed{\mathbf{\Phi M}^{(D_1)}}$ with $\mathbf{M}_{(D_1,H)}$, each row of $\boxed{\mathbf{\Phi M}^{(D_2)}}$ with $\mathbf{M}_{(D_2,H)}$, and sums the results. For a fixed row number, the total time over all $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{S})$ is bounded by $O(T(|\mathcal{P}_{\mathcal{T}}(\mathcal{S})|))$. So the total time for Line 26 in the procedure is $O(w \cdot T(\eta|S|))$.

Line 27 multiply each row of $\boxed{\mathbf{\Phi M}^{(H)}}$ with a vector and then performs a constant number of additions of $O(w)$-length vectors. Since $\boxed{\mathbf{\Phi M}^{(H)}}$ is computed for all $H \in T(|\mathcal{P}_{\mathcal{T}}(\mathcal{S})|)$ in $O(w \cdot T(\eta|S|))$ total time, this runtime must also be a bound on the number of total non-zero entries. Since each $\boxed{\mathbf{\Phi M}^{(H)}}$ is used once in Line 27 for a matrix-vector multiplication, the total runtime over all H is also $O(w \cdot T(\eta|S|))$. Lastly, the vector additions across all H takes $O(w \cdot \eta|S|)$ time.

Line 28 adds two vectors of length w. This is not the bottleneck.

SumAncestors(H): At the root, there are no ancestors, hence we return the zero matrix. When H is not the root, suppose P is the parent of H. Then we can recursively write

$$\sum_{\text{ancestor } A \text{ of } H} \mathbf{M}_{H \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z} = \mathbf{M}_{(H,P)} \left(\mathbf{I}_{F_P} \boldsymbol{z} + \sum_{\text{ancestor } A \text{ of } P} \mathbf{M}_{P \leftarrow A} \mathbf{I}_{F_A} \boldsymbol{z} \right).$$

The procedure implements the right hand side, and is therefore correct.

Estimate and Query: Their correctness follow from Lemmas 3.6.7 and 3.6.8, and the correctness of $\boxed{\mathbf{\Phi}\boldsymbol{y}|_{E(H)}}$ maintained by UPDATE.

Overall Estimate and Query time along N sampling paths: We show that if we call ESTIMATE along N sampling paths each from the root to a leaf, and we call QUERY on the leaves, the overall cost for these calls is $O(w \cdot T(\eta N))$:

Suppose the set of nodes visited is given by $\mathcal{H}$, then $|\mathcal{H}| \leq \eta N$. Since there is no update, and ESTIMATE is called for a node only after it is called for its parent, we know that SUMANCESTORS(H) is called exactly once for each $H \in \mathcal{H}$. Each SUMANCESTOR(H) multiplies a unique edge operator $\mathbf{M}_{(H,P)}$ with a vector. Hence, the total runtime of SUMANCESTORS is $T(|\mathcal{H}|)$. Furthermore, the total number of non-zero entries of the return values of these SUMANCESTORS is also $O(T(|\mathcal{H}|))$.

Finally, each QUERY applies a constant-time operator $\mathbf{J}_H$ to the output of a unique SUMANCESTORS call, so the overall runtime is certainly bounded by $O(T(|\mathcal{H}|))$. Adding a constant-sized $\boldsymbol{y}|_{E(H)}$ can be done efficiently. Similarly, each ESTIMATE multiplies $\boxed{\mathbf{\Phi}\mathbf{M}^{(H)}}$ with the output of a unique SUMANCESTORS call. This can be computed as w-many vectors each multiplied with the SUMANCESTORS output. Then two vectors of length w are added. Summing over all $H \in \mathcal{H}$, the overall runtime is $O(w \cdot T(|\mathcal{H}|)) = O(w \cdot T(\eta N))$.

Query time on N leaves: Since this is a subset of the work described above, the runtime must also be bounded by $O(w \cdot T(\eta N))$.

□

3.6.4 Proof of Theorem 3.2.6

We combine the previous three subsections for the overall approximation procedure. It is essentially ABSTRACTMAINTAINAPPROX in Algorithm 17, with the abstractions replaced by a data structure implementation. We did not provide the corresponding pseudocode.

Theorem 3.2.6 (Approximate vector maintenance with tree operator)**.** *Given a constant degree tree $\mathcal{T}$ with height η that supports tree operator $\mathbf{M}$ with complexity T, there is a randomized data structure MAINTAINAPPROX that takes as input the dynamic variables $\mathbf{M}, c, \boldsymbol{z}^{(\mathrm{prev})}, \boldsymbol{z}^{(\mathrm{sum})}, \boldsymbol{y}, \mathbf{D}$ at every IPM step, and maintains the approximation $\overline{\boldsymbol{x}}$ to $\boldsymbol{x} \overset{\mathrm{def}}{=} \boldsymbol{y} + \mathbf{M}\boldsymbol{z} = \boldsymbol{y} + \mathbf{M}(c \cdot \boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})})$ satisfying $\left\|\mathbf{D}^{1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\right\|_\infty \leq \delta$. It supports the following procedures:*

- *INITIALIZE(tree $\mathcal{T}$, tree operator $\mathbf{M}, c \in \mathbb{R}, \boldsymbol{z}^{(\mathrm{prev})} \in \mathbb{R}^n, \boldsymbol{z}^{(\mathrm{sum})} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m, \mathbf{D} \in \mathbb{R}^{n \times n}, \rho > 0, \delta > 0$): Initialize the data structure with initial vector $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}(c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})})$,* *diagonal scaling matrix* $\mathbf{D}$*, target approximation accuracy* δ*, success probability* $1-\rho$*, in* $O(m\eta^2 \log m \log(\frac{m}{\rho}))$ *time. Initialize* $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}$*.*

- Approximate$(\mathbf{M}, c, \boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})}, \boldsymbol{y}, \mathbf{D})$*: Update the internal variables to their new iterations as given. Then output a vector* $\overline{\boldsymbol{x}}$ *such that* $\|\mathbf{D}^{1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\|_\infty \leq \delta$ *for the current vector* $\boldsymbol{x}$ *and the current diagonal scaling* $\mathbf{D}$*.*

Suppose $\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_{\mathbf{D}^{(k+1)}} \leq \beta$ for all k*, where* $\mathbf{D}^{(k)}$ *and* $\boldsymbol{x}^{(k)}$ *are the* $\mathbf{D}$ *and* $\boldsymbol{x}$ *at the* k*-th call to* Approximate*. Then, for the* k*-th call to* Approximate*, we have*

- *the data structure first updates* $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k-1)}$ *for the coordinates* i *with* $\mathbf{D}_{ii}^{(k)} \neq \mathbf{D}_{ii}^{(k-1)}$*, then updates* $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k)}$ *for* $O(N_k \overset{\text{def}}{=} 2^{2\ell_k}(\beta/\delta)^2 \log^2 m)$ *coordinates, where* ℓ_k *is the largest integer* ℓ *with* $k = 0 \bmod 2^\ell$*.*

- *The amortized time cost of* Approximate *is*

$$\Theta(\eta^2 \log(\frac{m}{\rho}) \log m) \cdot T(\eta \cdot (N_{k-2^{\ell_k}} + |\mathcal{S}|)),$$

where $\mathcal{S}$ *is the set of nodes* H *where either* $\mathbf{M}_{(H,P)}$*,* $\mathbf{J}_H$*,* $\boldsymbol{z}^{(\text{prev})}|_{F_H}$*, or* $\boldsymbol{z}^{(\text{sum})}|_{F_H}$ *changed, or where* $\boldsymbol{y}_e$ *or* $\mathbf{D}_{e,e}$ *changed for some edge* e *in* H*, compared to the* $(k-1)$*-th step.*

Proof. The data structure AbstractMaintainApprox in Algorithm 17 performs the correct vector approximation maintenance, however, it is not completely implemented. MaintainApprox simply replaces the abstractions with a concrete implementation using the data structure MaintainSketch from Algorithm 18.

First, for notation purposes, let $\boldsymbol{z} \overset{\text{def}}{=} c\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})}$, and let $\boldsymbol{x} \overset{\text{def}}{=} \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$, so that at step k, Approximate procedure has $\boldsymbol{x}^{(k)}$ (in implicit form) as input, and return $\overline{\boldsymbol{x}}$.

Let $\ell \in \{1, \ldots, O(\log m)\}$. We define a new dynamic vector $\boldsymbol{x}_\ell$ *symbolically*, which is represented at each step k for $k \geq 2^\ell$ by

$$\boldsymbol{x}_\ell^{(k)} \overset{\text{def}}{=} \boldsymbol{y}_\ell^{(k)} + \mathbf{M}_\ell^{(k)} \boldsymbol{z}_\ell^{(k)},$$

where the new tree operator $\mathbf{M}_\ell$ at step k is given by

- $\mathbf{M}_{\ell}^{(k)}{}_{(H,P)} = \operatorname{diag}\left(\mathbf{M}_{(H,P)}^{(k)}, \mathbf{M}_{(H,P)}^{(k-2^{\ell})}\right)$ for each child-parent edge (H,P) in $\mathcal{T}$,
- $\mathbf{J}_{\ell}^{(k)}{}_{H} = \overline{\mathbf{D}}_{E(H),E(H)} \left[\mathbf{J}_{H}^{(k)}\ \mathbf{J}_{H}^{(k-2^{\ell})}\right]$ for each leaf node $H \in \mathcal{T}$,

where $\overline{\mathbf{D}}$ is the diagonal matrix defined in FINDLARGECOORDINATES, with $\overline{\mathbf{D}}_{i,i} = \mathbf{D}_{i,i}^{(k)}$ at step k if $\overline{\boldsymbol{x}}_i$ has not been updated after step $k - 2^{\ell}$, and zero otherwise.

At step k, the vector $\boldsymbol{y}_{\ell}$ is given by $\boldsymbol{y}_{\ell}^{(k)} = \overline{\mathbf{D}}^{1/2}\left(\boldsymbol{y}^{(k)} - \boldsymbol{y}^{(k-2^{\ell})}\right)$, and $\boldsymbol{z}_{\ell}$ by $\boldsymbol{z}_{\ell}^{(k)} \stackrel{\text{def}}{=} \left[\boldsymbol{z}^{(k)}\ \boldsymbol{z}^{(k-2^{\ell})}\right]^{\top}$. Then, at each step k with $k \geq 2^{\ell}$, we have

$$\begin{aligned}
\boldsymbol{x}_{\ell}^{(k)} &\stackrel{\text{def}}{=} \boldsymbol{y}_{\ell}^{(k)} + \mathbf{M}_{\ell}^{(k)} \boldsymbol{z}_{\ell}^{(k)} \qquad (3.27)\\
&= \left(\overline{\mathbf{D}}^{1/2}\boldsymbol{y}^{(k)} + \overline{\mathbf{D}}^{1/2}\mathbf{M}^{(k)}\boldsymbol{z}^{(k)}\right) - \left(\overline{\mathbf{D}}^{1/2}\boldsymbol{y}^{(k-2^{\ell})} + \overline{\mathbf{D}}^{1/2}\mathbf{M}^{(k-2^{\ell})}\boldsymbol{z}^{(k-2^{\ell})}\right)\\
&= \overline{\mathbf{D}}^{1/2}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^{\ell})}).
\end{aligned}$$

Note this is precisely the vector $\boldsymbol{q}$ for a fixed ℓ in FINDLARGECOORDINATES in Algorithm 17. It is straightforward to see that $\mathbf{M}_{\ell}$ indeed satisfies the definition of a tree operator. Furthermore, $\mathbf{M}_{\ell}$ has the same complexity as $\mathbf{M}$. MAINTAINAPPROX will contain $O(\log m)$ copies of the MAINTAINSKETCH data structures in total, where the ℓ-th copy sketches $\boldsymbol{x}_{\ell}$ as it changes throughout the IPM algorithm.

We now describe each procedure in words, and then prove their correctness and runtime.

Initialize$(\mathcal{T}, \mathbf{M}, c, \boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})}, \boldsymbol{y}, \mathbf{D}, \rho, \delta)$**:** This procedure implements the initialization of ABSTRACTMAINTAINAPPROX, where the dynamic vector $\boldsymbol{x}$ to be approximated is represented by $\boldsymbol{x} \stackrel{\text{def}}{=} \boldsymbol{y} + \mathbf{M}(c\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})})$. The initialization steps described in Algorithm 17 takes $O(wm)$ time. Let $\boldsymbol{\Phi}$ denote the JL-sketching matrix.

We initialize two copies of the MAINTAINSKETCH data structure, `ox_cur` and `ox_prev`. At step k, `ox_cur` will maintain sketches of $\boldsymbol{\Phi}\boldsymbol{x}^{(k)}$, and `ox_prev` will maintain sketches of $\boldsymbol{\Phi}\boldsymbol{x}^{(k-1)}$. (The latter is initialized at step 1, but we consider it as part of initialization.)

In addition, for each $0 \leq \ell \leq O(m)$, we initialize a copy $\texttt{sketch}_{\ell}$ of MAINTAINSKETCH. These are needed for the implementation of FINDLARGECOORDINATES(ℓ) in APPROXIMATE. Specifically, at step $k = 2^{\ell}$ of the IPM, we initialize $\texttt{sketch}_{\ell}$ by calling $\texttt{sketch}_{\ell}$.INITIALIZE$(\mathcal{T}, \boldsymbol{\Phi}, \mathbf{M}_{\ell}^{(k)}, \boldsymbol{z}_{\ell}^{(k)}, \boldsymbol{y}_{\ell}^{(k)})$.

(Although this occurs at step $k > 0$, we charge its runtime according to its function as part of initialization.)

The total initialization time is $O(wm \log m) = O(m\eta^2 \log m \log(\frac{m}{\rho}))$ by Lemma 3.6.9. By the existing pseudocode in Algorithm 17, it correctly initializes $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}$.

Approximate$(\mathbf{M}^{(\text{new})}, c^{(\text{new})}, \boldsymbol{z}^{(\text{prev})^{(\text{new})}}, \boldsymbol{z}^{(\text{sum})^{(\text{new})}}, \boldsymbol{y}^{(\text{new})}, \mathbf{D}^{(\text{new})})$**:** This procedure implements Approximate in Algorithm 17. We consider when the current step is k below.

First, we update the sketch data structures $\texttt{sketch}_\ell$ for each ℓ by calling $\texttt{sketch}_\ell$.Update. Recall at step k, $\texttt{sketch}_\ell$ maintains sketches for the vector $\boldsymbol{x}_\ell^{(k)} = \overline{\mathbf{D}}^{1/2}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^\ell)})$, although the actual representation in $\texttt{sketch}_\ell$ of the vector $\boldsymbol{x}_\ell$ is given by $\boldsymbol{x}_\ell = \boldsymbol{y}_\ell + \mathbf{M}_\ell \boldsymbol{z}_\ell$ as defined in (3.27).

Next, we execute the pseudocode given in Approximate in Algorithm 17:

To update $\overline{\boldsymbol{x}}_e$ to $\boldsymbol{x}_e^{(k-1)}$ for a single coordinate (Line 20 of Algorithm 17), we find the leaf node H containing the edge e, and call `ox_prev`.Query(H). This returns the subvector $\boldsymbol{x}^{(k-1)}|_{E(H)}$, from which we can make the assignment to $\overline{\boldsymbol{x}}_e$. To update $\overline{\boldsymbol{x}}_e$ to $\boldsymbol{x}_e^{(k)}$ for single coordinates (Line 29 of Algorithm 17), we do the same as above, except using the data structure `ox_cur`.

In the subroutine FindLargeCoordinates(ℓ), the vector $\boldsymbol{q}$ defined in the pseudocode is exactly $\boldsymbol{x}_\ell^{(k)}$. We get the value of $\boldsymbol{\Phi}_{E(u)}\boldsymbol{q}$ at a node u by calling $\texttt{sketch}_\ell$.Estimate(u), and we get the value of $\boldsymbol{q}|_{E(u)}$ at a leaf node u by calling $\texttt{sketch}_\ell$.Query(u).

Number of coordinates changed in $\overline{\boldsymbol{x}}$ during Approximate. In Line 20 of Approximate in Algorithm 17, $\overline{\boldsymbol{x}}$ is updated in every coordinate e where $\mathbf{D}_e$ differs compared to the previous step.

Next, the procedure collect a set of coordinates for which we update $\overline{\boldsymbol{x}}$, by calling FindLargeCoordinates(ℓ) for each $0 \leq \ell \leq \ell_k$, where ℓ_k is defined to be the number of trailing zeros in the binary representation of k. (These are exactly the values of ℓ such that $k \equiv 0 \mod 2^\ell$). In each call of FindLargeCoordinates(ℓ), There are $O(2^{2\ell}(\eta/\delta)^2 \log^2 m \log(m/\rho))$ iterations of the outer for-loop, and $O(1)$ iterations of the inner while-loop by the assumption of $\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\|_{\mathbf{D}^{(k+1)}} \leq \beta$ and Lemma 3.6.4. Each iteration of the while-loop adds a $O(1)$ sized set to the collection I of candidate coordinates. So overall, FINDLARGECOORDINATES(ℓ) returns a set of size $O(2^{2\ell}(\eta/\delta)^2 \log^2 m \log(m/\rho))$. Summing up over all calls of FINDLARGECOORDINATES, the total size of the set of coordinates to update is

$$N_k \stackrel{\text{def}}{=} \sum_{\ell=0}^{\ell_k} O(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho)) = O(2^{2\ell_k}(\beta/\delta)^2 \log^2 m). \tag{3.28}$$

We define $\ell_0 = N_0 = 0$ for convenience.

Changes to sketching data structures. Let $\mathcal{S}^{(k)}$ denote the set of nodes H, where one of (when applicable) $\mathbf{M}_{(H,P)}$, $\mathbf{J}_H$, $\boldsymbol{z}^{(\text{prev})}|_{F_H}$, $\boldsymbol{z}^{(\text{sum})}|_{F_H}$, $\boldsymbol{y}_{F_H}$, $\mathbf{D}_{E(H)}$ changes during step k. (They are entirely induced by changes in $\boldsymbol{v}$ and $\boldsymbol{w}$ at step k.) We store $\mathcal{S}^{(k)}$ for each step.

For each ℓ, the diagonal matrix $\overline{\mathbf{D}}$ is the same as $\mathbf{D}$, except $\overline{\mathbf{D}}_{ii}$ is temporarily zeroed out for 2^ℓ steps after $\overline{\boldsymbol{x}}_i$ changes at a step. Thus, the number of coordinate changes to $\overline{\mathbf{D}}$ at step k is the number of changes to $\mathbf{D}$, plus $N_{k-1} + N_{k-2^\ell}$: N_{k-1} entries are zeroed out because of updates to $\overline{\boldsymbol{x}}_i$ in step $k-1$. The N_{k-2^ℓ} entries that were zeroed out in step $k - 2^\ell + 1$ because of the update to $\overline{\boldsymbol{x}}_i$ in step $k - 2^\ell$ are back.

Hence, at step k, the updates to $\texttt{sketch}_\ell$ are induced by updates to $\overline{\mathbf{D}}$, and the updates to $\boldsymbol{x}$ at step k, and at step $k-2^\ell$. The updates to the two $\boldsymbol{x}$ terms are restricted to the nodes $\mathcal{S}^{(k-2^\ell)} \cup \mathcal{S}^{(k)}$ in $\mathcal{T}$ for Algorithm 18. Updates to `ox_cur` and `ox_prev` can be similarly analyzed.

Runtime of Approximate. First, we consider the time to update each $\texttt{sketch}_\ell$: At step k, the analysis above combined with Lemma 3.6.9 show that $\texttt{sketch}_\ell$.UPDATE with new iterations of the appropriate variables run in time

$$\begin{aligned} &O\left(w \cdot T\left(\eta \cdot (|\mathcal{S}^{(k)}| + |\mathcal{S}^{(k-2^\ell)}| + N_{k-1} + N_{k-2^\ell})\right)\right) \\ &\leq w \cdot O\left(T(\eta \cdot (|\mathcal{S}^{(k)}| + N_{k-1} + N_{k-2^\ell}))\right) + w \cdot O\left(T(\eta \cdot |\mathcal{S}^{(k-2^\ell)}|)\right), \end{aligned}$$

where we use the concavity of T. The second term can be charged to step $k - 2^{\ell}$. Thus, the amortized time cost for $\texttt{sketch}_{\ell}$.UPDATE at step k is

$$w \cdot O(T(\eta \cdot (|\mathcal{S}^{(k)}| + N_{k-1} + N_{k-2^{\ell_k}}))).$$

Summing over all $0 \leq \ell \leq O(\log m)$ for the different copies of $\texttt{sketch}_{\ell}$, we get an extra $O(\log m)$ factor in the overall update time.

