Fully Dynamic Spectral Vertex Sparsifiers and Applications

David Durfee [*†] Georgia Tech
Yu Gao[‡*†] Georgia Tech
Gramoz Goranci [*§] University of Vienna
Richard Peng [*†] Georgia Tech

June 26, 2019

Abstract

We study *dynamic* algorithms for maintaining spectral vertex sparsifiers of graphs with respect to a set of terminals T of our choice. Such objects preserve pairwise resistances, solutions to systems of linear equations, and energy of electrical flows between the terminals in T. We give a data structure that supports insertions and deletions of edges, and terminal additions, all in sublinear time. We then show the applicability of our result to the following problems.

(1) A data structure for dynamically maintaining solutions to Laplacian systems $\mathbf{Lx} = \mathbf{b}$, where $\mathbf{L}$ is the graph Laplacian matrix and $\mathbf{b}$ is a demand vector. For a bounded degree, unweighted graph, we support modifications to both $\mathbf{L}$ and $\mathbf{b}$ while providing access to ϵ-approximations to the energy of routing an electrical flow with demand $\mathbf{b}$, as well as query access to entries of a vector $\tilde{\mathbf{x}}$ such that $\left\|\tilde{\mathbf{x}} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon \left\|\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}$ in $\tilde{O}(n^{11/12}\epsilon^{-5})$ expected amortized update and query time.

(2) A data structure for maintaining fully dynamic All-Pairs Effective Resistance. For an intermixed sequence of edge insertions, deletions, and resistance queries, our data structure returns $(1 \pm \epsilon)$-approximation to all the resistance queries against an oblivious adversary with high probability. Its expected amortized update and query times are $\tilde{O}(\min(m^{3/4}, n^{5/6}\epsilon^{-2})\epsilon^{-4})$ on an unweighted graph, and $\tilde{O}(n^{5/6}\epsilon^{-6})$ on weighted graphs.

The key ingredients in these results are (1) the intepretation of Schur complement as a sum of random walks, and (2) a suitable choice of terminals based on the behavior of these random walks to make sure that the majority of walks are local, even when the graph itself is highly connected and (3) maintenance of these local walks and numerical solutions using data structures.

These results together represent the first data structures for maintaining key primitives from the Laplacian paradigm for graph algorithms in sublinear time without assumptions on the underlying graph topologies. The importance of routines such as effective resistance, electrical flows, and Laplacian solvers in the static setting make us optimistic that some of our components can provide new building blocks for dynamic graph algorithms.

[*]Emails: {ddurfee,ygao380}@gatech.edu, gramoz.goranci@univie.ac.at, richard.peng@gmail.com

[†]This material is based upon work supported by the National Science Foundation under Grant No. 1718533.

[‡]The author was supported by the ACO program at the Georgia Institute of Technology.

[§]This work was done while visiting the Georgia Institute of Technology. The research leading to these results has received funding from the Marshall Plan Foundation and the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

Problems arising from analyzing and understanding graph structures have motivated the development of many powerful tools for storing and compressing graphs and networks. One such tool that has received a considerable amount of attention over the past two decades is graph sparsification [10, 9]. Roughly speaking, a graph sparsifier is a "compressed" version of a large input graph that preserves important properties like distance information [56], cut value [10] or graph spectrum [65]. Graph Sparsifiers fall into two main categories: *edge sparsifiers*, which are graphs that reduce the number of edges, and *vertex sparsifiers*, which are graphs that reduce the number of vertices. Both categories have many applications in approximation algorithms [22, 59], machine learning [46, 45, 72], and most recently efficient graph algorithms [66, 47, 63, 36]. While edge sparsifiers have played an instrumental role in obtaining nearly linear time algorithms [9], their practical applicability is somewhat limited due to the fact most of the large networks are already sparse. On the other hand, vertex sparsifiers address the "real" compression of large networks by reducing the number of vertices.

While vertex sparisifers in general are significantly more difficult to generate [51, 13, 49], a notable exception is vertex sparsifiers for quadratic minimization problems, otherwise known as Schur complements. Concretely, given an undirected, weighted graph G, a subset of terminal vertices T and its corresponding Laplacian matrix, a graph H with $V(H) = T$ is a vertex resistance sparsifier of G with respect to T if the Laplacian matrix of H is obtained by the Schur complement of the Laplacian of G with respect to T. Schur complement is a central concept in physics and linear algebra with a wide range of applications including multi-grid solvers, Markov chains and finite-element analysis [16], and has also recently found extensive applications in graph algorithms [40, 41, 18, 19, 62, 61].

Most of the massive graphs in the real world, such as social networks, the web graph, are subject to frequent changes over time. This dynamic behavior of graphs has been studied for several important graph problems, where the basic idea is to maintain problem solutions as graphs undergo edge insertions and deletions in time faster than recomputing the solution from scratch. Dynamic graph algorithms have also been formulated for many problems that involve edge sparsifiers [32, 70, 35, 26], as well important variants of edge sparsifiers themselves, including minimum spanning trees [33, 52, 73, 53], spanners [8], spectral sparsifiers [2], and low-stretch spanning trees [27]. However, despite the increasing importance of high quality vertex sparsifiers in graph algorithms, very little is known about their maintenance in the dynamic setting.

In this paper we give the first non-trival *dynamic* algorithms for maintaining Schur complements of general graphs with respect to a set of terminal of our choice. Our data-structure maintains at any point of time a $(1 \pm \epsilon)$ approximation to the Schur complement while supporting insertions and deletions of edges, and arbitrary vertex additions to the terminal set. To the best of our knowledge, prior dynamic Schur complement algorithms were only known for minor-free graphs [24, 25].

Lemma 1.1. *Given an error parameter $\epsilon > 0$, an unweighted undirected multi-graph $G = (V, E)$ with n vertices, m edges, a subset of terminal vertices T' and a parameter $\beta \in (0, 1)$ such that $|T'| = O(\beta m)$, there is a data-structure* DynamicSC(G, T', β) *for maintaining a graph $\tilde{H}$ with $\mathbf{L}_{\tilde{H}} \approx_\epsilon \mathbf{SC}(G, T)$ for some T with $T' \subseteq T$, $|T| = O(\beta m)$, while supporting $O(\beta m)$ operations in the following running times:*

- Initialize(G, T', β)*: Initialize the data-structure, in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$[1] expected amortized time.*
- Insert(u, v)*: Insert the edge (u, v) to G in $\tilde{O}(1)$ amortized time.*

[1] We use $\tilde{O}(f)$ to denote $O(f \cdot \text{poly}(\log n))$ for a function f.

- DELETE(u, v): *Delete the existing edge* (u, v) *from* G *in* $\tilde{O}(1)$ *amortized time.*
- ADDTERMINAL(u): *Add* u *to* T' *in* $\tilde{O}(1)$ *amortized time.*

Our guarantees hold against an oblivious adversary.

Our algorithm extends to weighted graphs, albeit with slightly larger running time guarantees. Concretely we give an algorithm that maintains an approximate Schur Complement with $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ expected amortized time for initializing the data-structure, and $\tilde{O}(1)$ amortized time for the remaining operations.

The key algorithmic components behind the result in unweighted graphs are (1) the interpretation of Schur complement as a sum of random walks and (2) randomly picking a terminal vertex subset onto which the vertex resistance sparsifiers is constructed. Specifically, in a novel way we combine random walk based methods for generating resistance vertex sparsifiers [19] with results in combinatorics that bound the speed at which such walks spread among *distinct* edges [6]. Our result in the weighted case essentially follows the same idea except that the speed at which random walks visit distinct edges in weighted networks could be very slow. To control this, we show how to efficiently sample a distinct edge without the need to simulate every step of the walk. This interacts well with other parts of our data structure and leads to comparable running time guarantees.

We show the applicability of our dynamic Schur complement to two cornerstone problems in graph Laplacian literature, namely *dynamic* Laplacian solver [66] and *dynamic* All-Pairs Effective Resistances [67].

Solving linear systems lies at the heart of many problems arising in scientific computing, numerical linear algebra, optimization and computer science. An important subclass of linear systems are Laplacian systems, which arise in many natural contexts, including computation of voltages and currents in electrical network. Solving Laplacian system has received increasing attention over the past years after the breakthrough work of Spielman and Teng [66] who gave the first near-linear time algorithm. Motivated by fast Laplacian solvers in different model of computations [5, 58], we initiate the study of algorithms for dynamically solving Laplacian systems. Concretely, given a graph Laplacian $\mathbf{L} \in \mathbb{R}^{n\times n}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ in the range of $\mathbf{L}$, the goal is to maintain an $\mathbf{x}$ such that $\mathbf{Lx} = \mathbf{b}$, while off-diagonals of $\mathbf{L}$ and the entries of $\mathbf{b}$ change over time. To allow for sub-linear query times, here we focus on querying one (or a few) coordinates of $\mathbf{x}$. Formally, given any index $u \in \{1, \ldots, n\}$, the goal is to output $\tilde{\mathbf{x}}(u)$ for some approximation $\tilde{\mathbf{x}}$ of $\mathbf{L}^\dagger\mathbf{b}$. Our contribution is the first sub-linear dynamic Laplacian solver in bounded degree graphs.

Theorem 1.2. *For any given error threshold* $m^{-1} < \epsilon < 1$*, there is a data-structure for maintaining an unweighted, undirected bounded degree multi-graph* $G = (V, E)$ *with* n *vertices,* m *edges and a vector* $\mathbf{b} \in \mathbb{R}^n$ *that supports the following operations in* $\tilde{O}(n^{11/12}\epsilon^{-5})$ *expected amortized time:*

- INSERT(u, v): *Insert the edge* (u, v) *with resistance* 1 *in* G*.*
- DELETE(u, v): *Delete the edge* (u, v) *from* G*.*
- CHANGE$(u, \mathbf{b}'(u), v, \mathbf{b}'(v))$: *Change* $\mathbf{b}(u)$ *to* $\mathbf{b}'(u)$ *and* $\mathbf{b}(v)$ *to* $\mathbf{b}'(v)$ *while keeping* $\mathbf{b}$ *in the range of* $\mathbf{L}$*.*
- SOLVE(u): *Return* $\tilde{\mathbf{x}}(u)$ *with* $\tilde{\mathbf{x}}$ *such that* $\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon \left\|\mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}}$.

Our guarantees hold against an oblivious adversary.

Note that the $\tilde{\mathbf{x}}$ in the theorem above is not guaranteed to be inside the range of $\mathbf{L}_G$ and it only preserves the differences between vertices in the same connected component.

We observe that conditioning on the vector $\mathbf{b}$ having small support, i.e., a small number of non-zero elements, immediately leads to a dynamic solver by just including the corresponding vertices into the Schur complement, and maintaining a dynamic Schur complement onto these vertices augmented with some carefully chosen additional terminals. Upon receipt of a query index, we add the corresponding vertex to the current Schur complement and simply solve a linear system there. However, note that the demand vector may have a large number of non-zero entries, thus preventing us from obtaining a sub-linear time algorithm with this approach. We alleviate this by projecting this vector onto the set of current terminals and showing that such projection can be maintained dynamically while introducing controllable error in the approximation guarantee.

A consequence of dynamic Laplacian solver is that we can maintain the energy of the electrical flow when routing any vector $\mathbf{b}$ in the range of $\mathbf{L}$, which is a generalization of the All-Pairs Effective Resistance problem with $\mathbf{b}$ having exactly two non-zero entries, one of them being 1 and the other -1.

Another application of our technique is dynamic maintainance of effective resistance, a well studied quantity that has direct applications in random walks, spanning trees [48] and graph sparsification [67]. We maintain (approximate) All-Pairs Effective Resistances of a graph G among any pair of query vertices while supporting an intermixed sequence of edge insertions and deletions in G. Our study is also motivated in part by the wide usage of *commute distances*, a random walk-based similarity measure that has been successfully employed in important practical applications such as link predictions [44]. Since commute distance is a scaled version of effective resistance, our dynamic algorithm readily extends to this graph measure while achieving the same approximation and running time guarantees.

Theorem 1.3. *For any given error threshold $\epsilon > 0$, there is a data-structure for maintaining an unweighted, undirected multi-graph $G = (V, E)$ with up to m edges that supports the following operations in $\tilde{O}(m^{3/4}\epsilon^{-4})$ expected amortized time:*

- Insert(u, v)*: Insert the edge (u, v) with resistance 1 in G.*
- Delete(u, v)*: Delete the edge (u, v) from G.*
- EffectiveResistance(s, t)*: Return a $(1 \pm \epsilon)$-approximation to the effective resistance between s and t in the current graph G.*

Our guarantees hold against an oblivious adversary.

Our algorithm can also handle weighted graphs, albeit with a bound of $\tilde{O}(m^{5/6}\epsilon^{-4})$ on the expected amortized update and query time. By running this algorithm on the output of a dynamic spectral sparsifier [2], we obtain a bound of $\tilde{O}(n^{5/6}\epsilon^{-6})$ per operation, which is truly sub-linear irrespective of graph density. We discuss such improvements in Sections 4.3.

We are optimistic that our algorithmic ideas could be useful for dynamically maintaining a wider range of graph properties. Both the results that we give dynamic algorithms for, vertex sparsifiers and Schur complements, have wide ranges of applications in static settings, with the latter being at the core of the 'Laplacian paradigm' of graph algorithms [68, 69]. While it's less clear that solutions across multiple Laplacian solves can be propagated to each other as the input dynamically changes, repeated sparsification on the other hand represents a routine that composes and interacts well with a much wider range of primitives. As a result, we are optimistic that it can be used as a building block in dynamic versions of many existing applications of Laplacian solvers.

1.1 Related Works

The recent data structures for maintaining effective resistances in planar graphs [24, 25] drew direct connections between Schur complements and data structures for maintaining them in dynamic graphs. This connection is due to the preservation of effective resistances under vertex eliminations (Fact 2.2). From this perspective, the Schur complement can be viewed as a vertex sparsifier for preserving resistances among a set of terminal vertices.

The power of vertex or edge graph sparsifiers, which preserve certain properties while reducing problem sizes, has long been studied in data structures [20, 21]. Ideas from these results are central to recent works on offline maintenance for 3-connectivity [57], generating random spanning trees [18], and new notions of centrality for networks [43]. Our result is the first to maintain such vertex sparsifiers, specifically Schur complements, for *general* graphs in online settings.

While the ultimate goal is to dynamically maintain (approximate) minimum cuts and maximum flows, effective resistances represent a natural 'first candidate' for this direction of work due to them having perfect vertex sparsifiers. That is, for any subset of terminals, there is a sparse graph on them that approximately preserves the effective resistances among all pairs of terminals. This is in contrast to distances, where it's not known whether such a graph can be made sparse, or in contrast to cuts, where the existence of such a dense graph is not known (assuming that we are not content with large constant or poly-logarithmic approximations).

Dynamic Graph Algorithms. The maintenance of graph properties in dynamic algorithms has been a major area of ongoing research in data structures. The problems being maintained include $2-$ or $3-$connectivity [21, 32, 33], shortest paths [29, 30, 11, 1, 15], global minimum cut [31, 70, 42, 26], maximum matching [55, 28, 12], and maximal matching [7, 54, 64]. Perhaps most closely related to our work are dynamic algorithms that maintain properties related to paths [23, 21, 32, 35, 73, 53, 52]. In particular, the work of Wulff-Nilsen [73] also utilizes the behavior of random walks under edge deletions to keep track of low-conductance cuts.

Dynamic algorithms for evaluating algebraic functions such as matrix determinant and matrix inverse has also been considered [60]. One application of such algorithms is that they can be used to dynamically maintain single-pair effective resistance. Specifically, using the dynamic matrix inversion algorithm, one can dynamically maintain *exact* (s, t)-effective resistance in $O(n^{1.575})$ update time and $O(n^{0.575})$ query time.

Vertex Sparsifiers. Vertex sparsifiers have been studied in more general settings for preserving cuts and flows among terminal vertices [51, 13, 39]. Efficient versions of such routines have direct applications in data structures, even when they only work in restricted settings: terminal sparsifiers on quasi-bipartite graphs [4] were core routines in the data structure for maintaining flows in bipartite undirected graphs [2].

Our data structure utilizes vertex sparsifiers, but in even more limited settings as we get to control the set of vertices to sparsify onto. Specifically, the local maintenance of this sparsifier under insertions and deletions hinges upon the choice of a random subset of terminals, while vertex sparsifiers usually need to work for any subset of terminals. Evidence from numerical algorithms [40, 19] suggest this choice can significantly simplify interactions between algorithmic components. We hope this flexibility can motivate further studies of vertex sparsifiers in more restrictive, but still algorithmically useful settings.

Organization. The paper is organized as follows. We discuss preliminaries in Section 2 and give an overview of the key techniques in Section 3. After that we give a data-structure for dynamic

Schur complement on unweighted graphs in Section 4, which can be applied to the dynamic All-Pairs Effective Resistance problem. In Section 4.3, we briefly discuss the extension of our data-structure to weighted graphs. In Section 5, we give a high-level idea behind a data-structure that dynamically maintains the projection of a vector onto a subset of vertices of an unweighted bounded degree graph, which can be combined with dynamic Schur complement to give a dynamic Laplacian solver. In Appendix A, we show that our bound on the maximum load of a vertex from Lemma 5.5 is essentially tight. In Appendix B, we provide details on the graph approximation guarantees and properties of projections that our random walk sampling and other routines rely on. Finally, in Appendix C, we provide an algorithm for approximately sampling the sum of reciprocals of the edge weights of a random walk which allows us to generate long random walks without going through each step.

2 Preliminaries

In our dynamic setting, an undirected, weighted multi-graph undergoes both insertions and deletions of edges. We let $G = (V, E, \boldsymbol{w})$ always refer to the *current* version of the graph. We will use n and m to denote bounds on the number of vertices and edges at any point, respectively.

For an unweighted, undirected multi-graph G, let $\mathbf{A}_G$ denote its adjacency matrix and let $\mathbf{D}_G$ its degree diagonal matrix (counting edge multiplicities for both matrices). The graph *Laplacian* $\mathbf{L}_G$ of G is then defined as $\mathbf{L}_G = \mathbf{D}_G - \mathbf{A}_G$. Let $\mathbf{L}_G^\dagger$ denote the Moore-Penrose pseudo-inverse of $\mathbf{L}_G$. Subscripts will be often omitted when the underlying graph is clear from the context. We define the indicator vector $\mathbf{1}_u \in \mathbb{R}^n$ of a vertex u such that $\mathbf{1}_u(v) = 1$ if $v = u$, and $\mathbf{1}_u(v) = 0$ otherwise. Let $\boldsymbol{d}(u) = \sum_{v:(u,v)\in E} \boldsymbol{w}_{u,v}$ be the weighted degree of a vertex u.

A *walk* in G is a sequence of vertices such that consecutive vertices are connected by edges. A *random walk* in G is a walk that starts at a starting vertex w_0, and at step $i \geq 1$, the vertex w_i is chosen randomly among the neighbors of w_{i-1}. If graph G is unweighted, then each of its neighbors becomes w_i with equal probability. If G is weighted, the probability $\mathbb{P}_w[w_i = u \mid w_0, \ldots, w_{i-1}]$ is proportional to the edge weight $\boldsymbol{w}_{w_{i-1}u}$.

Effective Resistance. For our algorithm, it will be useful to define effective resistance using linear algebraic structures. Specifically, given any two vertices u and v in G, if $\boldsymbol{\chi}(u, v) := \mathbf{1}_u - \mathbf{1}_v$, then the *effective resistance* between u and v is given by

$$R_{\text{eff}}^G(u, v) := \boldsymbol{\chi}_{u,v}^\top \mathbf{L}_G^\dagger \boldsymbol{\chi}_{u,v}.$$

Linear systems in graph Laplacian matrices can be solved in nearly-linear time [38]. One prominent application of these solvers is the approximation of effective resistances.

Lemma 2.1. *Fix $\epsilon \in (0, 1)$ and let $G = (V, E)$ be any graph with two arbitrary distinguished vertices u and v. There is an algorithm that computes a value ϕ such that*

$$(1 - \epsilon)R_{\text{eff}}^G(u, v) \leq \phi \leq (1 + \epsilon)R_{\text{eff}}^G(u, v),$$

in $\tilde{O}(m + n/\epsilon^2)$ time with high probability.

Schur complement. Given a graph $G = (V, E)$, we can think of the *Schur complement* as the partially eliminated state of G. This relies on some partitioning of V into two disjoint subset of vertices T, called *terminals* and $F := V \setminus T$, called *non-terminals*, which in turn partition the Laplacian $\mathbf{L}$ into 4 blocks:

$$\mathbf{L} := \begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,T]} \\ \mathbf{L}_{[T,F]} & \mathbf{L}_{[T,T]} \end{bmatrix}. \tag{1}$$

The *Schur complement* onto T, denoted by $\mathbf{SC}(G,T)$, is the matrix obtained after eliminating the variables in F. Its closed form is given by

$$\mathbf{SC}(G,T) = \mathbf{L}_{[T,T]} - \mathbf{L}_{[T,F]}\mathbf{L}_{[F,F]}^{-1}\mathbf{L}_{[F,T]}.$$

It is well known that $\mathbf{SC}(G,T)$ is a Laplacian matrix of a graph on vertices in T. To simplify our exposition, we let $\mathbf{SC}(G,T)$ denote both the Laplacian and its corresponding graph. An important property of Schur complement which we exploit in this work is to view the Schur complement as a collection of random walks. This particular feature will be discussed in more detail in Section 3. The key role of Schur complements in our algorithms stems from the fact that they can be viewed as vertex sparsifiers that preserve pairwise effective resistances (see e.g., [24]).

Fact 2.2 (Vertex Resistance Sparsifier)**.** *For any graph $G = (V,E)$, any subset of vertices T, and any pair of vertices $u, v \in T$,*

$$R_{\text{eff}}^{G}(u,v) = R_{\text{eff}}^{\text{SC}(G,T)}(u,v).$$

Spectral Approximation

Definition 2.3 (Spectral Sparsifier)**.** Given a graph $G = (V,E)$ and $\epsilon \in (0,1)$, we say that a graph $H = (V,E')$ is a $(1 \pm \epsilon)$-*spectral sparsifier* of G (abbr. $H \approx_\epsilon G$) if $E' \subseteq E$, and for all $\mathbf{x} \in \mathbb{R}^n$

$$(1-\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x} \leq (1+\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x}.$$

In the dynamic setting, Abraham et al. [2] recently showed that $(1 \pm \epsilon)$-spectral sparsifiers of a dynamic graph G can be maintained efficiently. This algorithm will be invoked in several places throughout this paper.

Lemma 2.4 ([2], Theorem 4.1)**.** *Given a graph G with polynomially bounded edge weights, with high probability, we can dynamically maintain a $(1 \pm \epsilon)$-spectral sparsifier of size $\widetilde{O}(n\epsilon^{-2})$ of G in $O(\log^9 n\epsilon^{-2})$ expected amortized time per edge insertion or deletion. The running time guarantees hold against an oblivious adversary.*

The above result is useful because matrix approximations also preserve approximations of their quadratic forms. As a consequence of this fact, we get the following lemma.

Lemma 2.5. *If H is a $(1 \pm \epsilon)$-spectral sparsifier of G, then for any pair of vertices u and v*

$$(1-\varepsilon)R_{\text{eff}}^{G}(u,v) \leq R_{\text{eff}}^{H}(u,v) \leq (1+\varepsilon)R_{\text{eff}}^{G}(u,v).$$

2.1 Projection matrix and its properties

We next define a matrix that naturally appears when performing Gaussian elimination on the non-terminal vertices. Concretely, given a graph $G = (V,E)$ and terminals $T \subseteq V$, the *matrix-projection* of the non-terminals $F = V \setminus T$ onto T is given by

$$\mathbf{P}(T) := \begin{bmatrix} -\mathbf{L}_{[T,F]}\mathbf{L}_{[F,F]}^{-1} & \mathbf{I}_T \end{bmatrix}.$$

We now review some useful properties about the matrix projection $\mathbf{P}(T)$. Consider the Laplacian system $\mathbf{Lx} = \mathbf{b}$, where $\mathbf{L}$ is partitioned into block-matrices as in Equation (1). This in turn partitions the solution vector into non-terminals and terminals, i.e., $\mathbf{x} = \begin{bmatrix}\mathbf{x}_F & \mathbf{x}_T\end{bmatrix}^\top$.

Lemma 2.6. *Let $\mathbf{x}_T$ be a solution vector such that $\mathbf{SC}(G,T)\mathbf{x}_T = \mathbf{P}(T)\mathbf{b}$. Then there exists an extension $\mathbf{x}$ of $\mathbf{x}_T$ such that $\mathbf{L}\mathbf{x} = \mathbf{b}$.*

The following lemma draws a connection between the projection matrix and certain random walk probabilities which will allow us to take a combinatorial view on several problems we deal with.

Lemma 2.7. *Consider a graph $G = (V, E)$. For any subset of vertices $T \subseteq V$, a vertex $v \in T$, and a vertex $u \in F = V \setminus T$, let $\mathbb{P}_u\left[t_v < t_{T\setminus v}\right]$ be the probability that the random walk that starts at u hits v before hitting any other vertex from $T \setminus v$. Then we have that*

$$\left[\mathbf{P}(T)\mathbf{1}_u\right](v) = \mathbb{P}_u\left[t_v < t_{T\setminus v}\right].$$

In fact, $\{\mathbf{P}(T)\mathbf{1}_u\}_{v\in T}$ is a probability distribution for any fixed vertex $v \in F$.

Given a demand vector $\mathbf{b} \in \mathbb{R}^n$, we say that $\mathbf{P}(T) \cdot \mathbf{b}$ is the projection of $\mathbf{b}$ onto T. In general, the projection of $\mathbf{b}$ is shorter than the original vector $\mathbf{b}$. However, for the sake of exposition, often we consider $\mathbf{P}(T) \cdot \mathbf{b}$ to be an n-dimensional vector by assuming that all coordinates in F are 0.

