Final Report for Marshall Plan Scholarship Program

Approximation Algorithms for Dynamic Graph Problems

Gramoz Goranci

Research stay at Georgia Institute of Technology, January-April 2018

Abstract

In this paper we consider the *fully-dynamic* All-Pairs Effective Resistance problem, where the goal is to maintain effective resistances on a graph G among any pair of query vertices under an intermixed sequence of edge insertions and deletions in G. The effective resistance between a pair of vertices is a physics-motivated quantity that encapsulates both the congestion and the dilation of a flow. It is directly related to random walks, and it has been instrumental in the recent works for designing fast algorithms for combinatorial optimization problems, graph sparsification, and network science.

We give a data-structure that maintains $(1\pm\epsilon)$-approximations to all-pair effective resistances of a fully-dynamic unweighted, undirected multi-graph G with $\tilde{O}(m^{4/5}\epsilon^{-4})$ expected amortized update and query time, against an oblivious adversary. Key to our result is the maintenance of a dynamic *Schur complement* (also known as vertex resistance sparsifier) onto a set of terminal vertices of our choice.

This maintenance is obtained (1) by interpreting the Schur complement as a sum of random walks and (2) by randomly picking the vertex subset into which the sparsifier is constructed. We can then show that each update in the graph affects a small number of such walks, which in turn leads to our sub-linear update time. We believe that this local representation of vertex sparsifiers may be of independent interest.

This work answers an open question raised in the research assignment agreement entitled "Approximation Algorithms for Dynamic Graph Problems", between Marshall Foundation and Gramoz Goranci (see Section 3.3 of the agreement). The obtained results are joint work with David Durfee, Yu Gao and Richard (Yang) Peng. The manuscript is currently under submission, and available online at `https://arxiv.org/pdf/1804.04038.pdf`.

1 Introduction

The incorporation of numerical and optimization tools into graph algorithms has been a highly successful approach in algorithm design. It is key to the current best results for several fundamental problems in combinatorial optimization, such as approximate maximum flow [CKM$^+$11, She13, KLOS14, Pen16], multi-commodity flow [She17], shortest path and weighted matching [CMSV17]. For dynamic graphs undergoing edge modifications, core components from these algorithms such as graph partitioning, have also played an important role in recent developments on dynamic minimum spanning forest with worst-case guarantees [Wul17, NSW17, NS17]. The versatility of these tools in the static setting suggests that they can extend to wider ranges of problems on dynamic graphs.

Dynamic graph algorithms seek to maintain solutions to certain problems on graphs as they undergo edge insertions and deletions in time faster than recomputing the solution from scratch after each update. It is a subject that has been studied extensively in data structures, with problems being maintained including 2− or 3−connectivity [EGIN97, HDLT01, HRW15], shortest paths [HKN14, HKN16, BC16, ACK17], global minimum cut [Hen97, Tho07, LS11, GHT16], maximum matching [OR10, GP13, BHN16], and maximal matching [BGS15, NS16, Sol16]. On general graphs, these results give sub-linear time only under significant restrictions to the queries, e.g., global minimum cut, 3-edge connectivity, size of maximum matching (which equals to the value of a particular $s - t$ max-flow), and shortest paths from a fixed source. Relaxing such restrictions, specifically maintaining solutions to more general problems on dynamic graphs, is a major area of ongoing research in data structures.

A more unified view of these problems is through the optimization or the numerical algorithms perspective. In the undirected setting, where much work on dynamic graph algorithms has taken place, the maximum flow problem and the shortest path problem correspond to minimizing the ℓ_∞ and the ℓ_1 objectives of flows meeting a certain demand, respectively. A natural interpolation is the ℓ_2 objective, otherwise known as the electrical flow problem. In the static setting, ℓ_2 primitives have been instrumental in recent works on faster graph algorithms [CKM$^+$11, Mad13, CMMP13, Mad16, ALdOW17, CMTV17]. The objective value of this flow, effective resistances, is also a well studied quantity, and has direct applications in random walks, spanning trees [MST15], and importance of graph edges [SS11]. An in-depth treatment of such connections can be found in the monograph by Doyle and Snell [DS84], and a short summary of the connection between ℓ_p norm flows and resistances is given in Appendix B.

The applications of effective resistances, and its broader connections with optimization problems on graphs make it arguably one of the most important primitives in algorithm design. However, in the dynamic setting, prior to our work, sub-linear time algorithms for maintaining effective resistances were only known in minor-free graphs [GHP17, GHP18].

Our result. In this paper, we show the first sub-linear time fully-dynamic algorithm for maintaining approximate effective resistances in general graphs. Our algorithm is randomized and the approximation guarantee holds with probability at least $1 - 1/\text{poly}(n)$.

Theorem 1.1. *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 $\tilde{O}(m^{4/5}\epsilon^{-4})$ 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*.*

If we restrict to *simple* graphs, and only maintain the effective resistances between a small number of vertex-pairs (s_i, t_i), our algorithm can also give a running time of $\tilde{O}(n^{6/7}\epsilon^{-4})$ per operation, which is truly sub-linear irrespective of graph density. We discuss such an improvement in Section 6.

Our algorithm is motivated by two sequences of recent results. For special family of graphs, e.g. planar graphs, recent works [GHP17, GHP18] showed data structures for maintaining approximate effective resistances with $\tilde{O}(\sqrt{n}\epsilon^{-2})$ [1] update and query time. On the other hand, concurrent results on generating random spanning trees [DKP+17] and computing network centrality parameters [LZ18] implicitly rely on the ability to answer effective resistance queries along with edge deletions or contractions offline.

The key algorithmic component behind our result is the efficient maintenance of a graph reduction to a random subset of vertices, picked uniformly from the endpoints of edges. To this end, we leverage recent results based on vertex elimination schemes for solving linear systems in graph Laplacians [KLP+16b, KS16] and generating random spanning trees [DPPR17]. Specifically, we combine random walk based methods for generating effective resistance preservers on terminals with results in combinatorics that bound the speed at which such walks spread among vertices [BF96]. By showing that these walks are sufficiently local, we conclude that each update only affects a small number of such walks, and those can in turn be resampled efficiently. To the best of our knowledge, we are the first to utilize the behavior of random walks in data structures for maintaining graphs that undergo edge insertions and deletions.

Finally, we remark that the effective resistance between a pair of vertices is finite only if they are connected. Thus, our data structure also provides the first scheme for maintaining connectivity in dynamic graphs that does not utilize data structures for maintaining trees under edge updates, such as link-cut trees [ST83, ST85], Euler-tour trees [HK95], or Top-trees [AHLT05]. The many extensions of ideas stemming from dynamic connectivity, and the wide range of applications of random walks make us optimistic that our algorithmic ideas could be useful for dynamically maintaining other important graph properties.

1.1 Related Works

The recent data structures for maintaining effective resistances in planar graphs [GHP17, GHP18] 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 [Epp91, EGIN97]. Ideas from these results are central to recent works on offline maintenance for 3-connectivity [PSS17], generating random spanning trees [DKP+17], and new notions of centrality for networks [LZ18]. 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

[1] We use the notation of $\tilde{O}(f(n))$ to hide polylog factors. Specifically, for a function $f(n)$, some error parameter ϵ, and some constant $c > 0$, we have $\tilde{O}(f(n)) = O(f(n)\log^c f(n)\log^c(1/\epsilon))$.