Similarly, we can update `ox_prev` and `ox_cur` in the same amortized time.

Next, we consider the runtime for Line 20 in Algorithm 17: The number of coordinate accesses to $\boldsymbol{x}^{(k-1)}$ is $|\{i : \mathbf{D}_{ii}^{(k)} - \mathbf{D}_{ii}^{(k-1)} \neq 0\}| = O(\mathcal{S}^{(k)})$. Each coordinate is computed by calling `ox_cur`.QUERY, and by Lemma 3.6.9, the total time for these updates is $w \cdot O(T(\eta \cdot |\mathcal{S}^{(k)}|)$.

Finally, we analyze the remainder of the procedure, which consists of FINDLARGECOORDINATES(ℓ) for each $0 \leq \ell \leq \ell_k$ and the subsequent updates to entries of $\overline{\boldsymbol{x}}$: For each FINDLARGECOORDINATES(ℓ) call, by Lemma 3.6.4, $N_{k,\ell} \overset{\text{def}}{=} \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho))$ sampling paths are explored in the $\texttt{sketch}_{\ell}$ data structure, where each sampling path correspond to one iteration of the while-loop. We calculate $\|\mathbf{\Phi}_{E(H)} \boldsymbol{x}_{\ell}\|_2^2$ at a node H in the sampling path using $\texttt{sketch}_{\ell}$.ESTIMATE(H), and at a leaf node H using $\texttt{sketch}_{\ell}$.QUERY(H). The total time is $w \cdot O(T(\eta \cdot N_{k,\ell}))$ by Lemma 3.6.9. To update a coordinate $i \in E(H)$ that was identified to be large, we can refer to the output of $\texttt{sketch}_{\ell}$.QUERY(H) from the sampling step.

Summing over each $0 \leq \ell \leq \ell_k$, we see that the total time for the FINDLARGECOORDINATES calls and the subsequent updates fo $\overline{\boldsymbol{x}}$ is

$$\sum_{\ell=0}^{\ell_k} w \cdot O(T(\eta \cdot N_{k,\ell})) = w \cdot O(T(\eta \cdot N_k)),$$

where N_k is the number of coordinates that are updated in $\overline{\boldsymbol{x}}$ as shown in (3.28).

Combined with the update times, we conclude that the total amortized cost of APPROXIMATE at step k is

$$\Theta(\eta^2 \log(\frac{m}{\rho}) \log m) \cdot T(\eta \cdot (|\mathcal{S}^{(k)}| + N_{k-1} + N_{k-2^{\ell_k}})).$$

Observe that $N_{k-1} = N_{k-2^0}$ and N_{k-2^ℓ} are both bounded by $O(N_{k-2^{\ell_k}})$: When $\ell \neq \ell_k$, the number of trailing zeros in $k - 2^\ell$ is no more than ℓ_k. When $\ell = \ell_k$, the number of trailing zeros of $k - 2^{\ell_k}$ is $\ell_{k-2^{\ell_k}}$. In both cases, $\ell_{k-2^\ell} \leq \ell_{k-2^{\ell_k}}$. So we have the desired overall runtime. $\square$

3.7 Slack projection

In this section, we define the slack tree operator as required to use MaintainRep. We then give the full slack maintenance data structure.

3.7.1 Tree operator for slack

The full slack update at IPM step k with step direction $\boldsymbol{v}^{(k)}$ and step size $\bar{t}h$ is

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}(\bar{t}h\boldsymbol{v}^{(k)}),$$

where we require $\widetilde{\mathbf{P}}_{\boldsymbol{w}} \approx \mathbf{P}_{\boldsymbol{w}}$ and $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)} \in \text{Range}(\mathbf{W}^{1/2}\mathbf{B})$.

Let $\widetilde{\mathbf{L}}^{-1}$ denote the approximation of $\mathbf{L}^{-1}$ from (3.8), maintained and computable with a DynamicSC data structure. If we define

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

then $\widetilde{\mathbf{P}}_{\boldsymbol{w}} \approx_{\eta\delta} \mathbf{P}_{\boldsymbol{w}}$, and $\text{Range}(\widetilde{\mathbf{P}}_{\boldsymbol{w}}) = \text{Range}(\mathbf{P}_{\boldsymbol{w}})$ by definition, where η and δ are parameters in DynamicSC. Hence, this suffices as our approximate slack projection matrix. In order to use MaintainRep to maintain $\boldsymbol{s}$ throughout the IPM, it remains to define a slack tree operator $\mathbf{M}^{(\text{slack})}$ so that

$$\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \mathbf{M}^{(\text{slack})}\boldsymbol{z}^{(k)},$$

where $\boldsymbol{z}^{(k)} \stackrel{\text{def}}{=} \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}^{(k)}$ at IPM step k. We proceed by defining a tree operator $\mathbf{M}$ satisfying $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \mathbf{M}\boldsymbol{z}^{(k)}$. Namely, we show that $\mathbf{M} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{B}\mathbf{\Pi}^{(0)\top}\cdots\mathbf{\Pi}^{(\eta-1)\top}$ is indeed a tree operator. Then we set $\mathbf{M}^{(\text{slack})} \stackrel{\text{def}}{=} \mathbf{W}^{-1/2}\mathbf{M}$.

For the remainder of the section, we abuse notation and use $\boldsymbol{z}$ to mean $\boldsymbol{z}^{(k)}$ for one IPM step k.

Definition 3.7.1 (Slack projection tree operator). Let $\mathcal{T}$ be the separator tree from data structure DYNAMICSC, with Laplacians $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H)$ at each node $H \in \mathcal{T}$. We use $\mathbf{B}[H]$ to denote the adjacency matrix of G restricted to the region.

For a node $H \in \mathcal{T}$, define $V(H)$ and F_H required by the tree operator as $\partial(H) \cup F_H$ and F_H from the separator tree construction respectively. Note the slightly confusing fact that $V(H)$ *is not* the set of vertices in region H of the input graph G, *unless* H is a leaf node. Suppose node H has parent P, then define the tree edge operator $\mathbf{M}_{(H,P)} : \mathbb{R}^{V(P)} \mapsto \mathbb{R}^{V(H)}$ as:

$$\mathbf{M}_{(H,P)} \stackrel{\text{def}}{=} \mathbf{I}_{\partial(H)\cup F_H} - \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1} \mathbf{L}^{(H)}_{F_H,\partial(H)} = \mathbf{I}_{\partial(H)\cup F_H} - \mathbf{X}^{(H)\top}, \tag{3.29}$$

where $\mathbf{X}^{(H)}$ is defined in (3.15).

At each leaf node H of $\mathcal{T}$, define the leaf operator $\mathbf{J}_H = \mathbf{W}^{1/2}\mathbf{B}[H]$.

The remainder of this section proves the correctness of the tree operator.

Lemma 3.7.2. *Let* $\mathbf{M}$ *be the tree operator as defined in Definition 3.7.1. We have*

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

We begin with a few observations about the $\mathbf{\Pi}^{(i)}$'s:

Observation 3.7.3. *For any* $0 \leq i < \eta$, *and for any vector* $\boldsymbol{x}$, *we have* $\mathbf{\Pi}^{(i)\top}\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{y}_i$, *where* $\boldsymbol{y}_i$ *is a vector supported on* $F_i = \cup_{H\in\mathcal{T}(i)} F_H$. *Extending this observation, for* $0 \leq i < j < \eta$,

$$\mathbf{\Pi}^{(i)\top} \cdots \mathbf{\Pi}^{(j-1)\top}\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{y},$$

where $\boldsymbol{y}$ *is a vector supported on* $F_i \cup \cdots \cup F_{j-1} = \cup_{H: i\leq\eta(H)<j} F_H$. *Furthermore, if* $\boldsymbol{x}$ *is supported on* F_A *for* $\eta(A) = j$, *then* $\boldsymbol{y}$ *is supported on* $\cup_{H\in\mathcal{T}_A} F_H$.

The following helper lemma describes a sequence of edge operators from a node to a leaf.

Lemma 3.7.4. *For any leaf node* $H \in \mathcal{T}$, *and a node* A *with* $H \in \mathcal{T}_A$ *(*A *is an ancestor of* H *or* H *itself), we have*

$$\mathbf{M}_{H\leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{I}_{\partial H\cup F_H}\mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(\eta-1)\top}\boldsymbol{z}|_{F_A}. \tag{3.30}$$

Proof. For simplicity of notation, let $V(H) \stackrel{\text{def}}{=} \partial H \cup F_H$ for a node H.

To start, observe that for a node A at level $\eta(A)$, we have $\mathbf{\Pi}^{(i)}\boldsymbol{z}|_{F_A} = \boldsymbol{z}|_{F_A}$ for all $i \geq \eta(A)$. So it suffices to prove

$$\mathbf{M}_{H \leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{I}_{V(H)}\mathbf{\Pi}^{(0)\top} \cdots \mathbf{\Pi}^{(\eta(A)-1)\top}\boldsymbol{z}|_{F_A}.$$

Let the path from leaf H up to node A in $\mathcal{T}$ be denoted $(H_0 \stackrel{\text{def}}{=} H, H_1, \ldots, H_t \stackrel{\text{def}}{=} A)$, for some $t \leq \eta(A)$. We will prove by induction for k decreasing from t to 0:

$$\mathbf{M}_{H_k \leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{I}_{V(H_k)}\mathbf{\Pi}^{(\eta(H_k))\top}\mathbf{\Pi}^{(\eta(H_k)+1)\top} \cdots \mathbf{\Pi}^{(\eta(A)-1)\top}\boldsymbol{z}|_{F_A}. \tag{3.31}$$

For the base case of $H_t = A$, we have $\mathbf{M}_{H_t \leftarrow A}\boldsymbol{z}|_{F_A} = \boldsymbol{z}|_{F_A} = \mathbf{I}_{V(H_t)}\boldsymbol{z}|_{F_A}$.

For the inductive step at H_k, we first apply induction hypothesis for H_{k+1} to get

$$\mathbf{M}_{H_{k+1} \leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{I}_{V(H_{k+1})}\mathbf{\Pi}^{(\eta(H_{k+1}))\top} \cdots \mathbf{\Pi}^{(\eta(A)-1)\top}\boldsymbol{z}|_{F_A}. \tag{3.32}$$

Multiplying by the edge operator $\mathbf{M}_{(H_k, H_{k+1})}$ on both sides gives

$$\mathbf{M}_{H_k \leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{M}_{(H_k, H_{k+1})}\mathbf{I}_{V(H_{k+1})}\mathbf{\Pi}^{(\eta(H_{k+1}))\top} \cdots \mathbf{\Pi}^{(\eta(A)-1)\top}\boldsymbol{z}|_{F_A}. \tag{3.33}$$

Recall the edge operator $\mathbf{M}_{(H_k, H_{k+1})}$ maps vectors supported on $V(H_{k+1})$ to vectors supported on $V(H_k)$ and zeros otherwise. So we can drop the $\mathbf{I}_{V(H_{k+1})}$ term in the right hand side. Let $\boldsymbol{x} \stackrel{\text{def}}{=} \mathbf{\Pi}^{(\eta(H_{k+1}))\top} \cdots \mathbf{\Pi}^{(\eta(A)-1)\top}\boldsymbol{z}|_{F_A}$. Now, by the definition of the edge operator, the above equation becomes

$$\mathbf{M}_{H_k \leftarrow A}\boldsymbol{z}|_{F_A} = (\mathbf{I}_{V(H_k)} - \mathbf{X}^{(H_k)\top})\boldsymbol{x}. \tag{3.34}$$

On the other hand, we have

$$\begin{aligned}
\mathbf{I}_{V(H_k)}\mathbf{\Pi}^{(\eta(H_k))\top} \cdots \mathbf{\Pi}^{(\eta(H_{k+1})-1)\top}\boldsymbol{x} &= \mathbf{I}_{V(H_k)}\mathbf{\Pi}^{(\eta(H_k))\top}\left(\mathbf{\Pi}^{(\eta(H_k)+1)\top} \cdots \mathbf{\Pi}^{(\eta(H_{k+1})-1)\top}\boldsymbol{x}\right) \\
&= \mathbf{I}_{V(H_k)}\mathbf{\Pi}^{(\eta(H_k))\top}(\boldsymbol{x} + \boldsymbol{y}),
\end{aligned}$$

where $\boldsymbol{y}$ is a vector supported on $\cup F_R$ for nodes R at levels $\eta(H_k)+1, \cdots, \eta(H_{k+1})-1$ by Observation 3.7.3. In particular, $\boldsymbol{y}$ is zero on F_{H_k}. Also, $\boldsymbol{y}$ is zero on ∂H_k, since by Observation 3.4.12, $\partial H_k \subseteq \cup_{\text{ancestor } A' \text{ of } H_k} F_{A'}$, and ancestors of H_k are at level $\eta(H_{k+1})$ or higher. Then $\boldsymbol{y}$ is zero on $V(H_k) = \partial H_k \cup F_{H_k}$, and the right hand side is

$$= (\mathbf{I}_{V(H_k)} - \mathbf{X}^{(H_k)\top})\boldsymbol{x},$$

where we apply the definition of $\boldsymbol{\Pi}^{(\eta(H_k))\top}$ and expand the left-multiplication by $\mathbf{I}_{V(H_k)}$.

Combining with (3.34) and substituting back the definition of $\boldsymbol{x}$, we get

$$\mathbf{M}_{H_k \leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{I}_{V(H_k)}\boldsymbol{\Pi}^{(\eta(H_k))\top} \cdots \boldsymbol{\Pi}^{(\eta(A)-1)\top}\boldsymbol{z}|_{F_A}.$$

which completes the induction.

□

To prove Lemma 3.7.2, we apply the leaf operators to the result of the previous lemma and sum over all nodes and leaf nodes.

Proof of Lemma 3.7.2. Let H be a leaf node. We sum (3.30) over all A with $H \in \mathcal{T}_A$ to get

$$\begin{aligned}\sum_{A: H \in \mathcal{T}_A} \mathbf{M}_{H \leftarrow A}\boldsymbol{z}|_{F_A} &= \mathbf{I}_{\partial H \cup F_H} \sum_{A: H \in \mathcal{T}_A} \boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}\boldsymbol{z}|_{F_A} \\ &= \mathbf{I}_{\partial H \cup F_H}\boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}\boldsymbol{z},\end{aligned}$$

where we relax the sum in the right hand side to be over all nodes in $\mathcal{T}$, since by Observation 3.7.3, for any A with $H \notin \mathcal{T}_A$, we simply have $\mathbf{I}_{\partial H \cup F_H}\boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}\boldsymbol{z}|_{F_A} = \mathbf{0}$. Next, we apply the leaf operator $\mathbf{J}_H = \mathbf{W}^{1/2}\mathbf{B}[H]$ to both sides to get

$$\sum_{A: H \in \mathcal{T}_A} \mathbf{J}_H \mathbf{M}_{H \leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{W}^{1/2}\mathbf{B}[H]\mathbf{I}_{\partial H \cup F_H}\boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top}\boldsymbol{z}.$$

Since $\mathbf{B}[H]$ is zero on columns supported on $V(G) \setminus (\partial H \cup F_H)$, we can simply drop the $\mathbf{I}_{\partial H \cup F_H}$ in the right hand side.

Finally, we sum up the equation above over all leaf nodes. The left hand side is precisely the definition of $\mathbf{M}\boldsymbol{z}$. Recall the regions of the leaf nodes partition the original graph G, so we have

$$\sum_{H\in\mathcal{T}(0)}\sum_{A:H\in\mathcal{T}_A}\mathbf{J}_H\mathbf{M}_{H\leftarrow A}\boldsymbol{z}|_{F_A} = \mathbf{W}^{1/2}\left(\sum_{H\in\mathcal{T}(0)}\mathbf{B}[H]\right)\mathbf{\Pi}^{(0)\top}\cdots\mathbf{\Pi}^{(\eta-1)\top}\boldsymbol{z}$$

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

□

We now examine the slack tree operator complexity.

Lemma 3.7.5. *The complexity of the slack tree operator as defined in Definition 3.8.1 is $T(k) = \widetilde{O}(\sqrt{mk}\cdot\delta^{-2})$, where δ is the Schur complement approximation factor from data structure* DYNAMICSC*.*

Proof. Let $\mathbf{M}_{(D,P)}$ be a tree edge operator. Applying $\mathbf{M}_{(D,P)} = \mathbf{I}_{\partial(D)} - \left(\mathbf{L}^{(D)}_{F_D,F_D}\right)^{-1}\mathbf{L}^{(D)}_{F_D,\partial(D)}$ to the left or right consists of three steps which are applying $\mathbf{I}_{\partial(D)}$, applying $\mathbf{L}^{(D)}_{F_D,\partial(D)}$ and solving for $\mathbf{L}^{(D)}_{F_D,F_D}\boldsymbol{v} = \boldsymbol{b}$ for some vectors $\boldsymbol{v}$ and $\boldsymbol{b}$. Each of the three steps costs time $O(\delta^{-2}|\partial(D)\cup F_D|)$ by Lemma 3.4.22 and Theorem 3.3.1.

For any leaf node H, H has a constant number of edges, and it takes constant time to compute $\mathbf{J}_H\boldsymbol{u}$ for any vector $\boldsymbol{u}$. The number of vertices may be larger but the nonzeros of $\mathbf{J}_H = \mathbf{W}^{1/2}\mathbf{B}[H]$ only depends on the number of edges. To bound the total cost over k distinct edges, we apply Lemma 3.4.16, which then gives the claimed complexity. □

3.7.2 Proof of Theorem 3.2.7

Finally, we give the full data structure for maintaining the slack solution.

The tree operator $\mathbf{M}$ defined in Definition 3.7.1 satisfies $\mathbf{M}\boldsymbol{z}^{(k)} = \widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ at step k, by the definition of $\boldsymbol{z}^{(k)}$. To support the proper update $\boldsymbol{s} \leftarrow \boldsymbol{s} + \bar{t}h\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$, we define $\mathbf{M}^{(\text{slack})} \overset{\text{def}}{=} \mathbf{W}^{-1/2}\mathbf{M}$ and note it is also a tree operator:

Lemma 3.7.6. *Suppose $\mathbf{M}$ is a tree operator supported on $\mathcal{T}$ with complexity $T(K)$. Let $\mathbf{D}$ be a diagonal matrix in $\mathbb{R}^{E\times E}$ where $E = \bigcup_{\text{leaf } H\in\mathcal{T}} E(H)$. Then $\mathbf{DM}$ can be represented by a tree operator with complexity $T(K)$.*

Proof. Suppose $\mathbf{M} \in \mathbb{R}^{E\times V}$. For any vector $\boldsymbol{z} \in \mathbb{R}^V$, $\mathbf{DM}\boldsymbol{z} = \mathbf{D}(\mathbf{M}\boldsymbol{z})$. Thus, to compute $\mathbf{DM}\boldsymbol{z}$, we may first compute $\mathbf{M}\boldsymbol{z}$ and then multiply the i-th entry of $\mathbf{M}\boldsymbol{z}$ with $\mathbf{D}_{i,i}$. This can be achieved by defining a new tree operator $\mathbf{M}'$ with leaf operators $\mathbf{J}'$ such that $\mathbf{J}'_H = \mathbf{D}_{E(H),E(H)}\mathbf{J}_H$ and $\mathbf{M}'_{(H,P)} = \mathbf{M}_{(H,P)}$. The size of each leaf operator remains constant. All edge operators do not change from $\mathbf{M}$. Thus, the new operator $\mathbf{M}'$ has the same complexity as $\mathbf{M}$. $\square$

With the lemma above, we can use MAINTAINREP (Algorithm 16) to maintain the implicit representation of $\boldsymbol{s}$ and Theorem 3.2.6 to maintain an approximate vector $\overline{\boldsymbol{s}}$ as required in Algorithm 12. A single IPM step calls the procedures REWEIGHT, MOVE, APPROXIMATE in this order once. Note that we reinitialize the data structure when $\bar{t}$ changes, so within each instantiation, may assume $\bar{t} = 1$ by scaling. $\bar{t}$ changes only $\widetilde{O}(1)$ times in the IPM.