Lemma 2.8. *Consider a graph $G = (V, E)$. Let $T \subseteq V$ be a subset of vertices, and let $u \in F = V \setminus T$. Consider the demand vector $\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u$ that requests to send one unit of flow from u to T according to the probability distribution $\{\mathbf{P}(T)\mathbf{1}_u\}_{v\in T}$. Then the minimum energy needed to route this demand is given by*

$$\|\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u\|^2_{\mathbf{L}^\dagger} = (\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u)^\top \mathbf{L}^\dagger (\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u).$$

3 Overview

The core building block of our algorithms is a fast routine that generates and maintains an approximate Schur complement onto a set of terminals T of our choice under insertion and deletions of edges as well as terminal additions, with all of these operations being supported in sub-linear time. One of the key ideas is to view to the Schur complement as a sum of random walks, and then observe that sampling exactly one walk per edge in the original graph already yields the desired object. Concretely, we build upon ideas introduced in sparsifying random walk polynomials [14], and Schur complements [40, 19] to show that it suffices to keep a union of these walks. The following result is implicit in these works, and we will review its proof in the full version of this paper.

Theorem 3.1. *Let $G = (V, E, w)$ be an undirected, weighted multi-graph with a subset of vertices T. Furthermore, let $\epsilon \in (0, 1)$, and let ρ be some parameter related to the concentration of sampling given by*

$$\rho = O\left(\log n\epsilon^{-2}\right).$$

Let H be an initially empty graph, and for every edge $e = (u, v)$ of repeat ρ times the following procedure:

1. *Simulate a random walk starting from u until it* first *hits T at vertex t_1,*
2. *Simulate a random walk starting from v until it* first *hits T at vertex t_2,*
3. *Combine these two walks (including e) to get a walk $u = (t_1 = u_0, \ldots, u_\ell = t_2)$, where ℓ is the length of the combined walk.*

4. Add the edge (t_1, t_2) to H with weight

$$1/\left(\rho \sum_{i=0}^{\ell-1} \left(1/\boldsymbol{w}_{u_i u_{i+1}}\right)\right)$$

The resulting graph H satisfies $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, T)$ with high probability.

The output approximate Schur complement of H onto T has up to $\tilde{O}(m\epsilon^{-2})$ edges, and thus is very dense to be leveraged as a sparsifier for our applications. Fortunately, there already exist efficient dynamic spectral sparsifiers, and we can always afford to keep a sparsifier $\tilde{H}$ of H whose size is only $\tilde{O}(|T|\epsilon^{-2})$.

The performance of our data structure depends on how fast we can generate the random walks used to create H. Note that even on the length n path with terminals T concentrated on one end, the lengths of these walks may be as long as $\Omega(n^2)$. To overcome this we shorten the walks by augmenting T with roughly $O(\beta m)$ random vertices from a carefully chosen distribution. This random augmentation of T ensures that any vertex v in G is roughly $O(\beta^{-1})$ apart from a vertex in T, and then our problem reduces to understanding the rate at which a random walk spreads among *distinct* edges. Concretely, our goal is to efficiently generate the first k distinct edges visited by a walk in G. We distinguish the following cases.

1. For unweighted graphs, we utilize a result by Barnes and Feige [6] that shows that with high probability a walk reaches k distinct edges in about k^2 steps.
2. For weighted graphs, we employ an event driven simulation of walks. Specifically, by computing the exit probability on the current set of edges visited so far, we sample the first k new edges reached by the walk in poly(k) time. Then, because we know the order that each edges is first reached, the first among them that belongs to T gives the intersection of the walk with T.

Following Point (1), our dynamic Schur complement data-structure H with respect to a randomly augmented T is initialized by generating for each edge $e \in E$, ρ random walk of length roughly β^{-2}. This operation costs roughly $O(m\beta^{-2})$. We then make the observation that the ability to add terminals into T means we only need to consider insertions/deletions between vertices in T. Specifically, for each affected edge we append its endpoints to T. A further advantage of this approach is that additions to T only shorten random walks in H, and the cost of shortening or truncating these random walks in H can be charged to the cost of constructing them during the initialization. Thus, it follows that we can support terminal additions, and thus insert or delete edges in $O(1)$ amortized time. Maintaining a sparsifier $\tilde{H}$ of H introduces only polylogarithmic overheads, so this step does not affect much our running times. We next discuss the applicability of this result.

The data-structure we presented readily gives a *sub-linear* dynamic Laplacian solver for the case where $\mathbf{b}$ has small support, namely fewer than βm vertices of $\mathbf{b}$ are non-zero. This can be accomplished by simply appending the entries of $\mathbf{b}$ (more precisely, their corresponding vertices) to the Schur complement H, and solving the system on H upon receipt of an index query. The resulting solution vector can then be lifted back to the original Laplacian using Lemma 2.6. However, note that our data-structure can only support up to $O(|T|) = O(\beta m)$ operations if we want to keep the the size of H small. Thus, to limit the growth in $|T|$ we periodically rebuild the entire data structure (i.e., we resample the set of new terminals completely) after βm operations, which in turn

gives an amortized update time of $O(m\beta^{-2}/(\beta m)) = O(\beta^{-3})$. Combining this with the bound of $O(\beta m)$ on the query time we obtain the following trade-off

$$\tilde{O}(\beta^{-3} + \beta m),$$

which is minimized when $\beta = m^{-1/4}$, thus giving an update and query time of $O(m^{3/4})$.

So it remains to address the case where $\mathbf{b}$ has a large number of non-zero entries. We overcome this difficulty by projecting this vector onto the current set of terminals T using the matrix $\mathbf{P}(T)$ and analyzing the error incurred by this projection. Our main observation is that the standard notion of error in Laplacian solvers, namely the $\mathbf{L}$-norm, corresponds to energies of electrical flows. This allows us to incur error in some of the $\mathbf{b}(u)$ values and then bound the energy of fixing them. To find such flows, we once again consider our problem from a random walk perspective, namely we view the projection of $\mathbf{b}$ onto T being equivalent to moving $\mathbf{b}$ around via random walks (Lemma 2.7). As such walks are short on unweighted graphs, we can relate their energies to the length of the walks times $\mathbf{b}(u)^2$ (Lemma 2.8).

One final obstacle is that if we move some vertex u from outside of T into T, the walks affected may be from multiple $\mathbf{b}(u)$s. To address this, we bound the 'load' of a vertex, defined as the number of walks that go through it, by the total length of the walks. The latter follows from the uniform distribution of random walks being stationary. Thus, as long as we picked T so that all the entries in $V \setminus T$ have small magnitudes, each move of some u into T incurs some small error. Bounding the accumulation of such errors, and rebuilding appropriately gives the overall dynamic solver result.

One application of the dynamic Laplacian solver is that we can maintain the energy of electrical flow for routing $\mathbf{b}$. This can also be viewed as an extension of our dynamic effective resistances data-structure, which can only maintain the energy of electrical flows for $\mathbf{b}$ with two non-zeros. Some further extensions in this direction that we believe would be useful are providing implicit access to the dual electrical flows, as well as finding the k largest entries either in the flow edges or the solution vector $\mathbf{x}$. However, such extensions will likely require a better understanding of the graph sparsifier component [2], which is treated as a black box in this paper.

For dynamically maintaining effective resistance in unweighted graphs, we essentially follow the same approach as with the dynamic solver for small support demand vectors, and thus obtain a running time of $O(m^{3/4})$ on both update and query time. For weighted graphs, we employ the weighted dynamic Schur complement algorithm (following Point(2)) which gives slightly weaker guarantees, namely a bound of $\tilde{O}(m^{5/6})$ on the update and query time. Interestingly, this weighted version has another immediate advantage; by running the data-structure on the output of a dynamic spectral sparsifier (Lemma 2.4), we obtained a bound of $\tilde{O}(n^{5/6})$ per operation, which is truly sub-linear irrespective of graph density.

4 Dynamic Schur Complement

In this section we show how to dynamically maintain approximate Schur complements. We first restrict our attention to unweighted graphs (i.e., prove Lemma 1.1), and then discuss how this result extends to the weighted case. We also present one of the applications of our data-structure, namely dynamic maintenance of effective resistance on unweighted (Theorem 1.3).

4.1 Dynamic Schur Complement on Unweighted Graphs

In this section we design a data-structure for maintaining approximte Schur complements under the assumption that the dynamic graph remains unweighted throughout the updates. Specifically, we have the following lemma.

Lemma 4.1 (Restatement of Lemma 1.1)**.** *Given an error threshold $\epsilon > 0$, an unweighted undirected multi-graph $G = (V, E)$ with n vertices, m edges, a subset of terminal vertices T' and a parameter $\beta \in (0, 1)$ such that $|T'| = O(\beta m)$, there is a data-structure* DynamicSC(G, T', β) *for maintaining a graph $\tilde{H}$ with $\mathbf{L}_{\tilde{H}} \approx_\epsilon \mathbf{SC}(G, T)$ for some T with $T' \subseteq T$, $|T| = O(\beta m)$, while supporting $O(\beta m)$ operations in the following running times:*

- Initialize(G, T', β)*: Initialize the data-structure, in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$ expected amortized time.*
- Insert(u, v)*: Insert the edge (u, v) to G in $\tilde{O}(1)$ amort. time.*
- Delete(u, v)*: Delete the existing edge (u, v) from G in $\tilde{O}(1)$ amortized time.*
- AddTerminal(u)*: Add u to T' in $\tilde{O}(1)$ amortized time.*

Our guarantees hold against an oblivious adversary.

To prove the lemma above, we first review the interpretation of Schur Complements using random walks, and then discuss how to generate and maintain these walks under edge updates and addition of terminal vertices.

Given a graph $G = (V, E)$ and a subset of terminals T recall that $\mathbf{SC}(G, T)$ was defined using an algebraic expression that involved the Laplcian of G. However, since it is still unclear how to exploit this expression in the dynamic setting we instead take a different, more 'combinatorial', view on $\mathbf{SC}(G, T)$. Concretely, we will interpret $\mathbf{SC}(G, T)$ as a collection of random walks, each starting at an edge of G and terminating in T, as described in Theorem 3.1.

Let H be the output graph from the construction in Theorem 3.1. Recall that H is an approximate Schur Complement onto T that has up to $\rho m = \tilde{O}(m\epsilon^{-2})$ edges (that is, ρ for each edge in G, where $\rho = O(\log n \epsilon^{-2})$ is the sampling parameter). As we will next show, H does not change too much (in amortized sense) upon inserting or deleting an edge in G. We will be able to maintain H such that the distribution of H is the same as $H(G)$ of the current graph G. Therefore, we can maintain these changes using a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 2.4), and whenever a query comes, we answer it on $\tilde{H}$ in $\tilde{O}(|T|\,\epsilon^{-2}) = \tilde{O}(\beta m \epsilon^{-2})$ time.

While it is widely known how to generate random walks efficiently, we note that the length of the walks generated in Theorem 3.1 could be prohibitively large if T is being picked arbitrarily. To see this, recall our example where we considered a path of length n with terminals T being places in one end. The length of such random walks may be as long as $\Omega(n^2)$. To shorten these random walks, we augment T with a random subset of vertices. Coming back to the path example, βn uniformly random vertices will be roughly β^{-1} apart, and random walks will reach one of these βn vertices in about β^{-2} steps. Because G could be a multi-graph, and we want to support queries involving any vertex, we pick T as the end points of a uniform subset of edges. A case that illustrates the necessity of this choice is a path except one edge has n parallel edges. In this case it takes $\Theta(n)$ steps in expectation for a random walk to move away from the end points of that edge. This choice of T completes the definition of our data structure, which we summarize in Algorithm 1. The performance of our data structures hinge upon the properties of the generated random walks. We start by formalizing such a structure involving the set of the augmented terminals described above while parameterizing it with a more general probability β for including the endpoints of the edges.

Definition 4.2 (β-shorted walks)**.** Let G be an weighted, undirected multi-graph and $\beta \in (0, 1)$ a parameter. A collection of *β-shorted walks* W on G is a set of random walks created as follows:

1. Choose a subset of terminal vertices T, obtained by including the endpoints of each edge independently with probability at least β.

Algorithm 1: InitializeUnweighted(G, T', β)

Input : Unweighted graph G, set of vertices $T' \subseteq V$ such that $|T'| \leq O(m\beta)$, and $\beta \in (0, 1)$
Output: Approximate Schur Complement H and union of β-shorted walks W

Set $T \leftarrow T'$, $H \leftarrow (V, \emptyset)$ and $W \leftarrow \emptyset$
For each edge $e = (u, v)$ in G, let $T \leftarrow T \cup \{u, v\}$ with probability β
Let $\rho = O(\log n\epsilon^{-2})$ be the sampling overhead according to Theorem 3.1
for *each edge* $e = (u, v) \in E$ *and* $i = 1, \ldots, \rho$ **do**
    Generate a random walk $w_1(e, i)$ from u until $\Theta(\beta^{-1} \log n)$ different edges have been hit, it reaches T, or it has hit every edge in its component
    Generate a random walk $w_2(e, i)$ from v until $\Theta(\beta^{-1} \log n)$ different edges have been hit, it reaches T, or it has hit every edge in its component
    if *both walks reach* T *at* t_1 *and* t_2 *respectively* **then**
        Connect $w_1(e, i)$, e and $w_2(e, i)$ to form a walk $w(e, i)$ between t_1 and t_2
        Let $\ell \leftarrow \ell(w_1(e, i)) + \ell(w_2(e, i)) + 1$ be the length of $w(e, i)$
        Add an edge (t_1, t_2) with weight $1/(\rho\ell)$ to H
        Add $w(e, i)$ to W
return H and W

2. For each edge $e \in E$, generate $\rho = O(\log n\epsilon^{-2})$ walks from its endpoints either until $\Omega(\beta^{-1} \log n)$ different edges have been hit, they reach T, or they visited each edge that is in the same connected component as e.

As we will shortly seee, the main property of the collection W is that its random walks are short. Moreover, we will also prove that all walks in W will reach T with high probability. These guarantees are summarized in the following theorem.

Theorem 4.3. *Let $G = (V, E)$ be any undirected multi-graph, and $\beta \in (0, 1)$ a parameter. Any set of β-shorted walks W, as described in Definition 4.2, satisfies the following:*

- *With high probability, any random walk in W starting in a connected component containing a vertex from T terminates at a vertex in T.*

Note that Theorem 4.3 is conditioned upon the connected component having a vertex in T: this is necessary because walks stay inside a connected component. However, this does not affect our queries: our data-structure has an operation for making any vertex u a terminal, which we call during each query to ensure both s and t are terminal vertices. Such an operation interacts well with Theorem 4.3 because it can only increase the probability of an edge's endpoints being chosen.

Proving the theorem requires to determine the rate at which a random walk visits at least $\beta^{-1} \log n$ edges. It turns out that a random walk of length $\tilde{O}(\beta^{-2})$ is highly likely to achieve this. For formally showing this, we need the following result by Barnes and Feige [6].

Theorem 4.4 ([6], Theorem 1.2)**.** *There is an absolute constant c_{BF} such that for any undirected unweighted connected multi-graph G with n vertices and m edges, any vertex u and any value $\hat{m} \leq m$, the expected time for a random walk starting from u to visit at least $\hat{m}$* distinct *edges is at most $c_{BF}\hat{m}^2$.*

The above theorem can be amplified into a with high probability bound by repeating the walk $O(\log n)$ times.

Corollary 4.5. *In any undirected unweighted connected multi-graph G with m edges, for any starting vertex u, any length $\ell \leq m^2$, and a parameter $\delta \geq 1$, a walk of length $c_{BF} \cdot \delta \cdot \ell \log n$ from u visits at least $\ell^{1/2}$* distinct *edges with probability at least $1 - n^{-\delta}$.*

Proof. We can view each such walk as a concatenation of $\delta \log n$ sub-walks, each of length $2 \cdot c_{BF} \cdot \ell$.

We call a sub-walk *good* if the number of distinct edges that it visits is at least $\ell^{1/2}$. Applying Markov's inequality to the result of Theorem 4.4, a walk takes more than $2 \cdot c_{BF} \cdot \ell$ steps to visit $\ell^{1/2}$ distinct edges with probability at most $1/2$.

This means that each subwalk fails to be good with probability at most $1/2$. Thus, the probability that all subwalks fail to be good is at most $2^{-\delta \log n} = n^{-\delta}$. The result then follows from an union bound over all starting vertices $u \in V$. □

We now have all the tools to prove Theorem 4.3.

Proof of Theorem 4.3. For any walk w, we define $V(w)$ (respectively, $E(w)$) to be the set of distinct vertices (respectively, edges) that a walk w visits. Consider a random walk w that starts at u of length

$$\ell = c_{BF} \cdot \delta^3 \cdot \beta^{-2} \log^3 n$$

where δ is a constant related to the success probability.

If the connected component containing the walk has fewer than

$$\delta \cdot \beta^{-1} \cdot \log n$$

edges, then Corollary 4.5 gives that we have covered this entire component with high probability, and the guarantee follows from the assumption that this component contains a vertex of T.

Otherwise, we will show that w reached enough edges for one of their endpoints to be picked to be picked into T with high probability. The key observation is that because w is generated independently from T, we can bound the probability of this walk not hitting T by first generating w, and then T. Specifically, for any size threshold z, we have

$$\begin{aligned} \mathbb{P}_{T,w}\left[V\left(w\right) \cap T = \emptyset\right] &= \mathbb{P}_{w,T}\left[V\left(w\right) \cap T = \emptyset\right] \\ &\leq \mathbb{P}_{w}\left[\left|E\left(w\right)\right| \leq z\right] \\ &\quad + \mathbb{P}_{w:|E(w)|\geq z,T}\left[V\left(w\right) \cap T = \emptyset\right]. \end{aligned} \tag{2}$$

By Corollary 4.5 and the choice of ℓ, if we set

$$z = \delta \cdot \beta^{-1} \cdot \log n,$$

then the first term in Equation (2) is bounded by $n^{-\delta}$. For bounding the second term, we can now focus on a particular walk $\widehat{w}$ that visits at least $\delta \cdot \beta^{-1} \cdot \log n$ distinct edges, i.e.,

$$\left|E\left(\widehat{w}\right)\right| \geq \delta \cdot \beta^{-1} \log n.$$

Recall that we independently added the end points of each of these edges into T with probability β. If any of them is selected, we have a vertex that is both in $V(\widehat{w})$ and T. Thus, the probability that T contains no vertices from $V(\widehat{w})$ is at most

$$(1-\beta)^{|E(\widehat{w})|} \leq (1-\beta)^{\delta \cdot \beta^{-1} \log n} \leq e^{-\delta \log n} \leq n^{-\delta},$$

which completes the proof. □

Algorithm 2: AddTerminal(u)

Input : Vertex u such that $u \notin T$

1 Set $T \leftarrow T \cup \{u\}$

2 Shorten all random walks in W at the first point they meet u

3 Update the corresponding edges in H and $\tilde{H}$

Corollary 4.5 together with Theorem 4.3 yield the following lemma.

Lemma 4.6. *Algorithm 1 runs in $\tilde{O}(m\beta^{-2}\epsilon^{-2})$ time and outputs a graph H with $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, T)$, with high probability.*

Proof. By Corollary 4.5, the length of each walk generated in Algorithm 1 is bounded by $O(\beta^{-2}\log^3 n)$. In addition, note that each step in a random walk can be simulated in $O(1)$ time. This is due to the fact that we can sample an integer in $[0, n-1]$ by drawing $x \in [0, 1]$ uniformly and taking $\lfloor xn \rfloor$. Combining these with the fact that the algorithm generates $\rho m = \tilde{O}(m\epsilon^{-2})$ walks, it follows that the running time of the algorithm is dominated by $\tilde{O}(m\beta^{-2}\epsilon^{-2})$.

Note that the collection of generated walks form the set W of β-shorted walks. By Theorem 4.3, with high probability, each of the walks that starts at a component containing a vertex in T hits T. Conditioning on the latter, Theorem 3.1 gives that with high probability, $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, T)$. □

Handling edge updates and terminal additions. We start by observing that there is always a one-to-one correspondence between the collection of β-shorted walks W and our approximate Schur complement H. Accordingly, our primary concern will be supporting the Insert, Delete, and AddTerminal operations in the collection W. However, as W undergoes changes, we need to efficiently update the sparsifier H. To handle these updates, we would ideally have efficient access to walks in W that are affected by the corresponding updates.

To achieve this, we index into walks that utilize a vertex or an edge, and thus set up a reverse data structure pointing from vertices and edges to the walks that contain them. The following lemma says that we can modify this representation with minimal cost.

Lemma 4.7. *For the collection of β-shorted walks W, let W_v and W_e be the specific walks of W that contain vertex v and edge e, respectively. We can maintain a data structure for W such that for any vertex v or edge e it reports*

- *all walks in W_v or W_e in $O(|W_v|)$ or $O(|W_e|)$ time, respectively,*

with an additional $O(\log n)$ overhead for any changes made to W.

Proof. For every vertex (respectively, edge), we can maintain a balanced binary search tree consisting of all the walks that use it in time proportional to the number of vertices (respectively, edges) in the walks. Supporting rank and select operations on such trees then gives the claimed bound. □

As a result, any update made to the collection of walks can be updated in the approximate Schur complement H generated from these walks in $O(\log n)$ time. We now have all the necessary ingredients to prove Lemma 1.1.

Proof of Lemma 1.1. We give a two-level data-structure for dynamically maintaining Schur complements on unweighted graphs, which keeps at any time a terminal set T of size $\Theta(m\beta)$. This entails maintaining

1. an approximate Schur complement H of G with respect to T (Theorem 3.1),

2. a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 2.4).

We implement the procedure Initialize by running Algorithm 1, which produces a graph H and then computing a spectral sparsifier $\tilde{H}$ of H using Lemma 2.4. Note that by construction of our data-structure, every update in H will be handled by the black-box dynamic sparsifier $\tilde{H}$.

As we will shortly see, operations Insert and Delete will be reduced to adding terminals to the set T. Thus, the bulk of our effort is devoted to implementing the procedure AddTerminal. Let u be a non-terminal vertex that we want to append to T. We set $T \leftarrow T \cup \{u\}$, and then shorten all the walks at the first location they meet u. This shortening of walks induces in turn edge insertions and deletions to H, which are then processed by $\tilde{H}$. The pseudocode for this operation is summarized in Algorithm 2. To quickly locate the first appearances of u in the random walks from W, we make use of the data-structure from Lemma 4.7. Let us first describe the construction of such data-structure during the preprocessing phase. Let W_u be the balanced binary search tree consisting of all the walks that use the vertex u in W. Fix $w \in W_u$. For any $t \geq 0$, if w visits u after t steps, we check whether W_u contains w or not. If the latter holds, we know that u has appeared before in w and we do not need to add w to W_u. Otherwise, we add w to W_u as this is the first time the walk w visits u. After locating the first appearances of u, we cut the walks in these locations, delete the corresponding affected walks (together with their weight from H), and insert the new shorter walks to H. Note that we can simply use arrays to represent each random walk in W.

For implementing operations Insert and Delete we proceed as follows. Specifically, upon insertion or deletion of an edge $e = (u, v)$ in G, we append both u and v to the terminal set T. Now, all the walks that pass through u or v in W must be shorten at the first location they meet u or v. For inserting an edge (u, v) with weight $\boldsymbol{w}_{uv}$ in G (in fact, Lemma 1.1 is restricted to $\boldsymbol{w}_{u,v} = 1$), we simply add ρ trivial random walks (i.e., the edge (u, v)) of weight $\frac{\boldsymbol{w}_{uv}}{\rho}$ to H (which sum up to the edge (u, v) itself). For deleting the edge (u, v) with weight $\boldsymbol{w}_{uv}$ from G, simply delete these ρ random walks between u and v in H (which exist since we guaranteed that u and v are added as terminals to H).

We next analyze the performance of our data-structure. Let us start with the pre-processing time. First, by Lemma 4.6 we get that the cost for constructing H on a graph with m edges is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-2})$. Next, since H has $\tilde{O}(m\epsilon^{-2})$ edges, constructing $\tilde{H}$ takes $\tilde{O}(m\epsilon^{-4})$ time. Thus, the amortized time of Initialize operation is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$.

Following the above discussion, for analyzing the cost of the update operations it suffices to bound the time for adding a vertex to T, which in turn (asymptotically) bounds the update time for edge insertions and deletions. The main observation we make is that adding a vertex to T only shortens the existing walks, and Lemma 4.7 allows us to find such walks in time proportional to the amount of edges deleted from the walk. Since the walk needed to be generated in the Initialize operation, the deletion of these edges takes equivalent time to generating them. Moreover, we note that (1) handling the updates in $\tilde{H}$ induced by H introduces additional $O(\text{poly}(\log n)\epsilon^{-2})$ overheads, and (2) adding or deleting ρ edges until the next rebuild costs $\tilde{O}(\beta m\epsilon^{-2})$, since we process only up to βm operations. These together imply that the amortized cost for adding a terminal can be charged against the pre-processing time, which is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$, up to poly-logarithmic factors. Thus, it follows that the operations AddTerminal, Insert and Delete can be implemented in $\tilde{O}(1)$ amortized update time. □

4.2 Dynamic All-Pairs Effective Resistance on Unweighted Graphs

In this section we present one of the applications of our dynamic Schur complement data structure for unweighted graphs. Concretely, we design a dynamic algorithm that supports an intermixed sequence of edge insertions, deletion and pair-wise resistance queries, and returns a $(1 \pm \epsilon)$-approximation to all the resistance queries.

We start by reviewing two natural attempts for solving this problem.