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.

Dynamic Graph Algorithms. The maintenance of properties related to paths in dynamic graphs is a well studied topic on data structures [Fre85, EGIN97, HDLT01, KKM13, Wul17, NSW17, NS17]. A key difficulty facing paths on graphs is that general graphs are not decomposable: piecing together connectivity information from an arbitrary partition of a graph is difficult, and there are classes of graphs such as expanders that are not partitionable.

Dynamic algorithms for evaluating algebraic functions such as matrix determinant and matrix inverse has also been considered [San04]. 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 [Moi09, CLLM10, KR13]. 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 [AGK14] were core routines in the data structure for maintaining flows in bipartite undirected graphs [ADK$^+$16].

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 [KLP$^+$16b, DPPR17] 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, after which we present our data-structure in Section 3. The key properties of random walks that we use are given in Section 4, and we show the dynamic maintenance of approximate Schur complements in Section 5. In Section 6, we briefly discuss an alternate way of analyzing our data structure's performances in terms of the number of vertices. In Appendix A, we provide details on the graph approximation guarantees that our random walk sampling routines rely on. Finally, in Appendix B we provide brief details on the $p-$norm flow formulations of shortest paths, maximum flows, and effective resistances.

2 Preliminaries

In our dynamic setting, an undirected, unweighted multi-graph undergoes both insertions and deletions of edges. We let $G = (V, E)$ 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.

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 v_0, and at step $i \geq 1$, the vertex v_i is chosen randomly among the neighbors of v_{i-1}.

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 $\mathbf{L}_G$. We also need to define the indicator vector $\mathbf{1}_u \in \mathbb{R}^V$ of a vertex u such that $\mathbf{1}_u(v) = 1$ if $v = u$, and $\mathbf{1}_u(v) = 0$ otherwise.

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

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

Linear systems in graph Laplacian matrices can be solved in nearly-linear time [ST14]. 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*

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

in $\tilde{O}(m\epsilon^{-2})$ 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 and F, which in turn partition the Laplacian $\mathbf{L}_G$ into 4 blocks:

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

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

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

It is well known that $\text{SC}(G, T)$ is a Laplacian matrix of a graph on vertices in T. To simplify our exposition, we let $\text{SC}(G, T)$ denote both the Laplacian and its corresponding graph.

In this work, we will not utilize the above algebraic expression of Schur complement. Instead, our algorithm is built upon a view of the Schur complement as a collection of random walks. This particular view we be discussed in more details 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., [GHP17]).

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

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

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 $\mathbf{x} \in \mathbb{R}^n$

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

In the dynamic setting, Abraham et al. [ADK+16] recently showed that $(1\pm\epsilon)$-spectral sparsifiers of a dynamic graph G can be maintained efficiently.

Lemma 2.4 ([ADK+16], Theorem 4.1)**.** [2] *Given a graph G with polynomially bounded edge weights, with high probability, we can dynamically maintain a $(1\pm\epsilon)$-spectral sparsifier of size $\tilde{O}(n\epsilon^{-2})$ 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*

$$(1-\varepsilon)R_G(u,v) \leq R_H(u,v) \leq (1+\varepsilon)R_G(u,v).$$

3 Overview and Data Structure

In this section we start by discussing the key high level invariants that we maintain throughout our data structure. We then continue by describing how to use these invariants to dynamically maintain approximate Schur complements. Finally, we show that this leads to an algorithm for maintaining effective resistance, and thus proves our main result in Theorem 1.1.

We now review two natural attempts for addressing our 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 $\tilde{O}(m\epsilon^{-2})$ [DKP+17], 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.

The running time bottleneck then becomes computing and maintaining $\text{SC}(G,T)$ under edge updates to G. To speed up this process, we take a more local interpretation of a sparsifier of $\text{SC}(G,T)$ as a collection of random walks, each starting at an edge of G and terminating in T. In Appendix A, we review the following result, which is implicit in previous works on block elimination based algorithms for estimating determinants [DPPR17].

Theorem 3.1. *Let $G=(V,E)$ be an undirected, unweighted 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 G, repeat ρ times the following procedure:

[2] Version 1 `https://arxiv.org/pdf/1604.02094v1.pdf`.

1. *Simulate a random walk starting from u until it hits T at vertex t_1,*

2. *Simulate a random walk starting from v until it hits T at vertex t_2,*

3. *Let the total length of this combined walk (including edge e) be ℓ. Add the edge (k_1, k_2) to H with weight*

$$\frac{1}{\rho\ell}.$$

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

The output approximate Schur complement H onto T has up to $\rho m = \tilde{O}(m\epsilon^{-2})$ edges (that is, T for each edge in G). However, as we will show, H does not change too much upon inserting or deleting an edge in G. Therefore, we can maintain these changes using a dynamic spectral sparsifier $\tilde{H}$ of H, and whenever a query comes, we answer it on $\tilde{H}$ in $\tilde{O}(|T|\,\epsilon^{-2}) = \tilde{O}(\beta m\epsilon^{-2})$ time.

Thus the bulk of our effort is devoted to generating and maintaining the random walks described in Theorem 3.1. Specifically, upon insertion or deletion of an edge $e = (u, v)$ in G, we only need to regenerate walks that pass through u or v. The cost of this depends on both the length of a walk, as well as the maximum number of walks that passes through a vertex u (which we will refer to as the *load* of u). For T picked arbitrarily, e.g., the leftmost $n/2$ vertices of a length n path, both of these parameters can be large: a walk needs about n^2 steps to move across the path, and the load at the middle vertices is $\Theta(n)$.

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 Figure 1, and will discuss throughout the rest of this overview. A variant based on sampling vertices that obtains truly sublinear time per operation, but has more limitations on operations, is in Section 6.

The performance of our data structures hinge upon the properties of the random walks generated. We start by formalizing such a structure involving a set of augmented terminals, which we parameterize with a more general probability β.

Definition 3.2 (β-shorted walks)**.** Let G be an unweighted, 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.

2. For each edge $e \in E$, generate ρ walks from its endpoints either until $O(\beta^{-2}\log^3 n)$ steps have been taken, or they reach T.

The main property of the collection W is that its random walks are local. That is, with high probability all walks in W are short, and only a small number of such walks pass through each vertex u, i.e., the expected load of u with respect to W is small. These guarantees are summarized in the following theorem. Details on this behavior of random walks are deferred to Section 4.

Theorem 3.3. *Let $G = (V, E)$ be any undirected multi-graph, and $\beta \in (0, 1)$ a parameter such that $\beta m = \Omega(\log n)$. Any set of β-shorted walks W, as described in Definition 3.2, satisfies:*

1. A subset of terminal vertices T, obtained by including the endpoints of each edge independently, with probability at least $m^{-1/5}$.
2. A sampling overhead $\rho = O(\log n\varepsilon^{-2})$ (chosen according to Theorem 3.1).
3. Graph H created by repeating the following procedure for each edge $e = (u, v)$,
   (a) For $i = 1, \ldots, \rho$,
      i. Generate random walks $W(e, i)$ from u and v until either $O(m^{2/5} \log^3 n)$ steps have been taken, or they reach T.
      ii. If both walks reach T at t_1 and t_2 respectively, then
         A. Let ℓ be the number of edges on the walk $W(e, i)$.
         B. Add an edge between t_1 and t_2 to H with weight $\frac{1}{\rho\ell}$.