Theorem 3.2.7 (Slack maintenance)**.** *Given a modified planar graph G with m edges and its separator tree $\mathcal{T}$ with height η, the randomized data structure MAINTAINSLACK (Algorithm 19) implicitly maintains the slack solution $\boldsymbol{s}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{s}}$, and supports the following procedures with high probability against an adaptive adversary:*

- *INITIALIZE$(G, \boldsymbol{s}^{(\text{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \bar{\epsilon} > 0)$: Given a graph G, initial solution $\boldsymbol{s}^{(\text{init})}$, initial direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$ and target approximation accuracy $\bar{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time, and set the representations $\boldsymbol{s} \leftarrow \boldsymbol{s}^{(\text{init})}$ and $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{s}$.*
- *REWEIGHT$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$, given implicitly as a set of changed weights): Set the current weights to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*
- *MOVE$(\alpha \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ given implicitly as a set of changed coordinates): Implicitly update $\boldsymbol{s} \leftarrow \boldsymbol{s} + \alpha\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}$ for some $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ with $\|(\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}\|_2 \leq \eta\delta\,\|\boldsymbol{v}\|_2$, and $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} \in \mathrm{Range}(\mathbf{B})$. The total runtime is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ where K is the number of coordinates changed in $\boldsymbol{v}$.*

Algorithm 19 Slack Maintenance, Main Algorithm

```
1: data structure MaintainSlack extends MaintainRep
2: private: member
3:     MaintainRep maintainRep: data structure to implicitly maintain
                 $$s = y + \mathbf{W}^{-1/2}\mathbf{M}(cz^{(\mathrm{prev})} + z^{(\mathrm{sum})}).$$
                                                        ▷ $\mathbf{M}$ is defined by Definition 3.7.1
4:     MaintainApprox bar_s: data structure to maintain approximation $\overline{s}$ to $s$ (Theorem 3.2.6)
5:
6: procedure Initialize($G, s^{(\mathrm{init})} \in \mathbb{R}^m, v \in \mathbb{R}^m, w \in \mathbb{R}^m_{>0}, \delta > 0, \overline{\epsilon} > 0$)
7:     Build the separator tree $\mathcal{T}$ by Theorem 3.4.13
8:     maintainRep.Initialize($G, \mathcal{T}, \mathbf{W}^{-1/2}\mathbf{M}, v, w, s^{(\mathrm{init})}, \delta$)        ▷ initialize $s \leftarrow s^{(\mathrm{init})}$
9:     bar_s.Initialize($\mathbf{W}^{-1/2}\mathbf{M}, c, z^{(\mathrm{prev})}, z^{(\mathrm{sum})}, y, \mathbf{W}, n^{-5}, \overline{\epsilon}$)   ▷ initialize $\overline{s}$ approximating $s$
10: end procedure
11:
12: procedure Reweight($w^{(\mathrm{new})} \in \mathbb{R}^m_{>0}$)
13:     maintainRep.Reweight($w^{(\mathrm{new})}$)
14: end procedure
15:
16: procedure Move($\alpha, v^{(\mathrm{new})} \in \mathbb{R}^m$)
17:     maintainRep.Move($\alpha, v^{(\mathrm{new})}$)
18: end procedure
19:
20: procedure Approximate( )
21:                       ▷ the variables in the argument are accessed from maintainRep
22:     return $\overline{s}$ = bar_s.Approximate($\mathbf{W}^{-1/2}\mathbf{M}, c, z^{(\mathrm{prev})}, z^{(\mathrm{sum})}, y, \mathbf{W}$)
23: end procedure
24:
25: procedure Exact( )
26:     return maintainRep.Exact()
27: end procedure
```

- *$\textsc{Approximate}() \to \mathbb{R}^m$: Return the vector $\overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \leq \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{s}$.*
- *$\textsc{Exact}() \to \mathbb{R}^m$: Output the current vector $\boldsymbol{s}$ in $\widetilde{O}(m\delta^{-2})$ time.*

Suppose $\alpha\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to $\textsc{Move}$. Suppose in each step, $\textsc{Reweight}$, $\textsc{Move}$ and $\textsc{Approximate}$ are called in order. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $(k-1)$-th and k-th $\textsc{Reweight}$ and $\textsc{Move}$ calls. Then at the k-th $\textsc{Approximate}$ call,

- *the data structure first sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last $\textsc{Reweight}$, then sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k)}$ for $O(N_k \overset{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*
- *The amortized time for the k-th $\textsc{Approximate}$ call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})})$.*

Proof of Theorem 3.2.7. We prove the runtime and correctness of each procedure separately.

Recall by Lemma 3.7.4, the tree operator $\mathbf{M}$ has complexity $T(K) = O(\delta^{-2}\sqrt{mK})$.

Initialize: By the initialization of `maintainRep` (Theorem 3.2.5), the implicit representation of $\boldsymbol{s}$ in `maintainRep` is correct and $\boldsymbol{s} = \boldsymbol{s}^{(\text{init})}$. By the initialization of `approx`, $\overline{\boldsymbol{s}}$ is set to $\boldsymbol{s}$ to start.

Initialization of `maintainRep` takes $\widetilde{O}(m\delta^{-2})$ time by Theorem 3.2.5, and the initialization of `slackSketch` takes $\widetilde{O}(m)$ time by Theorem 3.2.6.

Reweight: In $\textsc{Reweight}$, the value of $\boldsymbol{s}$ does not change, but all the variables in `MaintainRep` are updated to depend on the new weights. The correctness and runtime follow from Theorem 3.2.5.

Move: `maintainRep`.$\textsc{Move}(\alpha, \boldsymbol{v}^{(k)})$ updates the implicit representation of $\boldsymbol{s}$ by

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \mathbf{W}^{-1/2}\mathbf{M}\alpha\boldsymbol{z}^{(k)}.$$

By the definition of the slack projection tree operator $\mathbf{M}$ and Lemma 3.7.2, this is equivalent to the update

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \alpha \mathbf{W}^{-1/2} \widetilde{\mathbf{P}}_{\boldsymbol{w}} \boldsymbol{v}^{(k)},$$

where $\widetilde{\mathbf{P}}_{\boldsymbol{w}} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{\Pi}^{(0)} \cdots \mathbf{\Pi}^{(\eta-1)}\widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\mathbf{B}^{\top}\mathbf{W}^{1/2}$. By Theorem 3.4.21, $\|\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\|_{\mathrm{op}} \leq \eta\delta$. From the definition, $\mathrm{Range}(\mathbf{W}^{1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}) \subseteq \mathrm{Range}(\mathbf{B})$.

By the guarantees of `maintainRep`, if $\boldsymbol{v}^{(k)}$ differs from $\boldsymbol{v}^{(k-1)}$ on K coordinates, then the runtime is $\widetilde{O}(\delta^{-2}\sqrt{mK})$. Furthermore, $\boldsymbol{z}^{(\mathrm{prev})}$ and $\boldsymbol{z}^{(\mathrm{sum})}$ change on F_H for at most $\widetilde{O}(K)$ nodes in $\mathcal{T}$.

Approximate: The returned vector $\overline{\boldsymbol{s}}$ satisfies $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \leq \overline{\epsilon}$ by the guarantee of `bar_s`.APPROXIMATE from Theorem 3.2.6.

Exact: The runtime and correctness directly follow from the guarantee of `maintainRep`.EXACT given in Theorem 3.2.5.

Finally, we have the following lemma about the runtime for APPROXIMATE. Let $\overline{\boldsymbol{s}}^{(k)}$ denote the returned approximate vector at step k.

Lemma 3.7.7. *Suppose $\alpha\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to MOVE. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $k-1$-th and k-th REWEIGHT and MOVE calls. Then at the k-th APPROXIMATE call,*

- *The data structure first sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last REWEIGHT, then sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k)}$ for $O(N_k \overset{\mathrm{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*
- *The amortized time for the k-th APPROXIMATE call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})})$.*

Proof. Since $\overline{\boldsymbol{s}}$ is maintained by `bar_s`, we apply Theorem 3.2.6 with $\boldsymbol{x} = \boldsymbol{s}$ and diagonal matrix $\mathbf{D} = \mathbf{W}$. We need to prove $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\|_{\mathbf{D}^{(k)}} \leq O(\beta)$ for all k first. The constant factor in $O(\beta)$

does not affect the guarantees in Theorem 3.2.6. The left-hand side is

$$\begin{aligned}
\left\|\boldsymbol{s}^{(k)} - \boldsymbol{s}^{(k-1)}\right\|_{\mathbf{W}^{(k)}} &= \left\|\alpha^{(k)}\mathbf{W}^{(k)^{-1/2}}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}\right\|_{\mathbf{W}^{(k)}} && \text{(by } \textsc{Move}\text{)}\\
&= \left\|\alpha^{(k)}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}\right\|_2\\
&\le (1+\eta\delta)\alpha^{(k)}\|\boldsymbol{v}^{(k)}\|_2 && \text{(by the assumption that } \alpha\|\boldsymbol{v}\|_2 \le \beta\text{)}\\
&\le 2\beta.
\end{aligned}$$

Where the second last step follows from $\|\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}}\|_{\mathrm{op}} \le \eta\delta$ and the fact that $\mathbf{P}_{\boldsymbol{w}}$ is an orthogonal projection. Now, we can apply Theorem 3.2.6 to conclude that at each step k, `bar_s`.APPROXIMATE first sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last REWEIGHT, then set $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k)}$ for $O(N_k \overset{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\epsilon})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.

For the second point, MOVE updates $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ on F_H for $\widetilde{O}(K)$ different nodes $H \in \mathcal{T}$ by Theorem 3.2.5. REWEIGHT then updates $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ on F_H for $\widetilde{O}(K)$ different nodes, and updates the tree operator $\mathbf{W}^{-1/2}\mathbf{M}$ on $\widetilde{O}(K)$ different edge and leaf operators. In turn, it updates $\boldsymbol{y}$ on $E(H)$ for $\widetilde{O}(K)$ leaf nodes H. Now, we apply Theorem 3.2.6 and the complexity of the tree operator to conclude the desired amortized runtime. □

□

3.8 Flow projection

In this section, we define the flow tree operator as required to use MAINTAINREP. We then give the full flow maintenance data structure.

During the IPM, we maintain $\boldsymbol{f} \overset{\text{def}}{=} \widehat{\boldsymbol{f}} - \boldsymbol{f}^{\perp}$ by maintaining the two terms separately. For IPM step k with direction $\boldsymbol{v}^{(k)}$ and step size h, we update them as follows:

$$\begin{aligned}
\widehat{\boldsymbol{f}} &\leftarrow \widehat{\boldsymbol{f}} + h\mathbf{W}^{1/2}\boldsymbol{v}^{(k)},\\
\boldsymbol{f}^{\perp} &\leftarrow \boldsymbol{f}^{\perp} + h\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)},
\end{aligned}$$

where $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)}$ satisfies $\left\|\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}\right\|_2 \leq \epsilon \left\|\boldsymbol{v}^{(k)}\right\|_2$ for some factor ϵ, and $\mathbf{B}^\top \mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}^{(k)}$. We will include the initial value of $\boldsymbol{f}$ in $\widehat{\boldsymbol{f}}$.

Maintaining $\widehat{\boldsymbol{f}}$ is straightforward; in the following section, we focus on $\boldsymbol{f}^\perp$.

3.8.1 Tree operator for flow

We hope to use MAINTAINREP to maintain $\boldsymbol{f}^\perp$ throughout the IPM. In order to do so, it remains to define a flow tree operator $\mathbf{M}^{(\text{flow})}$ so that

$$\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \mathbf{M}^{(\text{flow})}\boldsymbol{z}^{(k)},$$

where $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$ satisfies the constraints mentioned above, and $\boldsymbol{z}^{(k)} \stackrel{\text{def}}{=} \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}^{(k)}$. We will define a flow projection tree operator $\mathbf{M}$ so that $\widetilde{\boldsymbol{f}} \stackrel{\text{def}}{=} \mathbf{M}\boldsymbol{v}^{(k)}$ satisfies $\left\|\widetilde{\boldsymbol{f}} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\right\|_2 \leq O(\eta\delta)\left\|\boldsymbol{v}\right\|_2$ and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}} = \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}^{(k)}$. This means it is feasible to set $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}^{(k)} = \widetilde{\boldsymbol{f}}$. Then, we define $\mathbf{M}^{(\text{flow})} \stackrel{\text{def}}{=} \mathbf{W}^{-1/2}\mathbf{M}$.

For the remainder of the section, we abuse notation and use $\boldsymbol{z}$ to mean $\boldsymbol{z}^{(k)}$ for one IPM step k.

Definition 3.8.1 (Flow projection tree operator). Let $\mathcal{T}$ be the separator tree from data structure DYNAMICSC, with Laplacians $\mathbf{L}^{(H)}$ and $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H)$ at each node $H \in \mathcal{T}$. We use $\mathbf{B}[H]$ to denote the adjacency matrix of G restricted to the region.

To define the flow projection tree operator $\mathbf{M}$, we proceed as follows: The tree operator is supported on the tree $\mathcal{T}$. For a node $H \in \mathcal{T}$ with parent P, define the tree edge operator $\mathbf{M}_{(H,P)}$ as:

$$\mathbf{M}_{(H,P)} \stackrel{\text{def}}{=} (\mathbf{L}^{(H)})^{-1}\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H). \tag{3.35}$$

At each node H, we let F_H in the tree operator be the set F_H of eliminated vertices defined in the separator tree. At each leaf node H of $\mathcal{T}$, we have the leaf operator $\mathbf{J}_H = \mathbf{W}^{1/2}\mathbf{B}[H]$.

Before we give intuition and formally prove the correctness of the flow tree operator, we examine its complexity.

Lemma 3.8.2. *The complexity of the flow tree operator as defined in Definition 3.8.1 is $T(k) = \widetilde{O}(\sqrt{mk}\cdot\delta^{-2})$, where δ is the overall approximation factor from data structure* DYNAMICSC.

Proof. Let $\mathbf{M}_{(H,P)}$ be a tree edge operator. Note that it is a symmetric matrix. For any leaf node H, H has a constant number of edges, and it takes constant time to compute $\mathbf{J}_H \boldsymbol{u}$ for any vector $\boldsymbol{u}$. The number of vertices may be larger but the nonzeros of $\mathbf{J}_H = \mathbf{W}^{1/2}\mathbf{B}[H]$ only depends on the number of edges.

If H is not a leaf node, then $\mathbf{M}_{(H,P)}\boldsymbol{u}$ consists of multiplying with $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H)$ and solving the Laplacian system $\mathbf{L}^{(H)}$. By Lemma 3.4.22 and Theorem 3.3.1, this can be done in $\widetilde{O}(\delta^{-2} \cdot |\partial H|)$ time. To bound the total cost over k distinct edges, we apply Lemma 3.4.16, which gives the claimed complexity. $\square$

Theorem 3.8.3. *Let $\boldsymbol{v} \in \mathbb{R}^m$, and let $\boldsymbol{z} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$. Let $\mathbf{M}$ be the flow projection tree operator from Definition 3.8.1. Suppose $\delta = O(1/\log m)$ is the overall approximation factor from* DYNAMICSC. *Then $\widetilde{\boldsymbol{f}} \overset{\text{def}}{=} \mathbf{M}\boldsymbol{z}$ satisfies $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}} = \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$ and $\left\|\widetilde{\boldsymbol{f}} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\right\|_2 \leq O(\eta\delta)\left\|\boldsymbol{v}\right\|_2$.*

The remainder of the section is dedicated to proving this theorem.

Fix $\boldsymbol{v}$ for the remainder of this section. Let $\boldsymbol{d} \overset{\text{def}}{=} \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v} \in \mathbb{R}^n$; since it is supported on the vertices of G and its entries sum to 0, it is a *demand* vector. In the first part of the proof, we show that $\widetilde{\boldsymbol{f}}$ routes the demand $\boldsymbol{d}$. Let $\boldsymbol{f}^\star \overset{\text{def}}{=} \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$. In the second part of the proof, we show that $\widetilde{\boldsymbol{f}}$ is close to $\boldsymbol{f}^\star$. Finally, a remark about terminology:

Remark 3.8.4. If $\mathbf{B}$ is the incidence matrix of a graph, then any vector of the form $\mathbf{B}\boldsymbol{x}$ is a flow by definition. Often in this section, we have vectors of the form $\mathbf{W}^{1/2}\mathbf{B}\boldsymbol{x}$. In this case, we refer to it as a *weighted flow*. We say a weighted flow $\boldsymbol{f}$ routes a demand $\boldsymbol{d}$ if $(\mathbf{W}^{1/2}\mathbf{B})^\top\boldsymbol{f} = \boldsymbol{d}$.

We proceed with a series of lemmas and their intuition, before tying them together in the overall proof at the end of the section.

Lemma 3.8.5. *Let $\boldsymbol{z} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)}\mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$ be as given in Theorem 3.8.3. For each node $H \in \mathcal{T}$, let $\boldsymbol{z}|_{F_H}$ be the sub-vector of $\boldsymbol{z}$ supported on the vertices F_H, and define the demand*

$$\boldsymbol{d}^{(H)} \overset{\text{def}}{=} \mathbf{L}^{(H)}\boldsymbol{z}|_{F_H}.$$

Then $\boldsymbol{d} = \sum_{H\in\mathcal{T}} \boldsymbol{d}^{(H)}$.

Proof. In the proof, note that all $\mathbf{I}$ are $n \times n$ matrices, and we implicitly pad all vectors with the necessary zeros to match the dimensions. For example, $\boldsymbol{z}|_{F_H}$ below should be viewed as an n-dimensional vector supported on F_H. Define

$$\mathbf{X}^{(i)} = \sum_{H \in \mathcal{T}(i)} \mathbf{X}^{(H)}.$$

We have

$$\mathbf{\Pi}^{(i)} = \mathbf{I} - \mathbf{X}^{(i)} = \mathbf{I} - \sum_{H \in \mathcal{T}(i)} \mathbf{L}^{(H)}_{\partial(H),F_H} \left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1}.$$

Suppose H is at level i of $\mathcal{T}$. We have

$$\begin{aligned} \boldsymbol{z}|_{F_H} &= (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d} \\ &= (\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\mathbf{\Pi}^{(i-1)} \cdots \mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d}, \end{aligned} \tag{3.36}$$

where we use the fact $\mathrm{Im}(\mathbf{X}^{(H')}) \cap F_H = \emptyset$ if $\eta(H') \geq i$. From this expression for $\boldsymbol{z}|_{F_H}$, we have

$$\begin{aligned} \boldsymbol{d}^{(H)} &\stackrel{\text{def}}{=} \mathbf{L}^{(H)}\boldsymbol{z}|_{F_H} \\ &= \mathbf{L}^{(H)}_{\partial H,F_H}\boldsymbol{z}|_{F_H} + \mathbf{L}^{(H)}_{F_H,F_H}\boldsymbol{z}|_{F_H} \\ &= \mathbf{X}^{(H)}(\mathbf{\Pi}^{(i-1)} \cdots \mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d})_{F_H} + (\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d})|_{F_H}, \end{aligned}$$

where the last line follows from (3.36). By padding zeros to $\mathbf{X}^{(H)}$, we can write the equation above as

$$\boldsymbol{d}^{(H)} = \mathbf{X}^{(H)}\mathbf{\Pi}^{(i-1)} \cdots \mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d} + (\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d})|_{F_H}.$$

Now, computing the sum, we have

$$\begin{aligned}
\sum_{H\in\mathcal{T}} \boldsymbol{d}^{(H)} &= \sum_{i=0}^{\eta}\sum_{H\in\mathcal{T}(i)} \mathbf{X}^{(H)}\mathbf{\Pi}^{(i-1)}\cdots\mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d} + \sum_{i=0}^{\eta}\sum_{H\in\mathcal{T}(i)} (\mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d})|_{F_H} \\
&= \left(\sum_{i=0}^{\eta} \mathbf{X}^{(i)}\mathbf{\Pi}^{(i-1)}\cdots\mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d}\right) + \mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d} \qquad (F_H \text{ partition } V(G)) \\
&= \left(\sum_{i=0}^{\eta-1} (\mathbf{I}-\mathbf{\Pi}^{(i)})\mathbf{\Pi}^{(i-1)}\cdots\mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d}\right) + \mathbf{\Pi}^{(\eta-1)}\cdots\mathbf{\Pi}^{(1)}\mathbf{\Pi}^{(0)}\boldsymbol{d} \\
&= \boldsymbol{d}, \qquad \text{(telescoping sum)}
\end{aligned}$$

completing our proof. □

Next, we examine the feasibility of $\widetilde{\boldsymbol{f}}$. To begin, we introduce a decomposition of $\widetilde{\boldsymbol{f}}$ based on the decomposition of $\boldsymbol{d}$, and prove its feasibility.

Definition 3.8.6. Let $\mathbf{M}^{(H)}$ be the flow tree operator supported on the tree $\mathcal{T}_H \in \mathcal{F}$ (Definition 3.5.9). We define the flow $\widetilde{\boldsymbol{f}}^{(H)} \stackrel{\text{def}}{=} \mathbf{M}^{(H)}\boldsymbol{z} = \mathbf{M}^{(H)}\boldsymbol{z}|_{F_H}$.