- First, since spectral sparsifiers preserve effective resistances (Lemma 2.5), we could dynamically maintain a spectral sparsifier (Lemma 2.4), and then compute the (s, t) effective resistance on this sparsifier. This leads to a data structure with $\text{poly}(\log n, \epsilon^{-1})$ update time and $\tilde{O}(n\epsilon^{-2})$ query time.
- Second, by the preservation of effective resistances under Schur complements (Fact 2.2), we could also utilize Schur complements to obtain a faster query time among a set of βm terminals, T, for some reduction factor $\beta \in (0, 1)$, at the expense of a slower update time. Specifically, after each edge update, we recompute an approximate Schur complement of the sparsifier onto T in time $\tilde{O}(m\epsilon^{-2})$ [18], after which each query takes $\tilde{O}(\beta m\epsilon^{-2})$ time.

The first approach obtains sublinear update time, while the second approach gives sublinear query time. Our algorithm stems from combining these two methods, with the key additional observation being that adding more vertices to T makes the Schur complement algorithm more local. Specifically, using Lemma 1.1 leads to a data-structure for dynamically maintaining all-pairs effective resistances.

Proof of Theorem 1.3. Let $\mathcal{D}(\tilde{H})$ denote the data structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G (Lemma 1.1). Since $\mathcal{D}(\tilde{H})$ supports only up to βm operations, we rebuild $\mathcal{D}(\tilde{H})$ on the current graph G after such many operations. Note that the operations Insert and Delete on G are simply passed to $\mathcal{D}(\tilde{H})$. For processing the query operation EffectiveResistance(s, t), we declare s and t terminals (using the operation AddTerminal of $\mathcal{D}(\tilde{H})$), which ensures that they are both now contained in $\tilde{H}$. Finally, we compute the (approximate) effective resistance between s and t in $\tilde{H}$ using Lemma 2.1.

We now analyze the performance of our data-structure. Recall that the insertion or deletion of an edge in G can be supported in $\tilde{O}(1)$ expected amortized time by $\mathcal{D}(\tilde{H})$. Since our data-structure is rebuilt every βm operations, and rebuilding $\mathcal{D}(\tilde{H})$ can be implemented in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$, it follows that the amortized cost per edge insertion or deletion is

$$\frac{\tilde{O}(m\beta^{-2}\epsilon^{-4})}{\beta m} = \tilde{O}(\beta^{-3}\epsilon^{-4}).$$

The cost of any (s, t) query is dominated by (1) the cost of declaring s and t terminals and (2) the cost of computing the (s, t) effective resistance to ϵ accuracy on the graph $\tilde{H}$. Since (1) can be performed in $\tilde{O}(1)$ time, we only need to analyze (2). We do so by first giving a bound on the size of T. To this end, note that each of the m edges in the current graph adds two vertices to T with probability β independently. By a Chernoff bound, the number of random augmentations added to T is at most $2\beta m$ with high probability. In addition, since $\mathcal{D}(\tilde{H})$ is rebuilt every βm operations, the size of T never exceeds $4\beta m$ with high probability. The latter also bounds the size of $\tilde{H}$ by $\tilde{O}(\beta m\epsilon^{-2})$ and gives that the query cost is $\tilde{O}(\beta m\epsilon^{-4})$.

Combining the above bounds on the update and query time, we obtain the following trade-off

$$\tilde{O}\left((\beta m + \beta^{-3})\epsilon^{-4}\right),$$

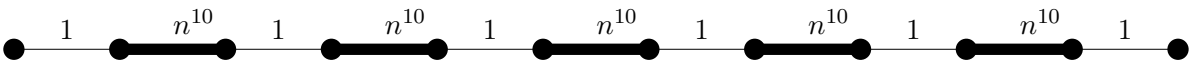

Figure 1: A weighted graph on which a random walk takes a long time to hit new edges

which is minimized when $\beta = m^{-1/4}$, thus giving an expected amortized update and query time of

$$\tilde{O}(m^{3/4}\epsilon^{-4}).$$ □

4.3 Dynamic Schur Complement on Weighted Graphs

In this section we present an extension of Lemma 1.1 to weighted graphs while slightly increasing the running time guarantees. Concretely, we prove the following lemma.

Lemma 4.8. *Given an error threshold $\epsilon > 0$, a weighted, undirected multi-graph $G = (V, E, \boldsymbol{w})$ with n vertices, m edges, a subset of terminal vertices T' and a parameter $\beta \in (0, 1)$ such that $|T'| = O(\beta m)$, there is a data-structure* WeightedDynamicSC(G, T', β) *for maintaining a graph $\tilde{H}$ with $\tilde{H} \approx_\epsilon \mathbf{SC}(G, T)$ for some T with $T' \subseteq T$, $|T| = O(\beta m)$, while supporting $O(\beta m)$ operations in the following running times:*

- Initialize(G, T', β)*: Initialize the data-structure in $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ expected amortized time.*
- Insert(u, v, w)*: Insert the edge (u, v) with weight w to G in $\tilde{O}(1)$ amortized time.*
- Delete(u, v)*: Delete the existing edge (uf, v) from G in $\tilde{O}(1)$ amortized time.*
- AddTerminal(u)*: Add u to T' in $\tilde{O}(1)$ amortized time.*

While the extension of our data-structure to weighted graphs builds upon the ideas we used in the unweighted case, there are a few obstacles that force us to introduce new components in our algorithm in order to make such an extension feasible. To illustrate, consider the weighted graph shown in Figure 1 and recall that the running time our data-structure depends on the speed at which random walks visit distinct edges in a graph. Due to the structure of the edge weights, a random walk in this graph is expected to take $\Theta(n^{10})$ steps before hitting four different edges. This shows that the naive generation of random walks in weighted graphs may be computationally prohibitive for our purposes.

Fast Generation of Random Walks in Weighted Graphs. To rectify the above issue, we make the important observation that it is not necessary to keep information for every single step of a random walk. Instead, it would suffice if we could efficiently determine the step at which the walk meets a new vertex along with the corresponding weight associated with the walk, which defines the edge weight that is added to the sparsifier. This high-level idea allows us to generate random walks much faster, and we next make this more precise.

Following the notation we used in the unweighted case, for an arbitrary vertex $v \in V$, a set of terminals $T \subseteq V$ and a parameter $\beta \in (0, 1)$, a *β-shorted walk* with respect to v and T is a random walk that starts at a given vertex $v \in V$ and halts whenever $\Omega(\beta^{-1} \log n)$ *different* vertices have been hit, it reaches a vertex in T, or it has hit every edge in the connected component containing v. The main contribution of this section is summarized in the following lemma.

Lemma 4.9. *Let $G = (V, E, \boldsymbol{w})$ be an undirected, weighted graph with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$, where c is a positive constant. Let $T \subseteq V$ be a set of terminals and $v \in V$ be an arbitrary vertex.*

Then there is an algorithm that generates a β-shorted random walk with respect to v and T and approximates its corresponding weight up to a $\pm\epsilon$ relative error in $\tilde{O}(\beta^{-4}\epsilon^{-2})$ time.

We first give an intuition behind the algorithm in the above lemma and also describe how this algorithm interacts with other parts of our dynamic data-structure. Let $w = (w_0, \ldots, w_t)$ be a random walk that starts at an endpoint of an edge, and define

$$s(w) := \sum_{i=1}^{t} \frac{1}{\boldsymbol{w}_{w_{i-1}w_i}}, \tag{3}$$

to be its corresponding weight. Recall that before adding the walk w to H, we must scale it proportionally to $1/s(w)$ (Theorem 3.1). Observe that throughout our dynamic algorithm, the only modification we might do to w is to truncate it at the first location it meets a new vertex u that is being declared a terminal. Moreover, after this modification, note that the old value of $s(w)$ is no longer valid and we need to extract $s(w)$ that corresponds to the new walk. To allow efficient access to such information, we can view the walk w as being split into sub-walk segments by the first locations w meets new vertices and store the weights of each such sub-walks. As we will next see, this bookkeeping alone allows us to proceed with the same algorithm as in the unweighted case.

We next give the three main components for implementing the algorithm stated in Lemma 4.9.

(A) Sample the number of steps needed for a random walk w to visit a new vertex.

(B) Sample a new *distinct* vertex that w hits, and its corresponding edge.

(C) Sample the (approximate) weight of a random walk between two given vertices.

After describing each of them, we will see that their combination naturally leads to our desired result.

Let us first discuss (A). For any $t \geq 0$, consider a t step random walk w and let $U = \{w_0, \ldots, w_t\}$ be the set of *distinct* vertices that w has visited up to step t. Define $u := w_t \in U$ to be the *current* vertex of the walk w. Our goal to to efficiently sample the number of steps the walk w needs to visit a vertex not in U. To this end, we start by introducing some useful notation. For any $i \geq 0$, let $p^{\text{new}}(i)$ be the probability that w meets a new vertex that is not in U in $w_{t+1}, \ldots, w_{t+i}$. For $v \in U$, let $\boldsymbol{p}_i(v)$ be the probability that $w^{(t+i)} = v$, conditioned on w not having met any new vertex in $w_{t+1}, \ldots, w_{t+i-1}$. Then it can be easily verified that both $p^{\text{new}}(i)$ and $\boldsymbol{p}_i(v)$ are just linear combinations of $p^{\text{new}}(i)$ and $\boldsymbol{p}_{i-1}(v)$

$$\boldsymbol{p}_i(v) = \sum_{u \in U} \left(\boldsymbol{p}_{i-1}(u) \cdot \frac{\boldsymbol{w}_{uv}}{\boldsymbol{d}_u} \right), \quad \forall v \in U. \tag{4}$$

$$p^{\text{new}}(i) = p^{\text{new}}(i-1) + \sum_{u \in V \setminus U} \sum_{v \in U} \left(\boldsymbol{p}_{i-1}(v) \cdot \frac{\boldsymbol{w}_{vu}}{\boldsymbol{d}_v} \right). \tag{5}$$

Next, using the linearity of the recurrences in (5) and (4) we can find a matrix $\boldsymbol{W}$ of dimension $(k+1) \times (k+1)$, where $k = |U|$, satisfying the following equality

$$\begin{bmatrix} \boldsymbol{p}_i \\ p^{\text{new}}(i) \end{bmatrix} = \boldsymbol{W} \cdot \begin{bmatrix} \boldsymbol{p}_{i-1} \\ p^{\text{new}}(i-1) \end{bmatrix}, \quad \forall i \geq 1. \tag{6}$$

The main advantage introducing such a matrix is that it allows us to efficiently compute $p^{\text{new}}(i)$ and $\boldsymbol{p}_i$ using fast exponentiation via repeated squaring. Specifically, let $\boldsymbol{p}_0$ be a unit vector of dimension k, where for the current vertex u of the walk w we have that $\boldsymbol{p}_0(u) = 1$, and 0 otherwise.

Algorithm 3: BINARYSEARCH($\boldsymbol{W}, \hat{\boldsymbol{p}}_0, M$)

Input : A $(k+1) \times (k+1)$ matrix $\boldsymbol{W}$, a $(k+1)$ dimensional vector $\hat{\boldsymbol{p}}_0$, and an integer M
Output: An integer

Set $\ell \leftarrow 0$, $r \leftarrow M$, $\ell p \leftarrow 0$ and $rp \leftarrow 1$
while $(\ell \neq r)$ **do**
  Set $\eta \leftarrow \lfloor (\ell + r)/2 \rfloor$
  Compute $\hat{\boldsymbol{p}}_\eta = \boldsymbol{W}^\eta \hat{\boldsymbol{p}}_0$ using Lemma 4.10
  Set $p^{\text{new}}(\eta) = \hat{\boldsymbol{p}}_\eta(k+1)$
  With probability $(p^{\text{new}}(\eta) - \ell p)/(rp - \ell p)$, set $r \leftarrow \eta$, and $rp = p^{\text{new}}(\eta)$, otherwise, with probability $(rp - p^{\text{new}}(\eta))/(rp - \ell p)$, set $\ell \leftarrow \eta + 1$, $\ell p = p^{\text{new}}(\eta)$
return ℓ

Let $\hat{\boldsymbol{p}}_0 = [\boldsymbol{p}_0 \quad p^{\text{new}}(0)]^\top$ be the extended $k+1$ dimension vector, where $p^{\text{new}}(0) = 0$. For any $i \geq 1$, repeatedly applying Equation (6) and letting $\hat{\boldsymbol{p}}_i := \boldsymbol{W}^i \hat{\boldsymbol{p}}_0$ yields

$$\hat{\boldsymbol{p}}_i(v) = \boldsymbol{p}_i(v), \ \forall v \in U \quad \text{and} \quad \hat{\boldsymbol{p}}_i(k+1) = p^{\text{new}}(i). \tag{7}$$

Using the above relation, we can use fast exponentiation via repeated squaring to compute $p^{\text{new}}(i)$ in $O(k^3 \log(i))$ time. This follows directly from the following well-known lemma, which we will exploit in a few other places throughout this work.

Lemma 4.10. *Let A be a matrix of dimension $n \times n$, and A^i denote the i-th power of A, for any $i \geq 1$. Then there is an algorithm that computes A^i in $O(n^3 \log(i))$ time.*

We now have all the tools to describe the sampling procedure for computing the number of steps that the walk needs to visit a vertex that is distinct from the vertices in U. We accomplish this using a "binary search"-inspired subroutine, which works as follows. As an input, our algorithm is given a $(k+1) \times (k+1)$ matrix $\boldsymbol{W}$ (as defined in Equation (6)), the vector $\hat{\boldsymbol{p}}_0$, and an integer M, which is an upper-bound on the cover time of G. The algorithm also maintains variables $\ell, r, \ell p, rp$ with the following initialization $\ell \leftarrow 0$, $r \leftarrow M$, $\ell p \leftarrow 0$ and $rp \leftarrow 1$. As long as $(\ell \neq r)$, it defines the average $\eta = \lfloor (\ell + r)/2 \rfloor$ and then proceeds to compute $\hat{\boldsymbol{p}}_\eta = W^\eta \hat{\boldsymbol{p}}_0$ using Lemma 4.10. Note that $p^{\text{new}}(\eta) = \hat{\boldsymbol{p}}_\eta(k+1)$ by Equation (7). Finally, the algorithm uses $p^{\text{new}}(\eta)$ to randomly decide whether w meets a new vertex in the next η steps or not. In other words, it updates the maintained variables using the rule below:

1. with probability $(p^{\text{new}}(\eta) - \ell p)/(rp - \ell p)$, set $r \leftarrow \eta$, and $rp = p^{\text{new}}(\eta)$,
2. otherwise, with probability $(rp - p^{\text{new}}(\eta))/(rp - \ell p)$, set $\ell \leftarrow \eta + 1$, $\ell p = p^{\text{new}}(\eta)$.

If $(\ell = r)$, then the algorithm returns ℓ. This procedure is summarized in Algorithm 3.

We next show the correctness of the above procedure. To do so, we first need the following notation. For a t-step random walk w and a current vertex $u = w_t \in U$, let $X(u, U)$ be the smallest number of steps of steps needed for w to visit a vertex not in U, i.e., $X(u, U) = \min\{i \mid i \geq 1, \ w_{t+i} \notin U\}$. Note that $X(u, U)$ is a random variable, and $X(u, U) \leq M$ by definition of M.

Lemma 4.11. *Let w be a t-step random walk and U, $u = w_t \in U$, and $k = |U|$ be the number of distinct vertices w has visited so far. For $\boldsymbol{W}$, $\hat{\boldsymbol{p}}_0$, and M defined as above,* BINARYSEARCH($\boldsymbol{W}, \hat{\boldsymbol{p}}_0, M$) *correctly samples $X(u, U)$, i.e., the number of steps w needs to visit a vertex not in U, in $O(k^3 \log^2 M)$ time.*

Proof. By Equation (7) and Line 4 in Algorithm 3, note that $p^{\text{new}}(\eta)$ is the probability that w meets a new vertex in the fist η steps. The correctness of BinarySearch can be proven using an inductive argument on the number of iterations of the while loop. Here, we just show the crucial parts for being able to apply such an argument. First, observe that right after Line 2 in the while loop, we have that

$$(rp - \ell p) = \mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq r\right].$$

The latter holds because $\ell p = \mathbb{P}_{X(u,U)}\left[X(u,U) \leq \ell\right]$ and $rp = \mathbb{P}_{X(u,U)}\left[X(u,U) \leq r\right]$, which in turn can be verified for each assignment of ℓ and r. Next, we prove that conditioning on $\ell \leq X(u,U) \leq r$ right after Line 2 in the while loop, Algorithm 3 samples $X(u,U)$ from the correct distribution. This is true when $(\ell = r)$, since the condition of the while loop is no longer satisfied and the algorithm returns ℓ. If, however $(\ell \neq r)$, then we need to compute the following probabilities: (1) $\mathbb{P}_{X(u,U)}\left[X(u,U) \leq \eta \mid \ell \leq X(u,U) \leq r\right]$ and (2) $\mathbb{P}_{X(u,U)}\left[X(u,U) > \eta \mid \ell \leq X(u,U) \leq r\right]$. To determine (1), we get that

$$\begin{aligned}
\mathbb{P}_{X(u,U)}\left[X(u,U) \leq \eta \mid \ell \leq X(u,U) \leq r\right] &= \frac{\mathbb{P}_{X(u,U)}\left[(X(u,U) \leq \eta) \wedge (\ell \leq X(u,U) \leq r)\right]}{\mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq r\right]} \\
&= \frac{\mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq \eta\right]}{\mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq r\right]} \\
&= \frac{(p^{\text{new}}(\eta) - \ell p)}{(rp - \ell p)}.
\end{aligned}$$

The probability from case (2) can be shown similarly. Since Line 5 in Algorithm 3 updates the search boundaries ℓ and r and their corresponding values ℓp and rp using probabilities (1) and (2), the correctness of the algorithm follows.

For the running time, observe that the number of iterations until the condition of the while loop is no longer satisfied is bounded by $O(\log M)$. Moreover, the running time of one iteration is dominated by the time needed to compute $\boldsymbol{W}^{\eta}\hat{\boldsymbol{p}}_0$. Since $\boldsymbol{W}$ is a $(k+1)\times(k+1)$ dimensional matrix and $\eta \leq M$, Lemma 4.10 implies that the matrix powering step can be computed in $O(k^3 \log M)$. Thus, it follows that Algorithm 3 can be implemented in $O(k^3 \log^2 M)$ time. □

We next explain how to sample a new distinct vertex, and its corresponding edge of a t-step random walk w, i.e., we discuss component (B). Let $X(u,U)$ be the index computed by BinarySearch routine. We first compute the probability dsitribution $\boldsymbol{q}$ over vertices in U after performing the next $(X(u,U)-1)$ steps of the random walk w, conditioning on w not leaving U. Next, we proceed to computing the probability distribution $\boldsymbol{r}$ over the edges in in $(U, V \setminus U)$ conditioning on $w_0, \ldots, w_t$ and $w_{t+X(u,U)}$ being the first vertex not in U. Formally, for $v \in U$, $z \in V \setminus U$, we have

$$\boldsymbol{r}(v,z) = \frac{\boldsymbol{q}(v)\boldsymbol{w}_{vz}}{R}, \text{ where } R := \sum_{v \in U, z \in V \setminus U} \boldsymbol{q}(v)\boldsymbol{w}_{vz}. \tag{8}$$

Finally, we sample $(w_{t+X(u,U)-1}, w_{t+X(u,U)})$ according to $\boldsymbol{r}$, where $w_{t+X(u,U)}$ is the first vertex not in U. The lemma below shows that we can efficiently sample from $\boldsymbol{r}$.

Lemma 4.12. *Let w be a t-step random walk and let U with $k = |U|$ be the set of distinct vertices w has visited so far. Given the number of steps $X(u,U)$ needed for w to visit a vertex not in U, there exists an algorithm that samples an edge leaving U, and the first vertex not in U in $O(k^3 \log M)$ time.*

Proof. We start by showing how to compute the distribution $\boldsymbol{q}$. To this end, recall that $\boldsymbol{p}_i(v)$ is the probability that $w_{t+i} = v$, conditioned on w not having met any vertex different from U in $w_{t+1}, \ldots, w_{t+i-1}$. Thus, by Equation (6), we can use the fast exponentiation routine (Lemma 4.10) to compute the vector $\hat{\boldsymbol{p}}_{X(u,U)-1} = \boldsymbol{W}^{X(u,U)-1}\hat{\boldsymbol{p}}_0$. Since by Equation (7) we have that $\hat{\boldsymbol{p}}_{X(u,U)-1}(v) = \boldsymbol{p}_{X(u,U)-1}(v)$ for each $v \in U$, it follows that $\boldsymbol{q}(v)$ is exactly $\boldsymbol{p}_{X(u,U)-1}(v)$. Note that the running time for implementing this step is $O(k^3 \log M)$ as $X(u,U) \leq M$.

We next describe how to efficiently sample from the distribution $\boldsymbol{r}$. First, it will be helpful to to sample a vertex $v \in U$ conditioning on the on $w_{t+X(u,U)}$ being the first vertex not in U. Specifically, we are interested in sampling a vertex $v \in U$ with probability

$$\frac{\boldsymbol{q}(v) \cdot \boldsymbol{w}(v, V \setminus U)}{R}, \text{ where } \boldsymbol{w}(v, V \setminus U) := \sum_{z \in V \setminus U} \boldsymbol{w}_{vz}. \tag{9}$$

For being able to efficiently sample from this distribution, we need to compute $\boldsymbol{w}(v, V \setminus U)$, which in turn may require examining up to $\Omega(n)$ edges incident to v. However, this is not sufficient for our purposes as our ultimate goal is to sample from $\boldsymbol{r}$ in time only proportional to k. To alleviate this, observe that $\boldsymbol{w}(v, V \setminus U) = (\boldsymbol{d}(v) - \sum_{z \in U} w_{v,z})$. Thus, maintaining *weighted* degree $\boldsymbol{d}(v)$ for each $v \in V$, allows us to compute $\boldsymbol{w}(v, V \setminus U)$ in $O(k)$ time. Similarly, rearranging the sums in the definition of R we get

$$R = \sum_{v \in U,\ z \in V \setminus U} \boldsymbol{q}(v)\boldsymbol{w}_{v,z} = \sum_{v \in U} \left(\boldsymbol{q}(v) \cdot \boldsymbol{w}(v, V \setminus U)\right),$$

which in turn implies that R can be computed in $O(k^2)$ time. The latter gives that the distribution defined in Euqation (9) can be computed in $O(k^2)$ time. For sampling a vertex $v \in U$ from this distribution we simply generate a uniformly-random value $x \in [0, 1]$, and then perform binary search on the prefix sum array of the probability distribution. Since computing the prefix sum array and performing binary search can be done in $O(k)$ and $O(\log n)$ time, respectively, we get that sampling $v \in U$ according to distribution defined in Equation (9) can be performed in $O(k^2)$ time.

We next explain how to sample an edge (v, z), where $z \in V \setminus U$ and $v \in U$ is the vertex we sampled from above. The probability distribution from which (v, z) is sampled is as follows

$$\frac{\boldsymbol{w}_{v,z}}{\boldsymbol{w}(v, V \setminus U)}. \tag{10}$$

To see the idea behind this choice, note that Equation (10) combined with Equation (9) yields the distribution $\boldsymbol{r}$ as defined in Equation (8), which ensures that the edge is sampled correctly. However, one complication we face with is that v may be incident to $\Omega(n)$ edges. Remember that for sampling an edge one needs access to the prefix sum array, which is expensive for our purposes. A natural attempt is to compute such an array during preprocessing. Nevertheless, this alone does not suffice as the set U will change over the course of our algorithm. Instead, for every vertex $v \in V$, we maintain an augmented Balanced Binary Tree (BBT) on the edge weights incident to v. Augmented BBT is a data-structure that supports operations such as (1) computing prefix sums and (2) updating the edge weights incident to v, both in $O(\log n)$ time.

We employ the augmented BBT data-structure as follows. First, for each vertex U and the sampled vertex $v \in U$, we update the weights of the edges from v to U to 0 in the augmented BBT of v. We then sample a uniformly-random value $x \in [0, W]$, and use the prefix sums computation in the tree to determine the range in which x lies together with the corresponding edge $(w_{t+X(u,U)-1}, w_{t+X(u,U)})$, where $w_{t+X(u,U)}$ is the first vertex not in U. After having sampled the

edge, we undo all the changes we performed in the augmented BBT of v. It follows that sampling an edge according to Equation (10) can be implemented in $O(k \log n)$. Putting together the above running times, we conclude that sampling an edge leaving U as well as the first vertex not in U can be implemented in $O(k^3 \log M)$ time. □

The last ingredient we need is an efficient way to sample the sum of weights in the random walk starting at w_t and ending at $w_{t+X(u,U)}$, where $X(u, U)$ is the number of steps needed for the walk to leave the vertex set U (Component (C)). In other words, we need to sample the following sum

$$\sum_{i=t+1}^{t+X(u,U)} \frac{1}{\boldsymbol{w}_{w_{i-1}w_i}}.$$

We accomplish this task by employing a doubling technique. To illustrate, for any pair of vertices $u, v \in V$ and $s(w)$ as defined in Equation (3), let

$$f^{u,v}_{s(w),\ell} \tag{11}$$

be the probability mass function of $s(w)$ conditioning on (1) w being a random walk that starts at u and ends at v, i.e., $w \sim w_{u,v}$ and (2) length of the walk $\ell(w)$ is ℓ in G. Then it can be shown that

$$f^{u,v}_{s(w),\ell} = \sum_{y \in V} \left(f^{u,y}_{s(w),\ell/2} * f^{y,v}_{s(w),\ell/2} \right),$$

where $*$ denotes the convolution between two probability mass functions. Equivalently, the convolution is the probability mass function of the sum of the two corresponding random variables. The above relation suggests that if (1) we have some *approximate* representation of the probability mass functions $f^{u,v}_{s(w),\ell/2}$ for all $u, v \in V$, and (2) we are able to compute the convolution of the two mass functions under such representation, we can produce approximations for $f^{u,v}_{s(w),\ell}$, where $u, v \in V$. This idea is formalized in the following lemma.