Figure 1: Overall data structure, which is a collection of β-shorted walks from Definition 3.2 with $\beta = m^{-1/5}$, reweighted according to Theorem 3.1.

1. *With high probability, any random walk in W starting in a connected component containing a vertex from T terminates at a vertex in T.*
2. *For any edge e, the expected load of e with respect to W is $O(\beta^{-2} \log^4 n\epsilon^{-2})$.*

Note that Part 1 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 3.3 because it can only increase the probability of an edge's endpoints being chosen.

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). We start by analyzing the update operations. Upon insertion or deletion of an edge e in the current graph G, the main idea is to regenerate all the walks that utilized e. This ensures that the collection of walks W that we maintain produces a valid approximate Schur Complement H. Since we know that the length of these walks is $\tilde{O}(\beta^{-2})$ (Definition 3.2), and Theorem 3.3 Part 2 also limits the load per edge, using a rejection sampling technique, we can regenerate these walks in $\tilde{O}(\beta^{-4}\epsilon^{-2})$ time.

However, note that declaring u to be a terminal forces us to truncate all the walks in W at the first location they meet u. Our key observation is that the cost of truncating these walks can be charged to the cost of constructing them during the pre-processing phase. Thus it follows that we can declare any vertex a terminal in $O(1)$ amortized time. On the other hand, we need to avoid extending these truncated walk when these query vertices are no longer needed in the terminals. We address this by retraining the queried vertices in T, but periodically rebuild the entire data structure (which which we resample the terminals completely) to limit the growth in $|T|$.

The above discussion leads to the data-structure DynamicSC(G, T, β): Given an undirected multi-graph $G = (V, E)$ and a subset of terminals T, maintain the collection of β-shorted walks W, and in turn the approximate Schur complement H (as outlined in Figure 1) while supporting the following operations:

- INITIALIZE(G, T, β): Construct the collection W and the sparsifier H.
- INSERT(u, v): Insert the edge (u, v) with resistance 1 in G.
- DELETE(u, v): Delete the edge (u, v) from G.
- ADDTERMINAL(u): Add the vertex u to the set of terminals in T.

We can now state the guarantees of the above data-structure using the defined operations. Specific implementation details are deferred to Section 5.

Lemma 3.4. *Given an undirected multi-graph $G = (V, E)$ a subset of terminal vertices T, and a parameter $\beta \in (0, 1)$ such that $\beta m = \Omega(\log n)$, DYNAMICSC(G, T, β) maintains the collection of β-shorted walks W, and in turn a graph H that is with high probability a sparsifier of $\mathrm{SC}(G, T)$, in a dynamic graph with at most $2m$ edges, while supporting its operations in the following running times:*

1. INITIALIZE(G, T, β) *in $O(m\beta^{-2} \log^5 n\epsilon^{-2})$ expected amortized time.*
2. INSERT(u, v) *in $O(\beta^{-4} \log^8 n\epsilon^{-2})$ expected amortized time.*
3. DELETE(u, v) *in $O(\beta^{-4} \log^8 n\epsilon^{-2})$ expected amortized time.*
4. ADDTERMINAL(u) *in $O(1)$ amortized time.*

Furthermore, each of these operations leads to an amortized number of changes to H bounded by the corresponding amortized costs.

Putting together the bounds in the above lemma proves our main result, i.e., Theorem 1.1.

Proof of Theorem 1.1. We present our two-level data-structure for dynamically maintaining all-pair effective resistances. Specifically, we keep the terminal set T of size $\Theta(\beta m)$. This entails maintaining

1. an approximate Schur complement H of G (Lemma 3.4),
2. a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 2.4),

and rebuilding our data-structure every βm operations due to the insertions into T caused by handling queries.

We now describe the update and query operations. All updates in the graph are passed to the first data-structure (which handles them by Lemma 3.4 Parts 2 and 3). Those updates in turn will trigger other updates in H, which are then handled by our second data-structure for $\tilde{H}$. Next, upon receiving a query about the (s, t) effective resistance, we declare both s and t terminals (by Lemma 3.4 Part 4), which ensures that they are now contained in $\tilde{H}$. Finally, we compute the (approximate) effective resistance between s and t in the graph $\tilde{H}$ using Lemma 2.1.

We next analyze the performance of our data-structure. Let us start with the pre-processing time. First, observe 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 our pre-processing is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$.

We now analyze the update operations. By construction, note that a single update in G triggers $\tilde{O}(\beta^{-4}\epsilon^{-2})$ updates in H, and those updates can be handled in $O(\mathrm{poly}(\log n)\epsilon^{-2})$ time using the dynamic spectral sparsifier $\tilde{H}$. Thus, we get that the expected amortized update time per insertion or deletion is $\tilde{O}(\beta^{-4}\epsilon^{-4})$.

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 $O(1)$ 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 the data-structure in Lemma 3.4 is re-built 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 $\tilde{O}(\beta m\epsilon^{-4})$.

Finally, note that each rebuild can be performed in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$ amortized time. Since we do rebuilds every βm operations, this leads to an amortized cost of

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

which is dominated by the update time.

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

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

which is minimized when $\beta = m^{-1/5}$, thus giving an expected amortized update and query time of

$$\tilde{O}\left(m^{4/5}\epsilon^{-4}\right).$$

□

Future directions. Our result raises several open questions for future works. Here we state those that we believe are closer to our results. (1) The most natural one is whether our update or query time bounds can be improved to $\text{poly}(\log n, 1/\epsilon)$. It is also interesting to investigate whether our update times can be made deterministic and/or worst-case. (2) As effective resistances can be interpreted in terms of electrical flows, we can ask whether the values in these flows, such as the flow value on edge $\hat{e} = (\hat{u}, \hat{v})$ when sending 1 unit of current from $\hat{u}$ to $\hat{v}$ can be dynamically maintained. (3) Finally, the use of Schur complement as vertex sparsifiers raises the question of whether modifications of Schur complements can be used to maintain more combinatorial problems.

4 Properties of Random Walks

In this section we give more details on the properties of β-shorted walks collection that our data-structure maintains. Specifically, we prove Theorem 3.3:

Theorem 3.3. *Let $G = (V, E)$ be any undirected multi-graph, and $\beta \in (0, 1)$ a parameter such that $\beta m = \Omega(\log n)$. Any set of β-shorted walks W, as described in Definition 3.2, satisfies:*

1. *With high probability, any random walk in W starting in a connected component containing a vertex from T terminates at a vertex in T.*
2. *For any edge e, the expected load of e with respect to W is $O(\beta^{-2}\log^4 n\epsilon^{-2})$.*

We start with the first property, which claims that we traversed sufficiently long to reach a vertex in T with high probability. For this, we need the following result by Barnes and Feige [BF96].

Theorem 4.1 ([BF96], Theorem 1.2)**.** *There is an absolute constant c_{BF} such that for any undirected, unweighted, 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 4.2. *In any undirected unweighted multi-graph G with m edges, for any starting vertex u, any length ℓ, 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 $c_{BF} \cdot \ell$.