Lemma 3.8.7. *We have that* $(\mathbf{W}^{1/2}\mathbf{B})^\top \widetilde{\boldsymbol{f}}^{(H)} = \boldsymbol{d}^{(H)}$. *In other words, the weighted flow* $\widetilde{\boldsymbol{f}}^{(H)}$ *routes the demand* $\boldsymbol{d}^{(H)}$ *using the edges of the original graph* G.

Proof. We will first show inductively that for each $H \in \mathcal{T}$, we have $\mathbf{B}^\top\mathbf{W}^{1/2}\mathbf{M}^{(H)} = \mathbf{L}^{(H)}$.

In the base case, if H is a leaf node of $\mathcal{T}$, then $\mathcal{F}_H$ is a tree with root H and a single leaf node under it. Then $\mathbf{M}^{(H)} = \mathbf{W}^{1/2}\mathbf{B}[H]$. It follows that

$$\mathbf{B}^\top\mathbf{W}^{1/2}\mathbf{M}^{(H)} = \mathbf{B}^\top\mathbf{W}^{1/2}\mathbf{W}^{1/2}\mathbf{B}[H] = \mathbf{L}^{(H)},$$

by definition of $\mathbf{L}^{(H)}$ for a leaf H of $\mathcal{T}$.

In the other case, H is not a leaf node of $\mathcal{T}$. Let D_1, D_2 be the two children of H. Then

$$\begin{aligned}
\mathbf{B}^\top \mathbf{W}^{1/2} \mathbf{M}^{(H)} &= \mathbf{B}^\top \mathbf{W}^{1/2} \left(\mathbf{M}^{(D_1)} \mathbf{M}_{(D_1,H)} + \mathbf{M}^{(D_2)} \mathbf{M}_{(D_2,H)} \right) \\
&= \mathbf{L}^{(D_1)} \mathbf{M}_{(D_1,H)} + \mathbf{L}^{(D_2)} \mathbf{M}_{(D_2,H)} && \text{(by induction)} \\
&= \mathbf{L}^{(D_1)} (\mathbf{L}^{(D_1)})^{-1} \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial D_1) + \mathbf{L}^{(D_2)} (\mathbf{L}^{(D_2)})^{-1} \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial D_2) \\
&= \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial D_1) + \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial D_2) \\
&= \mathbf{L}^{(H)}.
\end{aligned}$$

Finally, we conclude that $\mathbf{B}^\top \mathbf{W}^{1/2} \widetilde{\boldsymbol{f}}^{(H)} = \mathbf{B}^\top \mathbf{W}^{1/2} \mathbf{M}^{(H)} \boldsymbol{z}|_{F_H} = \mathbf{L}^{(H)} \boldsymbol{z}_{F_H} = \boldsymbol{d}^{(H)}$, where the last inequality follows by definition of $\boldsymbol{d}^{(H)}$. □

We observe an orthogonality property of the flows, which will become useful later:

Lemma 3.8.8. *For any nodes H, H' at the same level in $\mathcal{T}$,* Range$(\mathbf{M}^{(H)})$ *and* Range$(\mathbf{M}^{(H')})$ *are disjoint. Consequently, the flows $\widetilde{\boldsymbol{f}}^{(H)}$ and $\widetilde{\boldsymbol{f}}^{(H')}$ are orthogonal.*

Proof. Recall leaves of $\mathcal{T}$ correspond to pairwise edge-disjoint, constant-sized regions of the original graph G. Since H and H' are at the same level in $\mathcal{T}$, we know $\mathcal{T}_H$ and $\mathcal{T}_{H'}$ have disjoint sets of leaves. The range of $\mathbf{M}^{(H)}$ is supported on edges in the regions given by leaves of $\mathcal{T}_H$, and analogously for the range of $\mathbf{M}^{(H')}$. □

Next, we set up the tools for bounding $\left\| \widetilde{\boldsymbol{f}} - \mathbf{P}_{\boldsymbol{w}} \boldsymbol{v} \right\|_2$, involving an energy analysis drawing inspiration from electric flow routing. We begin with the canonical definitions and properties of electric-flow energy.

Definition 3.8.9. Let $\mathbf{W}^{1/2}\mathbf{B}$ be the edge-weighted incidence matrix of some graph G, and let $\mathbf{L} \stackrel{\text{def}}{=} \mathbf{B}^\top \mathbf{W} \mathbf{B}$ be the Laplacian. Let $\boldsymbol{d} \stackrel{\text{def}}{=} \mathbf{L}\boldsymbol{z}$ be a demand and $\boldsymbol{f}$ be any weighted flow that routes $\boldsymbol{d}$; that is, $(\mathbf{W}^{1/2}\mathbf{B})^\top \boldsymbol{f} = \boldsymbol{d}$. Then we say $\|\boldsymbol{f}\|_2^2$ is the *energy* of the flow $\boldsymbol{f}$.

There is a unique energy-minimizing flow $\boldsymbol{f}^\star$ routing the demand $\boldsymbol{d}$ on G. From the study of electric flows, we know $\boldsymbol{f}^\star = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$. Hence, we can refer to its energy as the energy of the demand $\boldsymbol{d}$ on the graph of $\mathbf{L}$, given by

$$\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}) \stackrel{\text{def}}{=} \min_{(\mathbf{W}^{1/2}\mathbf{B})^\top \boldsymbol{f}=\boldsymbol{d}} \|\boldsymbol{f}\|_2^2 = \boldsymbol{d}^\top(\mathbf{B}^\top\mathbf{W}\mathbf{B})^{-1}\boldsymbol{d} = \boldsymbol{d}^\top\mathbf{L}^{-1}\boldsymbol{d} = \boldsymbol{z}^\top\mathbf{L}\boldsymbol{z}. \tag{3.37}$$

We want to understanding how the energy changes when, instead of routing $\boldsymbol{d}$ using the edges of G, we use edges of some other graphs related to G. In particular, we are interested in the operations of graph decompositions and taking Schur complements. It turns out the energy behaves nicely:

Lemma 3.8.10. *Suppose G is a weighted graph that can be decomposed into weighted subgraphs G_1, G_2. That is, if $\mathbf{L}$ is the Laplacian of G, and $\mathbf{L}^{(i)}$ is the Laplacian of G_i, then $\mathbf{L} = \mathbf{L}^{(1)} + \mathbf{L}^{(2)}$. Suppose $\boldsymbol{d} \stackrel{\text{def}}{=} \mathbf{L}\boldsymbol{z}$ is a demand on the vertices of G. Then if we decompose $\boldsymbol{d} = \boldsymbol{d}^{(1)} + \boldsymbol{d}^{(2)}$, where $\boldsymbol{d}^{(i)} = \mathbf{L}^{(i)}\boldsymbol{z}$, then the energies are related as:*

$$\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}) = \mathcal{E}_{\mathbf{L}^{(1)}}(\boldsymbol{d}^{(1)}) + \mathcal{E}_{\mathbf{L}^{(2)}}(\boldsymbol{d}^{(2)}).$$

Proof. We have, by definition,

$$\begin{aligned}\mathcal{E}_{\mathbf{L}^{(1)}}(\boldsymbol{d}^{(1)}) + \mathcal{E}_{\mathbf{L}^{(2)}}(\boldsymbol{d}^{(2)}) &= \boldsymbol{z}^\top\mathbf{L}^{(1)}\boldsymbol{z} + \boldsymbol{z}^\top\mathbf{L}^{(2)}\boldsymbol{z} \\ &= \boldsymbol{z}^\top\mathbf{L}\boldsymbol{z} \\ &= \mathcal{E}_{\mathbf{L}}(\boldsymbol{d}).\end{aligned}$$

□

The following lemma shows if G' is a graph derived from G by taking Schur complement on a subset of the vertices C, and $\boldsymbol{d}$ is a demand supported on C, then the flow routing $\boldsymbol{d}$ on G will have lower energy than the flow routing $\boldsymbol{d}$ on G'.

Lemma 3.8.11. *Suppose G is a weighted graph with Laplacian $\mathbf{L}$. Let C be a subset of vertices of G. Let $\mathbf{L}' = \widetilde{\mathbf{Sc}}(\mathbf{L}, C)$ be an ϵ-approximate Schur complement. Then for the demand $\boldsymbol{d} = \mathbf{L}'\boldsymbol{z}$ supported on C,*

$$\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}) \leq_\epsilon \mathcal{E}_{\mathbf{L}'}(\boldsymbol{d}).$$

Proof. We have, by definition,

$$\begin{aligned}
\mathcal{E}_{\mathbf{L}}(\mathbf{L}'\boldsymbol{z}) &= \boldsymbol{z}^\top \mathbf{L}'\mathbf{L}^{-1}\mathbf{L}'\boldsymbol{z} \\
&\leq \boldsymbol{z}^\top \mathbf{L}'\mathbf{Sc}(\mathbf{L}, C)^{-1}\mathbf{L}'\boldsymbol{z} && \text{(since } \mathbf{Sc}(\mathbf{L}, C) \preccurlyeq \mathbf{L}\text{)} \\
&\approx_\epsilon \boldsymbol{z}^\top \mathbf{L}'\mathbf{L}'^{-1}\mathbf{L}'\boldsymbol{z} \\
&= \mathcal{E}_{\mathbf{L}'}(\mathbf{L}'\boldsymbol{z}).
\end{aligned}$$

□

For any $H \in \mathcal{T}$, we know $\widetilde{\boldsymbol{f}}^{(H)}$ routes $\boldsymbol{d}^{(H)}$ using the original graph G. Furthermore, we know the graph of $\mathbf{L}^{(H)}$ is related to G using the graph operations considered above. Suppose $\boldsymbol{f}^{(H)^\star}$ is the energy-minimizing flow routing $\boldsymbol{d}^{(H)}$ on the graph of $\mathbf{L}^{(H)}$. Then we want to relate the energies of $\widetilde{\boldsymbol{f}}^{(H)}$ and $\boldsymbol{f}^{(H)^\star}$:

Lemma 3.8.12. *Let H be a node at level i in $\mathcal{T}$. Given* any *$\boldsymbol{z}$, let $\boldsymbol{d} \stackrel{\text{def}}{=} \mathbf{L}^{(H)}\boldsymbol{z}$ be a demand. Then the weighted flow $\boldsymbol{f} \stackrel{\text{def}}{=} \mathbf{M}^{(H)}\boldsymbol{z}$ satisfies $\|\boldsymbol{f}\|_2^2 \leq_{i\delta} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d})$.*

Consequently, $\left\|\widetilde{\boldsymbol{f}}^{(H)}\right\|_2^2 \leq_{i\delta} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)})$.

Proof. We proceed by induction. In the base case, H is a leaf node, and we have

$$\left\|\mathbf{M}^{(H)}\boldsymbol{z}\right\|_2^2 = \boldsymbol{z}^\top(\mathbf{B}[H])^\top \mathbf{W}\mathbf{B}[H]\boldsymbol{z} = \boldsymbol{z}^\top \mathbf{L}^{(H)}\boldsymbol{z} = \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}).$$

Suppose H is at level $i > 0$ in $\mathcal{T}$, with children D_1 and D_2 at level at most $i-1$. Then

$$\begin{aligned}
&\left\|\mathbf{M}^{(H)}\boldsymbol{z}\right\|_2^2 \\
&= \left\|\left(\mathbf{M}^{(D_1)}\mathbf{M}_{(D_1,H)} + \mathbf{M}^{(D_2)}\mathbf{M}_{(D_2,H)}\right)\boldsymbol{z}\right\|_2^2
\end{aligned}$$

Since Range($\mathbf{M}^{(D_1)}$) and Range($\mathbf{M}^{(D_2)}$) are orthogonal, we have

$$\begin{aligned}
&= \left\| \mathbf{M}^{(D_1)} \mathbf{M}_{(D_1,H)} \boldsymbol{z} \right\|_2^2 + \left\| \mathbf{M}^{(D_2)} \mathbf{M}_{(D_2,H)} \boldsymbol{z} \right\|_2^2 \\
&\leq_{(i-1)\delta} \mathcal{E}_{\mathbf{L}^{(D_1)}} \left(\mathbf{L}^{(D_1)} \mathbf{M}_{(D_1,H)} \boldsymbol{z} \right) + \mathcal{E}_{\mathbf{L}^{(D_2)}} \left(\mathbf{L}^{(D_2)} \mathbf{M}_{(D_2,H)} \boldsymbol{z} \right) \\
&\hspace{6cm} \text{(by inductive hypothesis with } \boldsymbol{z} = \mathbf{M}_{(D_i,H)} \boldsymbol{z}\text{)} \\
&= \mathcal{E}_{\mathbf{L}^{(D_1)}} \left(\mathbf{L}^{(D_1)} (\mathbf{L}^{(D_1)})^{-1} \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial(D_1)) \boldsymbol{z} \right) + \mathcal{E}_{\mathbf{L}^{(D_2)}} \left(\mathbf{L}^{(D_2)} (\mathbf{L}^{(D_2)})^{-1} \widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial(D_2)) \boldsymbol{z} \right) \\
&\leq_{\delta} \mathcal{E}_{\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial(D_1))} \left(\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_1)}, \partial(D_1)) \boldsymbol{z} \right) + \mathcal{E}_{\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial(D_2))} \left(\widetilde{\mathbf{Sc}}(\mathbf{L}^{(D_2)}, \partial(D_2)) \boldsymbol{z} \right) \\
&= \mathcal{E}_{\mathbf{L}^{(H)}} (\mathbf{L}^{(H)} \boldsymbol{z}),
\end{aligned}$$

where the last two inequalities follow from Lemmas 3.8.10 and 3.8.11. □

Next, we want to relate the energy of routing $\boldsymbol{d}^{(H)}$ on the graph G and the energy on the graph of $\mathbf{L}^{(H)}$.

Lemma 3.8.13. *For a node H at level i in $\mathcal{T}$,*

$$\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}^{(H)}) \approx_{\delta} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}).$$

□

Proof. For one direction, we have

$$\begin{aligned}
\mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}) &= \boldsymbol{d}^{(H)^\top} \mathbf{L}^{(H)^{-1}} \boldsymbol{d}^{(H)} \\
&\approx_{\delta} \boldsymbol{d}^{(H)^\top} \mathbf{Sc}(\mathbf{L}[H], \partial(H) \cup F_H)^{-1} \boldsymbol{d}^{(H)} && \text{(by Theorem 3.2.4)} \\
&\leq \boldsymbol{d}^{(H)^\top} \mathbf{L}^{-1} \boldsymbol{d}^{(H)} && \text{(since } \mathbf{Sc}(\mathbf{L}, C) \succcurlyeq \mathbf{L}\text{)} \\
&= \mathcal{E}_{\mathbf{L}}(\boldsymbol{d}^{(H)}).
\end{aligned}$$

In the other direction, we note that $\widetilde{\boldsymbol{f}}^{(H)}$ is a weighted flow routing $\boldsymbol{d}^{(H)}$ on G. By Lemma 3.8.12 and the definition of energy,

$$\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}^{(H)}) \leq \left\| \widetilde{\boldsymbol{f}}^{(H)} \right\|_2^2 \approx_{i\delta} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}).$$

□

We need to further bound the sum of energies:

Lemma 3.8.14. *We have the following approximation of the energy of $\boldsymbol{d}$ on graph G:*

$$\sum_{H \in \mathcal{T}} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}) \approx_{\eta\delta} \mathcal{E}_{\mathbf{L}}(\boldsymbol{d}).$$

Proof. We need the following matrix multiplication property: For any matrices $\mathbf{A}, \mathbf{B}, \mathbf{D}$,

$$\begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^{\top} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{3.38}$$

Recall in our setting, all matrices are padded with zeros so that their dimension is $n \times n$, and vectors padded with zeros so their dimension is n.

Define $\boldsymbol{\beta} \stackrel{\text{def}}{=} \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(1)} \mathbf{\Pi}^{(0)} \boldsymbol{d}$ for simplicity. Recall $\boldsymbol{z} \stackrel{\text{def}}{=} \widetilde{\mathbf{\Gamma}} \mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)} \mathbf{B}^{\top} \mathbf{W}^{1/2} \boldsymbol{v}$. We can write

$$\boldsymbol{z}|_{F_H} = \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \boldsymbol{\beta}.$$

Then,

$$\begin{aligned} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}) &= \boldsymbol{z}^{\top}|_{F_H} \mathbf{L}^{(H)} \boldsymbol{z}|_{F_H} \\ &= \boldsymbol{\beta}^{\top} \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \mathbf{L}^{(H)} \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \boldsymbol{\beta} \\ &= \boldsymbol{\beta}^{\top} \left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \boldsymbol{\beta}. && \text{(by (3.38))} \end{aligned}$$

Summing over all $H \in \mathcal{T}$, we get

$$\begin{aligned}\sum_{H \in \mathcal{T}} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}) &= \boldsymbol{\beta}^\top \sum_{H \in \mathcal{T}} (\mathbf{L}^{(H)}_{F_H, F_H})^{-1} \boldsymbol{\beta} \\ &= \boldsymbol{d}^\top \boldsymbol{\Pi}^{(0)\top} \cdots \boldsymbol{\Pi}^{(\eta-1)\top} \left[\sum_H (\mathbf{L}^{(H)}_{F_H, F_H})^{-1} \right] \boldsymbol{\Pi}^{(\eta-1)} \cdots \boldsymbol{\Pi}^{(0)} \boldsymbol{d} \\ &\approx_{\eta\delta} \boldsymbol{d}^\top \mathbf{L}^{-1} \boldsymbol{d} \\ &= \mathcal{E}_{\mathbf{L}}(\boldsymbol{d}).\end{aligned}$$

where the last second step follows by Theorem 3.4.21. □

Lastly, the following lemma shows that our weighted flow $\widetilde{\boldsymbol{f}}$ routing $\boldsymbol{d}$ can be orthogonally decomposed in terms of the unique energy minimizer $\boldsymbol{f}^\star$, which in turn allows us to bound $\|\widetilde{\boldsymbol{f}} - \boldsymbol{f}^\star\|_2^2$.

Lemma 3.8.15. *Let* $\mathbf{L}$ *be a weighted Laplacian as above, and let* $\boldsymbol{d}$ *be a demand. Let* $\boldsymbol{f}^\star = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$ *be the weighted electric flow routing* $\boldsymbol{d}$ *attaining the minimum energy* $\mathcal{E}_{\mathbf{L}}(\boldsymbol{d})$*. For any other weighted flow* $\widetilde{\boldsymbol{f}}$ *satisfying* $\mathbf{B}^\top \mathbf{W}^{1/2} \widetilde{\boldsymbol{f}} = \boldsymbol{d}$*, if* $\|\widetilde{\boldsymbol{f}}\|_2^2 \leq_\epsilon \mathcal{E}_{\mathbf{L}}(\boldsymbol{d})$*, then*

$$\|\widetilde{\boldsymbol{f}} - \boldsymbol{f}^\star\|_2^2 \leq (e^\epsilon - 1) \|\boldsymbol{f}^\star\|_2^2.$$

Proof. Observe that

$$\boldsymbol{f}^{\star\top}(\widetilde{\boldsymbol{f}} - \boldsymbol{f}^\star) = \boldsymbol{d}^\top \mathbf{L}^{-1} \mathbf{B}^\top \mathbf{W}^{1/2} (\widetilde{\boldsymbol{f}} - \boldsymbol{f}^\star) = \boldsymbol{d}^\top \mathbf{L}^{-1} (\boldsymbol{d} - \boldsymbol{d}) = \boldsymbol{0}.$$

Hence, we have an orthogonal decomposition of $\widetilde{\boldsymbol{f}}$:

$$\|\widetilde{\boldsymbol{f}}\|_2^2 = \|\widetilde{\boldsymbol{f}}^\star\|_2^2 + \|\widetilde{\boldsymbol{f}} - \boldsymbol{f}^\star\|_2^2.$$

It follows that

$$\|\boldsymbol{f} - \boldsymbol{f}^\star\|^2 \leq (e^\epsilon - 1) \cdot \|\boldsymbol{f}^*\|_2^2.$$

□

Finally, we put all the lemmas together for the overall proof that $\widetilde{\boldsymbol{f}}$ is the desired weighted flow.