Lemma 4.13. *Let $G = (V, E, \boldsymbol{w})$ be a undirected, weighted graph with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$, where c is a positive constant. For any finite random walk w of length ℓ with $\ell \leq n^d$, where d is a positive constant, let $s(w)$ be the sum of the inverse of its edge weights, i.e.,*

$$s(w) = \sum_{i=1}^{\ell} \frac{1}{\boldsymbol{w}_{w_{i-1}w_i}}.$$

Moreover, for any $u, v \in V$, let

$$f^{u,v}_{s(w),\ell}$$

be the probability mass function of $s(w)$ conditioning on (1) w being a random walk that starts at u and ends at v, and (2) length of the walk $\ell(w)$ is ℓ in G. Then, for any pair $u, v \in V$, there exists an algorithm that that samples from $f^{u,v}_{s(w),\ell}$ and outputs a sampled $s(w)$ up to $\pm\epsilon$ relative error in $\tilde{O}(n^3\epsilon^{-2})$ time.

We finally describe a procedure that generates a β-shorted walk with respect to some vertex v and set of terminals T. Concretely, the algorithm maintains (1) a set U, initialized to $\{v\}$, of the distinct vertices visited so far by a random walk w starting at v, (2) the number of steps t the walk w has performed so far and (3) two lists L_w and L_s, initially set to empty, containing the first occurrences of distinct vertices of w and the weightes of the sub-walks induced by the

distinct vertices, respectively. Next, as long as w does not hit a vertex in T or there are vertices in the component containing v that are still not visited by w, for the next $\Theta(\beta^{-1}\log n)$ steps, the algorithm repeatedly generates a new vertex not in the current U by using components (A), (B) and (C). In each iteration, the maintained quantities U, t, L_w and L_s are updated accordingly. Note that this procedure indeed outputs all necessary information we need from a β-shorted walk. A detailed implementation of the algorithm is summarized in Figure 4.

We now have all the necessary tools to prove Lemma 4.9.

Proof of Lemma 4.9. We first show correctness. By Lemma 4.11 it follows that BinarySearch correctly samples the number of steps before a walk meets a new vertex. Next, Lemma 4.12 implies that the we can sample the new distinct vertex and its corresponding edge. Finally, by Lemma 4.13 we know that the weight of each sub-walk of a β-shorted walk is approximated within a $(1+\epsilon)$ relative error. Bringing these approximation together we get that the weight of the β-shorted walk itself is approximated within the same relative error.

We now analyse the running time of procedure GenerateSingleWalk. We start by bounding the cover time of G, which in turn bounds the number of steps for a random walk to meet a new vertex. To this end, note that it takes expected $O(m^2)$ time to meet a vertex in the same component ([3]). Thus, if we perform a random walk of length $O(m^3)$ we are guaranteed that it covers ever vertex in the component containing the starting vertex, with high probability.

Next, we analyze the running time for the steps executed within one iteration of the for loop. Observe that $k = |U| = O(\beta^{-1}\log n)$ at any point of time throughout our algorithm. The latter together with Lemma 4.11 give that it takes $O(k^3 \log^2 M) = \tilde{O}(\beta^{-3})$ time to sample the minimum number of steps for a random walk to visit a vertex not in U, where $M = O(m^3)$ by the discussion above. Furthermore, by Lemma 4.12 we can sample the new vertex not in U, and its corresponding edge in $\tilde{O}(\beta^{-3})$ time. Finally, Lemma 4.13 implies that the weight $s(w)$ of the random sub-walk between the current vertex and the new generated vertex can be approximately sampled in $\tilde{O}(\beta^{-3}\epsilon^{-2})$ time. The latter holds because Lemma 4.13 is invoked on top of the graph $G[U]$ for which $|V(G[U])| = O(\beta^{-1}\log n)$. Combining the above running times, we get that one iteration can be implemented in $\tilde{O}(\beta^{-3}\epsilon^{-2})$ time. Since there are $O(\beta^{-1}\log n)$ iterations, we conclude that the overall running time of our procedure is $\tilde{O}(\beta^{-4}\epsilon^{-2})$. □

We now present the procedure for generating a Schur complement on weighted graphs. The idea behind this algorithm is the same as in the unweighted setting, except that now we use GenerateSingleWalk to extract the information needed to simulate β-shorted walks. For the sake of completeness we summarize the details of this modified procedure in Algorithm 5.

Lemma 4.14. *Algorithm 5 runs in $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ time and outputs a graph H satisfying $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G,T)$, with high probability.*

Proof. We first bound the running time of Algorithm 5. By Lemma 4.9, the time needed to generate a β-shorted walk is $\tilde{O}(\beta^{-4}\epsilon^{-2})$. Combining this with the fact that the algorithm generates $\rho m = \tilde{O}(m\epsilon^{-2})$ walks, it follows that the running time of the algorithms is dominated by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$.

We next show the correctness of our procedure. First, note that procedure GenerateSingleWalk generates a valid β-shorted walk with its weight being approximated up to a $(1+\epsilon)$ relative error (Lemma 4.9). Assume for now that there is an oracle that fixes this approximate weight of a walk to its original exact weight. Then the collection of generated walks from Algorithm 5 forms the set W of β-shorted walks, and let $\hat{H}$ be the corresponding output graph. By Theorem 4.3, with high probability, each of the walks that starts at a component containing a vertex in T hits T. Conditioning on the latter, Theorem 3.1 gives that with high probability, $\mathbf{L}_{\hat{H}} \approx_\epsilon \mathbf{SC}(G,T)$.

Algorithm 4: GENERATESINGLEWALK(G, K, v)

Input : Weighted graph $G = (V, E, \boldsymbol{w})$ with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$ and $c > 0$, a set of vertices $K \subseteq V$, a vertex $v \in V$ such that the component containing v contains at least one vertex in K

Output: Two lists L_w and L_s containing the first occurrences of distinct vertices of a random walk w starting at v and the weights of the sub-walks induced by the distinct vertices, respectively

1 Set $U \leftarrow \{v\}$, $k \leftarrow |U|$, and let $u \leftarrow v$ be the current vertex

2 Let $t \leftarrow 0$ be the index of current step of random walk w, i.e., $w_t = u$

3 Let L_w and L_s be two lists, initially set to empty

4 **for** *each* $i = 1, \ldots, \Theta(\beta^{-1} \log n)$ **do**

5 Let $\boldsymbol{W}$ be a matrix of dimension $(k+1) \times (k+1)$ as defined in Equation (6)

6 Set $\hat{\boldsymbol{p}}_0 = [\boldsymbol{p}_0 \quad 0]^\top$, where $\boldsymbol{p}_0(u) \leftarrow 1$, and $\boldsymbol{p}_0(\hat{u}) \leftarrow 0$ for every $\hat{u} \in U \setminus u$

7 Set $X(u, U) \leftarrow$ BINARYSEARCH$(\boldsymbol{W}, \hat{\boldsymbol{p}}_0, O(m^3))$

8 Compute the probability distribution $\boldsymbol{q}$ over vertices in U after $(X(u,U) - 1)$ steps of the random walk w, conditioning on w not leaving U

9 Compute the probability distribution $\boldsymbol{r}$ over the the edges in $(U, V \setminus U)$ conditioning on $w_0, \ldots, w_t$ and $w_{t+X(u,U)}$ being the first vertex not in U. Concretely, for $v \in U$, $z \in V \setminus U$,

$$\boldsymbol{r}(v, z) = \frac{\boldsymbol{q}(v)\boldsymbol{w}_{v,z}}{R}, \text{ where } R \leftarrow \left(\sum_{v \in U,\ z \in V \setminus U} \boldsymbol{q}(v)\boldsymbol{w}_{v,z} \right).$$

10 Sample $(w_{t+X(u,U)-1}, w_{t+X(u,U)})$ according to $\mathbf{r}(w_{t+X(u,U)-1}, w_{t+X(u,U)})$

11 Set $e^{\text{new}} \leftarrow (w_{t+X(u,U)-1}, w_{t+X(u,U)})$

12 Invoke Lemma 4.13 in the inducted graph $G[U]$ to sample

13
$$s = \sum_{j=t+1}^{t+X(u,U)-1} \frac{1}{\boldsymbol{w}_{w_{j-1}, w_j}}.$$

14 Append $w_{t+X(u,U)}$ to L_w and $(s + 1/\boldsymbol{w}(e^{\text{new}}))$ to L_s

15 **if** $w_{t+X(u,U)} \in K$ **then**

16 Go to Line 20

17 **else**

18 Set $t \leftarrow (t + X(u, U))$, $u \leftarrow w_{t+X(u,U)}$, $U \leftarrow U \cup \{u\}$, and $k \leftarrow (k+1)$

19 If U covers the entire component, go to Line 20. Otherwise, $i \leftarrow (i+1)$

20 **return** lists L_w and L_s.

Algorithm 5: InitializeWeighted(G, K', β)

Input : Weighted graph $G = (V, E, \boldsymbol{w})$ with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$ and $c > 0$, set of vertices $K' \subseteq V$ such that $|K'| \leq O(\beta m)$, and $\beta \in (0, 1)$

Output: Approximate Schur Complement H and union of β-shorted walks W

Set $K \leftarrow K'$, $H \leftarrow (V, \emptyset)$ and $W \leftarrow \emptyset$

For each edge $e = (u, v)$ in G, let $K \leftarrow K \cup \{u, v\}$ with probability β

Let $\rho \leftarrow O(\log n \epsilon^{-2})$ be the sampling overhead according to Theorem 3.1

for *each edge* $e = (u, v) \in E$ *and each* $i = 1, \ldots, \rho$ **do**

  Using Algorithm 4, generate a random walk $w_1(e, i)$ from u until $\Theta(\beta^{-1} \log n)$ different vertices have been hit, it reaches K, or it has hit every edge in its component

  Using Algorithm 4, generate a random walk $w_2(e, i)$ from v until $\Theta(\beta^{-1} \log n)$ different vertices have been hit, it reaches K, or it hast hit every edge in its component

  if *both walks reach* K *at* t_1 *and* t_2 *respectively* **then**

    Connect $w_1(e, i)$, e and $w_2(e, i)$ to form a walk $w(e, i)$ between t_1 and t_2

    Let $s \leftarrow s(w_1(e, i)) + s(w_2(e_i)) + 1/\boldsymbol{w}_e$

    Add an edge (t_1, t_2) with weight $1/(\rho s)$ to H

    Add $w(e, i)$ to W

return H and W

Finally, let H be the graph where the edge weights are correct up to a $(\pm\epsilon)$ relative error. In other words, the weight of each edge e in H differs from the corresponding weight $\boldsymbol{w}_{\hat{H}}(e)$ in $\hat{H}$ by $\epsilon \boldsymbol{w}_{\hat{H}}(e)$. Summing over all the edges we get that $\mathbf{L}_H \approx_\epsilon \mathbf{L}_{\hat{H}}$. Since $\mathbf{L}_{\hat{H}} \approx_\varepsilon \mathbf{SC}(G, T)$ by the discussion above, we get that $\mathbf{L}_H \approx_{O(\epsilon)} \mathbf{SC}(G, T)$. Scaling ϵ appropriately completes the correctness. □

We now have all the necessary tools to present our dynamic algorithm for maintaining the collection of walks W (equivalently, the approximate Schur complement H), on weighted graphs.

Proof of Lemma 4.8. Similarly to the unweighted case, we give a two-level data-structure for dynamically maintaining Schur complements on weighted graphs. Specifically, we keep the terminal set T of size $\Theta(m\beta)$. This entails maintaining

1. an approximate Schur complement H of G with respect to T (Theorem 3.1),
2. a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 2.4).

We implement the procedure Initialize by running Algorithm 5, which produces a graph H and then computing a spectral sparsifier $\tilde{H}$ of H using Lemma 2.4. Note that by construction of our data-structure, every update in H will be handled by the black-box dynamic sparsifier $\tilde{H}$.

Similarly to the unweighted case, operations Insert and Delete are reduced to adding terminals to the set T and we refer the reader to the previous section for details on this reduction. Thus, the bulk of our effort is devoted to implementing the procedure AddTerminal. Let u be a non-terminal vertex that we want to append to T. We set $T \leftarrow T \cup \{u\}$, and then shorten all the walks at the first location they meet u. This shortening of walks induces in turn edge insertions and deletions to H, which are then processed by $\tilde{H}$. To quickly locate the first appearances of u in the random walks from W, we maintain a linked list W_u for each $u \in V$. This linked list contains the first appearances of w in the collections of random walks W. Note that constructing such lists can be performed at no additional costs during preprocessing phase, since Algorithm 4 directly gives the first appearances of vertices in every walk belonging to W. After locating the first appearances

of u, we cut the walks in these locations, delete the corresponding affected walks (together with their weight from H), and insert the new shorter walks to H. Note that we can simply use arrays to represent each random walk in W.

We next analyze the performance of our data-structure. Let us start with the preprocessing time. First, by Lemma 4.14 we get that the cost for constructing H on a graph with m edges is bounded by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$. Next, since H has $\tilde{O}(m\epsilon^{-2})$ edges, constructing $\tilde{H}$ takes $\tilde{O}(m\epsilon^{-4})$ time. Thus, the amortized time of INITIALIZE operation is bounded by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$.

We now analyze the update operations. By the above discussion, note that it suffices to bound the time for adding a vertex to T, which in turn (asymptotically) bounds the update time for edge insertions and deletions. The main observation we make is that adding a vertex to T only shortens the existing walks, and by the above discussion we can find such walks in time proportional to the amount of edges deleted from the walk. Since the walk needed to be generated in the INITIALIZE operation, the deletion of these edges take equivalent time to generating them. Moreover, we note that (1) handling the updates in $\tilde{H}$ induced by H introduces additional $O(\text{poly}(\log n)\epsilon^{-2})$ overheads, and (2) adding or deleting ρ edges until the next rebuild costs $\tilde{O}(\beta m\epsilon^{-2})$, since we process only up to βm operations. These together imply that the amortized cost for adding a terminal can be charged against the preprocessing time, which is bounded by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$, up to poly-logarithmic factors. Thus it follows that the operations ADDTERMINAL, INSERT and DELETE can be implemented in $\tilde{O}(1)$ amortized update time. □

4.4 Dynamic All-Pair Effective Resistance on Weighted Graphs

Following exactly the same arguments as in the proof of Theorem 1.3, we can use the above data-structure to efficiently maintain effective resistances on weighted, undirected dynamic graphs.

Theorem 4.15. *For any given error threshold $\epsilon > 0$, there is a data-structure for maintaining an weighted, undirected multi-graph $G = (V, E, \boldsymbol{w})$ with up to m edges that supports the following operations in $\tilde{O}(m^{5/6}\epsilon^{-4})$ expected amortized time:*

- INSERT(u, v, w)*: Insert the edge (u, v) with resistance $1/w$ in G.*
- DELETE(u, v)*: Delete the edge (u, v) from G.*
- EFFECTIVERESISTANCE(s, t)*: Return a $(1 \pm \epsilon)$-approximation to the effective resistance between s and t in the current graph G.*

Proof. Let $\mathcal{D}(\tilde{H})$ denote the data structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G (Lemma 4.8). Since $\mathcal{D}(\tilde{H})$ supports only up to βm operations, we rebuild $\mathcal{D}(\tilde{H})$ on the current graph G after such many operations. Note that the operations INSERT and DELETE on G are simply passed to $\mathcal{D}(\tilde{H})$. For processing the query operation EFFECTIVERESISTANCE(s, t), we declare s and t terminals (using the operation ADDTERMINAL of $\mathcal{D}(\tilde{H})$), which ensures that they are both now contained in $\tilde{H}$. Finally, we compute the (approximate) effective resistance between s and t in $\tilde{H}$ using Lemma 2.1.

We now analyze the performance of our data-structure. Recall that the insertion or deletion of an edge in G can be supported in $\tilde{O}(1)$ expected amortized time by $\mathcal{D}(\tilde{H})$. Since our data-structure is rebuilt every βm operations, and rebuilding $\mathcal{D}(\tilde{H})$ can be implemented in $\tilde{O}(m\beta^{-4}\epsilon^{-4})$, it follows that the amortized cost per edge insertion or deletion is

$$\frac{\tilde{O}(m\beta^{-4}\epsilon^{-4})}{\beta m} = \tilde{O}(\beta^{-5}\epsilon^{-4}).$$

The cost of any (s,t) query is dominated by (1) the cost of declaring s and t terminals and (2) the cost of computing the (s,t) effective resistance to ϵ accuracy on the graph $\tilde{H}$. Since (1) can be performed in $\tilde{O}(1)$ time, we only need to analyze (2). We do so by first giving a bound on the size of T. To this end, note that each of the m edges in the current graph adds two vertices to T with probability β independently. By a Chernoff bound, the number of random augmentations added to T is at most $2\beta m$ with high probability. In addition, since $\mathcal{D}(\tilde{H})$ is rebuilt every βm operations, the size of T never exceeds $4\beta m$ with high probability. The latter also bounds the size of $\tilde{H}$ by $\tilde{O}(\beta m \epsilon^{-2})$ and gives that the query cost is $\tilde{O}(\beta m \epsilon^{-2})$.

Combining the above bounds on the update and query time, we obtain the following trade-off

$$\tilde{O}\left((\beta m + \beta^{-5})\epsilon^{-4}\right),$$

which is minimized when $\beta = m^{-1/6}$, thus giving an expected amortized update and query time of

$$\tilde{O}(m^{5/6}\epsilon^{-4}). \qquad \square$$

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 for unweighted, bounded degree graphs. Specifically, as described in Theorem 1.2, our goal is to design a data-structure that maintains a solution to the Laplacian system $\mathbf{L}\mathbf{x} = \mathbf{b}$ under updates to both the underlying graph and the demand vector vector $\mathbf{b}$ 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.

Consider the dynamic Schur complement data-structure provided by Lemma 1.1. If the demand vector $\mathbf{b}$ has up to $O(\beta m)$ non-zero entries, for some parameter $\beta \in (0,1)$, we can simply incorporate the vertices corresponding to these entries in the terminal set T using operation AddTerminal of the dynamic Schur complement data-structure (Lemma 1.1). Upon receipt of a query index, we add the corresponding vertex to the Schur complement and (approximately) solve a Laplacian system on the maintained Schur complement. The obtained solution vector can then be lifted back to the Laplacian matrix using the following lemma, which we introduced in the preliminaries.

Lemma 2.6. *Let $\mathbf{x}_T$ be a solution vector such that $\mathbf{SC}(G,T)\mathbf{x}_T = \mathbf{P}(T)\mathbf{b}$. Then there exists an extension $\mathbf{x}$ of $\mathbf{x}_T$ such that $\mathbf{L}\mathbf{x} = \mathbf{b}$.*

However, the demand vector $\mathbf{b}$ may have a large number of non-zero entries, thus preventing us from obtaining a sub-linear algorithm with this approach. We alleviate this by projection this demand vector onto the current set of terminals and showing that such a projection can be maintained dynamically while introducing controllable error in the approximation guarantee. At a high level, our solver can be viewed as an one layer version of sparsified block-Cholesky algorithms [40].

We next discuss specific implementation details. Recall that $\mathbf{P}(T)$ is the matrix projection of non-terminal vertices F onto T. By Lemma 2.6, it is sufficient to maintain a solution $\mathbf{x}_T = \mathbf{SC}(G,T)^{\dagger}\mathbf{P}(T)\mathbf{b}$ dynamically. Since Lemma 1.1 already allows us to maintain a dynamic Schur complement, we need to devise a routine that maintains the projection $\mathbf{P}(T)\mathbf{b}$ of $\mathbf{b}$ under vertex additions to the terminal set.

To this end, we describe an algorithm that maintains such a projection which in turn allows us to again achieve sub-linear running times. The algorithm itself can be viewed as a numerically minded generalization of the approach for the small-support case. Concretely, let S denote the current set of terminals that the algorithm maintains (S and T will always be equal, and we differentiate between

them only for the sake of presentation). We initialize S with $O(\beta m)$ vertices from the corresponding entries in $\mathbf{b}$ that have the largest value. Our key structural observation is that if the entries of $\mathbf{b}$ are small, adding vertices to S does not change the projection significantly. To measure the error incurred by declaring some vertex a terminal, we exploit the fact that the projection $\mathbf{P}(S)\mathbf{b}$ itself is tightly connected to specific random walks in the underlying graph. In Subsection 5.3, we show that one can reuse earlier projections, even when new terminals are added to S, while paying an error corresponding to the lengths of these random walks and the magnitude of entries in $\mathbf{b}$. We then analyze how to control the accumulation of these errors over a sequence of terminal additions, and also describe an initialization procedure that involves solving a Laplacian system for computing the starting (approximate) projection vector. These together lead to the main lemma of this section, whose implementation details and analysis are deferred to Section 5.1.

Lemma 5.1. *Given an error parameter $\epsilon > 0$, an unweighted unweighted bounded-degree $G = (V, E)$ with n vertices, a vector $\mathbf{b} \in \mathbb{R}^n$ in the image of $\mathbf{L}$, a subset of terminal vertices S' and a parameter $\beta \in (0, 1)$ such that $|S'| = O(\beta m)$, there is a data-structure* DynamicProj(G, S', β) *for maintaining a vector $\tilde{\mathbf{b}}$ with $\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon \|\mathbf{b}\|_{\mathbf{L}^\dagger}$ for some S with $S' \subseteq S$, $|S| = O(\beta m)$, while supporting at most $\beta^3 m^{1/2} \epsilon (\operatorname{poly} \log n)^{-1}$ operations in the following running times:*

- Initialize(G, S', β)*: Initialize the data-structure $\tilde{O}(m)$ time.*
- Insert(u, v)*: Insert the edge (u, v) to G in $O(1)$ time while keeping G bounded-degree.*
- Delete(u, v)*: Delete the edge (u, v) from G in $O(1)$ time.*
- Change$(u, \mathbf{b}'(u), v, \mathbf{b}'(v))$*: Change $\mathbf{b}(u)$ to $\mathbf{b}'(u)$ and changes $\mathbf{b}(v)$ to $\mathbf{b}'(v)$ while keeping $\mathbf{b}$ in the range of $\mathbf{L}$ in $O(1)$ time.*
- AddTerminal(u)*: Add u to S in $O(1)$ time.*
- Query()*: Output the maintained $\tilde{\mathbf{b}}$ in $O(\beta m)$ time.*

The following lemma, whose proof will be shortly provided, allows us to combine the approximation guarantees of the data-structures (1) dynamic Schur complement and (2) dynamic Projection.

Lemma 5.2. *Let $0 < \epsilon \leq \frac{1}{2}$. Let k be a positive number such that $\|\mathbf{b}\|_{\mathbf{L}^\dagger} \leq k$. Suppose $\tilde{\mathbf{L}} \approx_\epsilon \mathbf{L}$, $\left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon k$ and $\left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^\dagger \tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}} \leq \epsilon \left\|\tilde{\mathbf{L}}^\dagger \tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}}$. Then $\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger \mathbf{b}\right\|_{\mathbf{L}} \leq 10\epsilon k$.*

We now have all the necessary tools to present the data-structure for solving Laplacian systems in bounded-degree graphs, which essentially entails combining Lemma 1.1 and Lemma 5.1.

Proof of Theorem 1.2. Let $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$ denote the data-structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G and an approximate dynamic Projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$, respectively. Set $\epsilon \leftarrow (\epsilon/10)$ for both data-structures. Our dynamic solver simultaneously maintains $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$. Since $\mathcal{D}(\tilde{\mathbf{b}})$ supports only up to $\beta^3 m^{1/2} \epsilon (\operatorname{poly} \log n)^{-1}$, we rebuild both data-structures after such many operations.

We now describe the implementation of the operations. First, we find the first βm entries with maximum value in $\mathbf{b}$. We then take the corresponding vertices and initialize S' and T' to be these βm vertices. The implementation of these data-structures involves including the endpoints of each edge with probability β to S and T, respectively. We couple these algorithms such that $S = T$, and this property will be maintained throughout the algorithm. The operations Insert and Delete on G are simply passed to $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$. The operation Change$(u, \mathbf{b}'(u), v, \mathbf{b}'(v))$ is passed to $\mathcal{D}(\tilde{\mathbf{b}})$. Upon receipt of a query $\mathbf{x}(u)$, for some vertex $u \in V$, i.e., operation Solve(u), we declare u

a terminal (using the operation AddTerminal(u) of both $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$). We then proceed by extracting an approximate Schur complement $\tilde{H}$ of G from $\mathcal{D}(\tilde{H})$ and an approximate projection vector $\tilde{\mathbf{b}}$ from $\mathcal{D}(\tilde{\mathbf{b}})$. Finally, using a black-box Laplacian solver [38], we compute a solution vector $\tilde{\mathbf{x}}_T$ to the system $\mathbf{L}_{\tilde{H}}\tilde{\mathbf{x}}_T = \tilde{\mathbf{b}}$ and output $\tilde{\mathbf{x}}_T(u)$ (this is possible since u was added to T).