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 4.1, a walk takes more than $O(\ell)$ 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 $u \in V$. □

This means that a walk of length $\tilde{O}(\beta^{-2})$ is highly likely to visit at least $\beta^{-1} \log n$ distinct edges, among which at least one should be added to T with high probability. If the connected component where the walk started in has fewer than $\beta^{-1} \log n$ edges, we get that the walk should have visited the entire component with high probability, and thus any (initial) vertex in T that belongs to that component.

Proof of Theorem 3.3 Part 1. 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

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

vertices, then Corollary 4.2 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 them 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

$$\begin{aligned} \Pr_{T,w}\left[V\left(w\right) \cap T = \emptyset\right] &= \Pr_{w,T}\left[V\left(w\right) \cap T = \emptyset\right] \\ &\leq \Pr_{w}\left[\left|E\left(w\right)\right| \leq z\right] + \Pr_{w:|E(w)|\geq z}\left[\Pr_{T}\left[V\left(w\right) \cap T = \emptyset\right]\right]. \end{aligned} \tag{1}$$

By Corollary 4.2 and the choice of ℓ, if we set

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

then the first term in Equation (1) 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.,

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

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 $V(w)$ 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. □

The bound on walk lengths also leads to a bound on the load of an edge, which is Part 2 of Theorem 3.3. We next show that this is the case. For the sake of simplicity, we will ignore the sampling overhead $\rho = O(\log n/\epsilon^2)$ in our preliminary discussions.

First, we observe that instead of terminating walks once they hit T (as described in Figure 1), we can run all the walks from all edges up to $\ell = O(\beta^{-2} \log^3 n)$ steps.

Note that the number of walks starting at each vertex u is $\deg(u)$ because we are starting one random walk per endpoint of each edge. For each $u \in V$, we let $W(u)$ be the union over $\deg(u)$ random walks of length ℓ starting from u. Furthermore, recall that $W = \cup_u W(u)$ is the collection of β-shorted walks that our sparsification routine maintains.

We want to obtain bounds on the load of any vertex $\hat{u} \in V$ (respectively, edge $\hat{e} \in E$) incurred by the random walks in W. For the purposes of the proof, it will be useful to introduce some random variables. The *load* of $\hat{u}$ (respectively, $\hat{e}$), denoted by $N_{\hat{u}}$ (respectively, $N_{\hat{e}}$), is the number of walks that pass through vertex $\hat{u}$ (respectively, edge $\hat{e}$) from the random walks in W. For $t \geq 0$, let $X_u(t)$ be the set of vertices visited in a random walk starting at u after t steps.

The first quantity we are interested in is the contribution of random walks from each $u \in V$ in the load of $\hat{u}$, which we denote by $Y_u(\hat{u})$. Concretely, $Y_u(\hat{u})$ is the total number of walks that pass through $\hat{u}$, from the random walks in W_u. Using the above random variables, we have that

$$Y_u(\hat{u}) = \sum_{0 \leq t \leq \ell} \deg(u) \cdot \mathbf{1}_{(\hat{u} \in X_u(t))}.$$

Now, observing that $N_{\hat{u}} = \sum_{u \in V} Y_u(\hat{u})$, we can expand the expectation of $N_{\hat{u}}$ as follows

$$\begin{aligned}
\mathbb{E}[N_{\hat{u}}] &= \sum_{u \in V} \mathbb{E}[Y_u(\hat{u})] \\
&= \sum_{u \in V} \sum_{0 \leq t \leq \ell} \deg(u) \cdot \Pr[\hat{u} \in X_u(t)] \\
&= \sum_{0 \leq t \leq \ell} \left(\sum_{u \in V} \deg(u) \cdot \Pr[\hat{u} \in X_u(t)] \right). \qquad (2)
\end{aligned}$$

It turns out that that the term contained in the brackets of Equation (2) equals $\deg(\hat{u})$. Formally, we have the following lemma.

Lemma 4.3. *Let G be an undirected, unweighted graph. For any vertex $\hat{u} \in V$ and any length $t \geq 0$, we have*

$$\sum_{u \in V} \deg(u) \cdot \Pr[\hat{u} \in X_u(t)] = \deg(\hat{u}).$$

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 $\hat{e}$ is 1 at any time. Below we verify this fact in a more principled manner.

Proof of Lemma 4.3. The proof is by induction on the length of the walks t. When $t = 0$, we have

$$\Pr\left[\hat{u} \in X_u(0)\right] = \begin{cases} 1 & \text{if } \hat{u} = u, \\ 0 & \text{otherwise,} \end{cases}$$

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

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

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

Substituting this into the summation to get

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

which upon rearranging of the two summations gives:

$$\sum_{\hat{v}:(\hat{u},\hat{v})\in E} \frac{1}{\deg(\hat{v})} \left(\sum_{u\in V} \deg(u) \cdot \Pr\left[\hat{v} \in X_u(t-1)\right] \right).$$

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

Proof of Theorem 3.3 Part 2. Plugging Lemma 4.3 into Equation 2 gives that

$$\mathbb{E}\left[N_{\hat{u}}\right] \leq \deg(\hat{u}) \cdot \ell.$$

Incorporating the sampling overhead ρ, which we initially ignored, we get

$$\mathbb{E}\left[N_{\hat{u}}\right] \leq \deg(\hat{u}) \cdot \ell \cdot \rho \leq O\left(\deg(\hat{u}) \cdot \beta^{-2} \log^4 n\epsilon^{-2}\right), \tag{3}$$

thus proving the desired bound on the load of $\hat{u}$.

To get the bound on the load of any edge $\hat{e} = (\hat{u}, \hat{v})$, we use the fact that

$$\mathbb{E}\left[N_{\hat{e}}\right] = \frac{1}{\deg(\hat{u})} \mathbb{E}\left[N_{\hat{u}}\right] + \frac{1}{\deg(\hat{v})} \mathbb{E}\left[N_{\hat{v}}\right].$$

Plugging the bound from Equation (3) in the above equation, we get that

$$\mathbb{E}\left[N_{\hat{e}}\right] \leq O\left(\beta^{-2} \log^4 n\epsilon^{-2}\right),$$

which proves the bound on the load of $\hat{e}$ and completes the proof. □

5 Dynamic Schur Complement

In this section we show that the approximate Schur complement given in Figure 1 can be maintained dynamically. The primary difficulty here is dynamically maintaining the collection of β-shorted walks (see Definition 3.2). We next show how to do this efficiently and combine it with an amortized cost analysis to prove Lemma 3.4.

Lemma 3.4. *Given an undirected multi-graph $G = (V, E)$ a subset of terminal vertices T, and a parameter $\beta \in (0, 1)$ such that $\beta m = \Omega(\log n)$, DynamicSC(G, T, β) maintains the collection of β-shorted walks W, and in turn a graph H that is with high probability a sparsifier of $\mathrm{SC}(G, T)$, in a dynamic graph with at most $2m$ edges, while supporting its operations in the following running times:*

1. Initialize(G, T, β) *in $O(m\beta^{-2} \log^5 n\epsilon^{-2})$ expected amortized time.*
2. Insert(u, v) *in $O(\beta^{-4} \log^8 n\epsilon^{-2})$ expected amortized time.*
3. Delete(u, v) *in $O(\beta^{-4} \log^8 n\epsilon^{-2})$ expected amortized time.*
4. AddTerminal(u) *in $O(1)$ amortized time.*

Furthermore, each of these operations leads to an amortized number of changes to H bounded by the corresponding amortized costs.