Proof of Theorem 3.8.3. We first decompose $\boldsymbol{d} = \sum_{H \in \mathcal{T}} \boldsymbol{d}^{(H)}$ according to Lemma 3.8.5. By definition of the flow tree operator,

$$\widetilde{\boldsymbol{f}} \stackrel{\text{def}}{=} \mathbf{M}\boldsymbol{z} \stackrel{\text{def}}{=} \sum_{H \in \mathcal{T}} \mathbf{M}^{(H)} \boldsymbol{z}|_{F_H} = \sum_{H \in \mathcal{T}} \widetilde{\boldsymbol{f}}^{(H)},$$

where $\widetilde{\boldsymbol{f}}^{(H)} \stackrel{\text{def}}{=} \mathbf{M}^{(H)} \boldsymbol{z}|_{F_H}$ routes demand $\boldsymbol{d}^{(H)}$ by Lemma 3.8.7. Hence,

$$(\mathbf{W}^{1/2}\mathbf{B})^\top \widetilde{\boldsymbol{f}} = \sum_{H \in \mathcal{T}} (\mathbf{W}^{1/2}\mathbf{B})^\top \widetilde{\boldsymbol{f}}^{(H)} = \sum_{H \in \mathcal{T}} \boldsymbol{d}^{(H)} = \boldsymbol{d},$$

meaning $\widetilde{\boldsymbol{f}}$ is feasible for routing $\boldsymbol{d}$ on G.

For each demand term $\boldsymbol{d}^{(H)}$, let $\boldsymbol{f}^{(H)\star}$ be the weighted flow on G that attains the minimum energy $\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}^{(H)})$ for routing it. By Definition 3.8.9 , $\boldsymbol{f}^{(H)\star} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}^{(H)}$. Recall $\boldsymbol{f}^\star \stackrel{\text{def}}{=} \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{W}^{1/2}\mathbf{B}\mathbf{L}^{-1}\boldsymbol{d}$. Hence,

$$\boldsymbol{f}^\star = \sum_{H \in \mathcal{T}} \boldsymbol{f}^{(H)\star}.$$

By Lemma 3.8.13, we know if H is at level i in $\mathcal{T}$, then $\widetilde{\boldsymbol{f}}^{(H)}$ satisfies

$$\left\| \widetilde{\boldsymbol{f}}^{(H)} \right\|_2^2 \leq_{i\delta} \mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}) \approx_{i\delta} \mathcal{E}_{\mathbf{L}}(\boldsymbol{d}^{(H)}) = \left\| \boldsymbol{f}^{(H)\star} \right\|_2^2. \tag{3.39}$$

This shows that in the flow tree operator, the output $\widetilde{\boldsymbol{f}}^{(H)}$ of each tree operator $\mathbf{M}^{(H)}$ is close to the natural corresponding term $\boldsymbol{f}^{(H)\star}$. Finally, we bound the overall approximation error:

$$\begin{aligned}
\left\|\widetilde{\boldsymbol{f}} - \boldsymbol{f}^{\star}\right\|_2^2 &= \left\|\sum_{H\in\mathcal{T}} \left(\widetilde{\boldsymbol{f}}^{(H)} - \boldsymbol{f}^{(H)\star}\right)\right\|_2^2 \\
&\le \left(\sum_{H\in\mathcal{T}} \left\|\widetilde{\boldsymbol{f}}^{(H)} - \boldsymbol{f}^{(H)\star}\right\|_2\right)^2 \\
&= \sum_{i=0}^{\eta} \sum_{H\in\mathcal{T}(i)} (e^{2i\delta} - 1)\mathcal{E}_{\mathbf{L}}(\boldsymbol{d}^{(H)}) && \text{(by Lemma 3.8.15 and (3.39))} \\
&\le \sum_{i=0}^{\eta} \sum_{H\in\mathcal{T}(i)} (e^{2i\delta} - 1)e^{i\delta}\mathcal{E}_{\mathbf{L}^{(H)}}(\boldsymbol{d}^{(H)}) && \text{(by Lemma 3.8.13)} \\
&\le e^{4\eta\delta} \sum_{H\in\mathcal{T}} \mathcal{E}_{\mathbf{L}}(\boldsymbol{d}) && \text{(by Lemma 3.8.14)} \\
&= O(\eta\delta)\left\|\boldsymbol{f}^{\star}\right\|^2,
\end{aligned}$$

which concludes the overall proof. □

3.8.2 Proof of Theorem 3.2.8

Finally, we present the overall flow maintenance data structure. It is analogous to slack, except during each MOVE operation, there is an additional term of $\alpha\mathbf{W}^{1/2}\boldsymbol{v}$.

Theorem 3.2.8 (Flow maintenance)**.** *Given a modified planar graph G with m edges and its separator tree $\mathcal{T}$ with height η, the randomized data structure MAINTAINFLOW (Algorithm 20) implicitly maintains the flow solution $\boldsymbol{f}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{f}}$, and supports the following procedures with high probability against an adaptive adversary:*

- *INITIALIZE$(G, \boldsymbol{f}^{(\mathrm{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0)$: Given a graph G, initial solution $\boldsymbol{f}^{(\mathrm{init})}$, initial direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$, and target approximation accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time and set the internal representation $\boldsymbol{f} \leftarrow \boldsymbol{f}^{(\mathrm{init})}$ and $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}$.*

- *REWEIGHT$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$ given implicitly as a set of changed weights): Set the current weights to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

- *MOVE($\alpha \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ given implicitly as a set of changed coordinates): Implicitly update $\boldsymbol{f} \leftarrow \boldsymbol{f} + \alpha \mathbf{W}^{1/2}\boldsymbol{v} - \alpha \mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$ for some $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$, where $\|\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\|_2 \leq O(\eta\delta)\,\|\boldsymbol{v}\|_2$ and $\mathbf{B}^\top \mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{B}^\top \mathbf{W}^{1/2}\boldsymbol{v}$. The runtime is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{mK})$, where K is the number of coordinates changed in $\boldsymbol{v}$.*

- *APPROXIMATE() $\rightarrow \mathbb{R}^m$: Output the vector $\overline{\boldsymbol{f}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \leq \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{f}$.*

- *EXACT() $\rightarrow \mathbb{R}^m$: Output the current vector $\boldsymbol{f}$ in $\widetilde{O}(m\delta^{-2})$ time.*

Suppose $\alpha\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to MOVE. Suppose in each step, REWEIGHT, MOVE and APPROXIMATE are called in order. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $(k-1)$-th and k-th REWEIGHT and MOVE calls. Then at the k-th APPROXIMATE call,

- *the data structure first sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last REWEIGHT, then sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k)}$ for $O(N_k \stackrel{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*

- *The amortized time for the k-th APPROXIMATE call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})})$.*

Proof of Theorem 3.2.8. We have the additional invariant that the IPM flow solution $\boldsymbol{f}$ can be recovered in the data structure by the identity

$$\boldsymbol{f} = \widehat{\boldsymbol{f}} - \boldsymbol{f}^\perp, \tag{3.40}$$

where $\boldsymbol{f}^\perp$ is implicit maintained by `maintainRep`, and $\widehat{\boldsymbol{f}}$ is implicitly maintained by the identity $\widehat{\boldsymbol{f}} = \widehat{\boldsymbol{f}}_0 + \widehat{c}\mathbf{W}\boldsymbol{v}$.

We prove the runtime and correctness of each procedure separately. Recall by Lemma 3.7.4, the tree operator $\mathbf{M}$ has complexity $T(K) = O(\delta^{-2}\sqrt{mK})$.

Algorithm 20 Flow Maintenance, Main Algorithm

```
1: data structure MaintainFlow extends MaintainZ
2: private: member
3:     $\boldsymbol{w} \in \mathbb{R}^m$: weight vector        ▷ we use the diagonal matrix $\mathbf{W}$ interchangeably
4:     $\boldsymbol{v} \in \mathbb{R}^m$: direction vector
5:     MaintainRep maintainRep: data structure to implicitly maintain
           $$\boldsymbol{f}^{\perp} \overset{\text{def}}{=} \boldsymbol{y} + \mathbf{W}^{1/2}\mathbf{M}(c\boldsymbol{z}^{(\text{prev})} + \boldsymbol{z}^{(\text{sum})}).$$
                                                         ▷ $\mathbf{M}$ is defined by Definition 3.8.1
6:     $\widehat{c} \in \mathbb{R}, \widehat{\boldsymbol{f}}_0 \in \mathbb{R}^m$: scalar and vector to implicitly maintain
           $$\widehat{\boldsymbol{f}} \overset{\text{def}}{=} \widehat{\boldsymbol{f}}_0 + \widehat{c} \cdot \mathbf{W}\boldsymbol{v}.$$
7:     MaintainApprox bar_f: data structure to maintain approximation $\overline{\boldsymbol{f}}$ to $\boldsymbol{f}$ (Theorem 3.2.6)
8:
9: procedure Initialize($G, \boldsymbol{f}^{(\text{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \delta > 0, \overline{\epsilon} > 0$)
10:     Build the separator tree $\mathcal{T}$ by Theorem 3.4.13
11:     maintainRep.Initialize($G, \mathcal{T}, \mathbf{W}^{1/2}\mathbf{M}, \boldsymbol{v}, \boldsymbol{w}, \mathbf{0}, \delta$)        ▷ initialize $\boldsymbol{f}^{\perp} \leftarrow \mathbf{0}$
12:     $\boldsymbol{w} \leftarrow \boldsymbol{w}, \boldsymbol{v} \leftarrow \boldsymbol{v}$
13:     $\widehat{c} \leftarrow 0, \widehat{\boldsymbol{f}}_0 \leftarrow \boldsymbol{f}^{(\text{init})}$        ▷ initialize $\widehat{\boldsymbol{f}} \leftarrow \boldsymbol{f}^{(\text{init})}$
14:     bar_f.Initialize($-\mathbf{W}^{1/2}\mathbf{M}, c, \boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})}, -\boldsymbol{y} + \widehat{\boldsymbol{f}}_0 + \widehat{c} \cdot \mathbf{W}\boldsymbol{v}, \mathbf{W}^{-1}, n^{-5}, \overline{\epsilon}$)
15:                                                        ▷ initialize $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}^{(\text{init})}$
16: end procedure
17:
18: procedure Reweight($\boldsymbol{w}^{(\text{new})} \in \mathbb{R}^m_{>0}$)
19:     maintainRep.Reweight($\boldsymbol{w}^{(\text{new})}$)
20:     $\Delta\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})} - \boldsymbol{w}$
21:     $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})}$
22:     $\widehat{\boldsymbol{f}}_0 \leftarrow \widehat{\boldsymbol{f}}_0 - \widehat{c}(\Delta\mathbf{W})^{1/2}\boldsymbol{v}$
23: end procedure
24:
25: procedure Move($\alpha, \boldsymbol{v}^{(\text{new})} \in \mathbb{R}^m$)
26:     maintainRep.Move($\alpha, \boldsymbol{v}^{(\text{new})}$)
27:     $\Delta\boldsymbol{v} \leftarrow \boldsymbol{v}^{(\text{new})} - \boldsymbol{v}$
28:     $\boldsymbol{v} \leftarrow \boldsymbol{v}^{(\text{new})}$
29:     $\widehat{\boldsymbol{f}}_0 \leftarrow \widehat{\boldsymbol{f}}_0 - \widehat{c}\mathbf{W}^{1/2}\Delta\boldsymbol{v}$
30:     $\widehat{c} \leftarrow \widehat{c} + \alpha$
31: end procedure
32:
33: procedure Approximate( )
34:                   ▷ the variables in the argument are accessed from maintainRep
35:     return bar_f.Approximate($-\mathbf{W}^{1/2}\mathbf{M}, c, \boldsymbol{z}^{(\text{prev})}, \boldsymbol{z}^{(\text{sum})}, -\boldsymbol{y} + \widehat{\boldsymbol{f}}_0 + \widehat{c} \cdot \mathbf{W}\boldsymbol{v}, \mathbf{W}^{-1}$)
36: end procedure
37:
38: procedure Exact( )
39:     $\boldsymbol{f}^{\perp} \leftarrow$ maintainRep.Exact()
40:     return $(\widehat{\boldsymbol{f}}_0 + \widehat{c} \cdot \mathbf{W}\boldsymbol{v}) - \boldsymbol{f}^{\perp}$
41: end procedure
```

Initialize: By the initialization of `maintainRep` (Theorem 3.2.5), the implicit representation of $\boldsymbol{f}^{\perp}$ in `maintainRep` is correct and $\boldsymbol{f}^{\perp} = \mathbf{0}$. We then set $\widehat{\boldsymbol{f}} \stackrel{\text{def}}{=} \widehat{\boldsymbol{f}}_0 + \widehat{c}\mathbf{W}\boldsymbol{v} = \boldsymbol{f}^{(\text{init})}$. So overall, we have $\boldsymbol{f} \stackrel{\text{def}}{=} \widehat{\boldsymbol{f}} + \boldsymbol{f}^{\perp} = \boldsymbol{f}^{(\text{init})}$. By the initialization of `approx`, $\overline{\boldsymbol{f}}$ is set to $\boldsymbol{f} = \boldsymbol{f}^{(\text{init})}$ to start.

Initialization of `maintainRep` takes $\widetilde{O}(m\delta^{-2})$ time by Theorem 3.2.5, and the initialization of `approx` takes $\widetilde{O}(m)$ time by Theorem 3.2.6.

Reweight: The change to the representation in $\boldsymbol{f}^{\perp}$ is correct via `maintainRep` in exactly the same manner as the proof for the slack solution. For the representation of $\widehat{\boldsymbol{f}}$, the change in value caused by the update to $\boldsymbol{w}$ is subtracted from the $\widehat{\boldsymbol{f}}_0$ term, so that the representation is updated while the overall value remains the same.

Move: This is similar to the proof for the slack solution. `maintainRep`.MOVE$(\alpha, \boldsymbol{v}^{(k)})$ updates the implicit representation of $\boldsymbol{f}^{\perp}$ by

$$\boldsymbol{f}^{\perp} \leftarrow \boldsymbol{f}^{\perp} + \mathbf{W}^{1/2}\mathbf{M}\alpha\boldsymbol{z}^{(k)},$$

where $\mathbf{M}$ is the flow projection tree operator defined in Definition 3.8.1. By Lemma 3.7.2, this is equivalent to the update

$$\boldsymbol{f}^{\perp} \leftarrow \boldsymbol{f}^{\perp} + \alpha\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}},$$

where $\left\|\widetilde{\boldsymbol{f}} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}\right\|_2 \leq O(\eta\delta)\left\|\boldsymbol{v}^{(k)}\right\|_2$ and $\mathbf{B}^{\top}\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}} = \mathbf{B}^{\top}\mathbf{W}^{1/2}\boldsymbol{v}^{(k)}$ by Theorem 3.8.3.

For the $\widehat{\boldsymbol{f}}$ term, let $\widehat{\boldsymbol{f}}_0', \widehat{c}', \boldsymbol{v}'$ be the state of $\widehat{\boldsymbol{f}}_0, \widehat{c}$ and $\boldsymbol{v}$ at the start of the procedure, and similarly let $\widehat{\boldsymbol{f}}'$ be the state of $\widehat{\boldsymbol{f}}$ at the start. At the end of the procedure, we have

$$\widehat{\boldsymbol{f}} \stackrel{\text{def}}{=} \widehat{\boldsymbol{f}}_0 + \widehat{c}\mathbf{W}\boldsymbol{v} = \widehat{\boldsymbol{f}}_0' - \widehat{c}'\mathbf{W}^{1/2}\Delta\boldsymbol{v} + (\widehat{c}' + \alpha)\mathbf{W}\boldsymbol{v} = \widehat{\boldsymbol{f}}_0' + \widehat{c}'\mathbf{W}^{1/2}\boldsymbol{v}' + \alpha\mathbf{W}^{1/2}\boldsymbol{v} = \widehat{\boldsymbol{f}}' + \alpha\mathbf{W}^{1/2}\boldsymbol{v},$$

so we have the correct update $\widehat{\boldsymbol{f}} \leftarrow \widehat{\boldsymbol{f}} + \alpha\mathbf{W}^{1/2}\boldsymbol{v}$. Combined with $\boldsymbol{f}^{\perp}$, the update to $\boldsymbol{f}$ is

$$\boldsymbol{f} \leftarrow \boldsymbol{f} + \alpha\mathbf{W}^{1/2}\boldsymbol{v} - \alpha\mathbf{W}^{1/2}\widetilde{\boldsymbol{f}}.$$

By Theorem 3.2.5, if $\boldsymbol{v}^{(k)}$ differs from $\boldsymbol{v}^{(k-1)}$ on K coordinates, then the runtime of `maintainRep` is

$\widetilde{O}(\delta^{-2}\sqrt{mK})$. Furthermore, $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ change on F_H for at most $\widetilde{O}(K)$ nodes in $\mathcal{T}$. Updating $\widehat{\boldsymbol{f}}$ takes $O(K)$ time where $K \le O(m)$, giving us the overall claimed runtime.

Approximate: By the guarantee of `bar_f`.APPROXIMATE from Theorem 3.2.6, the returned vector satisfies $\|\mathbf{W}^{-1/2}\left(\overline{\boldsymbol{f}} - (\widehat{\boldsymbol{f}} - \boldsymbol{f}^{\perp})\right)\|_\infty \le \overline{\epsilon}$, where $\widehat{\boldsymbol{f}}$ and $\boldsymbol{f}^{\perp}$ are maintained in the current data structure.

Exact: The runtime and correctness follow from the guarantee of `maintainRep`.EXACT given in Theorem 3.2.5 and the invariant that $\boldsymbol{f} = \widehat{\boldsymbol{f}} - \boldsymbol{f}^{\perp}$.

Finally, we have the following lemma about the runtime for APPROXIMATE. Let $\overline{\boldsymbol{f}}^{(k)}$ denote the returned approximate vector at step k.

Lemma 3.8.16. *Suppose $\alpha\|\boldsymbol{v}\|_2 \le \beta$ for some β for all calls to MOVE. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $k-1$-th and k-th REWEIGHT and MOVE calls. Then at the k-th APPROXIMATE call,*

- *The data structure first sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last REWEIGHT, then sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k)}$ for $O(N_k \overset{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*
- *The amortized time for the k-th APPROXIMATE call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\sqrt{m(K + N_{k-2^{\ell_k}})})$.*

Proof. The proof is similar to the one for slack. Since $\overline{\boldsymbol{f}}$ is maintained by `bar_f`, we apply Theorem 3.2.6 with $\boldsymbol{x} = \overline{\boldsymbol{f}}$ and diagonal matrix $\mathbf{D} = \mathbf{W}^{-1}$. We need to prove $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\|_{\mathbf{D}^{(k)}} \le O(\beta)$ for all k first. The constant factor in $O(\beta)$ does not affect the guarantees in Theorem 3.2.6. The left-hand side is

$$\begin{aligned}
\left\|\boldsymbol{f}^{(k)} - \boldsymbol{f}^{(k-1)}\right\|_{\mathbf{W}^{(k)-1}} &= \left\|-\alpha^{(k)}\mathbf{M}\boldsymbol{z}^{(k)} + \alpha^{(k)}\boldsymbol{v}^{(k)}\right\|_2 && \text{(by MOVE)}\\
&\le \left\|-\alpha^{(k)}\mathbf{M}\boldsymbol{z}^{(k)}\right\|_2 + \left\|\alpha^{(k)}\boldsymbol{v}^{(k)}\right\|_2 \\
&\le (2 + O(\eta\delta))\alpha^{(k)}\|\boldsymbol{v}^{(k)}\|_2 && \text{(by the assumption that } \alpha\|\boldsymbol{v}\|_2 \le \beta)\\
&\le 3\beta.
\end{aligned}$$

Now, we can apply the conclusions from Theorem 3.2.6 to get that at the k-th step, the data structure first sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last Reweight, then sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k)}$ for $O(N_k \stackrel{\text{def}}{=} 2^{2\ell_k}(\frac{\beta}{\epsilon})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.

For the second point, Move updates $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ on F_H for $\widetilde{O}(K)$ different nodes $H \in \mathcal{T}$ by Theorem 3.2.5. Reweight then updates $\boldsymbol{z}^{(\text{prev})}$ and $\boldsymbol{z}^{(\text{sum})}$ on F_H for $\widetilde{O}(K)$ different nodes, and updates the tree operator $\mathbf{W}^{-1/2}\mathbf{M}$ on $\widetilde{O}(K)$ different edge and leaf operators. In turn, it updates $\boldsymbol{y}$ on $E(H)$ for $\widetilde{O}(K)$ leaf nodes H. The changes of $\widehat{\boldsymbol{f}}$ cause $O(K)$ changes to the vector $-\boldsymbol{y} + \widehat{\boldsymbol{f}}_0 + \widehat{c} \cdot \mathbf{W}\boldsymbol{v}$, which is the parameter $\boldsymbol{y}$ of Theorem 3.2.6. Now, we apply Theorem 3.2.6 and the complexity of the tree operator to conclude the desired amortized runtime. □

□

3.9 Min-cost flow for separable graphs

In this section, we extend our result to α-separable graphs.