We next show the correctness of the operation Solve(u). The Laplacian solver guarantees that the vector $\tilde{\mathbf{x}}_T$ satisfies

$$\left\|\tilde{\mathbf{x}}_T - \mathbf{L}_{\tilde{H}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}_{\tilde{H}}} \leq (\epsilon/10)\left\|\mathbf{L}_{\tilde{H}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}_{\tilde{H}}}. \tag{12}$$

Data-structure $\mathcal{D}(\tilde{\mathbf{b}})$ guarantees that

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(T)\mathbf{b}\right\|_{\mathbf{SC}(G,T)^{\dagger}} \leq (\epsilon/10)\left\|\mathbf{b}\right\|_{\mathbf{L}^{\dagger}}. \tag{13}$$

By Lemma 5.8, we know $\|\mathbf{P}(T)\mathbf{b}\|_{\mathbf{SC}(G,T)^{\dagger}} \leq \|\mathbf{b}\|_{\mathbf{L}^{\dagger}}$. Bringing together Equations (12) and (13) and applying Lemma 5.2 with $k = \|\mathbf{b}\|_{\mathbf{L}^{\dagger}}$, $\mathbf{L} := \mathbf{SC}(G,T)$, $\mathbf{b} := \mathbf{P}(T)\mathbf{b}$, $\mathbf{L}_{\tilde{H}}$ and $\tilde{\mathbf{b}}$ yield

$$\left\|\tilde{\mathbf{x}}_T - \mathbf{SC}(G,T)^{\dagger}\mathbf{P}(T)\mathbf{b}\right\|_{\mathbf{SC}(G,T)} \leq \epsilon k.$$

Using Lemma 2.6 we can lift the vector $\tilde{\mathbf{x}}_T$ to a solution $\tilde{\mathbf{x}}$ such that

$$\left\|\tilde{\mathbf{x}} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon k = \epsilon\left\|\mathbf{b}\right\|_{\mathbf{L}^{\dagger}} = \epsilon\left\|\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}.$$

Finally, we bound the running time of our dynamic solver. Changes in the demand vector $\mathbf{b}$ can be performed in $O(1)$ times, thus having negligible affect in our running times. The insertion or deletion of an edge in G can be supported in $\tilde{O}(1)$ expected amortized time by both $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$. Since we build our data-structures every $\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}$ operations, and the total rebuild cost is dominated by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$, it follows that the amortized cost per edge insertion or deletion is

$$\frac{\tilde{O}(m\beta^{-2}\epsilon^{-4})}{\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}} = \tilde{O}(m^{1/2}\beta^{-5}\epsilon^{-5}).$$

The cost of any query is dominated by (1) the cost of declaring the queried vertex u a terminal and (2) the cost of extracting $\tilde{H}$ and $\tilde{\mathbf{b}}$. Since (1) can be performed in $\tilde{O}(1)$ amortized time, we only need to analyze (2). Size of the terminal set $S = T$, which can be easily shown to be $O(\beta m)$ with high probability, immediately implies that the running time for (2) is dominated by $\tilde{O}(\beta m\epsilon^{-2}) = \tilde{O}(\beta m\epsilon^{-5})$, which also bounds the query cost.

Combining the above bounds on the query and update time, we obtain the following trade-off

$$\tilde{O}\left((m^{1/2}\beta^{-5} + \beta m)\epsilon^{-5}\right)$$

which is minimized when $\beta = m^{-1/12}$, thus giving an expected amortized update and query time of

$$\tilde{O}(m^{11/12}\epsilon^{-5}).$$

We can replace m by n in the above running time guarantee since by assumption G has bounded-degree throughout the algorithm. □

We next prove Lemma 5.2.

Proof of Lemma 5.2. We will use triangle inequality to decompose the error as:

$$\begin{aligned}\left\|\tilde{\mathbf{x}}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} &= \left\|\tilde{\mathbf{x}}-\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}+\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}-\tilde{\mathbf{L}}^{\dagger}\mathbf{b}+\tilde{\mathbf{L}}^{\dagger}\mathbf{b}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \\ &\leq \left\|\tilde{\mathbf{x}}-\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}}+\left\|\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}-\tilde{\mathbf{L}}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}+\left\|\tilde{\mathbf{L}}^{\dagger}\mathbf{b}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}, \end{aligned} \tag{14}$$

and bound each of them separately.

1. The first term can be bounded by first invoking the similarity of $\mathbf{L}$ and $\tilde{\mathbf{L}}$ to change the norm to $\tilde{\mathbf{L}}$, and applying the guarantees of the solve involving $\tilde{\mathbf{L}}$:

   $$\left\|\tilde{\mathbf{x}}-\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}} \leq \sqrt{(1+2\epsilon)}\left\|\tilde{\mathbf{x}}-\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}} \leq 2\left\|\tilde{\mathbf{x}}-\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}} \leq 2\epsilon\left\|\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}^{\dagger}}.$$

   This norm can in turn be transferred back to $\mathbf{L}$, and the discrepancy between $\mathbf{b}$ and $\tilde{\mathbf{b}}$ absorbed using triangle inequality:

   $$\leq 3\epsilon\left\|\tilde{\mathbf{b}}\right\|_{\mathbf{L}^{\dagger}} \leq 3\epsilon\left(\|\mathbf{b}\|_{\mathbf{L}^{\dagger}}+\left\|\tilde{\mathbf{b}}-\mathbf{b}\right\|_{\mathbf{L}^{\dagger}}\right) \leq 3\epsilon(1+\epsilon)k \leq 5\epsilon k.$$

2. The second term follows from combining the norms in $\tilde{\mathbf{L}}$ and $\mathbf{L}$ using the approximations between these matrices:

   $$\left\|\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}-\tilde{\mathbf{L}}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} = \left\|\tilde{\mathbf{b}}-\mathbf{b}\right\|_{\tilde{\mathbf{L}}^{\dagger}\mathbf{L}\tilde{\mathbf{L}}^{\dagger}} \leq 2\left\|\tilde{\mathbf{b}}-\mathbf{b}\right\|_{\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{L}}\tilde{\mathbf{L}}^{\dagger}} = 2\left\|\tilde{\mathbf{b}}-\mathbf{b}\right\|_{\tilde{\mathbf{L}}^{\dagger}},$$

   and once again converting the norm back from $\tilde{\mathbf{L}}$ to $\mathbf{L}$:

   $$\leq 4\left\|\tilde{\mathbf{b}}-\mathbf{b}\right\|_{\mathbf{L}^{\dagger}} \leq 4\epsilon k.$$

3. The third term can first be written in terms of the norm of $\mathbf{b}$ against a matrix involving the difference between $\mathbf{L}$ and $\tilde{\mathbf{L}}$:

   $$\left\|\tilde{\mathbf{L}}^{\dagger}\mathbf{b}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} = \left\|\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{b}\right\|_{\mathbf{L}} = \left\|\mathbf{L}^{\dagger/2}\mathbf{b}\right\|_{\left(\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}\right)^2}$$

   where because $\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}$ is a symmetric matrix, we have by the definition of operator norm:

   $$\leq \left\|\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}\right\|_2^2\left\|\mathbf{L}^{\dagger/2}\mathbf{b}\right\|_2 = \left\|\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}\right\|_2^2\|\mathbf{b}\|_{\mathbf{L}^{\dagger}}. \tag{15}$$

   Composing both sides of $\tilde{\mathbf{L}} \approx_{\epsilon} \mathbf{L}$ by $\mathbf{L}^{1/2}$ gives $\mathbf{L}^{1/2}\tilde{\mathbf{L}}\mathbf{L}^{1/2} \approx_{\epsilon} \mathbf{L}^{1/2}\mathbf{L}\mathbf{L}^{1/2}$, or upon rearranging:

   $$-\epsilon\mathbf{I} \preceq \mathbf{L}^{1/2}(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger})\mathbf{L}^{1/2} \preceq \epsilon\mathbf{I},$$

   or $\left\|\mathbf{L}^{1/2}(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger})\mathbf{L}^{1/2}\right\|_2^2 \leq \epsilon$. Substituting this bound into Equation 15 above then gives the result.

Summing up these three cases as in Equation 14 then gives the overall result

$$\left\|\tilde{\mathbf{x}}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \leq 10\epsilon k.$$

□

5.1 Dynamic Projection

We next discuss the main ideas behind the dynamic algorithm that maintains an approximate projection in Lemma 5.1 and then formally describe the implementation of this data-structure together with its running time guarantees. To this end, suppose we are given an approximate projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$ satisfying the following inequality

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon \left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger} \tag{16}$$

The crucial idea is to exploit the fact that the right hand side of the above inequality $\|\mathbf{b}\|_{\mathbf{L}^\dagger}$ corresponds to the square root of the energy need by the electrical flow to route $\mathbf{b}$ (see Lemma 2.1 in [50]). Since we assume that our dynamic graph G has bounded-degree, this energy is lower-bounded by

$$\left\|\mathbf{b}\right\|_{L^\dagger} \geq \sqrt{\sum_{u \in V} \left(\frac{|\mathbf{b}(u)|}{\deg(u)}\right)} = \Omega\left(\sqrt{\sum_{u \in V} |\mathbf{b}(u)|}\right).$$

Let S' be the set of βm vertices such that their corresponding coordinates in $\mathbf{b}$ have the largest values. Without loss of generality, scale all the entries in the vector $\mathbf{b}$ such that

$$|\mathbf{b}(u)| \geq 1, \quad \forall u \in S' \quad \text{and} \quad |\mathbf{b}(u)| \leq 1, \quad \forall u \in V \setminus S' \tag{17}$$

By definition of S', after up to $(\beta m)/2$ operations in our data-structure, we know that at least half of the vertices in S' will keep their corresponding $\mathbf{b}$ values. Thus the allowable error from right hand side of Equation (16), $\|\mathbf{b}\|_{\mathbf{L}^\dagger}$, is lower bounded by $\Omega(\sqrt{\beta m})$. Our goal is to control the error between the maintained approximate projection $\tilde{\mathbf{b}}$ and the true projection $\mathbf{P}(S)\mathbf{b}$. Our algorithm has two main components. First, it shows how to use a Laplacian solver that computes an approximate projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$ satisfying Equation (16) in nearly-linear time. Second, it gives a way to control the error of the projection $\mathbf{P}(S)\mathbf{b}$ under terminal additions to S with respect to the $\|\cdot\|_{\mathbf{L}^\dagger}$ norm.

The initialization lemma, whose proof is deferred to Subsection 5.2 is given below.

Lemma 5.3. *Given an unweighted graph $G = (V, E)$ with n vertices and m edges, a demand vector $\mathbf{b} \in \mathbb{R}^n$, set of vertices $S \subseteq V$ and an error parameter $\epsilon > 0$, there is an $\tilde{O}(m)$ time algorithm that computes a vector $\tilde{\mathbf{b}}$ such that*

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon \left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger}.$$

To elaborate on the second component of the algorithm, consider the error induced on $\mathbf{P}(S)\mathbf{b}$ when we add a vertex u to some terminal set S

$$\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger}, \quad \text{where } \tilde{S} = S \cup \{u\}.$$

In other words, the above expression gives the error when we simply keep the same vector $\mathbf{b}$ under a terminal addition to the set S. We will show that over a certain number of such addition we can bound the compounded error by $O(\sqrt{\beta m})$. Since the latter is a lower bound on $\|\mathbf{b}\|_{\mathbf{L}^\dagger}$, it follows that the maintained projection still provides a good approximation guarantee. The following lemma, whose proof is deferred to Subsection 5.3, bounds the error after one terminal addition.

Lemma 5.4. *Consider an unweighted undirected bounded-degree graph $G = (V, E)$, a demand vector $\mathbf{b} \in \mathbb{R}^n$ and a parameter $\beta \in (0, 1)$. Let $S \subseteq V$ with $|S| = O(\beta m)$ such that $|\mathbf{b}(u)| \geq 1$ for all $u \in S$, and $|\mathbf{b}(u)| \leq 1$ for all $u \in V \setminus S$. For each edge in G, include its endpoints to S independently, with probability at least β. Then, for any vertex $u \in V \setminus S$, with high probability*

$$\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} = \tilde{O}(\beta^{-5/2}), \quad \text{where } \tilde{S} = S \cup \{u\}.$$

We now have all the necessary tools to give a dynamic data-structure that maintains an approximate projection, i.e., prove Lemma 5.1.

Proof of Lemma 5.1. Given the input demand vector $\mathbf{b}$, let S' be the set of βm vertices such that their corresponding coordinates in $\mathbf{b}$ have the largest values. Without loss of generality, scale $\mathbf{b}$ according to Equation (17). For each edge in G include its endpoints to S' independently, with probability at least β.

We next describe the implementation of the operations. For implementing procedure Initialize(G, S', β), we invoke Lemma 5.3 with $\epsilon/2$. Let $\tilde{\mathbf{b}}$ be the output approximate projection satisfying Equation (16) with error parameter $\epsilon/2$ and set $S = S'$. As we will shortly see, operations Insert and Delete will be reduced to adding terminals to the set S. Thus we first discuss the implementation of the operation AddTerminal. To this end, let u be a non-terminal vertex that we want to append to S. We set $S = S \cup \{u\}$ and simply add an entry $\tilde{\mathbf{b}}(u) = 0$ to $\tilde{\mathbf{b}}$ while keeping the rest of the entries unaffected. To insert or delete an edge from the current graph, we simply run AddTerminal procedure for the edge endpoints.

Consider the operation Change$(u, \mathbf{b}(u)', v, \mathbf{b}(v'))$. We first invoke AddTerminal on both u and v and then add $\mathbf{b}(u)' - \mathbf{b}(u)$ to $\tilde{\mathbf{b}}(u)$ and $\mathbf{b}(v)' - \mathbf{b}(v)$ to $\tilde{\mathbf{b}}(v)$. Finally, to implement Query we simply return the approximate projection $\tilde{\mathbf{b}}$.

We next analyze the correctness of our data-structure which solely depends on the correctness of AddTerminal and Change operations. We will show that after k many such operations, our maintained approximate projection $\tilde{\mathbf{b}}$ satisfies

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \tilde{O}(k\beta^{-5/2}) + (\epsilon/2)\left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger}, \tag{18}$$

where S denotes the set of terminals after k operations. Note that when $(k = 0)$, the above inequality holds by Lemma 5.3 that implements the initialization. Let us analyze the error when a single terminal is added to S, i.e., $(k = 1)$. Then Lemma 5.4 implies that

$$\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} = \tilde{O}(\beta^{-5/2}), \quad \text{where } \tilde{S} = S \cup \{u\}.$$

Combining these two guarantees and applying triangle inequality, we get that the error after one terminal addition is

$$\begin{aligned}
\left\|\tilde{\mathbf{b}} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} &= \left\|\tilde{\mathbf{b}} - \mathbf{P}(\tilde{S})\mathbf{b} + \mathbf{P}(S)\mathbf{b} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \\
&\leq \left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} + \left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} \\
&\leq \tilde{O}(\beta^{-5/2}) + (\epsilon/2)\left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger}.
\end{aligned}$$

Next, to analyze the changes in the values of $\mathbf{b}$, let the updated $\mathbf{b}'$ be the updated vector $\mathbf{b}$. Let $\tilde{\mathbf{b}}'$ be the updated $\tilde{\mathbf{b}}$ and let $\mathbf{P}(S)\mathbf{b}'$ be the updated $\mathbf{P}(S)\mathbf{b}$. Using the fact that u and v are added to S, we get that

$$(\tilde{\mathbf{b}} - \tilde{\mathbf{b}}') = (\mathbf{b} - \mathbf{b}') = (\mathbf{P}(S)\mathbf{b} - \mathbf{P}(S)\mathbf{b}'),$$

which in turn implies that

$$(\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}) = (\tilde{\mathbf{b}}' - \mathbf{P}(S)\mathbf{b}'),$$

and thus the error vector does not change.

We showed that after each operation, either the correct vector moves by at most $\tilde{O}(\beta^{-5/2})$ with respect to its $\|\cdot\|_{\mathbf{L}^\dagger}$ norm, or $\tilde{\mathbf{b}} - \mathbf{P}(S)$ does not changes. Thus repeating the above argument k times yields Equation (18). Setting $k = c_{\text{EN}} \cdot m^{1/2}\beta^3\epsilon(\text{poly}\log n)^{-1})$ such that $\|\mathbf{b}\|_{\mathbf{L}^\dagger} = c_{\text{EN}} \cdot \sqrt{\beta m}$, we get that

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq (\epsilon/2)c_{\text{EN}}\sqrt{\beta m} + (\epsilon/2)\|\mathbf{b}\|_{\mathbf{L}^\dagger} \leq \epsilon\|\mathbf{b}\|_{\mathbf{L}^\dagger}.$$

For the running time, Lemma 5.3 implies that the initialization cost is bounded by $\tilde{O}(m)$. Since the size of the maintained vector $\tilde{\mathbf{b}}$ is bounded by $|S|$, it follows that the query cost is $O(\beta m)$. All the remaining operations can be implemented in $O(1)$ time. □

5.2 Initialization of Approximate Projection Vector

In this subsection we show how to compute an initial approximate projection vector of $\mathbf{P}(S)\mathbf{b}$, i.e., we prove Lemma 5.3.

Proof of Lemma 5.3. Define $F = V \setminus S$ and let G' be an n'-vertex graph obtained from G by contracting all vertices in S within G into a single vertex s and keeping parallel edges. Let $\mathbf{L}'$ denote the corresponding Laplacian matrix of G' and consider the induced vertex mapping $\gamma : V \to V(G')$ with $\gamma(u) = u$ for $u \in F$ and $\gamma(u) = s$ for $u \in S$. Let $\mathbf{b}' \in \mathbb{R}^{n'}$ be the corresponding demand vector in G' such that for $u \in V$, $\mathbf{b}'(\gamma(u)) = \mathbf{b}(u)$ if $\gamma(u) = u$ and $\mathbf{b}'(\gamma(u)) = \sum_{v \in S}\mathbf{b}(v)$ otherwise. For the given error parameter $\epsilon > 0$, we can invoke a black-box Laplacian solver to compute an approximate solution vector $\tilde{\mathbf{v}}'$ to $\mathbf{v}' = \mathbf{L}'^\dagger\mathbf{b}'$ such that

$$\left\|\tilde{\mathbf{v}}' - \mathbf{v}'\right\|_{\mathbf{L}'} \leq \epsilon\left\|\mathbf{v}'\right\|_{\mathbf{L}'}. \tag{19}$$

Now, to lift back the vector $\tilde{\mathbf{v}}'$ to G we define new vectors $\tilde{\mathbf{v}}$ and $\mathbf{v}$ such that for all $u \in V$

$$\tilde{\mathbf{v}}(u) := \tilde{\mathbf{v}}'(\gamma(u)) \text{ and } \mathbf{v}(u) := \mathbf{v}'(\gamma(u)).$$

Observe that for any edge $e = (u, v)$ in G, we have that

$$(\tilde{\mathbf{v}}(u) - \tilde{\mathbf{v}}(v)) = (\tilde{\mathbf{v}}'(u) - \tilde{\mathbf{v}}(v)) \text{ and } (\mathbf{v}(u) - \mathbf{v}(v)) = (\mathbf{v}'(u) - \mathbf{v}'(v)).$$

The above relations imply that the approximation guarantee from Equation (19) can be written as follows

$$\|\tilde{\mathbf{v}} - \mathbf{v}\|_{\mathbf{L}} \leq \epsilon\|\mathbf{v}\|_{\mathbf{L}}. \tag{20}$$

It is well known that if we interpret G as a resistor network, $\mathbf{v}$ represents the voltage vector on the vertices induced by the electrical flow that routes a certain demand in the network (see e.g., [17]). Thus, by linearity of electrical flows and our construction, we can view $\mathbf{v}$ as being the sum of the voltage vectors corresponding to the electrical flows that route $\mathbf{b}(u)$ amount of flow to S, where sum is taken over all $u \in F$. By Lemma 2.8, for each $u \in F$, the demand corresponding to the electrical flow that send $\mathbf{b}(u)$ units of flow to S is given by

$$\mathbf{b}(u)(\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u).$$

Summing over all $u \in F$ we get the demand vector corresponding to $\mathbf{v}$

$$\sum_{u \in F} \mathbf{b}(u)(\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u) = (\mathbf{b}|_F - \mathbf{P}(S)\mathbf{b}|_F) = (\mathbf{b}|_F - \mathbf{P}(S)(\mathbf{b} - \mathbf{b}|_S))$$
$$= (\mathbf{b}|_F - \mathbf{P}(S)\mathbf{b} - \mathbf{b}|_S)\,,$$

where $\mathbf{b}|_U$ is the restriction of $\mathbf{b}$ on the subset U with $\mathbf{b}|_U(u) = \mathbf{b}(u)$ if $u \in U$, and $\mathbf{b}|_U(u) = 0$ otherwise. Since we determined the demand vector corresponding to $\mathbf{v}$, we get that

$$\mathbf{L}\mathbf{v} = (\mathbf{b}|_F - \mathbf{P}(S)\mathbf{b} - \mathbf{b}|_S)\,. \tag{21}$$

Define the approximate project vector $\tilde{\mathbf{b}}$ that our algorithm outputs using the following relation

$$\tilde{\mathbf{b}} := (\mathbf{b}|_F - \mathbf{L}\tilde{\mathbf{v}} - \mathbf{b}|_S)\,, \tag{22}$$

where $\tilde{\mathbf{v}}$ is the extended voltage vector we defined above. To complete the proof of the lemma, it remains to bound the difference between $\tilde{\mathbf{b}}$ and $\mathbf{P}(S)\mathbf{b}$ with respect to the $\mathbf{L}^\dagger$ norm. To this end, using Equations (21) and (22) we have

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} = \|\mathbf{b}|_F - \mathbf{L}\tilde{\mathbf{v}} - \mathbf{b}|_S - (\mathbf{b}|_F - \mathbf{L}\mathbf{v} - \mathbf{b}|_S)\|_{\mathbf{L}^\dagger}$$
$$= \|\mathbf{L}\tilde{\mathbf{v}} - \mathbf{L}\mathbf{v}\|_{\mathbf{L}^\dagger} = \|\tilde{\mathbf{v}} - \mathbf{v}\|_{\mathbf{L}}\,.$$

Using the approximate guarantee in Equation (20) we have that

$$\|\tilde{\mathbf{v}} - \mathbf{v}\|_{\mathbf{L}} \leq \epsilon \|\mathbf{v}\|_{\mathbf{L}} = \epsilon \left\|\mathbf{v}'\right\|_{\mathbf{L}'} = \epsilon \left\|\mathbf{b}'\right\|_{\mathbf{L}'^\dagger} \leq \epsilon \|\mathbf{b}\|_{\mathbf{L}^\dagger}\,,$$

where the last inequality follows from the fact that the minimum energy needed to route $\mathbf{b}$ becomes smaller when contracting vertices. □

5.3 Stability of Projected Vectors

In this subsection we prove our core structural observation, namely that the projection vectors remain stable under the addition of a new terminal vertex, as stated in Lemma 5.4.

We start by considering the projection vector $\mathbf{P}(S)\mathbf{1}_u$, where $u \in F = V \setminus S$. Recall that for $s \in S$, Lemma 2.7 gives that $[\mathbf{P}(S)\mathbf{1}_u](s)$ is the probability that the random walk that starts at u hits the set S at the vertex s. Equivalently, we can view the probability of this walk as routing a fraction of $\mathbf{1}_u$ from u to s. Now, consider the operation of adding a non-terminal $u \in F$ to S, i.e., $\tilde{S} = S \cup \{u\}$. We observe that the fraction of $\mathbf{1}_u$ that we routed to some vertex v in S might have used the vertex $u \in F$. This indicates that this this fraction should have stopped at u, instead of going to other vertices in S, which in turn implies that the old projection vector $\mathbf{P}(S)\mathbf{1}_u$ is not valid anymore. We will later show that this change is tightly related to the load that random walks from other vertices in F put on the new terminal vertex u. In the following we focus on showing a provable bound on this load, which in turn will allow us to control the error for the maintained projection vector.

Concretely, for each vertex $u \in F$, we want to bound the load incurred by the random walks of the other vertices $v \in F \setminus u$ to the set S. For the purposes of our proof, it will be useful to introduce some random variable. For $v \in F$, let $Z_v(S)$ be the set of vertices visited in a random walk starting at v and ending at some vertex in S. For $t \geq 0$, let $X_v(t)$ be the set of vertices visited in a random

walk starting at $v \in F$ after t steps. For a demand vector $\mathbf{b}$ and any two vertices $u, v \in F$, the contribution of v to the load of u, denoted by $Y_v(u)$, is defined as follows

$$Y_v(u) = \mathbf{b}(v) \cdot \mathbf{1}_{(u \in Z_v(S))}.$$

The *load* of a vertex $u \in F$, denoted by N_u, is obtained by summing the contributions over all vertices in F, i.e.,

$$N_u = \sum_{v \in F} Y_v(u).$$

The following lemma gives a bound on the expected load of every non-terminal vertex.

Lemma 5.5. *For a parameter $\beta \in (0,1)$ and every vertex $u \in F$ we have that $\mathbb{E}[N_u] = \tilde{O}(\beta^{-2})$.*

For proving the above lemma it will be useful to rewrite the load quantity. To this end, recall that in the proof of Theorem 4.3 we have shown that any random walk that start at a vertex v of length $\ell = \tilde{O}(\beta^{-2})$ hits a vertex in the terminal set T with probability at least $1 - 1/n^c$, for some large constant c. Note that by construction of S in Lemma 5.4, the exact same argument applies to the set S. Thus, instead of terminating the random walks once they hit S, we can run all the walks from the vertices in F up to ℓ steps. The latter together with the assumption $\mathbf{b}(v) \leq 1$ for all $v \in F$ (provided by Lemma 5.4) give that

$$\begin{aligned}
\mathbb{E}[N_u] &= \sum_{v \in F} \mathbf{b}(v) \cdot \mathbb{P}_v[v \in Z_v(S)] \\
&\leq \sum_{v \in F} \left(\mathbb{P}_v[\text{walk } w \text{ from } v \text{ uses } u \text{ in its first } \ell \text{ steps}] + \mathbb{P}_v[|w| > \ell] \right) \\
&\leq \sum_{v \in F} \left(\sum_{0 \leq t \leq \ell} \mathbb{P}_v[u \in X_v(t)] + 1/n^c \right) \\
&\leq \sum_{0 \leq t \leq \ell} \left(\sum_{v \in V} \deg(v) \cdot \mathbb{P}_v[u \in X_v(t)] \right) + o(1). \qquad (23)
\end{aligned}$$

It turns out that that the term contained in the brackets of Equation (23) equals $\deg(u)$. Formally, we have the following lemma.