For our running time analysis, it is important to first 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 $\lfloor xn \rfloor$. Therefore, we will only need to consider the length of the walks in our runtime. As we will later see, the initialization costs will then follow from our construction of the approximate Schur complement H in Figure 1.

In addition, we note 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 which walks in W 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 5.1. *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 either*

1. *All walks in W_v or W_e in $O(|W_v|)$ or $O(|W_e|)$ time, respectively, or*
2. *The i^{th} walk (in order of time generated) of W_v or W_e in $O(\log n)$ time, for any value i,*

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. □

AddTerminal(u)
Input: vertex u such that $u \notin T$.

1. $T \leftarrow T \cup \{u\}$.
2. Shorten all random walks to the first location they meet u.
3. Update the corresponding edges in H.

Figure 2: Pseudocode for Adding a vertex to the set of terminals T

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. Thus, we can fully devote our attention to supporting the Insert, Delete, and AddTerminal operations in W. The procedure AddTerminal will be straightforward and its cost will be incorporated into the amortized analysis. The pseudocode for this operation is summarized in Figure 2.

Procedures Insert and Delete are more involved. The primary difficulty is that an edge update in the current graph changes the random walk distribution in the new graph and our walks may no longer be sampled according to this distribution. Recomputing each walk would be far too costly, so we instead observe that a random walk is a localized procedure. In particular, if some edge (u, v) is inserted or deleted, the only changes in the random walk procedure occur when the walk visits vertex u or v. This implies that all random walks in the original graph which did not visit vertex u or v have the same probability of occurring in the new graph with (u, v) inserted or deleted. We will then use certain properties of our collection of random walks proven in the previous section, along with a rejection sampling technique, to give stronger bounds on the number of walks we need to regenerate when the graph undergoes an edge update.

5.1 Deletions

As mentioned above, due to the localized nature of random walks, if we delete an edge (u, v), then all random walks in our collection that did not visit vertex u or v remain unaffected. A simple update procedure would then regenerate all walks that visit vertex u or v. However, a consequence of Theorem 3.3 Part 2 is that the expected number of walks visiting vertex u is $O(\deg(u) \cdot \beta^{-2} \log^4 n\epsilon^{-2})$, which is too costly if $\deg(u)$ is large. Ideally, we would then only deal with walks that use edge (u, v), and we will next show that this is in fact that case.

To this end, note that the only random walks whose probability is affected are those that visit vertex u or v. Consider a random walk that visits vertex u and let $\deg(u)$ be the degree of u in the new graph. For every remaining edge incident to u, the probability of it being used in the graph with (u, v) deleted is $1/\deg(u)$. Note that in the original graph they were used with probability $1/(\deg(u) + 1)$. However, if we condition upon the random walk not using the edge (u, v) in the original graph, then it is easy to see that any other edge incident to u is chosen with probability $1/\deg(u)$, exactly as desired. Consequently, the only random walk probabilities that are affected are the ones that utilize edge (u, v). In Figure 3, we give a routine which updates the walks that used the deleted edge (u, v). The running time guarantees of our update algorithm are given in the following lemma.

DELETE(u, v)
Input: vertices u and v for which (u, v) is an edge in G.
Output: an updated data-structure for $G \setminus \{(u, v)\}$.

1. Delete (u, v) from G.
2. Delete all walks starting from (u, v), as well as their associated edges in H.
3. For each walk w that uses the edge (u, v):
   (a) Regenerate the walk from at the point it first reaches u or v using the remaining edges, until either $O(\beta^{-2} \log^3 n)$ steps have been taken, or it reaches T.
   (b) Update in H the edge corresponding to this walk.

Figure 3: Pseudocode for maintaining the collection of β-shorted walks W, and the corresponding graph H after deleting edge (u, v).

Lemma 5.2. *The operation* DELETE(u, v) *takes* $O(\beta^{-4} \log^8 n\epsilon^{-4})$ *expected amortized time, and updates* $O(\beta^{-2} \log^4 n\epsilon^{-2})$ *edges in* H.

Proof. Theorem 3.3 Part 2 gives that the number of walks that utilize (u, v) is at most $O(\beta^{-2} \log^4 n\epsilon^{-2})$, which in turn gives us the bound on the number of edges that need to be updated.

By the bound on the walk lengths in the collection of β-shorted walks in Definition 3.2, resampling each of these walks takes time $O(\beta^{-2} \log^3 n\epsilon^{-2})$. Adding an extra $O(\log n)$ for translating between the collection of walks and H by Lemma 5.1, gives $O(\beta^{-4} \log^8 n\epsilon^{-4})$ amortized expected time. □

5.2 Insertions

Handling insertions will be more involved because we now have to consider every random walk that visits u or v. However, to bound the number of these walks that need to be regenerated, we can use rejection sampling. This will utilize the fact that for any random walk that visits vertex u or v, the probability that it uses edge (u, v) is proportional to the degree of u and v, respectively. In particular, if we let $\deg(u)$ be the degree of u in our new graph, then for each of our random walks that visit u we instead use the edge (u, v) with probability $1/\deg(u)$ and generate the remaining random walk from that point on. Note that for any other edge incident to u, the probability of that edge being used in the original random walk was $1/(\deg(u) - 1)$ and the probability that we keep the walk is $(\deg(u) - 1)/\deg(u)$, whose product gives the desired probability of $1/(\deg(u))$.

However, if we run this sampling procedure for each walk incident to u, the expected number of walks sampled will be $\tilde{O}(\deg(u) \cdot \beta^{-2}\epsilon^{-2})$, which is too costly if the degree is large. To address this, we implicitly run this sampling procedure on all walks by instead finding the instances in which we use (u, v) in just $O(\log n)$ time. More specifically, at each sample we use (u, v) with probability $1/\deg(u)$, so the probability that we use (u, v) for the first time in the i-th sample will be

$$\frac{1}{\deg(u)} \left(1 - \frac{1}{\deg(u)}\right)^{i-1}.$$

INSERT(u, v)
Input: vertices u and v.
Output: an updated data-structure for $G \cup \{(u, v)\}$.

1. With probability β
   (a) ADDTERMINAL(u) and ADDTERMINAL(v)
2. Add edge (u, v) to G.
3. Sample ρ walks starting from u and v ending in T, add corresponding edges to H.
4. For each end point $\hat{u} \in \{u, v\}$
   (a) Let d be the new degree of $\hat{u}$.
   (b) (implicitly) With probability $1/d$ for each occurrence of $\hat{u}$ in a random walk w:
      i. Regenerate w starting from that occurrence, with (u, v) as the edge taken.
      ii. Update in H the edge corresponding to this walk.

Figure 4: Pseudocode for maintaining the collection of β-shorted walks W, and the corresponding graph H after inserting edge (u, v).