Corollary 3.1.2 (Separable min-cost flow)**.** *Let $\mathcal{C}$ be an 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 convex function s. Given a graph $G \in \mathcal{C}$ with n vertices and m edges, 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 $\widetilde{O}((m + m^{1/2+\alpha}) \log M + s(m))$ expected time.*

The change in running time essentially comes from the parameters of the separator tree which we shall discuss in Section 3.9.1. We then calculate the total running time and prove Corollary 3.1.2 in Section 3.9.2.

3.9.1 Separator tree for separable graphs

Since our algorithm only exploits the separable property of the planar graphs, it can be applied to other separable graphs directly and yields different running times. Similar to the planar case, by adding two extra vertices to any α-separable graph, it is still α-separable with the constant c in Definition 3.4.7 increased by 2.

Recall the definition of separable graphs:

Definition 3.4.7 (Separable graph)*.* A graph $G = (V, E)$ is α-separable if there exists two constants $c > 0$ and $b \in (0, 1)$ such that every nonempty subgraph $H = (V(H) \subseteq V, E(H) \subseteq E)$ with $|E(H)| \geq 2$ of G can be partitioned into H_1 and H_2 such that

- $E(H_1) \cup E(H_2) = E(H)$, $E(H_1) \cap E(H_2) = \emptyset$,
- $|V(H_1) \cap V(H_2)| \leq c\lceil |E(H)|^{\alpha} \rceil$,
- $|E(H_i)| \leq b|E(H)|$, for $i = 1, 2$.

We call $S(H) \overset{\text{def}}{=} V(H_1) \cap V(H_2)$ the *balanced vertex separator* of H.

We define a separator tree $\mathcal{T}$ for an α-separable graph G in the same way as for a planar graph.

Definition 3.9.1 (Separator tree $\mathcal{T}$ for α-separable graph)*.* Let G be an α-separable graph. A separator tree $\mathcal{T}$ is a binary tree whose nodes represent subgraphs of G such that the children of each node H form a balanced partition of H.

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 $\partial(H)$ 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 $\partial(D_2) = (\partial(H) \cup S(H)) \cap V(D_2)$. Define $F_H = S(H) \setminus \partial(H)$.

3. If a region H contains a constant number of edges, 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.

Let $\eta(H)$ denote the height of node H which is defined as the maximum number of edges on a tree path from H to one of its descendants. $\eta(H) = 0$ if H is a leaf. Note that the height difference between a parent and child node could be greater than one. Let η denote the height of $\mathcal{T}$ which is defined as the maximum height of nodes in $\mathcal{T}$. We say H is at *level i* if $\eta(H) = i$.

The only two differences between the separator trees for planar and α-separable graphs are their construction time and update time (for k-sparse updates). For the planar case, these are bounded by Theorem 3.4.13 and Lemma 3.4.16 respectively. We shall prove their analogs Lemma 3.9.2 and Lemma 3.9.3.

[162] showed that the separator tree can be constructed in $O(s(n)\log n)$ time for any class of 1/2-separable graphs where $s(n)$ is the time for computing the separator. The proof can be naturally extended to α-separable graphs. We include the extended proofs in Section 3.10 for completeness.

Lemma 3.9.2. *Let $\mathcal{C}$ be an α-separable class such that we can compute a balanced separator for any graph in $\mathcal{C}$ with n vertices and m edges in $s(m)$ time for some convex function $s(m) \geq m$. Given an α-separable graph, there is an algorithm that computes a separator tree $\mathcal{T}$ in $O(s(m)\log m)$ time.*

Note that $s(\cdot)$ does not depend on n because we may assume the graph is connected so that $n = O(m)$.

We then prove the update time. Same as the planar case, we define $\mathcal{P}_{\mathcal{T}}(H)$ to be the set of all ancestors of H in the separator tree and $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ to be the union of $\mathcal{P}_{\mathcal{T}}(H)$ for all $H \in \mathcal{H}$. Then we have the following bound:

Lemma 3.9.3. *Let G be an α-separable graph with separator tree $\mathcal{T}$. Let $\mathcal{H}$ be a set of K nodes in $\mathcal{T}$. Then*

$$\sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})} |\partial(H)| + |S(H)| \leq \widetilde{O}(K^{1-\alpha} m^{\alpha}).$$

By setting α as $1/2$, we get Lemma 3.4.16 for planar graphs as a corollary.

3.9.2 Proof of running time

In this section, we prove Corollary 3.1.2. The data structures (except for the construction of the separator tree) will use exactly the same pseudocode as for the planar case. Thus, the correctness can be proven in the same way. We prove the runtimes only.

For the planar case, after constructing the separator tree by Theorem 3.4.13, Lemma 3.4.16 is the lemma that interacts with other parts of the algorithm. For α-separable graphs, we first construct the separator tree in $O(s(m)\log m)$ time by Lemma 3.9.2. Then we propagate the change in runtime ($\widetilde{O}(\sqrt{mK})$ from Lemma 3.4.16 to $\widetilde{O}(m^\alpha K^{1-\alpha})$ from Lemma 3.9.3) to all the data structures and to the complexity $T(\cdot)$ of the flow and slack tree operators.

We first propagate the change to the implicit representation maintenance data structure, which is the common component for maintaining the flow and the slack vectors.

Theorem 3.9.4. *Given an α-separable graph G with n vertices and m edges, and its separator tree $\mathcal{T}$ with height η, the deterministic data structure* MaintainRep *(Algorithm 16) maintains the following variables correctly at the end of every IPM step:*

- *the dynamic edge weights $\boldsymbol{w}$ and step direction $\boldsymbol{v}$ from the current IPM step,*
- *a* DynamicSC *data structure on $\mathcal{T}$ based on the current edge weights $\boldsymbol{w}$,*
- *an implicitly represented tree operator $\mathbf{M}$ supported on $\mathcal{T}$ with complexity $T(K)$,* computable using information from DynamicSC*,*
- *scalar c and vectors $\boldsymbol{z}^{(\mathrm{prev})}, \boldsymbol{z}^{(\mathrm{sum})}$, which together represent $\boldsymbol{z} = c\boldsymbol{z}^{(\mathrm{prev})} + \boldsymbol{z}^{(\mathrm{sum})}$, such that at the end of step k,*
$$\boldsymbol{z} = \sum_{i=1}^{k} \alpha^{(i)} \boldsymbol{z}^{(i)},$$
where $\alpha^{(i)}$ is the step size α given in Move *for step i,*
- *$\boldsymbol{z}^{(\mathrm{prev})}$ satisfies $\boldsymbol{z}^{(\mathrm{prev})} = \widetilde{\mathbf{\Gamma}}\mathbf{\Pi}^{(\eta-1)} \cdots \mathbf{\Pi}^{(0)} \mathbf{B}^\top \mathbf{W}^{1/2} \boldsymbol{v}$,*

- *an offset vector $\boldsymbol{y}$ which together with $\mathbf{M}, \boldsymbol{z}$ represent $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$, such that after step k,*

$$\boldsymbol{x} = \boldsymbol{x}^{(\text{init})} + \sum_{i=1}^{k} \mathbf{M}^{(i)}(\alpha^{(i)}\boldsymbol{z}^{(i)}),$$

  where $\boldsymbol{x}^{(\text{init})}$ is an initial value from Initialize*, and $\mathbf{M}^{(i)}$ is the state of $\mathbf{M}$ after step i.*

The data structure supports the following procedures:

- Initialize*$(G, \mathcal{T}, \mathbf{M}, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \boldsymbol{x}^{(\text{init})} \in \mathbb{R}^m, \epsilon_{\mathbf{P}} > 0)$: Given a graph G, its separator tree $\mathcal{T}$, a tree operator $\mathbf{M}$ supported on $\mathcal{T}$ with complexity T, initial step direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, initial vector $\boldsymbol{x}^{(\text{init})}$, and target projection matrix accuracy $\epsilon_{\mathbf{P}}$, preprocess in $\widetilde{O}(\delta^{-2}m + T(m))$ time and set $\boldsymbol{x} \leftarrow \boldsymbol{x}^{(\text{init})}$.*

- Reweight*$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$ given implicitly as a set of changed coordinates): Update the weights to $\boldsymbol{w}^{(\text{new})}$. Update the implicit representation of $\boldsymbol{x}$ without changing its value, so that all the variables in the data structure are based on the new weights.*

  The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}K^{1-\alpha}m^{\alpha} + T(K))$ total time, where K is an upper bound on the number of coordinates changed in $\boldsymbol{w}$ and the number of leaf or edge operators changed in $\mathbf{M}$. There are most $\widetilde{O}(K)$ nodes $H \in \mathcal{T}$ for which $\boldsymbol{z}^{(\text{prev})}|_{F_H}$ and $\boldsymbol{z}^{(\text{sum})}|_{F_H}$ are updated.

- Move*$(\alpha \in \mathbb{R}$, $\boldsymbol{v} \in \mathbb{R}^n$ given implicitly as a set of changed coordinates): Update the current direction to $\boldsymbol{v}$, and then $\boldsymbol{z}^{(\text{prev})}$ to maintain the claimed invariant. Update the implicit representation of $\boldsymbol{x}$ to reflect the following change in value:*

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathbf{M}(\alpha \boldsymbol{z}^{(\text{prev})}).$$

  The procedure runs in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}K^{1-\alpha}m^{\alpha})$ time, where K is the number of coordinates changed in $\boldsymbol{v}$ compared to the previous IPM step.

- Exact*(): Output the current exact value of $\boldsymbol{x} = \boldsymbol{y} + \mathbf{M}\boldsymbol{z}$ in $\widetilde{O}(T(m))$ time.*

Proof. The bottlenecks of Move is PartialProject. For each $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$, recall from Theorem 3.2.4 that $\mathbf{L}^{(H)}$ is supported on the vertex set $F_H \cup \partial(H)$ and has $\widetilde{O}(\delta^{-2}|F_H \cup \partial(H)|)$ edges.

Hence, $(\mathbf{L}^{(H)}_{F_H,F_H})^{-1}\boldsymbol{u}|_{F_H}$ can be computed by an exact Laplacian solver in $\widetilde{O}(\delta^{-2}|F_H \cup \partial(H)|)$ time, and the subsequent left-multiplying by $\mathbf{L}^{(H)}_{\partial(H),F_H}$ also takes $\widetilde{O}(\delta^{-2}|F_H \cup \partial(H)|)$ time. By Lemma 3.9.3, PartialProject takes $\widetilde{O}(\delta^{-2}K^{1-\alpha}m^{\alpha})$ time. Move also runs in $\widetilde{O}(\delta^{-2}K^{1-\alpha}m^{\alpha})$ time.

Reweight calls PartialProject and ReversePartialProject for $O(1)$ times and ComputeMz once. ReversePartialProject costs the same as PartialProject. The runtime of ComputeMz is still bounded by the complexity of the tree operator, $O(T(K))$. Thus, PartialProject takes $\widetilde{O}(\delta^{-2}K^{1-\alpha}m^{\alpha})$ time. Move also runs in $\widetilde{O}(\delta^{-2}K^{1-\alpha}m^{\alpha} + T(K))$ time.

Runtimes of other procedures and correctness follow from the same argument as in the proof for Theorem 3.2.5. □

Then we may use Theorem 3.9.4 and Theorem 3.2.6 to maintain vectors $\overline{\boldsymbol{f}}, \overline{\boldsymbol{s}}$, with the updated complexity of the operators.

Lemma 3.9.5. *For any α-separable graph G with separator tree $\mathcal{T}$, the flow and slack operators defined in Definitions 3.8.1 and 3.7.1 both have complexity $T(K) = O(\delta^{-2}K^{1-\alpha}m^{\alpha})$.*

Proof. The leaf operators of both the flow and slack tree operators has constant size. Let $\mathbf{M}_{(H,P)}$ be a tree edge operator. Note that it is a symmetric matrix. For the slack operator, Applying $\mathbf{M}_{(D,P)} = \mathbf{I}_{\partial(D)} - \left(\mathbf{L}^{(D)}_{F_D,F_D}\right)^{-1}\mathbf{L}^{(D)}_{F_D,\partial(D)}$ to the left or right consists of three steps which are applying $\mathbf{I}_{\partial(D)}$, applying $\mathbf{L}^{(D)}_{F_D,\partial(D)}$ and solving for $\mathbf{L}^{(D)}_{F_D,F_D}\boldsymbol{v} = \boldsymbol{b}$ for some vectors $\boldsymbol{v}$ and $\boldsymbol{b}$. For the flow operator, $\mathbf{M}_{(H,P)}\boldsymbol{u}$ consists of multiplying with $\widetilde{\mathbf{Sc}}(\mathbf{L}^{(H)}, \partial H)$ and solving the Laplacian system $\mathbf{L}^{(H)}$.

Each of the steps costs time $O(\delta^{-2}|\partial(D) \cup F_D|)$ by Lemma 3.4.22 and Theorem 3.3.1. To bound the total cost over K distinct edges, we apply Lemma 3.9.3 instead of Lemma 3.4.16, which gives the claimed complexity. □

We then have the following lemmas for maintaining the flow and slack vectors:

Theorem 3.9.6 (Slack maintenance for α-separable graphs)**.** *Given a modified planar graph G with n vertices and m edges, and its separator tree $\mathcal{T}$ with height η, the randomized data structure MaintainSlack (Algorithm 19) implicitly maintains the slack solution $\boldsymbol{s}$ undergoing IPM* *changes, and explicitly maintains its approximation $\overline{\boldsymbol{s}}$, and supports the following procedures with high probability against an adaptive adversary:*

- *Initialize*$(G, \boldsymbol{s}^{(\mathrm{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0)$*: Given a graph G, initial solution $\boldsymbol{s}^{(\mathrm{init})}$, initial direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$ and target approximation accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time, and set the representations $\boldsymbol{s} \leftarrow \boldsymbol{s}^{(\mathrm{init})}$ and $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{s}$.*

- *Reweight*$(\boldsymbol{w} \in \mathbb{R}^m_{>0}$*, given implicitly as a set of changed weights): Set the current weights to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}K^{1-\alpha}m^{\alpha})$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

- *Move*$(t \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ *given implicitly as a set of changed coordinates): Implicitly update $\boldsymbol{s} \leftarrow \boldsymbol{s} + t\mathbf{W}^{-1/2}\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v}$ for some $\widetilde{\mathbf{P}}_{\boldsymbol{w}}$ with $\|(\widetilde{\mathbf{P}}_{\boldsymbol{w}} - \mathbf{P}_{\boldsymbol{w}})\boldsymbol{v}\|_2 \leq \eta\delta \|\boldsymbol{v}\|_2$, and $\widetilde{\mathbf{P}}_{\boldsymbol{w}}\boldsymbol{v} \in \mathrm{Range}(\mathbf{B})$. The total runtime is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}K^{1-\alpha}m^{\alpha})$ where K is the number of coordinates changed in $\boldsymbol{v}$.*

- *Approximate*$() \to \mathbb{R}^m$*: Return the vector $\overline{\boldsymbol{s}}$ such that $\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_{\infty} \leq \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{s}$.*

- *Exact*$() \to \mathbb{R}^m$*: Output the current vector $\boldsymbol{s}$ in $\widetilde{O}(m\delta^{-2})$ time.*

Suppose $t\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to Move. Suppose in each step, Reweight, Move and Approximate are called in order. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $(k-1)$-th and k-th Reweight and Move calls. Then at the k-th Approximate call,

- *the data structure first sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last Reweight, then sets $\overline{\boldsymbol{s}}_e \leftarrow \boldsymbol{s}_e^{(k)}$ for $O(N_k \stackrel{\mathrm{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2 \log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^{\ell}$ when $k \neq 0$ and $\ell_0 = 0$.*

- *The amortized time for the k-th Approximate call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}(m^{\alpha}(K + N_{k-2^{\ell_k}})^{1-\alpha}))$.*

Proof. Because $T(m) = \widetilde{O}(\delta^{-2}m)$ (Lemma 3.9.5), the runtime of Initialize is still $\widetilde{O}(\delta^{-2}m)$ by Theorem 3.9.4 and Theorem 3.2.6. The runtime of Reweight, Move, and Exact follow from the guarantees of Theorem 3.9.4. The runtime of Approximate follows from Theorem 3.2.6 with $T(K) = O(K^{1-\alpha}m^{\alpha})$ (Lemma 3.9.5). □

Theorem 3.9.7 (Flow maintenance for α-separable graphs)**.** *Given a α-separable graph G with n vertices and m edges, and its separator tree $\mathcal{T}$ with height η, the randomized data structure* MAINTAINFLOW *(Algorithm 20) implicitly maintains the flow solution $\boldsymbol{f}$ undergoing IPM changes, and explicitly maintains its approximation $\overline{\boldsymbol{f}}$, and supports the following procedures with high probability against an adaptive adversary:*

- INITIALIZE*($G, \boldsymbol{f}^{(\mathrm{init})} \in \mathbb{R}^m, \boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^m_{>0}, \epsilon_{\mathbf{P}} > 0, \overline{\epsilon} > 0$): Given a graph G, initial solution $\boldsymbol{f}^{(\mathrm{init})}$, initial direction $\boldsymbol{v}$, initial weights $\boldsymbol{w}$, target step accuracy $\epsilon_{\mathbf{P}}$, and target approximation accuracy $\overline{\epsilon}$, preprocess in $\widetilde{O}(m\epsilon_{\mathbf{P}}^{-2})$ time and set the internal representation $\boldsymbol{f} \leftarrow \boldsymbol{f}^{(\mathrm{init})}$ and $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}$.*

- REWEIGHT*($\boldsymbol{w} \in \mathbb{R}^m_{>0}$ given implicitly as a set of changed weights): Set the current weights to $\boldsymbol{w}$ in $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}\alpha)$ time, where K is the number of coordinates changed in $\boldsymbol{w}$.*

- MOVE*($t \in \mathbb{R}, \boldsymbol{v} \in \mathbb{R}^m$ given implicitly as a set of changed coordinates): Implicitly update $\boldsymbol{f} \leftarrow \boldsymbol{f} + t\mathbf{W}^{1/2}\boldsymbol{v} - t\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$ for some $\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v}$, where $\|\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}\|_2 \leq O(\eta\delta)\,\|\boldsymbol{v}\|_2$ and $\mathbf{B}^\top\mathbf{W}^{1/2}\widetilde{\mathbf{P}}'_{\boldsymbol{w}}\boldsymbol{v} = \mathbf{B}^\top\mathbf{W}^{1/2}\boldsymbol{v}$. The runtime is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}K^{1-\alpha}m^{\alpha})$, where K is the number of coordinates changed in $\boldsymbol{v}$.*

- APPROXIMATE*() $\to \mathbb{R}^m$: Output the vector $\overline{\boldsymbol{f}}$ such that $\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{f}} - \boldsymbol{f})\|_\infty \leq \overline{\epsilon}$ for the current weight $\boldsymbol{w}$ and the current vector $\boldsymbol{f}$.*

- EXACT*() $\to \mathbb{R}^m$: Output the current vector $\boldsymbol{f}$ in $\widetilde{O}(m\delta^{-2})$ time.*

Suppose $t\|\boldsymbol{v}\|_2 \leq \beta$ for some β for all calls to MOVE*. Suppose in each step,* REWEIGHT*,* MOVE *and* APPROXIMATE *are called in order. Let K denote the total number of coordinates changed in $\boldsymbol{v}$ and $\boldsymbol{w}$ between the $(k-1)$-th and k-th* REWEIGHT *and* MOVE *calls. Then at the k-th* APPROXIMATE *call,*

- *the data structure first sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k-1)}$ for all coordinates e where $\boldsymbol{w}_e$ changed in the last* REWEIGHT*, then sets $\overline{\boldsymbol{f}}_e \leftarrow \boldsymbol{f}_e^{(k)}$ for $O(N_k \stackrel{\mathrm{def}}{=} 2^{2\ell_k}(\frac{\beta}{\overline{\epsilon}})^2\log^2 m)$ coordinates e, where ℓ_k is the largest integer ℓ with $k = 0 \mod 2^\ell$ when $k \neq 0$ and $\ell_0 = 0$.*

- *The amortized time for the k-th* APPROXIMATE *call is $\widetilde{O}(\epsilon_{\mathbf{P}}^{-2}(m^\alpha(K + N_{k-2^{\ell_k}})^{1-\alpha}))$.*

The proof is the same as Theorem 3.9.6.

Finally, we can prove Corollary 3.1.2.

Proof of Corollary 3.1.2. The correctness is exactly the same as the proof for Theorem 3.1.1.