Lemma 5.6. *Let G be an undirected unweighted graph. For any vertex $u \in V$ and any length $t \geq 0$, we have*

$$\sum_{v \in V} \deg(v) \cdot \mathbb{P}[u \in X_v(t)] = \deg(u).$$

To prove this, we use the reversibility of random walks, along with the fact that the total probability over all edges of a walk starting at e is 1 at any time. Below we verify this fact in a more principled manner.

Proof of Lemma 5.6. The proof is by induction on the length of the walks t. When $t = 0$, we have

$$\mathbb{P}[u \in X_v(0)] = \begin{cases} 1 & \text{if } u = v, \\ 0 & \text{otherwise,} \end{cases}$$

which gives a total of $\deg(u)$.

For the inductive case, assume the result is true for $t-1$. The probability of a walk reaching u after t steps can then be written in terms of its location at time $t-1$, the neighbor x of u, as well as the probability of reaching there:

$$\mathbb{P}\left[u \in X_v(t)\right] = \sum_{x:(u,x)\in E} \frac{1}{\deg(x)} \mathbb{P}\left[\hat{v} \in X_v(t-1)\right].$$

Substituting this into the summation to get

$$\sum_{v\in V} \deg(v) \cdot \mathbb{P}\left[u \in X_v(t)\right] = \sum_{v\in V} \deg(v) \sum_{x:(u,x)\in E} \frac{1}{\deg(x)} \mathbb{P}\left[\hat{v} \in X_v(t-1)\right],$$

which upon rearranging of the two summations gives:

$$\sum_{x:(u,x)\in E} \frac{1}{\deg(x)} \left(\sum_{v\in V} \deg(v) \cdot \mathbb{P}\left[x \in X_v(t-1)\right] \right).$$

By the inductive hypothesis, the term contained in the bracket is precisely $\deg(x)$, which cancels with the division, and leaves us with $\deg(u)$. Thus the inductive hypothesis holds for t as well. □

Plugging Lemma 5.6 in Equation (23), along with the fact that by assumption G has bounded degree we get that

$$\mathbb{E}\left[N_u\right] \leq \deg(u) \cdot \ell = \tilde{O}(\beta^{-2}),$$

thus proving Lemma 5.5.

We now have all the tools to prove Lemma 5.4.

Proof of Lemma 5.4. Recall that $\tilde{S} = S \cup \{u\}$, where u is vertex in $F = V \setminus S$. We want to obtain a bound on the difference $(\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b})$ with respect to the $\mathbf{L}^\dagger$ norm. We distinguish the following types of entries of the difference vector: (1) newly added terminal u, (2) the old terminals S and (3) the remaining non-terminal vertices $F \setminus \{u\}$. Note that $\mathbf{P}(S)\mathbf{b}$ and $\mathbf{P}(\tilde{S})\mathbf{b}$ are not n-dimensional vectors, so we assume that all missing entries are appended with zeros. This also makes it possible to compute the $\mathbf{L}^\dagger$ norm.

In what follows, we will repeatedly make use of the following relation by Lemma 2.7 for vertices $u \in F$ and $v \in S$

$$\mathbf{P}(S)\mathbf{1}_u(v) = \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F,\\ u_k=v}} \frac{\prod_{i=0}^{k-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{k-1} \boldsymbol{d}(u_i)}.$$

For the type (1) entry, i.e., newly added terminal u, using the definition of the load N_u, we get:

$$\begin{aligned}
[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}](u) &= - \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F\setminus\{u\},\\ u_k=u}} \mathbf{b}(u_0) \cdot \frac{\prod_{i=0}^{k-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{k-1} \boldsymbol{d}(u_i)} \\
&= - \sum_{u_0\in F} \mathbb{E}\left[Y_{u_0}(u)\right] = -\mathbb{E}\left[N_u\right]. \qquad (24)
\end{aligned}$$

Note that for type (3) entries, i.e., the remaining non-terminals $v \in F \setminus \{u\}$, we have that

$$[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}](v) = 0. \qquad (25)$$

Finally, for type (2) entries, i.e., old terminals $v \in S$, similarly to the type (1) entries we get

$$\begin{aligned}
&[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}](v) \\
&\quad= \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F,\\ u_k=v}} \mathbf{b}(u_0)\cdot\frac{\prod_{i=0}^{k-1}\boldsymbol{w}_{u_iu_{i+1}}}{\prod_{i=1}^{k-1}\boldsymbol{d}(u_i)} - \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F\setminus\{u\},\\ u_k=v}} \mathbf{b}(u_0)\cdot\frac{\prod_{i=0}^{k-1}\boldsymbol{w}_{u_iu_{i+1}}}{\prod_{i=1}^{k-1}\boldsymbol{d}(u_i)} \\
&\quad= \sum_{\substack{u_0=u,\ldots,u_k=u,\\ u_{k+1},\ldots,u_{r-1}\in F,u_r=v}} \mathbf{b}(u_0)\cdot\frac{\prod_{i=0}^{r-1}\boldsymbol{w}_{u_iu_{i+1}}}{\prod_{i=1}^{r-1}\boldsymbol{d}(u_i)} \\
&\quad= \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F\setminus\{u\},\\ u_k=v}} \mathbf{b}(u_0)\cdot\frac{\prod_{i=0}^{k-1}\boldsymbol{w}_{u_iu_{i+1}}}{\prod_{i=1}^{k-1}\boldsymbol{d}(u_i)} \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F,\\ u_k=v}} \frac{\prod_{i=0}^{k-1}\boldsymbol{w}_{u_iu_{i+1}}}{\prod_{i=1}^{k-1}\boldsymbol{d}(u_i)} \\
&\quad= \mathbb{E}\left[N_u\right]\cdot[\mathbf{P}(S)\mathbf{1}_u](v). \qquad (26)
\end{aligned}$$

Bringing together Equations (24), (25) and (26) we get that

$$[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}] = -(\mathbb{E}\left[N_u\right](\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u)).$$

The right-hand side of the equation can be interpreted as routing $\mathbb{E}\left[N_u\right]$ unit of flows from u to S. Thus, to measure the error, we simply need to upper-bound the square root of the energy need to route $\mathbb{E}\left[N_u\right]$ amount of flow from u to S (Lemma 2.8),i.e.,

$$\left\|\mathbb{E}\left[N_u\right](\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u)\right\|_{\mathbf{L}^\dagger}.$$

By the simplifying assumption that G is connected and the fact that each endpoint of an edge in E is added to S independently, with probability at least β, it is easy to show that with high probability, there exists a path $p(v,S)$ from u to S that uses at most $O(\beta^{-1}\log n)$ edges. Hence, if we route $\mathbb{E}\left[N_u\right]$ units of flow from u to S along the path $p(v,S)$, the energy of such a flow is upper-bounded by

$$(\mathbb{E}\left[N_u\right])^2\cdot\tilde{O}(\beta^{-1}) = \tilde{O}((\mathbb{E}\left[N_u\right])^2\beta^{-1}).$$

Using the latter we get that

$$\begin{aligned}
\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} &= \left\|\mathbb{E}\left[N_u\right](\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u)\right\|_{\mathbf{L}^\dagger} \\
&\leq \tilde{O}\left(\sqrt{(\mathbb{E}\left[N_u\right])^2\beta^{-1}}\right) \\
&= \tilde{O}(\mathbb{E}\left[N_u\right]\beta^{-1/2}) \\
&= \tilde{O}(\beta^{-5/2}),
\end{aligned}$$

where the last inequality uses the fact that $\mathbb{E}\left[N_u\right] = \tilde{O}(\beta^{-2})$ by Lemma 5.5. This completes the proof the lemma. □

5.4 Maintaining Energy of the Electrical Flow

In this subsection we show how our dynamic solver can be extended to maintain the energy of electrical flows for routing an arbitrary demand vector $\mathbf{b}$. This extension can be thought as a generalization of the All-Pair Effective Resistance problem with $\mathbf{b} = (\mathbf{1}_u - \mathbf{1}_v)$.

Theorem 5.7. *For any given error threshold $m^{-1} < \epsilon < 1$, there is a data-structure for maintaining an unweighted, undirected bounded degree multi-graph $G = (V, E)$ with n vertices, m edges and a vector $\mathbf{b} \in \mathbb{R}^n$ that supports the following operations in $\tilde{O}(\epsilon^{-25/6} m^{5/6})$ expected amortized time:*

- INSERT(u, v)*: Insert the edge (u, v) with resistance 1 in G.*
- DELETE(u, v)*: Delete the edge (u, v) from G.*
- CHANGE$(u, \mathbf{b}'_u, v, \mathbf{b}'_v)$*: Change $\mathbf{b}_u$ to $\mathbf{b}'_u$ and $\mathbf{b}_v$ to $\mathbf{b}'_v$ while keeping $\mathbf{b}$ in the range of $\mathbf{L}$.*
- ENERGY()*: Return the energy of the electrical flow routing $\mathbf{b}$ on G, up to an $\pm\epsilon$ relative error.*

The main idea behind the proof of the above theorem is to express the energy of the electrical flow routing $\mathbf{b}$ as sum of the energy to route $\mathbf{b}_F$ to S and the energy to route $\mathbf{P}(S)\mathbf{b}$ inside some arbitrary vertex set $S \subseteq V$, where $F = V \setminus S$ and $\mathbf{P}(S)$ is the projection vector. We formalize this approach in the lemma below.

Lemma 5.8. *For any undirected graph $G = (V, E)$, demand vector $\mathbf{b}$ in the image of $\mathbf{L}$ and vertex set $S \subseteq V$,*

$$\|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{[F,F]}} + \|\mathbf{P}(S)\mathbf{b}\|^2_{\mathbf{SC}(G,S)^\dagger} = \|\mathbf{b}\|^2_{\mathbf{L}^\dagger}, \text{ where } F = V \setminus S.$$

Proof. Since it is easy to verify that

$$\begin{bmatrix} \mathbf{L}_{[F,F]} & 0 \\ 0 & \mathbf{SC}(G, C) \end{bmatrix} = \begin{bmatrix} \mathbf{I}_F & 0 \\ -\mathbf{L}_{[S,F]}\mathbf{L}^\dagger_{[F,F]} & \mathbf{I}_S \end{bmatrix} \mathbf{L} \begin{bmatrix} \mathbf{I}_F & -\mathbf{L}^\dagger_{[F,F]}\mathbf{L}_{[F,S]} \\ 0 & \mathbf{I}_S \end{bmatrix},$$

it follows that

$$\mathbf{L}^\dagger = \begin{bmatrix} \mathbf{I}_F & -\mathbf{L}^\dagger_{[F,F]}\mathbf{L}_{[F,S]} \\ 0 & \mathbf{I}_S \end{bmatrix} \begin{bmatrix} \mathbf{L}^\dagger_{[F,F]} & 0 \\ 0 & \mathbf{SC}^\dagger(G, S) \end{bmatrix} \begin{bmatrix} \mathbf{I}_F & 0 \\ -\mathbf{L}_{[S,F]}\mathbf{L}^\dagger_{[F,F]} & \mathbf{I}_S \end{bmatrix}.$$

Multiplying the above equation with $\mathbf{b}^\top$ from the left and $\mathbf{b}$ from the right yields

$$\begin{aligned} \mathbf{b}^\top \mathbf{L}^\dagger_G \mathbf{b} &= \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_S \end{bmatrix}^\top \begin{bmatrix} \mathbf{I}_F & -\mathbf{L}^\dagger_{[F,F]}\mathbf{L}_{[F,S]} \\ 0 & \mathbf{I}_S \end{bmatrix} \begin{bmatrix} \mathbf{L}^\dagger_{[F,F]} & 0 \\ 0 & \mathbf{SC}^\dagger(G, S) \end{bmatrix} \begin{bmatrix} \mathbf{I}_{\bar{S}} & 0 \\ -\mathbf{L}_{[S,F]}\mathbf{L}^\dagger_{[F,F]} & \mathbf{I}_S \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_S \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{b}_F \\ \mathbf{P}(S)\mathbf{b} \end{bmatrix}^\top \begin{bmatrix} \mathbf{L}^\dagger_{[F,F]} & 0 \\ 0 & \mathbf{SC}^\dagger(G, S) \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ \mathbf{P}(S)\mathbf{b} \end{bmatrix} \\ &= \|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{[F,F]}} + \|\mathbf{P}(S)\mathbf{b}\|^2_{\mathbf{SC}(G,S)^\dagger}. \end{aligned}$$

□

Let us have a closer look at the equation of Lemma 5.8. In particular, consider the sum on the left-hand side of this equation and its two terms. Observe that the second term can be approximated using a black-box Laplacian solver since we already know how to dynamically maintain a sparsifier $\tilde{H}$ of $\mathbf{SC}(G, S)$ as well as an approximation projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$. For the first term, similar to the algorithm for maintaining projections, we employ lazy updates. Concretely, we show that that this term does not change too much when moving a vertex u to S, which is equivalent to deleting it from F. The latter is formalized in the lemma below.

Lemma 5.9. *Consider an unweighted, undirected, bounded-degree graph $G = (V, E)$, a demand vector $\mathbf{b} \in \mathbb{R}^n$ and a parameter $\beta \in (0, 1)$. Let $S \subseteq V$ with $|S| = O(\beta m)$ such that $|\mathbf{b}(u)| \geq 1$ for all $u \in S$ and $\mathbf{b}(u) \leq 1$ for all $u \in V \setminus S$. For each edge in G, include its endpoints to S with probability at least β. For any $u \in (F = V \setminus S)$, define $\tilde{F} = F \setminus \{u\}$. Then the following two equations hold with high probability,*

$$\left| \|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{[F,F]}} - \left\|\mathbf{b}_{\tilde{F}}\right\|^2_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}} \right| = \tilde{O}(\beta^{-5/2}) \max\left(\|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}}, \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}} \right), \tag{27}$$

and

$$\left| \|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}} - \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}} \right| = \tilde{O}(\beta^{-5/2}). \tag{28}$$

Proof. We first show Equation (28). To this end, let us start by showing that

$$\|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}} = \|\mathbf{b}_F - \mathbf{P}(S)\mathbf{b}_F\|_{\mathbf{L}^\dagger}.$$

Indeed, the following chain of equations give that

$$\begin{aligned}
(\mathbf{b}_F - \mathbf{P}(S)\mathbf{b}_F)^\top \mathbf{L}^\dagger (\mathbf{b}_F - \mathbf{P}(S)\mathbf{b}_F) &= \begin{bmatrix} \mathbf{b}_F \\ -\mathbf{P}(S)\mathbf{b}_F \end{bmatrix}^\top \begin{bmatrix} \mathbf{I}_F & -\mathbf{L}^\dagger_{[F,F]}\mathbf{L}_{[F,S]} \\ 0 & \mathbf{I}_S \end{bmatrix} \\
&\quad \cdot \begin{bmatrix} \mathbf{L}^\dagger_{[F,F]} & 0 \\ 0 & \mathbf{SC}^\dagger(G,S) \end{bmatrix} \begin{bmatrix} \mathbf{I}_F & 0 \\ -\mathbf{L}_{[S,F]}\mathbf{L}^\dagger_{[F,F]} & \mathbf{I}_S \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ -\mathbf{P}(S)\mathbf{b}_F \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{b}_F \\ 0 \end{bmatrix}^\top \begin{bmatrix} \mathbf{L}^\dagger_{[F,F]} & 0 \\ 0 & \mathbf{SC}^\dagger(G,S) \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ 0 \end{bmatrix} \\
&= \|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{[F,F]}}.
\end{aligned}$$

Using this equality and letting $\tilde{S} = S \cup \{u\}$ we get that

$$\begin{aligned}
\left| \|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}} - \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}} \right| &= \left| \|\mathbf{b}_F - \mathbf{P}(S)\mathbf{b}_F\|_{\mathbf{L}^\dagger} - \left\|\mathbf{b}_{\tilde{F}} - \mathbf{P}(\tilde{S})\mathbf{b}_F\right\|_{\mathbf{L}^\dagger} \right| \\
&\leq \left\|\mathbf{b}_F - \mathbf{P}(S)\mathbf{b}_F - \mathbf{b}_{\tilde{F}} + \mathbf{P}(\tilde{S})\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger} \\
&\leq \left\|\mathbf{b}_u - \mathbf{P}(S)\mathbf{b}_F + \mathbf{P}(\tilde{S})\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger}.
\end{aligned}$$

Using the fact that $\mathbf{P}(S)\mathbf{b}_S = \mathbf{b}_S$ for any $\mathbf{b}$ and $S \subseteq V$ yields

$$\begin{aligned}
\left\|\mathbf{b}_u - \mathbf{P}(S)\mathbf{b}_F + \mathbf{P}(\tilde{S})\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger} &= \left\|-\mathbf{P}(S)\left(\mathbf{b}_F + \mathbf{b}_S\right) + \mathbf{P}(\tilde{S})\left(\mathbf{b}_{\tilde{F}} + \mathbf{b}_S + \mathbf{b}_u\right)\right\|_{\mathbf{L}^\dagger} \\
&= \left\|-\mathbf{P}(S)\mathbf{b} + \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} \\
&= \tilde{O}(\beta^{-5/2}),
\end{aligned}$$

where the last equality follows from Lemma 5.4, thus proving Equation (28).

Equation (27) follows from the following chain of inequalities

$$\begin{aligned}\left|\|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{[F,F]}} - \left\|\mathbf{b}_{\tilde{F}}\right\|^2_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}}\right| &= \left|\|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}} - \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}}\right| \cdot \left(\|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}} + \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}}\right) \\ &\leq \tilde{O}(\beta^{-5/2}) \cdot \left(\|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}} + \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}}\right) \\ &= \tilde{O}(\beta^{-5/2}) \cdot \max\left(\|\mathbf{b}_F\|_{\mathbf{L}^\dagger_{[F,F]}}, \left\|\mathbf{b}_{\tilde{F}}\right\|_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}}\right). \end{aligned}$$

□

Proof of Theorem 5.7. For the sake of exposition, we rename the energy terms described in Lemma 5.8. Specifically, let $\mathcal{E}(\mathbf{b}) := \|\mathbf{b}\|^2_{\mathbf{L}^\dagger}$, $\mathcal{E}(\mathbf{b}_F) := \|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{F,F}}$ and $\mathcal{E}(\mathbf{P}(S)\mathbf{b}) := \|\mathbf{P}(S)\mathbf{b}\|^2_{\mathbf{SC}(G,S)^\dagger}$.

Let $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$ denote the data-structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G and an approximate dynamic projection $\tilde{b}$ of $\mathbf{P}(S)\mathbf{b}$, respectively. Set $\epsilon \leftarrow (\epsilon/7)$. Similar to the dynamic Laplacain solver, our algorithm simultaneously maintains $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$ and it rebuilds after $\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}$ operations.

We now describe the implementation of the operations. The operations InserTe and Delete are implemented similarly to the dynamic solver so we only focus on the differences here. In the initialization step, we calculate an approximation $\tilde{\mathcal{E}}(\mathbf{b}) := \left\|\tilde{\mathbf{x}} - \mathbf{L}_G^\dagger \mathbf{b}\right\|^2_{\mathbf{L}}$ to the energy $\mathcal{E}(\mathbf{b})$ by using a black-box approximate Laplacian solver. Moreover, using the approximate projection $\tilde{\mathbf{b}}$ and the approximate Schur Complement $\tilde{H}$, we calculate an approximation $\tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b})$ to $\mathcal{E}(\mathbf{P}(S)\mathbf{b})$ using black-box Laplacian solver. Finally, we define

$$\tilde{\mathcal{E}}(\mathbf{b}_F) := \tilde{\mathcal{E}}(\mathbf{b}) - \tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b}),$$

and we will shortly show that this a good approximation to $\tilde{\mathcal{E}}(\mathbf{b}_F)$. We note that the approximation $\tilde{\mathcal{E}}(\mathbf{b}_F)$ is remains unchanged until we rebuild the data-structure from scratch.

To handle the Query operation, i.e., output the (approximate) energy needed to route $\mathbf{b}$ in G, we calculate $\tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b})$ using black-box Laplacian solver as in the initialization step, and output

$$\tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b}) + \tilde{\mathcal{E}}(\mathbf{b}_F).$$

We next show the correctness of our algorithm. Consider the errors introduced when approximating $\mathcal{E}(\mathbf{b})$ and $\mathcal{E}(\mathbf{P}(S)\mathbf{b})$ during initialization. For the former we have that

$$\begin{aligned}\left|\tilde{\mathcal{E}}(\mathbf{b}) - \mathcal{E}(\mathbf{b})\right| &= \left|\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|^2_{\mathbf{L}} - \left\|\mathbf{L}^\dagger\mathbf{b}\right\|^2_{\mathbf{L}}\right| \\ &\leq \left|\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} - \left\|\mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}}\right| \left(\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} + \left\|\mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}}\right) \\ &\leq \epsilon \left\|\mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} \left(\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} + \left\|\mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}}\right) \\ &\leq 2\epsilon \left\|\mathbf{L}^\dagger\mathbf{b}\right\|^2_{\mathbf{L}} = 2\mathcal{E}(\mathbf{b}).\end{aligned}$$

For the latter, we can similarly show that $\tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b})$ approximates $\mathcal{E}(\mathbf{P}(S)\mathbf{b})$ within a $\pm 2\epsilon\mathcal{E}(\mathbf{b})$ absolute error. Combining these two errors gives that $\tilde{\mathcal{E}}(\mathbf{b}_F)$ approximates $\mathcal{E}(\mathbf{b}_F)$ within a $\pm 4\epsilon\mathcal{E}(\mathbf{b})$ absolute error.

We next study the correctness of the Query operation. Let us first consider the error incurred by keeping the same $\tilde{\mathcal{E}}(\mathbf{b}_F)$ until the next rebuild. By Lemma 5.9, Equation 27, the change of $\mathcal{E}(\mathbf{b}_F)$ is bounded by

$$\left| \mathcal{E}'(\mathbf{b}_F) - \left\| \mathbf{b}'_{\tilde{F}} \right\|^2_{\mathbf{L}^\dagger_{[\tilde{F},newF]}} \right| \leq \tilde{O}(\beta^{-5/2}) \sqrt{\mathcal{E}_{\max}(\mathbf{b}_F, \mathbf{b}_{\tilde{F}})}$$

where $\mathcal{E}'(\mathbf{b}_F)$ and $\mathbf{b}'$ are the corresponding values before the update and

$$\mathcal{E}_{\max}(\mathbf{b}_F, \mathbf{b}_{\tilde{F}}) := \max \left(\|\mathbf{b}_F\|^2_{\mathbf{L}^\dagger_{[F,F]}}, \left\|\mathbf{b}_{\tilde{F}}\right\|^2_{\mathbf{L}^\dagger_{[\tilde{F},\tilde{F}]}} \right).$$

Observe that $\mathcal{E}(\mathbf{b})$ is $O(\beta m)$. If $\mathcal{E}_{\max}(\mathbf{b}_F, \mathbf{b}_{\tilde{F}})$ is $O(\beta m)$, then the above error is bounded by $\tilde{O}((\beta^{-5/2}/\sqrt{\beta m}) \cdot \mathcal{E}(\mathbf{b}))$. Otherwise, $\mathcal{E}_{\max}(\mathbf{b}_F, \mathbf{b}_{\tilde{F}})$ is at least $c\sqrt{\beta m}$ for some constant c. Since we only process up to $\beta^3 m^{1/2} \epsilon (\text{poly} \log n)^{-1}$ operations and $\sqrt{\mathcal{E}(\mathbf{b})}$ can change by at most $\tilde{O}(\beta^{-5/2})$ per update (Lemma 5.9 Equation 28), the current $\sqrt{\mathcal{E}(\mathbf{b})}$ is at least $O(\sqrt{\beta m} - \epsilon\sqrt{\beta m}) = \Omega(\sqrt{\beta m})$. So again the above error is bounded by $\tilde{O}((\beta^{-5/2}/\sqrt{\beta m}) \cdot \mathcal{E}(\mathbf{b}))$.

As the number of updates until next rebuild is bounded by $\beta^3 m^{1/2} \epsilon (\text{poly} \log n)^{-1}$, we get the total error incurred by keeping the same $\tilde{\mathcal{E}}(\mathbf{b}_F)$ is $\tilde{O}(\frac{\beta^{-5/2}}{\sqrt{\beta m}} \mathcal{E}(\mathbf{b}) \cdot \epsilon \beta^3 m^{1/2} (\text{poly} \log n)^{-1})) = \epsilon \mathcal{E}(\mathbf{b})$.

Recall that upon a query our algorithm outputs $\tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b}) + \tilde{\mathcal{E}}(\mathbf{b}_F)$. We next show that this value approximates $\mathcal{E}(\mathbf{b})$ within a $\pm\epsilon\mathcal{E}(\mathbf{b})$ absolute error, thus concluding the correctness proof. Indeed, summing over the error of $\tilde{\mathcal{E}}(\mathbf{P}(S)\mathbf{b})$, the initial error of $\tilde{\mathcal{E}}(\mathbf{b}_F)$ and the error incurred during the updates for $\tilde{\mathcal{E}}(\mathbf{b}_F)$ yields an approximation with $\pm 7\epsilon\mathcal{E}(\mathbf{b}) = \pm\epsilon\mathcal{E}(\mathbf{b})$ absolute error, where the last equality follows by our choice of ϵ.

The time complexities of our update and query operations in the data-structure are similar to the dynamic solver and can be shown to be $\tilde{O}(n^{11/12}\epsilon^{-5})$. □

Acknowledgements

We thank Daniel D. Sleator for helpful comments on an earlier version of this paper, and Xiaorui Sun for showing us the tight example for the load of random walks as discussed in Appendix A.

References

[1] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Symposium on Discrete Algorithms (SODA)*, pages 440–452, 2017.

[2] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *Symposium on Foundations of Computer Science (FOCS)*, pages 335–344, 2016.

[3] R. Aleliunas, R. J. Lipton, L. Lovasz, C. Rackoff, and R. M. Karp. Random walks, universal traversal sequences, and the complexity of maze problems. In *Symposium on Foundations of Computer Science (FOCS)*, pages 218–223, 1979.