In order to efficiently find the value i at which we first sample (u, v), we simply draw a uniformly random number $x \in [0, 1]$. Using geometric series properties, we can compute the probability that $i \leq j$ for any value j, and we binary search on $i \leq 2^j$ by increasing j and checking if x is above or below this value. Our expected running time will then be $O(\log \deg(u)) \leq O(\log n)$. Iterating this procedure efficiently finds all walks that use edge (u, v).

Finally, note that we need to consider an additional case where the inserted edge causes both of its endpoints to be added to T. If this occurs, we simply truncate all walks to those vertices. The above discussion is summarized in the pseudocode given in Figure 4

Lemma 5.3. *The operation* INSERT(u, v) *takes* $O(\beta^{-4} \log^8 n\epsilon^{-4})$ *expected amortized time, and updates* $O(\beta^{-2} \log^4 n\epsilon^{-2})$ *edges in* H.

Proof. For now, let us assume that the ADDTERMINAL operation takes $O(1)$ amortized time. We next bound the expected amortized time for the remainder of the procedure.

Theorem 3.3 Part 2 implies that the number of occurrences of u in all the walks is bounded by

$$O\left(\deg(u) \cdot \beta^{-2} \log^4 n\epsilon^{-2}\right).$$

Since, by the discussion above, each of these is updated with probability $1/\deg(u)$, the expected number of walks that we regenerate is $O(\beta^{-2} \log^4 n\epsilon^{-2})$, giving our bound on the number of edge updates. The total amortized cost then follows analogous to the proof of Lemma 5.2. □

5.3 Amortized Analysis

The overall bound requires amortizing the costs of shortening the walks caused by ADDTERMINAL(u) to the cost of creating these walks in the first place. This can be handled using a standard amortized

analysis.

Proof of Lemma 3.4. We first examine the operation INITIALIZE(G, T, β). Forming H requires generating $\rho = O(\log n\epsilon^{-2})$ walks from each edge of G up to a length of at most $O(\beta^{-2}\log^3 n)$, which with the overhead of maintaining reverse pointers from Lemma 5.1 gives a cost of

$$O\left(m\beta^{-2}\log^5 n\epsilon^{-2}\right).$$

The expected amortized time of the operations DELETE(u, v) and INSERT(u, v) follow from Lemma 5.2 and 5.3, respectively, where we note that Lemma 5.3 assumed that ADDTERMINAL(u) only required $O(1)$ amortized time. It then remains to show that this is the case.

Adding a vertex to T only shortens the existing walks, and Lemma 5.1 allows us to find such walks in time proportional to the amount of edges deleted from the walk. Since this walk needed to be generated in either the INITIALIZE, INSERT, or DELETE, then the deletion of these edges will take equivalent time to generating them. As a result, we can account for this amortized cost by just doubling the cost of INITIALIZE, INSERT, and DELETE, which does not affect their asymptotic runtime.

□

6 Better Guarantees for (s, t)-Resistance on Simple Dense Graphs

In this section we discuss a different parameterization of our data structure where we restrict to *simple* graphs and only maintain the effective resistance between a small number of *fixed* vertex-pairs (s_i, t_i). For the sake of exposition, we only consider the case where we want to maintain the effective resistance between a single (s, t) pair. It is then easy to extend our data-structure to support up to $\tilde{O}(n^{6/7})$ fixed vertex-pairs.

Theorem 6.1. *For any given error threshold $\epsilon > 0$, and a vertex-pair (s, t), there is a data structure for maintaining a n-vertex simple graph $G = (V, E)$ while supporting edge insertions and deletions in G as well as (s, t)-effective resistances queries in $\tilde{O}(n^{6/7}\epsilon^{-4})$ expected update and query time.*

In the above theorem, the improvement on the running time (for sufficiently dense graphs) comes from a result by Barnes and Feige [BF96], who give a bound on the number of distinct vertices visited in a random walk of certain length. In what follows, we will describe how to modify both the data-structure and the algorithm for maintaining dynamic Schur complements, which in turn will prove the theorem.

We start with the modification of the data-structure. In comparison to the data-structure in Figure 1, the key difference here is that we directly sample vertices with some probability and include them in the set of terminals T. This forces us to change the length of the random walks, as shown in Figure 5. However, we remark that the Theorem 3.1 holds for arbitrary T, and thus it readily applies to our modification of the terminal set.

Now, for any parameter $\beta \in (0, 1)$, similarly to Definition 3.2, we can define the collection of β-shorted walks W for the vertex version of our data-structure. Specifically, (1) we pick a subset of terminal vertices T, obtained by including each vertex independently, with probability at least β and (2) the length of the random walks we generate is replaced by $O(\beta^{-3}\log^4 n)$.

We next review the result by Barnes and Feige [BF96].

Lemma 6.2 ([BF96], Theorem 1.1)**.** *There is an absolute constant c_{BF} such that for any undirected, unweighted, simple G with n vertices, any vertex u and any value $\hat{n} \leq n$, the expected time for a random walk starting from u to visit at least $\hat{n}$* distinct *vertices is at most $c_{BF}\hat{n}^3$.*

1. A subset of terminal vertices T, obtained by including each vertex independently, with probability at least $n^{-1/7}$.
2. A sampling overhead $\rho = O(\log n\varepsilon^{-2})$ (chosen according to Theorem 3.1).
3. Graph H created by repeating the following procedure for each edge $e = (u, v)$,
   (a) For $i = 1, \ldots, \rho$,
      i. Generate random walks $W(e, i)$ from u and v until either $O(n^{3/7} \log^3 n)$ steps have been taken, or they reach T.
      ii. If both walks reach T at t_1 and t_2 respectively, then
         A. Let ℓ be the number of edges on the walk $W(e, i)$.
         B. Add an edge between t_1 and t_2 to H with weight $\frac{1}{\rho\ell}$.

Figure 5: Overall data structure, which is a collection of β-shorted walks from with $\beta = n^{-1/7}$, reweighted according to Theorem 3.1.

Now, using the above lemma and following essentially the same reasoning as in Section 4, we get the analogue of Theorem 3.3.

Theorem 6.3. *Let $G = (V, E)$ be any undirected simple graph, and $\beta \in (0, 1)$ a parameter such that $\beta n = \Omega(\log n)$. Any set of β-shorted walks W, as described above, satisfies:*

1. *With high probability, any random walk in W starting in a connected component containing a vertex from T terminates at a vertex in T.*
2. *For any edge e, the expected load of e with respect to W is $O(\beta^{-3} \log^5 n\epsilon^{-2})$.*

Following the ideas we presented in Section 3, we can use the above theorem to construct the data-structure that maintains a dynamic Schur complement, i.e., DynamicSC(G, T, β). However, one difference here is that the terminal additions are not supported, and the update times are no longer amortized. We implement the insertions and deletions of edges using Lemmas 5.2 and 5.3. Note that because we randomly pick vertex subsets, we do not need to regenerate augmentations to T, i.e., Line 1 of Figure 4 is no longer useful in our Insert routine. The guarantees of these modifications are summarized in the lemma below.