For the runtime, we use the data structure runtimes given in Theorem 3.9.6 and Theorem 3.9.7. We may assume $\alpha > 1/2$ because otherwise the graph is 1/2-separable and the runtime follows from Theorem 3.1.1. The amortized time for the k-th IPM step is

$$\widetilde{O}(\delta^{-2} m^{\alpha} (K + N_{k-2^{\ell_k}})^{1-\alpha}).$$

where $N_k \overset{\text{def}}{=} 2^{2\ell_k}(\beta/\alpha)^2 \log^2 m = O(2^{2\ell_k} \log^2 m)$, where $\alpha = O(1/\log m)$ and $\epsilon_{\mathbf{P}} = O(1/\log m)$ are defined in CENTERINGIMPL.

Observe that $K + N_{k-2^{\ell_k}} = O(N_{k-2^{\ell_k}})$. Now, summing over all T steps, the total time is

$$\begin{aligned} O(m^{\alpha} \log m) \sum_{k=1}^{T} (N_{k-2^{\ell_k}})^{1-\alpha} &= O(m^{\alpha} \log^2 m) \sum_{k=1}^{T} 2^{2(1-\alpha)\ell_{(k-2^{\ell_k})}} \\ &= O(m^{\alpha} \log^2 m) \sum_{k'=1}^{T} 2^{2(1-\alpha)\ell_{k'}} \sum_{k=1}^{T} [k - 2^{\ell_k} = k'], \\ &= O(m^{\alpha} \log^2 m \log T) \sum_{k'=1}^{T} 2^{2(1-\alpha)\ell_{k'}}. \end{aligned} \tag{3.41}$$

Without $1-\alpha$ in the exponent, recall from the planar case that

$$\sum_{k'=1}^{T} 2^{\ell_{k'}} = \sum_{i=0}^{\log T} 2^i \cdot T/2^{i+1} = O(T \log T).$$

The summation from (3.41) is

$$\begin{aligned}
\sum_{k=1}^{T} 2^{2(1-\alpha)\ell_k} &= \sum_{k=1}^{T} (2^{\ell_k})^{2-2\alpha} \\
&\leq \left(\sum_{k=1}^{T} 1^{1/(2\alpha-1)}\right)^{2\alpha-1} \left(\sum_{k=1}^{T} \left(\left(2^{\ell_k}\right)^{2-2\alpha}\right)^{1/(2-2\alpha)}\right)^{2-2\alpha} && \text{(by Hölder's Inquality)} \\
&= \widetilde{O}\left(T^{2\alpha-1}(T\log T)^{2-2\alpha}\right) \\
&= \widetilde{O}(\sqrt{m}\log M \log T),
\end{aligned}$$

where we use $T = O(\sqrt{m}\log n \log(nM))$ from Theorem 3.2.1. So the runtime for CenteringImpl is $\widetilde{O}(m^{1/2+\alpha}\log M)$. By Lemma 3.9.2, the overall runtime is $\widetilde{O}(m^{1/2+\alpha}\log M + s(m))$. □

3.10 Omitted Proofs

Lemma 3.4.16. *Let G be a modified planar graph with separator tree $\mathcal{T}$. Let $\mathcal{H}$ be a set of K nodes in $\mathcal{T}$. Then*

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

Proof. Note that F_H is always a subset of $S(H)$. We will instead prove

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

First, we decompose the quantity we want to bound by levels in $\mathcal{T}$:

$$\sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})} |\partial(H)| + |S(H)| = \sum_{i=0}^{\eta} \sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} |\partial(H)| + |S(H)|. \tag{3.42}$$

We first bound $\sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} |\partial(H)| + |S(H)|$ for a fixed i. Our main observation is that we can bound the total number of boundary vertices of nodes at level i by the number of boundary and separator vertices of nodes at level $(i+1)$. Formally, our key claim is the following

$$\sum_{H\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} |\partial(H)| \le \sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},i+1)} \left(|\partial(H')| + 2|S(H')|\right). \tag{3.43}$$

Without loss of generality, we may assume that if node H is included in the left hand sum, then its sibling is included as well. Next, recall by the definition of $\mathcal{T}$, for siblings H_1, H_2 with parent H', their boundaries are defined as

$$\partial(H)_i = (S(H') \cup \partial(H')) \cap V(H_i) = (S(H') \cap V(H_i)) \cup ((\partial H' \setminus S(H')) \cap V(H_i)),$$

for $i = 1, 2$. Furthermore, $V(H_1) \cup V(H_2) = V(H)$. Another crucial observation is that a vertex from $\partial(H)'$ exists in both H_1 and H_2 if and only if that vertex belongs to the separator $S(H')$.

$$\begin{aligned}|\partial(H_1)| + |\partial(H_2)| &\le |S(H')| + |(\partial(H') \setminus S(H')) \cap V(H_1)| + |S(H')| + |(\partial(H') \setminus S(H')) \cap V(H_2')| \\ &\le |\partial(H')| + 2|S(H')|. \end{aligned} \tag{3.44}$$

By summing (3.44) over all pairs of siblings in $\mathcal{P}_{\mathcal{T}}(\mathcal{H}, i)$, we get (3.43). By repeatedly applying (3.43) until we reach the root at height η, we have

$$\sum_{H\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} |\partial(H)| \le 2\sum_{j=i+1}^{\eta} \sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)} |S(H')|. \tag{3.45}$$

Summing over all the levels in $\mathcal{T}$, we have

$$\begin{aligned}\sum_{i=0}^{\eta} \sum_{H\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},i)} (|\partial(H)| + |S(H)|) &\le 2\sum_{j=0}^{\eta}(j+1) \sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)} |S(H')| && \text{(by (3.45))} \\ &\le 2c\sum_{j=0}^{\eta}(j+1) \sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)} \sqrt{|E(H')|}, && (3.46)\end{aligned}$$

where c is the constant such that $|S(H')| \le c\,(|E(H')|)^{1/2}$ in the definition of being 1/2-separable. Furthermore, the set of ancestors of $\mathcal{H}$ at level j has size $|\mathcal{P}_{\mathcal{T}}(\mathcal{H}, j)| \le |\mathcal{H}| = K$. Applying the

Cauchy-Schwarz inequality, we get that

$$\begin{aligned}
\sum_{H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})} (|\partial(H)| + |S(H)|) &\leq 2c \sum_{j=0}^{\eta} (j+1) \sqrt{|\mathcal{P}_{\mathcal{T}}(\mathcal{H}, j)|} \cdot \left(\sum_{H' \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, j)} |E(H')| \right)^{1/2} \\
&\leq 2c \sum_{j=0}^{\eta} (j+1) \sqrt{K} \cdot \left(\sum_{H' \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, j)} |E(H')| \right)^{1/2} \\
&\leq 2c\eta\sqrt{K} \sum_{j=0}^{\eta} \left(\sum_{H' \in \mathcal{P}_{\mathcal{T}}(\mathcal{H}, j)} |E(H')| \right)^{1/2} \\
&\leq O(\eta^2 \sqrt{mK}),
\end{aligned}$$

where the final inequality follows from the fact that nodes at the same level form an edge partition of G. As $\eta = O(\log m)$, the lemma follows. □

Lemma 3.9.2. *Let $\mathcal{C}$ be an α-separable class such that we can compute a balanced separator for any graph in $\mathcal{C}$ with n vertices and m edges in $s(m)$ time for some convex function $s(m) \geq m$. Given an α-separable graph, there is an algorithm that computes a separator tree $\mathcal{T}$ in $O(s(m) \log m)$ time.*

Proof. First, we let G be the root node of $\mathcal{T}(G)$. Let G_1 and G_2 be the two disjoint components of G obtained after the removal of the vertices in $S(G)$. We define the children $\mathrm{c}_1(G), \mathrm{c}_2(G)$ of G as follows: $V(\mathrm{c}_i(G)) = V(G_i) \cup S(G)$ and $E(\mathrm{c}_i(G)) = E(G_i)$, for $i = 1, 2$. Edges connecting some vertex in G_i and another vertex in $S(G)$ are added to $E(\mathrm{c}_i(G))$. For each edge connecting two vertices in $S(G)$, we append it to $E(\mathrm{c}_1(G))$ or $E(\mathrm{c}_2(G))$, whichever has less edges. By construction, property 2 in the definition of $\mathcal{T}(G)$ holds. We continue by repeatedly splitting each child $\mathrm{c}_i(G)$ that has at least one edge in the same way as we did for G, whenever possible. There are $O(m)$ components, each containing exactly 1 edge. The components containing exactly 1 edge form the *leaf nodes* of $\mathcal{T}(G)$. Note that the height of $\mathcal{T}(G)$ is bounded by $O(\log m) = O(\log m)$ as for any child H' of a node H, $|E(H')| \leq b|E(H)|$.

The running time of the algorithm is bounded by the total time to construct the separator for all nodes in the tree. Because the tree has height $O(\log m)$ and nodes with the same depth does not share any edge, the sum of edges over all tree nodes is $O(m \log m)$. Since $s(m)$ is convex, the algorithm runs in no more than $O(s(m)\log m)$ time.

$\square$

Lemma 3.9.3. *Let G be an α-separable graph with separator tree $\mathcal{T}$. Let $\mathcal{H}$ be a set of K nodes in $\mathcal{T}$. Then*

$$\sum_{H\in\mathcal{P}_{\mathcal{T}}(\mathcal{H})} |\partial(H)| + |S(H)| \leq \widetilde{O}(K^{1-\alpha}m^{\alpha}).$$

Proof. Using the separator tree, we have (3.46) in exactly the same way as for the planar case.

$$\sum_{H\in\mathcal{P}_{\mathcal{T}}(\mathcal{H})} (|\partial(H)| + |S(H)|) \leq 2c\sum_{j=0}^{\eta}(j+1)\sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)}\sqrt{|E(H')|}$$

Applying Hölder's Inequality instead of Cauchy-Schwarz for the planar case, we get

$$\begin{aligned}
&\leq 2c\sum_{j=0}^{\eta}(j+1)|\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)|^{1-\alpha}\cdot\left(\sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)}|E(H')|\right)^{\alpha}\\
&\leq 2c\sum_{j=0}^{\eta}(j+1)K^{1-\alpha}\cdot\left(\sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)}|E(H')|\right)^{\alpha}\\
&\leq 2c\eta K^{1-\alpha}\sum_{j=0}^{\eta}\left(\sum_{H'\in\mathcal{P}_{\mathcal{T}}(\mathcal{H},j)}|E(H')|\right)^{\alpha}\\
&\leq O(\eta^2K^{1-\alpha}m^{\alpha}),
\end{aligned}$$

where the final inequality follows from the fact that nodes at the same level form an edge partition of G. As $\eta = O(\log m)$, the lemma follows. $\square$

REFERENCES

[1] P. G. Doyle and J. L. Snell, *Random Walks and Electric Networks* (Carus Mathematical Monographs). Mathematical Association of America, 1984, vol. 22, Available at https://arxiv.org/abs/math/0001057.

[2] P. Bak, C. Tang, and K. Wiesenfeld, “Self-organized criticality: An explanation of the $1/f$ noise,” *Physical Review Letters*, vol. 59, no. 4, p. 381, 1987.

[3] X. Zhu, Z. Ghahramani, and J. D. Lafferty, “Semi-supervised learning using gaussian fields and harmonic functions,” in *ICML*, 2003.

[4] M. Belkin and P. Niyogi, “Semi-supervised learning on riemannian manifolds,” *Machine Learning*, vol. 56, pp. 209–239, 2004.

[5] A. Pothen, H. D. Simon, and K.-P. Liou, “Partitioning sparse matrices with eigenvectors of graphs,” *SIAM journal on matrix analysis and applications*, vol. 11, no. 3, pp. 430–452, 1990.

[6] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’14, New York, New York, USA: Association for Computing Machinery, 2014, pp. 701–710, ISBN: 9781450329569.

[7] J. Qiu *et al.*, “Netsmf: Large-scale network embedding as sparse matrix factorization,” in *The World Wide Web Conference*, ser. WWW ’19, San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 1509–1520, ISBN: 9781450366748.

[8] A. Madry, “Computing maximum flow with augmenting electrical flows,” in *57th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, Available at https://arxiv.org/abs/1608.06016, IEEE Computer Society, 2016, pp. 593–602.

[9] S. I. Daitch and D. A. Spielman, “Faster approximate lossy generalized flow via interior point algorithms,” in *Proceedings of the 40th annual ACM Symposium on Theory of Computing, STOC 2008, Victoria, BC, Canada, May 17-20, 2008*, Available at http://arxiv.org/abs/0803.0988, New York, NY, USA: ACM, 2008, pp. 451–460, ISBN: 978-1-60558-047-0.

[10] A. Choure and S. Vishwanathan, “Random walks, electric networks and the transience class problem of sandpiles,” in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Society of Industrial and Applied Mathematics, 2012, pp. 1593–1611.

[11] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, "Spectral sparsification of graphs: Theory and algorithms," *Communications of the ACM*, vol. 56, no. 8, pp. 87–94, Aug. 2013.

[12] A. A. Benczúr and D. R. Karger, "Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 47–55, ISBN: 0897917855.

[13] D. A. Spielman and S. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC 2004, Chicago, IL, USA, June 13-16, 2004*, Available at https://arxiv.org/abs/0809.3232, https://arxiv.org/abs/0808.4134, https://arxiv.org/abs/cs/0607105, 2004, pp. 81–90.

[14] A. Jambulapati and A. Sidford, "Ultrasparse ultrasparsifiers and faster laplacian system solvers," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2021, pp. 540–559.

[15] D. Durfee, Y. Gao, G. Goranci, and R. Peng, "Fully dynamic spectral vertex sparsifiers and applications," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, Available at https://arxiv.org/abs/1906.10530, ACM, 2019, pp. 914–925.

[16] S. Dong *et al.*, "Nested dissection meets ipms: Planar min-cost flow in nearly linear time," 2021, To appear in SODA 2022.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009, ISBN: 978-0-262-03384-8.

[18] A. V. Goldberg and R. E. Tarjan, "Efficient maximum flow algorithms," *Communications of the ACM*, vol. 57, no. 8, pp. 82–89, 2014, Available at https://cacm.acm.org/magazines/2014/8/177011-efficient-maximum-flow-algorithms.

[19] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S. Teng, "Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs," in *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, June 6-8 2011*, Available at https://arxiv.org/abs/1010.2921, ACM, 2011, pp. 273–282.

[20] J. Sherman, "Nearly maximum flows in nearly linear time," in *54th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2013, Berkeley, CA, USA, October 26-29, 2013*, Available at https://arxiv.org/abs/1304.2077, 2013, pp. 263–269.

[21] A. Madry, "Navigating central path with electrical flows: From flows to matchings, and back," in *54th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2013,*

Berkeley, CA, USA, October 26-29, 2013, Available at https://arxiv.org/abs/1307.2205, IEEE Computer Society, 2013, pp. 253–262.

[22] T. Kathuria, Y. P. Liu, and A. 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*, IEEE, 2020, pp. 119–130.

[23] J. v. d. Brand *et al.*, "Bipartite matching in nearly-linear time on moderately dense graphs," in *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, 2020, pp. 919–930.

[24] J. v. d. Brand *et al.*, "Minimum cost flows, mdps, and ℓ_1-regression in nearly linear time for dense instances," in *STOC*, Available at https://arxiv.org/abs/2101.05719, ACM, 2021, pp. 859–869.

[25] Y. Gao, Y. P. Liu, and R. Peng, "Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao," *FOCS 2021*, 2021, Available at https://arxiv.org/abs/2101.07233.

[26] J. van den Brand *et al.*, "Faster maxflow via improved dynamic spectral vertex sparsifiers," 2021, Under submission to STOC 2022.

[27] D. Durfee, M. Fahrbach, Y. Gao, and T. Xiao, "Nearly tight bounds for sandpile transience on the grid," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, Available at https://arxiv.org/abs/1704.04830, SIAM, 2018, pp. 605–624.

[28] P. Bak, *How Nature Works: The Science of Self-Organized Criticality.* Copernicus, 1996.

[29] N. W. Watkins, G. Pruessner, S. C. Chapman, N. B. Crosby, and H. J. Jensen, "25 years of self-organized criticality: Concepts and controversies," *Space Science Reviews*, vol. 198, no. 1-4, pp. 3–44, 2016.

[30] R. J. Wijngaarden, M. S. Welling, C. M. Aegerter, and M. Menghini, "Avalanches and self-organized criticality in superconductors," *The European Physical Journal B–Condensed Matter and Complex Systems*, vol. 50, no. 1, pp. 117–122, 2006.

[31] A. E. Biondo, A. Pluchino, and A. Rapisarda, "Modeling financial markets by self-organized criticality," *Physical Review E*, vol. 92, no. 4, p. 042 814, 2015.

[32] J. A. Scheinkman and M. Woodford, "Self-organized criticality and economic fluctuations," *The American Economic Review*, vol. 84, no. 2, pp. 417–421, 1994.

[33] H. Saba, J. Miranda, and M. Moret, "Self-organized critical phenomenon as a q-exponential decay–Avalanche epidemiology of dengue," *Physica A: Statistical Mechanics and its Applications*, vol. 413, pp. 205–211, 2014.

[34] J. Phillips, "Fractals and self-organized criticality in proteins," *Physica A: Statistical Mechanics and Its Applications*, vol. 415, pp. 440–448, 2014.

[35] M. Aschwanden, *Self-Organized Criticality in Astrophysics: The Statistics of Nonlinear Processes in the Universe*. Springer Science & Business Media, 2011.

[36] S. Mineshige, M. Takeuchi, and H. Nishimori, "Is a black hole accretion disk in a self-organized critical state?" *The Astrophysical Journal*, vol. 435, pp. L125–L128, 1994.

[37] O. Ramos, E. Altshuler, and K. Maløy, "Avalanche prediction in a self-organized pile of beads," *Physical Review Letters*, vol. 102, no. 7, p. 078 701, 2009.

[38] L. Brochini, A. de Andrade Costa, M. Abadi, A. C. Roque, J. Stolfi, and O. Kinouchi, "Phase transitions and self-organized criticality in networks of stochastic spiking neurons," *Scientific reports*, vol. 6, p. 35 831, 2016.

[39] A. Levina, J. M. Herrmann, and T. Geisel, "Dynamical synapses causing self-organized criticality in neural networks," *Nature physics*, vol. 3, no. 12, pp. 857–860, 2007.

[40] D. Dhar, "Theoretical studies of self-organized criticality," *Physica A: Statistical Mechanics and its Applications*, vol. 369, no. 1, pp. 29–70, 2006.

[41] S. Manna, "Two-state model of self-organized criticality," *Journal of Physics A: Mathematical and General*, vol. 24, no. 7, p. L363, 1991.

[42] A. Sornette and D. Sornette, "Self-organized criticality and earthquakes," *Europhysics Letters*, vol. 9, no. 3, p. 197, 1989.

[43] T. Kron and T. Grund, "Society as a self-organized critical system," *Cybernetics & Human Knowing*, vol. 16, no. 1, pp. 65–82, 2009.

[44] D. Dhar, "Self-organized critical state of sandpile automaton models," *Physical Review Letters*, vol. 64, no. 14, p. 1613, 1990.

[45] A. Björner, L. Lovász, and P. W. Shor, "Chip-firing games on graphs," *European Journal of Combinatorics*, vol. 12, no. 4, pp. 283–291, 1991.

[46] L. Babai and I. Gorodezky, "Sandpile transience on the grid is polynomially bounded," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Society for Industrial and Applied Mathematics, 2007, pp. 627–636.

[47] D. Dhar, P. Ruelle, S. Sen, and D.-N. Verma, "Algebraic aspects of abelian sandpile models," *Journal of Physics A: Mathematical and General*, vol. 28, no. 4, p. 805, 1995.

[48] L. Becchetti, V. Bonifaci, M. Dirnberger, A. Karrenbauer, and K. Mehlhorn, "Physarum can compute shortest paths: Convergence proofs and complexity bounds," in *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming* (ICALP), Springer, 2013, pp. 472–483.

[49] V. Bonifaci, K. Mehlhorn, and G. Varma, "Physarum can compute shortest paths," in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Society for Industrial and Applied Mathematics, 2012, pp. 233–240.

[50] K. Mehlhorn, "Physarum computations," in *Proceedings of the 30th International Symposium on Theoretical Aspects of Computer Science* (STACS), Schloss Dagstuhl, 2013, pp. 5–6.

[51] D. Straszak and N. K. Vishnoi, *IRLS and slime mold: Equivalence and convergence*, Preprint, `arXiv:1601.02712v1`, 2016.