[4] Alexandr Andoni, Anupam Gupta, and Robert Krauthgamer. Towards $(1+\epsilon)$-approximate flow sparsifiers. In *Symposium on Discrete algorithms (SODA)*, pages 279–293, 2014.

[5] Alexandr Andoni, Robert Krauthgamer, and Yosef Pogrow. On solving linear systems in sublinear time. In *Innovations in Theoretical Computer Science Conference (ITCS)*, pages 3:1–3:19, 2019.

[6] Greg Barnes and Uriel Feige. Short random walks on graphs. *SIAM Journal on Discrete Mathemathics*, 9(1):19–28, 1996.

[7] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM Journal on Computing*, 44(1):88–113, 2015. Announced at FOCS'11.

[8] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms (TALG)*, 8(4):35, 2012.

[9] Joshua Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM*, 56(8):87–94, August 2013.

[10] Andras A. Benczur and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *Symposium on Theory of computing (STOC)*, pages 47–55, 1996.

[11] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the O(mn) bound. In *Symposium on Theory of Computing (STOC)*, pages 389–397, 2016.

[12] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Symposium on Theory of Computing (STOC)*, pages 398–411, 2016.

[13] Moses Charikar, Tom Leighton, Shi Li, and Ankur Moitra. Vertex sparsifiers and abstract rounding algorithms. In *Symposium on Foundations of Computer Science (FOCS)*, pages 265–274, 2010.

[14] Dehua Cheng, Yu Cheng, Yan Liu, Richard Peng, and Shang-Hua Teng. Efficient sampling for Gaussian graphical models via spectral sparsification. *Conference on Learning Theory (COLT)*, pages 364–390, 2015.

[15] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[16] Florian Dörfler and Francesco Bullo. Kron reduction of graphs with applications to electrical networks. *IEEE Trans. on Circuits and Systems*, 60-I(1):150–163, 2013.

[17] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*, volume 22 of *Carus Mathematical Monographs*. Mathematical Association of America, 1984.

[18] David Durfee, Rasmus Kyng, John Peebles, Anup B Rao, and Sushant Sachdeva. Sampling random spanning trees faster than matrix multiplication. In *Symposium on Theory of Computing (STOC)*, pages 730–742, 2017.

[19] David Durfee, John Peebles, Richard Peng, and Anup B. Rao. Determinant-preserving sparsification of SDDM matrices with applications to counting and sampling spanning trees. In *Symposium on Foundations of Computer Science (FOCS)*, pages 926–937, 2017.

[20] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. In *Workshop on Algorithms and Data Structures (WADS)*, pages 392–399, 1991.

[21] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification: a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997. Announced at FOCS'92.

[22] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. Syst. Sci.*, 69(3):485–497, 2004.

[23] Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985. Announced at STOC'84.

[24] Gramoz Goranci, Monika Henzinger, and Pan Peng. The power of vertex sparsifiers in dynamic graph algorithms. In *European Symposium on Algorithms (ESA)*, pages 45:1–45:14, 2017.

[25] Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate schur complement on separable graphs. In *European Symposium on Algorithms (ESA)*, pages 40:1–40:15, 2018.

[26] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in polylogarithmic amortized update time. In *European Symposium on Algorithms (ESA)*, pages 46:1–46:17, 2016.

[27] Gramoz Goranci and Sebastian Krinninger. Dynamic low-stretch trees via dynamic low-diameter decompositions. *CoRR*, abs/1804.04928, 2018. Available at: http://arxiv.org/abs/1804.04928.

[28] Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *Symposium on Foundations of Computer Science (FOCS)*, pages 548–557, 2013.

[29] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 146–155, 2014.

[30] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the O(mn) barrier and derandomization. *SIAM Journal on Computing*, 45(3):947–1006, 2016. Announced at FOCS'13.

[31] Monika Rauch Henzinger. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *Journal of Algorithms*, 24(1):194–220, 1997.

[32] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. Announced at STOC'98.

[33] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *European Symposium on Algorithms (ESA)*, pages 742–753, 2015.

[34] Gorav Jindal, Pavel Kolev, Richard Peng, and Saurabh Sawlani. Density independent algorithms for sparsifying k-step random walks. In *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pages 14:1–14:17, 2017.

[35] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013.

[36] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Symposium on Discrete Algorithms (SODA)*, pages 217–226, 2014.

[37] Ioannis Koutis, Alex Levin, and Richard Peng. Faster spectral sparsification and numerical algorithms for SDD matrices. *ACM Transactions on Algorithms*, 12(2):17:1–17:16, 2016.

[38] Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly-m log n time solver for SDD linear systems. In *Symposium on Foundations of Computer Science (FOCS)*, pages 590–598, 2011.

[39] Robert Krauthgamer and Inbal Rika. Mimicking networks and succinct representations of terminal cuts. In *Symposium on Discrete algorithms (SODA)*, pages 1789–1799, 2013.

[40] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A Spielman. Sparsified cholesky and multigrid solvers for connection laplacians. In *Symposium on Theory of Computing (STOC)*, pages 842–850, 2016.

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

[42] Jakub Lacki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in o (n loglogn) time. In *European Symposium on Algorithms (ESA)*, pages 155–166, 2011.

[43] Huan Li and Zhongzhi Zhang. Kirchhoff index as a measure of edge centrality in weighted networks: Nearly linear time algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 2377–2396, 2018.

[44] David Liben-Nowell and Jon M. Kleinberg. The link prediction problem for social networks. In *International Conference on Information and Knowledge Management (CIKM)*, pages 556–559, 2003.

[45] Andreas Loukas. Graph reduction by local variation. *CoRR*, abs/1808.10650, 2018.

[46] Andreas Loukas and Pierre Vandergheynst. Spectrally approximating large graphs with smaller graphs. In *ICML*, volume 80 of *JMLR Workshop and Conference Proceedings*, pages 3243–3252. JMLR.org, 2018.

[47] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 245–254, 2010.

[48] Aleksander Madry, Damian Straszak, and Jakub Tarnawski. Fast generation of random spanning trees and the effective resistance metric. In *Symposium on Discrete Algorithms (SODA)*, pages 2019–2036, 2015.

[49] Konstantin Makarychev and Yury Makarychev. Metric extension operators, vertex sparsifiers and lipschitz extendability. In *Symposium on Foundations of Computer Science (FOCS)*, pages 255–264, 2010.

[50] Gary L. Miller and Richard Peng. Approximate maximum flow on separable undirected graphs. In *Symposium on Discrete Algorithms (SODA)*, pages 1151–1170, 2013.

[51] Ankur Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *Symposium on Foundations of Computer Science (FOCS)*, pages 3–12, 2009.

[52] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $O(n^{1/2-\epsilon})$-time. In *Symposium on Theory of Computing (STOC)*, pages 1122–1129, 2017.

[53] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 950–961, 2017.

[54] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. Announced at STOC'13.

[55] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Symposium on Theory of computing (STOC)*, pages 457–464, 2010.

[56] David Peleg and Alejandro A Schäffer. Graph spanners. *Journal of graph theory*, 13(1):99–116, 1989.

[57] Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Offline dynamic higher connectivity. *CoRR*, abs/1708.03812, 2017. Available at: http://arxiv.org/abs/1708.03812.

[58] Richard Peng and Daniel A. Spielman. An efficient parallel solver for SDD linear systems. In *Symposium on Theory of Computing (STOC)*, pages 333–342, 2014.

[59] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Symposium on Theory of computing (STOC)*, pages 255–264, 2008.

[60] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *Symposium on Foundations of Computer Science (FOCS)*, pages 509–517, 2004.

[61] Aaron Schild. An almost-linear time algorithm for uniform random spanning tree generation. In *Symposium on Theory of Computing (STOC)*, pages 214–227, 2018.

[62] Aaron Schild, Satish Rao, and Nikhil Srivastava. Localization of electrical flows. In *Symposium on Discrete Algorithms (SODA)*, pages 1577–1584, 2018.

[63] Jonah Sherman. Nearly maximum flows in nearly linear time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 263–269, 2013.

[64] Shay Solomon. Fully dynamic maximal matching in constant update time. In *Foundations of Computer Science (FOCS)*, pages 325–334, 2016.

[65] D. Spielman and S. Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

[66] D. Spielman and S. Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM Journal on Matrix Analysis and Applications*, 35(3):835–885, 2014.

[67] Daniel Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[68] Daniel A. Spielman. Algorithms, Graph Theory, and Linear Equations in Laplacian Matrices. In *Proceedings of the International Congress of Mathematicians*, 2010.

[69] Shang-Hua Teng. The Laplacian Paradigm: Emerging Algorithms for Massive Graphs. In *Theory and Applications of Models of Computation*, pages 2–14, 2010.

[70] Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. Announced at STOC'01.

[71] Joel A. Tropp. User-friendly tail bounds for sums of random matrices. *Foundations of Computational Mathemathics*, 12(4):389–434, August 2012.

[72] Tal Wagner, Sudipto Guha, Shiva Prasad Kasiviswanathan, and Nina Mishra. Semi-supervised learning on data streams via temporal label propagation. In *International Conference on Machine Learning (ICML)*, pages 5082–5091, 2018.

[73] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Symposium on Theory of Computing (STOC)*, pages 1130–1143, 2017.

A Lower Bound on Load of Edge

We show that our bound on the maximum load of a vertex from Lemma 5.5 is close to tight. This bound is crucial for bounding the effect of a single insertion/deletion on the random walks since all walks that interact with that vertex go into the error term.

Claim A.1. *For any value k, there exists a bounded degree graph such that if we take a random walk starting at every vertex until it reaches k distinct vertex, the maximum load of an edge is $\Omega(k^2/\log k)$.*

Note that the bounded degree assumption allow us to ignore the distinction between edge and vertex loads.

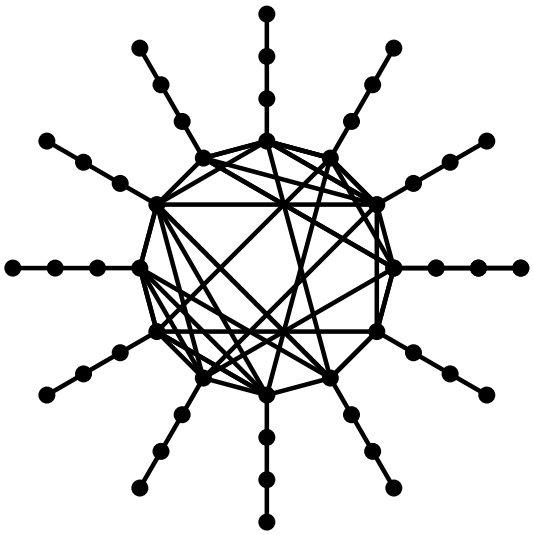

Figure 2: An instance of the graph described in Definition A.2

We build our hard instance by having a very well connected core graph, and paths leading from it. The main observation is that vertices from these paths need to walk on this core piece in order to reach k distinct vertices. An example of such a graph is shown in Figure 2 and a formal definition is given below.

Definition A.2. We say that a graph $G = (V, E)$ is a *path augmented expander* if it is defined as follows

1. Let G be a bounded-degree graph of constant expansion on k vertices, denoted as the *core*.
2. Extend G by adding for each core vertex u, a path of length $k/10$, denoted as the 'ray' of u.

For our overall bound on the number of steps that a random walk takes in the core component, we need the following bound on the expected number of steps of a walk within each ray.

Lemma A.3. *The probability that a random walk starting from one end of a path reaches n vertices to the right before returning to the end of the path is at most $1/n$.*

Proof. The resistance between the second vertex and the vertex n is $(n-1)$, while the resistance between the vertices 1 and 2 is 1. Since the long term behavior of random walks is akin to electrical flow, we get that such a random walk reaches vertex n before vertex 1 with probability $1/(1+n-1) = 1/n$. □

We now have all the tools to prove our lower bound.

Proof of Claim A.1. Consider the graph as given in Definition A.2. Specifically, consider the random walk starting at each vertex from some ray. Since each ray has at most $k/10$ vertices, at least $k/2$ distinct vertices must come from outside of this ray. We show that with probability at least $1/2$, such a walk must reach $\Omega(k/\log k)$ core vertices before visiting k distinct vertices.

At each core vertex, Lemma A.3 gives that the probability that the walk hits at least i vertices is

$$\sum_{j=1}^{i} \frac{1}{i} \leq O\left(\log k\right).$$

Thus the expected number of distinct vertices visited after k steps is at most most $k/10$. The later gives that with probability at least $1/2$, such a walk takes at least $\Omega(k/\log k)$ steps among core vertices. As the core graph is an expander, the number of distinct core vertices visited is at least $\Omega(k/\log k)$ as well.

Since the random walks from each vertex are independent, applying Chernoff bound gives that with high probability at least $\Omega(k^2)$ walks spend at least $\Omega(k^2/\log k)$ steps in the core. As there are k vertices in the core, one of them must have load $\Omega(k^2/\log k)$. □

B Proofs about Schur Complement

Lemma 2.6. *Let $\mathbf{x}_T$ be a solution vector such that $\mathbf{SC}(G,T)\mathbf{x}_T = \mathbf{P}(T)\mathbf{b}$. Then there exists an extension $\mathbf{x}$ of $\mathbf{x}_T$ such that $\mathbf{L}\mathbf{x} = \mathbf{b}$.*

Proof of Lemma 2.6. We assume without loss of generality that the underlying graph G is connected. Consider the following extended linear system

$$\begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,T]} \\ \mathbf{0} & \mathbf{SC}(G,T) \end{bmatrix} \begin{bmatrix} \mathbf{x}_F \\ \mathbf{x}_T \end{bmatrix} = \begin{bmatrix} \mathbf{I}_F & \mathbf{0} \\ \mathbf{P}(T) \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_T \end{bmatrix}$$

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

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

Multiplying both sides from the left with

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

we get that

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

what we wanted to show.

□

Lemma 2.7. *Consider a graph $G = (V, E)$. For any subset of vertices $T \subseteq V$, a vertex $v \in T$, and a vertex $u \in F = V \setminus T$, let $\mathbb{P}_u\left[t_v < t_{T\setminus v}\right]$ be the probability that the random walk that starts at u hits v before hitting any other vertex from $T \setminus v$. Then we have that*

$$\left[\mathbf{P}(T)\mathbf{1}_u\right](v) = \mathbb{P}_u\left[t_v < t_{T\setminus v}\right].$$

In fact, $\{\mathbf{P}(T)\mathbf{1}_u\}_{v\in T}$ is a probability distribution for any fixed vertex $v \in F$.

Proof of Lemma 2.7. First, note that if there is no path from vertices in T to $F = V \setminus T$, then the lemma holds trivially. Thus assume T and F are connected by paths. Next, let

$$\mathbf{L}_{[F,F]} = \mathbf{D}_F - \mathbf{A}_F,$$

where $\mathbf{D}_F$ is the diagonal of $\mathbf{L}_{[F,F]}$ and $\mathbf{A}_F$ is the negation of the off-diagonal entries, and then expand $\mathbf{L}_{[F,F]}^{-1}$ using the Jacobi series:

$$\begin{aligned} \mathbf{L}_{[F,F]}^{-1} = (\mathbf{D}_F - \mathbf{A}_F)^{-1} &= \mathbf{D}^{-1/2}\left(\mathbf{I} - \mathbf{D}_F^{-1/2}\mathbf{A}_F\mathbf{D}_F^{-1/2}\right)^{-1}\mathbf{D}_F^{-1/2} \\ &= \mathbf{D}_F^{-1/2}\left(\sum_{\ell=0}^{\infty}(\mathbf{D}_F^{-1/2}\mathbf{A}_F\mathbf{D}_F^{-1/2})^{\ell}\right)\mathbf{D}_F^{-1/2} = \sum_{\ell=0}^{\infty}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1}. \end{aligned}$$

The above series converges due to the fact that $\mathbf{L}_{[F,F]}$ is strictly diagonally dominant. Concretely, the latter implies $(\mathbf{A}_F\mathbf{D}_F^{-1})^{\ell}$ tends to zero as ℓ tends to infinity. Substituting this in the definition of $\mathbf{P}(T)$ and letting $\mathbf{1}_u = \begin{bmatrix}\mathbf{1}_u^F & \mathbf{1}_u^T\end{bmatrix}^{\top}$ we get that

$$\begin{aligned} \mathbf{P}(T)\mathbf{1}_u &= \begin{bmatrix} -\sum_{\ell=0}^{\infty}\mathbf{L}_{[T,F]}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1} & \mathbf{I}_T \end{bmatrix}\begin{bmatrix}\mathbf{1}_u^F \\ \mathbf{1}_u^T\end{bmatrix} \\ &= \sum_{\ell=0}^{\infty} -\mathbf{L}_{[T,F]}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1}\mathbf{1}_u^F. \end{aligned}$$

In particular, it follows that for any $v \in T$

$$\left[\sum_{\ell=0}^{\infty} -\mathbf{L}_{[T,F]}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1}\mathbf{1}_u^F\right](v) = \sum_{\substack{u_0=u,\ldots,u_{\ell-1}\in F,\\ u_\ell = v}} \frac{\prod_{i=0}^{\ell-1}\boldsymbol{w}_{u_iu_{i+1}}}{\prod_{i=1}^{\ell-1}\boldsymbol{d}(u_i)}.$$

□

Lemma 2.8. *Consider a graph $G = (V, E)$. Let $T \subseteq V$ be a subset of vertices, and let $u \in F = V \backslash T$. Consider the demand vector $\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u$ that requests to send one unit of flow from u to T according to the probability distribution $\{\mathbf{P}(T)\mathbf{1}_u\}_{v \in T}$. Then the minimum energy needed to route this demand is given by*

$$\|\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u\|^2_{\mathbf{L}^\dagger} = (\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u)^\top \mathbf{L}^\dagger (\mathbf{1}_u - \mathbf{P}(T)\mathbf{1}_u).$$

Proof of Lemma 2.8. Given a valid demand vector $\mathbf{b}$ with $\mathbf{b}^\top \mathbf{1} = 0$, Lemma 2.1 due to Miller and Peng [50] shows that the minimum energy for routing $\mathbf{b}$ is given by $\mathbf{b}^\top \mathbf{L}^\dagger \mathbf{b}$. Since by construction we have that $[\mathbf{1}_u - \mathbf{P}(K)\mathbf{1}_u]^\top \mathbf{1} = 0$, substituting this demand vector in place of $\mathbf{b}$ gives the lemma. □

Now we prove Theorem 3.1, which states that sampling random walks generates sparsifiers of Schur complements:

Theorem 3.1. *Let $G = (V, E, w)$ be an undirected, weighted multi-graph with a subset of vertices T. Furthermore, let $\epsilon \in (0, 1)$, and let ρ be some parameter related to the concentration of sampling given by*

$$\rho = O\left(\log n \epsilon^{-2}\right).$$

Let H be an initially empty graph, and for every edge $e = (u, v)$ of repeat ρ times the following procedure:

1. *Simulate a random walk starting from u until it* first *hits T at vertex t_1,*
2. *Simulate a random walk starting from v until it* first *hits T at vertex t_2,*
3. *Combine these two walks (including e) to get a walk $u = (t_1 = u_0, \ldots, u_\ell = t_2)$, where ℓ is the length of the combined walk.*
4. *Add the edge (t_1, t_2) to H with weight*

$$1 / \left(\rho \sum_{i=0}^{\ell-1} \left(1 / \boldsymbol{w}_{u_i u_{i+1}}\right) \right)$$

The resulting graph H satisfies $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, T)$ with high probability.

Note that this rescaling by $1/\rho \left(\sum_{1 \le i \le \ell} \frac{1}{\boldsymbol{w}_{w_{i-1} w_i}} \right)$ is quite natural: Consider the degenerate case where $T = V$. This routine generates ρ copies of each edge, which then need to be rescaled by $1/\rho$ to ensure approximation to the original graph.

Similar to other randomized graph sparsification algorithms [67, 37, 2, 19, 34], our sampling scheme directly interacts with Chernoff bounds. Our random matrices are 'groups' of edges related to random walks starting from the edge e. We will utilize Theorem 1.1 due to [71], which we paraphrase in our notion of approximations.

Theorem B.1. *Let $\mathbf{X}_1, \mathbf{X}_2 \ldots \mathbf{X}_k$ be a set of random matrices satisfying the following properties:*

1. *Their expected sum is a projection operator onto some subspace, i.e.,*

$$\sum_i \mathbb{E}\left[\mathbf{X}_i\right] = \mathbf{\Pi}.$$

2. *For each $\mathbf{X}_i$, its entire support satisfies:*

$$0 \preceq \mathbf{X}_i \preceq \frac{\epsilon^2}{O\left(\log n\right)} \mathbf{I}.$$

Then, with high probability, we have

$$\sum_i \mathbf{X}_i \approx_\epsilon \mathbf{\Pi}.$$

Re-normalizations of these bounds similar to the work of [67] give the following graph theoretic interpretation of the theorem above.

Corollary B.2. *Let $E_1 \dots E_k$ be distributions over random edges satisfying the following properties:*

1. *Their expectation sums to the graph G, i.e.,*

$$\sum_i \mathbb{E}\left[E_i\right] = G.$$

2. *For each E_i, any edge in its support has low leverage score in G, i.e.,*

$$\boldsymbol{w}_e R^G_{\text{eff}}\left(e\right) \leq \frac{\epsilon^2}{O\left(\log n\right)}.$$

Then, with high probability, we have

$$\sum_i \mathbf{L}_{E_i} \approx_\epsilon \mathbf{L}_G.$$

To fit the sampling scheme outlined in Theorem 3.1 into the requirements of Corollary B.2, we need (1) a specific interpretation of Schur complements in terms of walks, and (2) a bound on the effective resistances between two vertices at a given distance.

Given a walk $w = u_0, \dots, u_\ell$ of length ℓ in G with a subset a vertices T, we say that w is a *terminal-free* walk iff $u_0, u_\ell \in T$ and $u_1, \dots, u_{\ell-1} \in V \setminus T$.

Fact B.3 ([19], Lemma 5.4)**.** *For any undirected, weighted graph G and any subset of vertices $T \subseteq V$, the Schur complement $\mathbf{SC}(G, T)$ is given as an union over all multi-edges corresponding to terminal-free walks $u_0, \dots, u_\ell$ with weight*

$$\frac{\prod_{i=0}^{\ell-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{\ell-1} \boldsymbol{d}\left(u_i\right)}.$$

The fact below follows by repeatedly applying the triangle inequality of the effective resistances between two vertices.

Fact B.4. *In an weighted undirected graph G, the effective resistance between two vertices that are connected by a path $p = (p_0, \dots, p_\ell)$ is at most $\sum_{i=0}^{\ell-1} 1/\boldsymbol{w}_{p_i p_{i+1}}$.*

Combining the above results gives the guarantees of our sparsification routine.

Proof of Theorem 3.1. For every edge $e \in E$, let W_e be the random graph corresponding the the terminal-free random walk that started at edge e. Define $H = \rho \cdot \sum_e W_e$ to be the output graph by our sparsification routine, where $\rho = O(\log n \epsilon^{-2})$ is the sampling overhead. To prove that $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, T)$ with high probability, we need to show that (1) $\mathbb{E}\left[H\right] = \mathbf{SC}(G, T)$ and (2) for any edge f in W_e, its leverage score $\boldsymbol{w}_f R^{W_e}_{\text{eff}}(f)$ is at most $\leq \epsilon^2 / \log n$ (by Corollary B.2). Note that (2) immediately follows from the effective resistance bound of Fact B.4 and the choice of $\rho = O(\log n/\epsilon^2)$. We next show (1).

To this end, we start by describing the decomposition of $\mathbf{SC}(G, T)$ into random multi-edges, which correspond to random terminal-free walks in Fact B.3. The main idea is to sub-divide each

walk $u_0 \dots u_\ell$ of length ℓ in G into ℓ walks of the same length, each starting at one of the ℓ edges on the walk, and each having weight

$$\frac{\prod_{i=0}^{\ell-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{\ell-1} \boldsymbol{d}(u_i)}$$

By construction of our sparsification routine, note that every random graph W_e is a distribution over walks $u_0 \dots u_\ell$, each picked with probability

$$\frac{1}{\boldsymbol{w}_e} \frac{\prod_{i=0}^{\ell-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{\ell-1} \boldsymbol{d}(u_i)}.$$

Thus, to retain expectation, when such a walk is picked, our routine correctly adds it to H with weight $1/(\rho \sum_{i=0}^{\ell-1} 1/\boldsymbol{w}_{u_i u_{i+1}})$.

Formally, we get the following chain of equalities

$$\begin{aligned}
\mathbb{E}[H] &= \rho \cdot \sum_e \mathbb{E}[W_e] \\
&= \rho \cdot \sum_e \sum_{w=u_0,u_1 \dots u_{\ell(w)}: w \ni e} \frac{1}{\rho \left(\sum_{i=0}^{\ell-1} 1/\boldsymbol{w}_{u_i u_{i+1}} \right)} \cdot \frac{1}{\boldsymbol{w}_e} \cdot \frac{\prod_{i=0}^{\ell-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{\ell-1} \boldsymbol{d}(u_i)} \\
&= \sum_{w=u_0,u_1 \dots u_{\ell(w)}} \sum_{e: e \in w} \frac{1}{\left(\sum_{i=0}^{\ell-1} 1/\boldsymbol{w}_{u_i u_{i+1}} \right)} \cdot \frac{1}{\boldsymbol{w}_e} \cdot \frac{\prod_{i=0}^{\ell-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{\ell-1} \boldsymbol{d}(u_i)} \\
&= \sum_{w=u_0,u_1 \dots u_{\ell(w)}} \frac{\prod_{i=0}^{\ell-1} \boldsymbol{w}_{u_i u_{i+1}}}{\prod_{i=1}^{\ell-1} \boldsymbol{d}(u_i)} \\
&= \mathbf{SC}(G, T).
\end{aligned}$$

□

C Sampling Weights of a Random Walk

In this section, we show that given a random walk w of length ℓ in a weighted G with polynomially bounded weights, we can efficiently sample an approximation to $s(w) = \sum_{i=1}^{\ell} (1/\boldsymbol{w}_{w_{i-1} w_i})$. Concretely, we prove the following lemma from Section 4.3.