Lemma 6.4. *Given an undirected multi-graph $G = (V, E)$ a subset of terminal vertices T, and a parameter β such that $\beta n = \Omega(\log n)$, DynamicSC(G, T, β) maintains the collection of β-shorted walks W, and in turn a graph H that is with high probability a sparsifier of $\mathrm{SC}(G, T)$, while supporting its operations in the following running times:*

1. Initialize(G, T, β) *in $O(m\beta^{-3} \log^5 n\epsilon^{-2})$ expected time.*
2. Insert(u, v) *in $O(\beta^{-6} \log^9 n\epsilon^{-2})$ expected time.*
3. Delete(u, v) *in $O(\beta^{-6} \log^9 n\epsilon^{-2})$ expected time.*

Furthermore, each of these operations leads to a number of changes to H bounded by the corresponding costs.

Putting together the bounds in the above lemma proves our vertex based bounds, i.e., Theorem 6.1.

Proof of Theorem 6.1. We present our two-level data-structure for dynamically maintaining (s,t)-effective resistances. Specifically, we include both s and t to T, and keep the terminal set T of size $\Theta(\beta n)$. This entails maintaining

1. an approximate Schur complement H of G (Lemma 6.4),
2. a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 2.4).

We now describe the update and query operations. Specifically, whenever an edge insertion or deletion is performed in the current graph, we pass the corresponding update to the first data-structure (which handles this by Lemma 6.4 Parts 2 and 3). This update in turn will trigger other updates in H, which are then handled by our second data-structure for $\tilde{H}$. Next, upon receiving a query about the (s,t) effective resistance, we compute the (approximate) effective resistance between s and t in the graph $\tilde{H}$ using Lemma 2.1.

Similarly to the proof of Theorem 1.1, we can show that the pre-processing time is $\tilde{O}(m\beta^{-3}\epsilon^{-4})$, the expected update time is $\tilde{O}(\beta^{-6}\epsilon^{-4})$, and the query time is $\tilde{O}(\beta n\epsilon^{-4})$.

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

$$\tilde{O}\left((\beta n + \beta^{-6})\epsilon^{-4}\right),$$

which is minimized when $\beta = n^{-1/7}$, thus giving an expected update and query time of

$$\tilde{O}\left(n^{6/7}\epsilon^{-4}\right).$$

□

Acknowledgements

We thank Daniel D. Sleator for helpful comments on an earlier draft of the manuscript.

Appendix

A Schur Complement Sparsifier from Sum of Random Walks

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

Theorem 3.1. *Let $G = (V, E)$ be an undirected, unweighted 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 G, repeat ρ times the following procedure:

1. *Simulate a random walk starting from u until it hits T at vertex t_1,*
2. *Simulate a random walk starting from v until it hits T at vertex t_2,*

3. *Let the total length of this combined walk (including edge e) be ℓ. Add the edge (k_1, k_2) to H with weight*

$$\frac{1}{\rho\ell}.$$

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

Note that this rescaling by $1/\rho\ell$ is quite natural: in 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 [SS11, KLP16a, ADK+16, DPPR17, JKPS17], 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 [Tro12], which we paraphrase in our notion of approximations.

Theorem A.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 [SS11] give the following graph theoretic interpretation of the theorem above.

Corollary A.2. *Let $E_1 \ldots E_k$ 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.,*

$$\boldsymbol{w}_e \mathcal{R}_{\mathrm{eff}}^{E_i}(e) \leq \frac{\epsilon^2}{O\left(\log n\right)}.$$

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 A.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 $w = u_0, \ldots, u_\ell$ 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 $u_1, \ldots, u_{\ell-1} \in V \setminus T$.

Fact A.3 ([DPPR17], Lemma 5.4)**.** [3] *For any undirected, unweighted graph G and any subset of vertices $T \subseteq V$, the Schur complement $\mathrm{SC}(G,T)$ is given as an union over all multi-edges corresponding to terminal-free walks $u_0, \ldots, u_\ell$ with weight*

$$\prod_{i=1}^{\ell-1} \frac{1}{\deg(u_i)}.$$

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

Fact A.4. *In an unweighted undirected graph G, the effective resistance between two vertices that are distance ℓ apart is at most ℓ.*

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 $\mathbf{L}_H \approx_\epsilon \mathrm{SC}(G,T)$ with high probability, we need to show that (1) $\mathbb{E}[H] = \mathrm{SC}(G,T)$ and (2) for any edge f in W_e, its leverage score $\boldsymbol{w}_f \mathcal{R}_{\mathrm{eff}}^{W_e}(f)$ is at most $\leq \epsilon^2/\log n$ (by Corollary A.2). Note that (2) immediately follows from the effective resistance bound of Fact A.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 $\mathrm{SC}(G,T)$ into random multi-edges, which correspond to random terminal-free walks in Fact A.3. The main idea is to sub-divide each walk $u_0 \ldots u_\ell$ 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

$$\frac{1}{\ell} \cdot \prod_{i=1}^{\ell-1} \frac{1}{\deg(u_i)}.$$

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

$$\prod_{i=1}^{\ell-1} \frac{1}{\deg(u_i)}.$$

Thus, to retain expectation, when such a walk is picked, our routine correctly adds it to H with weight $1/(\rho\ell)$.

[3] We state the lemma for unit weighted graphs. The version cited is `https://arxiv.org/pdf/1705.00985v1.pdf` There may be updates to this arXiv manuscript in the near future.

Formally, we get the following chain of equalities

$$\begin{aligned}
\mathbb{E}[H] &= \rho \cdot \sum_e \mathbb{E}[W_e] \\
&= \rho \cdot \sum_e \sum_{w=u_0,u_1\ldots u_{\ell(w)}:w\ni e} \frac{1}{\rho\ell(w)} \cdot \prod_{i=1}^{\ell(w)-1} \frac{1}{\deg(u_i)} \\
&= \sum_{w=u_0,u_1\ldots u_{\ell(w)}} \sum_{e:e\in w} \frac{1}{\ell(w)} \cdot \prod_{i=1}^{\ell(w)-1} \frac{1}{\deg(u_i)} \\
&= \sum_{w=u_0,u_1\ldots u_{\ell(w)}} \prod_{i=1}^{\ell(w)-1} \frac{1}{\deg(u_i)} \\
&= \mathrm{SC}(G,T).
\end{aligned}$$

□

B A Unified View of Flows and Paths

We provide a brief overview of numerical formulations of flows that capture combinatorial problems including shortest paths, maximum flows, and effective resistances. This view is well known in the literature of using continuous methods for combinatorial optimization problems [CKM+11, Mąd11].

For an orientation of edges of a graph G with n vertices and m edges, we can define the edge-vertex incidence matrix $\mathbf{B} \in \mathbb{R}^{m\times n}$ as:

$$\mathbf{B}_{e,u} := \begin{cases} 1 & \text{if } e \text{ is the head of } u, \\ -1 & \text{if } e \text{ is the tail of } u, \\ 0 & \text{otherwise.} \end{cases}$$

Then a flow from s to t is a vector $\boldsymbol{f}$ on edges such that

$$\mathbf{B}^T\boldsymbol{f} = \boldsymbol{\chi}_{st},$$

where $\boldsymbol{\chi}_{st}$ is the indicator vector with 1 at t, -1 at s, and 0 everywhere else.