[52] D. Straszak and N. K. Vishnoi, "Natural algorithms for flow problems," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Society of Industrial and Applied Mathematics, 2016, pp. 1868–1883.

[53] D. Straszak and N. K. Vishnoi, "On a natural dynamics for linear programming," in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science* (ITCS), Association for Computing Machinery, 2016, p. 291.

[54] C. Everett and P. Stein, "The combinatorics of random walk with absorbing barriers," *Discrete Mathematics*, vol. 17, no. 1, pp. 27–45, 1977.

[55] P. Bhakta, B. Cousins, M. Fahrbach, and D. Randall, "Approximately sampling elements with fixed rank in graded posets," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Society for Industrial and Applied Mathematics, 2017, pp. 1828–1838.

[56] D. C. Jerison, L. Levine, and J. Pike, *Mixing time and eigenvalues of the abelian sandpile Markov chain*, Preprint, `arXiv:1511.00666v1`, 2015.

[57] A. Ramachandran and A. Schild, "Sandpile prediction on a tree in near linear time," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), Society of Industrial and Applied Mathematics, 2017, pp. 1115–1131.

[58] D. B. Wilson, "Dimension of the loop-erased random walk in three dimensions," *Physical Review E*, vol. 82, no. 6, p. 062 102, 2010.

[59] A. E. Holroyd, L. Levine, K. Mészáros, Y. Peres, J. Propp, and D. B. Wilson, "Chip-firing and rotor-routing on directed graphs," *In and Out of Equilibrium 2*, pp. 331–364, 2008.

[60] L. Lovász, “Random walks on graphs: A survey,” *Combinatorics, Paul Erdős is Eighty*, vol. 2, pp. 1–46, 1993.

[61] W. Ellens, F. Spieksma, P. Van Mieghem, A. Jamakovic, and R. Kooij, “Effective graph resistance,” *Linear Algebra and its Applications*, vol. 435, no. 10, pp. 2491–2506, 2011.

[62] G.-C. Rota and K. A. Baclawski, *Introduction to Probability and Random Processes.* 1979.

[63] D. A. Levin, Y. Peres, and E. L. Wilmer, *Markov Chains and Mixing Times.* American Mathematical Society, 2009.

[64] P. Tetali, “Random walks and the effective resistance of networks,” *Journal of Theoretical Probability*, vol. 4, no. 1, pp. 101–109, 1991.

[65] A. V. Karzanov, “On finding maximum flows in networks with special structure and some applications,” *Matematicheskie Voprosy Upravleniya Proizvodstvom*, vol. 5, pp. 81–94, 1973.

[66] S. Even and R. E. Tarjan, “Network flow and testing graph connectivity,” *SIAM Journal on Computing*, vol. 4, no. 4, pp. 507–518, 1975.

[67] A. V. Goldberg and R. E. Tarjan, “Finding minimum-cost circulations by successive approximation,” *Math. Oper. Res.*, vol. 15, no. 3, pp. 430–466, 1990.

[68] A. V. Goldberg and S. Rao, “Beyond the flow decomposition barrier,” *Journal of the ACM*, vol. 45, no. 5, pp. 783–797, 1998, Announced at FOCS’97.

[69] D. Adil, R. Kyng, R. Peng, and S. Sachdeva, “Iterative refinement for ℓ_p-norm regression,” in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2019, pp. 1405–1424.

[70] R. Kyng, R. Peng, S. Sachdeva, and D. Wang, “Flows in almost linear time via adaptive preconditioning,” in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, Available at https://arxiv.org/abs/1906.10340, ACM, 2019, pp. 902–913.

[71] M. B. Cohen, A. Madry, P. Sankowski, and A. Vladu, “Negative-weight shortest paths and unit capacity minimum cost flow in $\widetilde{O}(m^{10/7} \log W)$ time (extended abstract),” in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, Available at https://arxiv.org/abs/1605.01717, SIAM, 2017, pp. 752–771.

[72] Y. P. Liu and A. Sidford, “Faster energy maximization for faster maximum flow,” in *Procceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, Available at https://arxiv.org/abs/1910.14276, ACM, 2020, pp. 803–814.

[73] K. Axiotis, A. Madry, and A. Vladu, "Circulation control for faster minimum cost flow in unit-capacity graphs," pp. 93–104, 2020, Available at: https://arxiv.org/abs/2111.10368v1.

[74] Y. T. Lee, S. Rao, and N. Srivastava, "A new approach to computing maximum flows using electrical flows," in *Proceedings of the 45th Annual ACM Symposium on Theory of Computing, STOC 2013, Palo Alto, CA, USA, June 1-4, 2013*, ACM, 2013, pp. 755–764.

[75] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, "An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations," in *Symposium on Discrete Algorithms (SODA)*, Available at https://arxiv.org/abs/1304.2338, 2014, pp. 217–226.

[76] R. Peng, "Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, SIAM, 2016, pp. 1862–1867.

[77] J. Sherman, "Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 2017, pp. 452–460.

[78] A. Sidford and K. Tian, "Coordinate methods for accelerating ℓ_∞ regression and faster approximate maximum flow," in *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, Available at https://arxiv.org/abs/1808.01278, 2018, pp. 922–933.

[79] Y. T. Lee and A. Sidford, "Solving linear programs with sqrt(rank) linear system solves," *CoRR*, vol. abs/1910.08033, 2019. arXiv: 1910.08033.

[80] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, 1984.

[81] P. M. Vaidya, "Speeding-up linear programming using fast matrix multiplication (extended abstract)," in *30th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1989, Research Triangle Park, NC, USA, October 30 - November 1, 1989*, IEEE Computer Society, 1989, pp. 332–337.

[82] K. Axiotis, A. Mądry, and A. Vladu, "Faster sparse minimum cost flow by electrical flow localization," in *FOCS*, IEEE, 2021.

[83] J. Nelson and H. Yu, "Optimal bounds for approximate counting," *arXiv preprint arXiv:2010.02116*, 2020.

[84] S. Dong, Y. T. Lee, and G. Ye, "A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path," in *STOC*, ACM, 2021, pp. 1784–1797.

[85] C. Dwork, A. Roth, *et al.*, “The algorithmic foundations of differential privacy.,” *Found. Trends Theor. Comput. Sci.*, vol. 9, no. 3-4, pp. 211–407, 2014.

[86] S. Dong *et al.*, “Nested dissection meets ipms : Planar min-cost flow in nearly-linear time,” in *SODA*, SIAM, 2022.

[87] J. Renegar, “A polynomial-time algorithm, based on newton’s method, for linear programming,” *Mathematical Programming*, vol. 40, no. 1-3, pp. 59–93, 1988.

[88] Y. T. Lee and A. Sidford, “Efficient inverse maintenance and faster algorithms for linear programming,” in *56th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, October 17-20, 2015*, Available at https://arxiv.org/abs/1503.01752, IEEE Computer Society, 2015, pp. 230–249.

[89] M. B. Cohen, Y. T. Lee, and Z. Song, “Solving linear programs in the current matrix multiplication time,” *Journal of the ACM (JACM)*, vol. 68, no. 1, pp. 1–39, 2021.

[90] Y. T. Lee, Z. Song, and Q. Zhang, “Solving empirical risk minimization in the current matrix multiplication time,” in *Conference on Learning Theory, COLT 2019, Phoenix, AZ, USA, June 25-28, 2019*, ser. Proceedings of Machine Learning Research, Available at https://arxiv.org/abs/1905.04447, vol. 99, PMLR, 2019, pp. 2140–2157.

[91] J. v. d. Brand, “A deterministic linear program solver in current matrix multiplication time,” in *SODA*, Available at https://arxiv.org/abs/1910.11957, SIAM, 2020, pp. 259–278.

[92] J. v. d. Brand, Y. T. Lee, A. Sidford, and Z. Song, “Solving tall dense linear programs in nearly linear time,” in *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, Available at https://arxiv.org/abs/2002.02304, ACM, 2020, pp. 775–788.

[93] S. Jiang, Z. Song, O. Weinstein, and H. Zhang, “Faster dynamic matrix inverse for faster lps,” *CoRR*, vol. abs/2004.07470, 2020.

[94] J. van den Brand, “Unifying matrix data structures: Simplifying and speeding up iterative algorithms,” in *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, Available at https://arxiv.org/abs/2010.13888, SIAM, 2021, pp. 1–13.

[95] D. Nanongkai and T. Saranurak, “Dynamic spanning forest with worst-case update time: Adaptive, Las Vegas, and $O(n^{1/2-\epsilon})$-time,” in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, Available at https://arxiv.org/abs/1611.03745, 2017, pp. 1122–1129.

[96] C. Wulff-Nilsen, “Fully-dynamic minimum spanning forest with improved worst-case update time,” in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of*

Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017, Available at https://arxiv.org/abs/1611.02864, 2017, pp. 1130–1143.

[97] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen, "Dynamic minimum spanning forest with subpolynomial worst-case update time," in *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, Available at https://arxiv.org/abs/1708.03962, IEEE Computer Society, 2017, pp. 950–961.

[98] T. Saranurak and D. Wang, "Expander decomposition and pruning: Faster, stronger, and simpler," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, Available at https://arxiv.org/abs/1812.08958, SIAM, 2019, pp. 2616–2635.

[99] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak, "A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond," in *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, Available at https://arxiv.org/abs/1910.08025, IEEE, 2020, pp. 1158–1167.

[100] A. Bernstein *et al.*, "Fully-dynamic graph sparsifiers against an adaptive adversary," *CoRR*, vol. abs/2004.08432, 2020, Available at https://arxiv.org/abs/2004.08432.

[101] A. Bernstein and S. Chechik, "Deterministic decremental single source shortest paths: Beyond the O(mn) bound," in *Symposium on Theory of Computing (STOC)*, 2016, pp. 389–397.

[102] A. Bernstein and S. Chechik, "Deterministic partially dynamic single source shortest paths for sparse graphs," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2017, pp. 453–469.

[103] J. Chuzhoy and S. Khanna, "A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, 2019, pp. 389–400.

[104] M. P. Gutenberg and C. Wulff-Nilsen, "Deterministic algorithms for decremental approximate shortest paths: Faster and simpler," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2020, pp. 2522–2541.

[105] J. Chuzhoy, "Decremental all-pairs shortest paths in deterministic near-linear time," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, 2021, pp. 626–639.

[106] A. Bernstein, M. P. Gutenberg, and T. Saranurak, "Deterministic decremental SSSP and approximate min-cost flow in almost-linear time," in *62st IEEE Annual Symposium on Foundations of Computer Science, FOCS2021*, IEEE, 2021.

[107] M. B. Giles, "Multilevel monte carlo methods," *Acta Numerica*, vol. 24, pp. 259–328, 2015.

[108] J. H. Blanchet and P. W. Glynn, "Unbiased monte carlo for optimization and functions of expectations via multi-level randomization," in *2015 Winter Simulation Conference (WSC)*, IEEE, 2015, pp. 3656–3667.

[109] H. Asi, Y. Carmon, A. Jambulapati, Y. Jin, and A. Sidford, "Stochastic bias-reduced gradient methods," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[110] R. H. Morris Sr., "Counting large numbers of events in small registers," *Commun. ACM*, vol. 21, no. 10, pp. 840–842, 1978.

[111] D. M. Kane, J. Nelson, E. Porat, and D. P. Woodruff, "Fast moment estimation in data streams in optimal space," in *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, June 6-8 2011*, Available at https://arxiv.org/abs/1007.4191, ACM, 2011, pp. 745–754.

[112] H. Li and A. Schild, "Spectral subspace sparsification," in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2018, pp. 385–396.

[113] I. Koutis, G. L. Miller, and R. Peng, "Approaching optimality for solving SDD linear systems," in *51th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2010, Las Vegas, NV, USA, October 23-26, 2010*, Available at https://arxiv.org/abs/1003.2958, 2010, pp. 235–244.

[114] I. Koutis, G. L. Miller, and R. Peng, "A nearly-m log n time solver for SDD linear systems," in *52nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, Available at https://arxiv.org/abs/1102.4842, 2011, pp. 590–598.

[115] J. A. Kelner, L. Orecchia, A. Sidford, and Z. Allen Zhu, "A simple, combinatorial algorithm for solving SDD systems in nearly-linear time," in *Proceedings of the 45th Annual ACM Symposium on Theory of Computing, STOC 2013, Palo Alto, CA, USA, June 1-4, 2013*, Available at https://arxiv.org/abs/1301.6628, 2013, pp. 911–920.

[116] Y. T. Lee and A. Sidford, "Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems," in *2013 ieee 54th annual symposium on foundations of computer science*, IEEE, 2013, pp. 147–156.

[117] M. B. Cohen *et al.*, "Solving SDD linear systems in nearly $m \log^{1/2} n$ time," in *Proceedings of the 46th Annual ACM Symposium on Theory of Computing, STOC 2014, New York, NY, USA, June 1-3, 2014*, 2014, pp. 343–352.

[118] R. Kyng, Y. T. Lee, R. Peng, S. Sachdeva, and D. 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, Available at https://arxiv.org/abs/1512.01892, 2016, pp. 842–850.

[119] R. Kyng and S. Sachdeva, "Approximate gaussian elimination for Laplacians - fast, sparse, and simple," in *57th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2016, Hyatt Regency, New Brunswick, NJ, USA, October 9-11, 2016*, Available at https://arxiv.org/abs/1605.02353, 2016, pp. 573–582.

[120] J. A. Kelner and A. Madry, "Faster generation of random spanning trees," in *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, 2009, pp. 13–21.

[121] A. Madry, D. Straszak, and J. Tarnawski, "Fast generation of random spanning trees and the effective resistance metric," in *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, SIAM, 2014, pp. 2019–2036.

[122] D. Durfee, R. Kyng, J. Peebles, A. B. Rao, and S. Sachdeva, "Sampling random spanning trees faster than matrix multiplication," in *Symposium on Theory of Computing (STOC)*, 2017, pp. 730–742.

[123] A. Schild, "An almost-linear time algorithm for uniform random spanning tree generation," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, Available at https://arxiv.org/abs/1711.06455, ACM, 2018, pp. 214–227.

[124] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[125] J. H. Reif, "Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 71–81, 1983.

[126] R. Hassin and D. B. Johnson, "An O$(n \log^2 n)$ algorithm for maximum flow in undirected planar networks," *SIAM Journal on Computing*, vol. 14, no. 3, pp. 612–624, 1985.

[127] G. Borradaile and P. N. Klein, "An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph," *J. ACM*, vol. 56, no. 2, 9:1–9:30, 2009.

[128] E. W. Chambers, J. Erickson, and A. Nayyeri, "Homology flows, cohomology cuts," *SIAM J. Comput.*, vol. 41, no. 6, pp. 1605–1634, 2012.

[129] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," in *Proceedings of the Conference on Analysing Problem Classes and Programming for Parallel Computing*, ser. CONPAR '81, Berlin, Heidelberg: Springer-Verlag, 1981, pp. 314–327, ISBN: 3540108270.

[130] R. Peng and D. A. Spielman, "An efficient parallel solver for sdd linear systems," in *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, New York, New York: Association for Computing Machinery, 2014, pp. 333–342, ISBN: 9781450327107.

[131] I. Koutis and S. C. Xu, "Simple parallel and distributed algorithms for spectral graph sparsification," *ACM Trans. Parallel Comput.*, vol. 3, no. 2, Aug. 2016.

[132] R. E. Tarjan, "An efficient planarity algorithm," Tech. Rep., 1971.

[133] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized nested dissection," *SIAM journal on numerical analysis*, vol. 16, no. 2, pp. 346–358, 1979.

[134] K. D. Gremban, "Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems," Ph.D. dissertation, Carnegie Mellon University, 1996.

[135] J. Orlin, "A faster strongly polynomial minimum cost flow algorithm," in *Proceedings of the Twentieth annual ACM symposium on Theory of Computing*, 1988, pp. 377–387.

[136] A. Karczmarz and P. Sankowski, "Min-cost flow in unit-capacity planar graphs," in *27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany*, ser. LIPIcs, vol. 144, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 66:1–66:17.

[137] H. Kaplan and Y. Nussbaum, "Min-cost flow duality in planar networks," *arXiv preprint arXiv:1306.6728*, 2013.

[138] B. Vaidyanathan and R. K. Ahuja, "Fast algorithms for specially structured minimum cost flow problems with applications," *Operations Research*, vol. 58, no. 6, pp. 1681–1696, 2010.

[139] V. King, S. Rao, and R. Tarjan, "A faster deterministic maximum flow algorithm," *Journal of Algorithms*, vol. 17, no. 3, pp. 447–474, 1994.

[140] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows.* Prentice Hall, 1988.

[141] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva, "Maximum flow and minimum-cost flow in almost-linear time," *CoRR*, vol. abs/2203.00671, 2022. arXiv: 2203.00671.

[142] G. Borradaile, *Exploiting Planarity for Network Flow and Connectivity Problems.* Brown University, 2008.

[143] A. Itai and Y. Shiloach, "Maximum flow in planar networks," *SIAM Journal on Computing*, vol. 8, no. 2, pp. 135–150, 1979.

[144] R. Hassin, "Maximum flow in (s, t) planar networks," *Information Processing Letters*, vol. 13, no. 3, p. 107, 1981.

[145] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 3–23, 1997.

[146] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. 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*, ACM, 2011, pp. 313–322.

[147] K. Weihe, "Maximum (s, t)-flows in planar networks in $O(|V| \log |V|)$ time," *Journal of Computer and System Sciences*, vol. 55, no. 3, pp. 454–475, 1997.

[148] G. L. Miller and J. Naor, "Flow in planar graphs with multiple sources and sinks," *SIAM J. Comput.*, vol. 24, no. 5, pp. 1002–1017, 1995.

[149] G. Borradaile, P. N. Klein, S. Mozes, Y. Nussbaum, and C. Wulff-Nilsen, "Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time," *SIAM J. Comput.*, vol. 46, no. 4, pp. 1280–1303, 2017.

[150] G. L. Miller and R. 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*, SIAM, 2013, pp. 1151–1170.

[151] H. Imai and K. Iwano, "Efficient sequential and parallel algorithms for planar minimum cost flow," in *Algorithms, International Symposium SIGAL '90, Tokyo, Japan*, ser. Lecture Notes in Computer Science, vol. 450, Springer, 1990, pp. 21–30.

[152] M. K. Asathulla, S. Khanna, N. Lahn, and S. 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, New Orleans, LA, USA*, SIAM, 2018, pp. 457–476.

[153] N. Lahn and S. 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*, SIAM, 2019, pp. 569–588.

[154] S. Khuller, J. Naor, and P. Klein, "The lattice structure of flow in planar graphs," *SIAM Journal on Discrete Mathematics*, vol. 6, no. 3, pp. 477–490, 1993.

[155] D. Adil and S. Sachdeva, "Faster p-norm minimizing flows, via smoothed q-norm problems," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2020, pp. 892–910.

[156] D. Adil, B. Bullins, R. Kyng, and S. Sachdeva, "Almost-Linear-Time Weighted ℓ_p-norm Solvers in Slightly Dense Graphs via Sparsification," in *48th International Colloquium on Au-*

tomata, Languages, and Programming (ICALP 2021), vol. 198, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 9:1–9:15.

[157] M. B. Cohen, A. Madry, P. Sankowski, and A. Vladu, "Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{O}(m^{10/7} \log W)$ time (extended abstract)," in *Symposium on Discrete Algorithms (SODA)*, 2017, pp. 752–771.

[158] B. Huang, S. Jiang, Z. Song, and R. Tao, "Solving tall dense SDPs in the current matrix multiplication time," *arXiv preprint arXiv:2101.08208*, 2021.

[159] J. R. Gilbert and R. E. Tarjan, "The analysis of a nested dissection algorithm," *Numer. Math.*, vol. 50, no. 4, pp. 377–404, Feb. 1987.

[160] J. Fakcharoenphol and S. Rao, "Planar graphs, negative weight edges, shortest paths, and near linear time," *Journal of Computer and System Sciences*, vol. 72, no. 5, pp. 868–889, 2006.

[161] R. Kyng, "Approximate gaussian elimination," Ph.D. dissertation, Yale University, 2017.

[162] G. Goranci, M. Henzinger, and P. 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*, ser. LIPIcs, Available at https://arxiv.org/abs/1802.09111, vol. 112, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 40:1–40:15.

[163] S. Dong, Y. T. Lee, and G. Ye, "A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path," *arXiv preprint arXiv:2011.05365v2*, 2021.

[164] R. Lipton and R. Tarjan, "A Planar Separator Theorem," *SIAM Journal of Applied Mathematics*, vol. 36, no. 2, pp. 177–189, 1979.