Lemma 4.13. *Let $G = (V, E, \boldsymbol{w})$ be a undirected, weighted graph with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$, where c is a positive constant. For any finite random walk w of length ℓ with $\ell \leq n^d$, where d is a positive constant, let $s(w)$ be the sum of the inverse of its edge weights, i.e.,*

$$s(w) = \sum_{i=1}^{\ell} \frac{1}{\boldsymbol{w}_{w_{i-1} w_i}}.$$

Moreover, for any $u, v \in V$, let

$$f^{u,v}_{s(w),\ell}$$

be the probability mass function of $s(w)$ conditioning on (1) w being a random walk that starts at u and ends at v, and (2) length of the walk $\ell(w)$ is ℓ in G. Then, for any pair $u, v \in V$, there exists an algorithm that that samples from $f^{u,v}_{s(w),\ell}$ and outputs a sampled $s(w)$ up to $\pm\epsilon$ relative error in $\tilde{O}(n^3 \epsilon^{-2})$ time.

Algorithm 6: CONVOLUTE$(g^{(1)}, g^{(2)}, \epsilon, j)$

Input : Two (ϵ, j)-approximations $g^{(1)}$ and $g^{(2)}$ of two probability mass functions $f^{(1)}$ and $f^{(2)}$

Output: An $(\epsilon, (j+1))$-approximation $g := g^{(1)} * g^{(2)}$ of $f := f^{(1)} * f^{(2)}$

Set $g \leftarrow \mathbf{0}$

for $(k_1, k_2) \in \{0, \ldots, L\}^2$ **do**

  Find k_3 such that $(1+\epsilon)^{k_1} + (1+\epsilon)^{k_2} \in I^1_{k_3}$

  Set $g(k_3) \leftarrow g(k_3) + g^{(1)}(k_1) \cdot g^{(2)}(k_2)$

return g

To prove the above lemma, we employ a doubling technique. Specifically, for any pair of vertices $u, v \in V$, and a random walk w of length ℓ that starts at u and ends at v, it is easy to see that

$$f^{u,v}_{s(w),\ell} = \sum_{y \in V} \left(f^{u,y}_{s(w),\ell/2} * f^{y,v}_{s(w),\ell/2} \right),$$

where $*$ denotes the convolution between two probability mass functions. However, one challenge here is that we cannot afford dealing with exact representations of probability mass functions as this would be computationally expensive. Instead, we introduce an *approximate* representation of such functions, and then give an algorithm that allows computing the convolution between such approximate representations. Before proceeding further, note that we can scale down the edge weights so that $\boldsymbol{w}_e \leq 1$, and thus $1/\boldsymbol{w}_e \geq 1$ for every $e \in E$. In addition, we remark that $\boldsymbol{w}$ does not need to be integral.

Let us introduce a compact way to represent any given probability mass function approximately f. The main idea is to 'move' each number in the support of f by $(1+\epsilon)$, which in turn results in a $(1+\epsilon)$ approximation of the sampled value for f. Formally, let f be a probability mass function such that $f(x) = 0$, for each $x \notin \{0, \ldots, n^c\}$, where c is a positive constant. For $j \geq 1$, let I^j_k be the interval $[(1+\epsilon)^k, (1+\epsilon)^{k+j})$ for $k \in \{0, \ldots, L\}$ where $L = O\left((c+d)\epsilon^{-1} \log n\right)$. Note that the upper bound L is chosen in such a way that $\cup_k I^1_k$ covers the range of $f^{u,\ell}_{s(w),\ell}$ for every possible triplet (u, v, ℓ). For $j \geq 1$ and $\epsilon > 0$, we say that g is an (ϵ, j)-approximation of a probability mass function f iff there exists a matrix $\boldsymbol{H}$ satisfying the following properties:

(a) $\sum_{k=0}^{L} \boldsymbol{H}_{x,k} = f(x), \ \forall x \in \{0, \ldots, n^c\},$

(b) $\sum_{x=0}^{n^c} \boldsymbol{H}_{x,k} = g(k), \ \forall k \in \{0, \ldots, L\},$

(c) $\boldsymbol{H}_{x,k} = 0, \ \forall x \notin I^j_k.$

Note that an (ϵ, j)-approximation of f is also an $(\epsilon, (j+1))$-approximation of f. Moreover, observe that the intervals $\{I^1_k\}_{k \in \{0\} \cup L}$ are disjoint for different k but I^j_k overlaps with $I^j_{k'}$ whenever $j \geq 2$ and $|k - k'| < j$.

Next we show how to compute the convolution of two probability mass functions under their approximate representations. Let and $g^{(1)}$ and $g^{(2)}$ be (ϵ, j)-approximations of probability mass functions $f^{(1)}$ and $f^{(2)}$, respectively. Now consider two intervals $I^j_{k_1}$ and $I^j_{k_2}$. Without loss of

generality, assume that $k_1 \leq k_2$. If $x \in I^j_{k_1}$ and $y \in I^j_{k_2}$, then

$$x + y \in I' := [\text{le}, \text{ri}), \text{ where le} := \left((1+\epsilon)^{k_1} + (1+\epsilon)^{k_2}\right), \quad \text{ri} := \left((1+\epsilon)^{k_1+j} + (1+\epsilon)^{k_2+j}\right).$$

Furthermore, let $I^1_{k_3}$ be an interval such that $\text{le} \in I^1_{k_3}$. The latter implies that $(1+\epsilon)^{k_3} \leq \text{le} < (1+\epsilon)^{k_3+1}$. Since $\text{ri} = \text{le} \cdot (1+\epsilon)^j$, it follows that $\text{ri} < (1+\epsilon)^{k_3+j+1}$. Bringing together the above bounds we get that $(1+\epsilon)^{k_3} \leq \text{le} < (1+\epsilon)^{k_3+j+1}$, i.e., $I' \subseteq I^{j+1}_{k_3}$. Since k_3 depends on k_1, k_2, and j we sometimes write $k_3(k_1, k_2, j)$ instead of k_3.

Since the above approach gives us a way to combine two different intervals, it is now straightforward to compute the convolution between two probability mass functions. This task is performed in the standard way and we review its implementation details in Algorithm 6 for the sake of completeness.

Lemma C.1. *Let $j \geq 1$ and $\epsilon > 0$ by two parameters. Given any two (ϵ, j)-approximations $g^{(1)}$ and $g^{(2)}$ of probability mass functions $f^{(1)}$ and $f^{(2)}$, $\textsc{Convolute}(g^{(1)}, g^{(2)}, \epsilon, j)$ (Algorithm 6) computes in $\tilde{O}(\epsilon^2)$ time an $(\epsilon, (j+1))$-approximation $g := g^{(1)} * g^{(2)}$ of the convolution $f := f^{(1)} * f^{(2)}$.*

Proof. We first show the correctness. Since $g^{(1)}$ and $g^{(2)}$ are (ϵ, j)-approximations to $f^{(1)}$ and $f^{(2)}$ by assumption of the lemma, we know that there exists matrices $\boldsymbol{H}^{(1)}$ and $\boldsymbol{H}^{(2)}$ satisfying properties (a), (b) and (c). To show that the output g is correct we need to construct a matrix $\boldsymbol{H}$ that satisfies each of these properties. By construction of the algorithm, the new matrix $\boldsymbol{H}$ is defined as follows:

$$\boldsymbol{H}_{z,k_3} := \sum_{\substack{x \in I^j_{k_1}, x \in I^j_{k_2} \\ x+y=z, k_3 = k_3(k_1,k_2,j)}} \boldsymbol{H}^{(1)}_{x,k_1} \cdot \boldsymbol{H}^{(2)}_{y,k_2}, \quad z \in \{0, \ldots, n^c\},\ k_3 \in \{0, \ldots, L\}.$$

We start by showing property (a) for $\boldsymbol{H}$. Concretely, for any $z \in \{0, \ldots, n^c\}$ we get that

$$\begin{aligned}
\sum_{k_3=0}^{L} \boldsymbol{H}_{z,k_3} &= \sum_{k_3=0}^{L} \sum_{\substack{x \in I^j_{k_1}, x \in I^j_{k_2} \\ x+y=z, k_3 = k_3(k_1,k_2,j)}} \boldsymbol{H}^{(1)}_{x,k_1} \cdot \boldsymbol{H}^{(2)}_{y,k_2} \\
&= \sum_{x \in I^j_{k_1}, y \in I^j_{k_2}, x+y=z} \boldsymbol{H}^{(1)}_{x,k_1} \cdot \boldsymbol{H}^{(2)}_{y,k_2} \\
&= \sum_{x+y=z} \left(\sum_{x \in I^j_{k_1}} \boldsymbol{H}^{(1)}_{x,k_1} \right) \left(\sum_{y \in I^j_{k_2}} \boldsymbol{H}^{(2)}_{y,k_2} \right) \\
&= \sum_{x+y=z} f^{(1)}(x) \cdot f^{(1)}(y) \\
&= \left(f^{(1)} * f^{(2)} \right)(z) = f(z).
\end{aligned}$$

Next, $\boldsymbol{H}$ satisfies property (b) since for any $k_3 \in \{0, \ldots, L\}$ we get that

$$\begin{aligned}
\sum_{z=0}^{n^c} \boldsymbol{H}_{z,k_3} &= \sum_{z=0}^{n^c} \sum_{\substack{x \in I_{k_1}^j, x \in I_{k_2}^j \\ x+y=z, k_3=k_3(k_1,k_2,j)}} \boldsymbol{H}^{(1)}_{x,k_1} \cdot \boldsymbol{H}^{(2)}_{y,k_2} \\
&= \sum_{x \in I_{k_1}^j, y \in d_{k_2}^j, k_3=k_3(k_1,k_2,j)} \boldsymbol{H}^{(1)}_{x,k_1} \cdot \boldsymbol{H}^{(2)}_{y,k_2} \\
&= \sum_{k_3=k_3(k_1,k_2,j)} \left(\sum_{x \in I_{k_1}^j} \boldsymbol{H}^{(1)}_{x,k_1} \right) \left(\sum_{y \in I_{k_2}^j} \boldsymbol{H}^{(2)}_{y,k_2} \right) \\
&= \sum_{k_3=k_3(k_1,k_2,j)} g^{(1)}_{k_1} \cdot g^{(2)}_{k_2} \\
&= \left(g^{(1)} * g^{(2)} \right)(k_3) = g(k_3).
\end{aligned}$$

where the penultimate equality follows by Algorithm 6.

Finally, for every $x \notin I_k^j$, we have that $\boldsymbol{H}_{x,k} = 0$, i.e., property (c) holds for $\boldsymbol{H}$. The latter holds since $x \in I_{k_1}^j$ and $y \in I_{k_2}^j$ gives that $x+y \in I_{k_3(k_1,k_2,j)}^{j+1}$. Thus, by definition of approximate probability mass function, it follows that $g = g^{(1)} * g^{(2)}$ is an $(\epsilon, (j+1))$-approximation of $f = f^{(1)} * f^{(2)}$.

For the running time first recall that $L = O\left((c+d)\epsilon^{-1} \log n\right) = \tilde{O}(\epsilon^{-1})$. Since the cost for implementing CONVOLUTE is bounded by $\tilde{O}(L^2)$, it follows that we can implement this procedure in $\tilde{O}(\epsilon^{-2})$ time. □

The last ingredient we need is to show that given a family of probability mass functions, and their corresponding approximations, choosing one of these functions according to some probability distribution yields a random approximation in the natural way. Specifically, for an index set Q, let $\{f^{(q)}\}_{q \in Q}$ be a set of probability mass functions. Let $\hat{q}$ be be a random variable (independent from $\{f^{(q)}\}_{q \in Q}$) such that for every $q \in Q$, $\Pr[\hat{q} = q] = \boldsymbol{p}(q)$, and $\sum_{q \in Q} \boldsymbol{p}(q) = 1$. Furthermore, define

$$f := f^{(\hat{q})} = \sum_{q \in Q} \boldsymbol{p}(q) f^{(q)}$$

.

Lemma C.2. *Suppose $g^{(q)}$ is an (ϵ, j)-approximation of the probability mass function f^q, for all $q \in Q$. Let f be the probability mass function as defined above. Then*

$$g := \sum_{q \in Q} \boldsymbol{p}(q) g^{(q)}$$

is an (ϵ, j)-approximation of f.

Proof. By definition of an (ϵ, j)-approximation, we know that there exist matrices $\{\boldsymbol{H}^{(q)}\}_{q \in Q}$ for $\{g^{(q)}\}_{q \in Q}$ satisfying properties (a), (b) and (c). We need to show that for g as defined in the lemma, there exist a suitable matrix $\boldsymbol{H}$ that satisifes each of these properties. To this end, define $\boldsymbol{H}$ as follows

$$\boldsymbol{H}_{x,k} := \sum_{q \in Q} \boldsymbol{p}(q) \boldsymbol{H}^{(q)}_{x,k}, \quad x \in \{0, \ldots, n^c\},\ k \in \{0, \ldots, L\}.$$

We start by showing property (a). Concretely, for any $z \in \{0, \ldots, n^c\}$ we get that

$$\sum_{k=0}^{L} \boldsymbol{H}_{x,k} = \sum_{k=0}^{L} \sum_{q \in Q} \boldsymbol{p}(q) \boldsymbol{H}^{(q)}_{x,k} = \sum_{q \in Q} \boldsymbol{p}(q) \sum_{k=0}^{L} \boldsymbol{H}^{(q)}_{x,k} = \sum_{q \in Q} \boldsymbol{p}(q) f^{(q)}(x) = f(x).$$

Next, $\boldsymbol{H}$ satisfies property (b) since for any $k \in \{0, \ldots, L\}$ we get that

$$\sum_{x=0}^{n^c} \boldsymbol{H}_{x,k} = \sum_{x=0}^{n^c} \sum_{q \in Q} \boldsymbol{p}(q) \boldsymbol{H}^{(q)}_{x,k} = \sum_{q \in Q} \boldsymbol{p}(q) \sum_{x=0}^{n^c} \boldsymbol{H}^{(q)}_{x,k} = \sum_{q \in Q} \boldsymbol{p}(q) g^{(q)}(k) = g(k).$$

Finally, for every $x \notin I^j_k$, we have that $\boldsymbol{H}_{x,k} = 0$, i.e., property (c) is satisfied for $\boldsymbol{H}$. The latter holds since for all $x \notin I^j_k$ we have that $\boldsymbol{H}^{(q)}_{x,k} = 0$ and thus $\boldsymbol{H}_{x,k} = \sum_{q \in Q} \boldsymbol{p}(q) \boldsymbol{H}^{(q)}_{x,k} = 0$. As a result we conclude that g is an (ϵ, j)-approximation of f with matrix $\boldsymbol{H}$ satisfying all the required properties. □

We now describe how to compute a probability distribution that will in turn allow us to sample approximately from $f^{u,v}_{s(w),\ell}$. At a high level we accomplish this task by employing the "doubling technique" together with the approximate representations and their convolution. As an input, the algorithm receives a weighted graph G with polynomially bounded weights, a length parameter $\ell \geq 1$, an error parameter $\epsilon > 0$ and two vertices $u, v \in V$. The procedure computes and outputs a vector $\left(j_{u,v}, g^{u,v}_{\ell}, p^{u,v}_{\ell}\right)$, where $j_{u,v} \geq 1$ is a precision parameter, $g^{u,v}_{\ell}$ is an $(\epsilon, j_{u,v})$-approximation of $f^{u,v}_{s(w),\ell}$, and $p^{u,v}_{\ell} = \mathbb{P}_u[w_\ell = v]$ is the probability that the random walk w that originates at u hits v after ℓ steps.

If $(\ell = 1)$, then there are two possibilities depending on whether $(u, v) \in E$ or not. If the former holds, then the algorithms simply returns $(1, \vec{0}, 0)$ as it is not possible to reach v after performing one step of the random walk from v. Otherwise, we simply return $(1, g^{u,v}_1, p^{u,v}_{\ell})$, where $g^{u,v}_1(\frac{1}{\boldsymbol{w}_{u,v}}) = 1$ and $p^{u,v}_{\ell} = \frac{\boldsymbol{w}_{u,v}}{\boldsymbol{d}_G(v)}$.

However, if $(\ell > 1)$, then it first halves ℓ into two parts $\ell' = \lfloor \ell/2 \rfloor$ and $\ell'' = \lceil \ell/2 \rceil$. Next, for each $y \in V$ it recursively calls itself with input parameters $(G, u, y, \ell', \epsilon)$ and $(G, y, v, \ell'', \epsilon)$. The outputs from these two calls are then combined using the convolution manipulations described above to produce the final output. Exact details for implementing this procedure are summarized in Algorithm 7. The following lemma proves the correctness and the running time of the algorithm.

Lemma C.3. *Given a weighted graph $G = (V, E, \boldsymbol{w})$ with $\boldsymbol{w}_e \in [1, n^c]$ for each $e \in E$ and $c > 0$, two vertices $u, v \in V$, a length parameter $\ell \in [1, n^d]$ and an error parameter $\epsilon > 0$,* ComputeDistrib$(G, u, v, \ell, \epsilon)$ *(Algorithm 7) correctly computes a vector $(j_{u,v}, g^{u,v}_{\ell}, p^{u,v}_{\ell})$ in $\tilde{O}(n^3 \epsilon^{-2})$ time, where $g^{u,v}_{\ell}$ is an $(\epsilon, j_{u,v})$-approximation to $f^{u,v}_{s(w),\ell}$ and $p^{u,v}_{\ell}$ is the probability that the random walk w the starts at u hits v after ℓ steps. Moreover, the output $j_{u,v}$ cannot exceed $O(\log n)$.*

Proof. We first prove that the third coordinate of the output vector equals $\mathbb{P}_u[w_\ell = v]$. We proceed by induction on the length of the walk ℓ. If $(\ell = 1)$, it is easy to check that the condition holds by construction of the algorithm. Next assume $(\ell \geq 2)$ and note that $(\ell' < \ell)$ and $(\ell'' < \ell)$. Applying induction hypothesis on each recursion call, we know $p^{u,y}_{\ell'}$ is $\mathbb{P}_u[w_\ell = y]$ and $p^{y,v}_{\ell''}$ is $\mathbb{P}_y[w_{\ell''} = u]$. The latter along with the fact that $(\ell' + \ell'' = \ell)$ imply

Algorithm 7: ComputeDistrib$(G, u, v, \ell, \epsilon)$

Input : Weighted graph $G = (V, E, \boldsymbol{w})$, with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$ and $c > 0$, two vertices $u, v \in V$, a length parameter $\ell \in [1, n^d]$ and an error parameter $\epsilon > 0$

Output: A vector $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v})$, where $j_{u,v} \geq 1$ is a precision parameter, $g_\ell^{u,v}$ is an $(\epsilon, j_{u,v})$-approximation of $f_{s(w),\ell}^{u,v}$, and $p_\ell^{u,v}$ is the probability that the random walk w that starts at u hits v after ℓ steps

if $(\ell = 1)$ **then**
  If $(u, v) \notin E$, return $(1, \mathbf{0}, 0)$
  If $(u, v) \in E$, return $(1, g_\ell^{u,v}, p_\ell^{u,v})$, where $g_1^{u,v}(\frac{1}{\boldsymbol{w}(u,v)}) \leftarrow 1$ and $p_\ell^{u,v} \leftarrow \frac{\boldsymbol{w}(u,v)}{\boldsymbol{d}(v)}$
if $(\ell \geq 2)$ **then**
  Set $\ell' \leftarrow \lfloor \ell/2 \rfloor$ and $\ell'' \leftarrow \lceil \ell/2 \rceil$
  for *every* $y \in V$ **do**
    Invoke ComputeDistrib$(G, u, y, \ell', \epsilon)$ and ComputeDistrib$(G, y, v, \ell'', \epsilon)$
    Let $(j_{u,y}, g_{\ell'}^{u,y}, p_{\ell'}^{u,y})$ and $(j_{y,v}, g_{\ell''}^{y,v}, p_{\ell''}^{y,v})$ be the corresponding outputs
  Set $g_\ell^{u,v} \leftarrow \sum_{y \in V} p_{\ell'}^{u,y} p_{\ell''}^{y,v} \cdot \left(g_{\ell'}^{u,y} * g_{\ell''}^{y,v}\right)$
  Return $\left((\max_{y \in V} \max(j_{u,y}, j_{y,v})) + 1, g_\ell^{u,v}, \sum_{y \in V} p_{\ell'}^{u,y} \cdot p_{\ell''}^{y,v}\right)$

$$\sum_{y \in V} \left(p_{\ell'}^{u,y} \cdot p_{\ell''}^{y,v}\right) = \sum_{y \in V} \left(\mathbb{P}_u\left[w_\ell = y\right] \cdot \mathbb{P}_y\left[w_{\ell''} = u\right]\right) = \mathbb{P}_u\left[w_\ell = v\right].$$

We next prove that the second coordinate $g_\ell^{u,v}$ is an (ϵ, j)-approximation of $f_{s(w),\ell}^{u,v}$. First, since $j_{u,v} = (\max_y \max(j_{v,y}, j_{y,u})) + 1$, Lemma C.1 implies that $\left(g_{\ell'}^{u,y} * g_{\ell''}^{y,v}\right)$ is an $(\epsilon, j_{u,v})$-approximation of $f_{s(w),\ell',\ell'}^{u,y,v}$ where we define $f_{s(w),\ell',\ell'}^{u,y,v}$ to be the probability mass function of $s(w)$, conditioning on $w \sim w_{v,u}$, $\ell(w) = \ell' + \ell''$ and $w_{\ell'} = y$. Second, consider the triplets $\{(u, y, v)\}_{y \in V}$, and let $g_y = g_{\ell'}^{u,y} * g_{\ell''}^{y,v}$ and $p_y = p_{\ell'}^{u,y} \cdot p_{\ell'} y, v$. Then by Lemma C.2 we get that $g_\ell^{u,v} = \sum_{y \in V} p_y \cdot g_y$ is the desired $(\epsilon, j_{u,v})$-approximation.

Finally, we prove that $j_{u,v} = O(\log n)$. We will inductively show that the first coordinate $j_{u,v}$ of the output vector from ComputeDistrib$(G, u, v, \ell, \epsilon)$ is at most $k + 1$, for $\ell \leq 2^k$. For the base case $k = 0$, which implies that $\ell = 1$ and and the claim trivially holds. Now assume $that k \geq 1$. Since $\ell \leq 2^k$, by construction we get that $\ell' \leq 2^{k-1}$ and $\ell'' \leq 2^{k-1}$. By induction hypothesis, the first coordinates returned by all of the recursion calls are no more than $(k - 1) + 1 = k$. Thus, the returned $j_{u,v}$ at most $k + 1 = O(\log n)$.

For the running time, note that in all recursion calls of the procedure ComputeDistrib there are at most n^2 possible pairs (u, v) and $O(\log n)$ possible values of ℓ. In each of these calls, we invoke the procedure Convolute exactly n times, where each invocation costs $\tilde{O}(\epsilon^{-2})$ by Lemma C.1. Thus the total running time is bounded by $\tilde{O}(n^3 \epsilon^{-2})$.

□

We now have all the tools to prove Lemma 4.13.

Proof of Lemma 4.13. Our algorithm for sampling $s(w)$ is implemented as follows. First, it invokes the procedure computeDistrib$(G, u, v, \ell, \epsilon)$ and obtains the resulting vector $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v})$. Then it samples from the distribution by choosing the interval $I_k^{j_{u,v}} = [\text{le}, \text{ri}]$ with probability $g_\ell^{u,v}(k)$,

Algorithm 8: SAMPLE$(G, u, v, \ell, \epsilon)$

Input : Weighted graph $G = (V, E, \boldsymbol{w})$, with $\boldsymbol{w}_e = [1, n^c]$ for each $e \in E$ and some $c > 0$, two vertices $u, v \in V$, a length parameter $\ell \in [1, n^d]$ and an error parameter $\epsilon > 0$

Output: A sampled $s(w)$ up to a $(1+\epsilon)$ relative error, where w is a random walk of length ℓ that starts at u and ends at v

1 Set $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v}) \leftarrow$ COMPUTEDISTRIB$(G, u, v, \ell, O(\frac{\epsilon}{\log n}))$

2 Let k_0 be the index of the interval $I_{k_0}^{j_{u,v}}$ that is sampled according to distribution $g_\ell^{u,v}$

3 Return $\left(1 + O(\frac{\epsilon}{\log n})\right)^{k_0 + j_{u,v}}$

where le $:= (1 + O(\frac{\epsilon}{\log n}))^k$ and ri $:= (1 + O(\frac{\epsilon}{\log n}))^{k+j_{u,v}}$. Finally the algorithm outputs ri. This procedure is summarized in Algorithm 8.

We next argue about the correctness. Note that by property (b) in the definition of approximation $g_\ell^{u,v}$ of $f_{s(w),\ell}^{u,v}$, this sampling process can be viewed as sampling the pair (x, i) from the distribution $\boldsymbol{H}_{x,i}$, without knowing x. Furthermore, by property (a), each x is sampled with the correct probability $f_{s(w),\ell}^{u,v}(x)$. Since $\epsilon \leq 0.5$ it follows by Lemma C.3 that ri/le $= (1+O(\frac{\epsilon}{\log n}))^{j_{u,v}} \leq (1+\epsilon)$. Thus by property (c) we get that ri is within $[x, (1+\epsilon)x]$ for the (unknown) sampled x.

The running time of our sampling procedure is asymptotically dominated by the running time of COMPUTEDISTRIB, which is in turn bounded by $O(n^3\epsilon^{-2})$, as desired. □