Furthermore, for any $p \geq 1$, we can define the p-norm of a flow $\boldsymbol{f}$ via

$$\|\boldsymbol{f}\|_p := \left(\sum_e |\boldsymbol{f}_e|^p\right)^{1/p}.$$

Shortest paths, maximum flows, and electrical flows (effective resistances) on undirected graphs are all instances of the following optimization problem:

$$\begin{aligned} \min \quad & \|\boldsymbol{f}\|_p \\ \text{subject to:} \quad & \mathbf{B}^T\boldsymbol{f} = \boldsymbol{\chi}_{st}. \end{aligned}$$

Specifically, we distinguish the following cases:

1. When $p = 1$, we get the shortest path problem between s and t. Replacing $\boldsymbol{\chi}_{st}$ with a more general demand vector $\boldsymbol{d}$, we get the transshipment problem [She17].

2. When $p = \infty$, we get the problem of minimizing congestion, which is equivalent to routing the maximum amount of flow from s to t subject to at most 1 unit per edge, or in turn the undirected max-flow/min-cut problem.

3. When $p = 2$, we get the $s - t$ electrical flow problem. Here, since $\|\boldsymbol{f}\|_2^2$ is differentiable, we have

$$\boldsymbol{f}^T \Delta = 0$$

for any 'change' that is a circulation, i.e, $\mathbf{B}^T \Delta = 0$. The properties of column or row spaces then imply that

$$\boldsymbol{f} = \mathbf{B}\boldsymbol{\phi},$$

for a voltage vector $\boldsymbol{\phi}$, for which we can then solve to get

$$\mathbf{B}^T \mathbf{B} \boldsymbol{\phi} = \boldsymbol{\chi}_{s,t},$$

or $\boldsymbol{\phi} = \mathbf{L}^\dagger \boldsymbol{\chi}_{s,t}$, since $\mathbf{L} = \mathbf{B}^T \mathbf{B}$. The energy of the resulting flow $\boldsymbol{f}$ is then:

$$\|\boldsymbol{f}\|_2^2 = \left\|\mathbf{B}\mathbf{L}^\dagger \boldsymbol{\chi}_{s,t}\right\|_2^2 = \boldsymbol{\chi}_{s,t}^T \mathbf{L}^\dagger \boldsymbol{\chi}_{s,t},$$

which is exactly the definition of s-t effective resistance from Section 2.

References

[ACK17] 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.

[ADK+16] 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.

[AGK14] Alexandr Andoni, Anupam Gupta, and Robert Krauthgamer. Towards $(1 + \epsilon)$-approximate flow sparsifiers. In *Symposium on Discrete algorithms (SODA)*, pages 279–293, 2014.

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

[ALdOW17] Zeyuan Allen-Zhu, Yuanzhi Li, Rafael Mendes de Oliveira, and Avi Wigderson. Much faster algorithms for matrix scaling. In *Symposium on Foundations of Computer Science (FOCS)*, pages 890–901, 2017. Available at: https://arxiv.org/abs/1704.02315.

[BC16] 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.

[BF96] Greg Barnes and Uriel Feige. Short random walks on graphs. *SIAM Journal on Discrete Mathemathics*, 9(1):19–28, 1996.

[BGS15] 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.

[BHN16] 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.

[CKM+11] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Symposium on Theory of Computing (STOC)*, pages 273–282, 2011.

[CLLM10] 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.

[CMMP13] Hui Han Chin, Aleksander Madry, Gary L Miller, and Richard Peng. Runtime guarantees for regression problems. In *Innovations in Theoretical Computer Science (ITCS)*, pages 269–282, 2013.

[CMSV17] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{O}(m^{10/7} \log W)$ time (extended abstract). In *Symposium on Discrete Algorithms (SODA)*, pages 752–771, 2017.

[CMTV17] Michael B. Cohen, Aleksander Madry, Dimitris Tsipras, and Adrian Vladu. Matrix scaling and balancing via box constrained newton's method and interior point methods. In *Symposium on Foundations of Computer Science (FOCS)*, pages 902–913, 2017. Available at: https://arxiv.org/abs/1704.02310.

[DKP+17] 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.

[DPPR17] 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.

[DS84] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*, volume 22 of *Carus Mathematical Monographs*. Mathematical Association of America, 1984.

[EGIN97] 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.

[Epp91] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. In *Workshop on Algorithms and Data Structures (WADS)*, pages 392–399, 1991.

[Fre85] 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.

[GHP17] 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.

[GHP18] Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate schur complement on separable graphs. *CoRR*, abs/1802.09111, 2018. Available at: https://arxiv.org/abs/1802.09111.

[GHT16] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in poly-logarithmic amortized update time. In *European Symposium on Algorithms (ESA)*, pages 46:1–46:17, 2016.

[GP13] Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *Symposium on Foundations of Computer Science (FOCS)*, pages 548–557, 2013.

[HDLT01] 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.

[Hen97] 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.

[HK95] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Symposium on Theory of Computing (STOC)*, pages 519–527, 1995.

[HKN14] 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.

[HKN16] 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.

[HRW15] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *European Symposium on Algorithms (ESA)*, pages 742–753, 2015.

[JKPS17] Gorav Jindal, Pavel Kolev, Richard Peng, and Saurabh Sawlani. Density independent algorithms for sparsifying k-step random walks. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM*, pages 14:1–14:17, 2017.

[KKM13] 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.

[KLOS14] 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.

[KLP16a] 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.

[KLP+16b] 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.

[KR13] Robert Krauthgamer and Inbal Rika. Mimicking networks and succinct representations of terminal cuts. In *Symposium on Discrete algorithms (SODA)*, pages 1789–1799, 2013.

[KS16] 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.

[LS11] 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.

[LZ18] 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.

[Mąd11] Aleksander Mądry. *From graphs to matrices, and back: new techniques for graph algorithms*. PhD thesis, Massachusetts Institute of Technology, 2011. Available at: http://people.csail.mit.edu/madry/docs/thesis.pdf.

[Mad13] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Symposium on Foundations of Computer Science (FOCS)*, pages 253–262, 2013.

[Mad16] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *Symposium on Foundations of Computer Science (FOCS)*, pages 593–602, 2016.

[Moi09] 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.

[MST15] 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.

[NS16] 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.

[NS17] 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.

[NSW17] 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.

[OR10] 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.

[Pen16] Richard Peng. Approximate undirected maximum flows in $O(mpolylog(n))$ time. In *Symposium on Discrete Algorithms (SODA)*, pages 1862–1867, 2016.

[PSS17] 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.

[San04] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *Symposium on Foundations of Computer Science (FOCS)*, pages 509–517, 2004.

[She13] Jonah Sherman. Nearly maximum flows in nearly linear time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 263–269, 2013.

[She17] Jonah Sherman. Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow. In *Symposium on Theory of Computing (STOC)*, pages 452–460, 2017.

[Sol16] Shay Solomon. Fully dynamic maximal matching in constant update time. In *Foundations of Computer Science (FOCS)*, pages 325–334, 2016.

[SS11] Daniel Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[ST83] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983. Announced at STOC'81.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[ST14] 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.

[Tho07] Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. Announced at STOC'01.

[Tro12] Joel A. Tropp. User-friendly tail bounds for sums of random matrices. *Foundations of Computational Mathemathics*, 12(4):389–434, August 2012.

[Wul17] 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.