Sampling Random Spanning Trees Faster Than Matrix Multiplication

David Durfee
Georgia Tech, USA
ddurfee@gatech.edu

Rasmus Kyng*
Yale University, USA
rasmus.kyng@yale.edu

John Peebles†
Massachusetts Institute of Technology, USA
jpeebles@mit.edu

Anup B. Rao
Georgia Tech, USA
anup.rao@gatech.edu

Sushant Sachdeva
Google,USA
sachdevasushant@gmail.com

ABSTRACT

We present an algorithm that, with high probability, generates a random spanning tree from an edge-weighted undirected graph in $\widetilde{O}(n^{5/3}m^{1/3})$ time. The tree is sampled from a distribution where the probability of each tree is proportional to the product of its edge weights. This improves upon the previous best algorithm due to Colbourn et al. that runs in matrix multiplication time, $O(n^{\omega})$. For the special case of unweighted graphs, this improves upon the best previously known running time of $\tilde{O}(\min\{n^{\omega}, m\sqrt{n}, m^{4/3}\})$ for $m \gg n^{7/4}$ (Colbourn et al. '96, Kelner-Madry '09, Madry et al. '15).

The effective resistance metric is essential to our algorithm, as in the work of Madry et al., but we eschew determinant-based and random walk-based techniques used by previous algorithms. Instead, our algorithm is based on Gaussian elimination, and the fact that effective resistance is preserved in the graph resulting from eliminating a subset of vertices (called a Schur complement). As part of our algorithm, we show how to compute ϵ-approximate effective resistances for a set S of vertex pairs via approximate Schur complements in $\widetilde{O}(m+(n+|S|)\epsilon^{-2})$ time, without using the Johnson-Lindenstrauss lemma which requires $\widetilde{O}(\min\{(m+|S|)\epsilon^{-2}, m+n\epsilon^{-4}+|S|\epsilon^{-2}\})$ time. We combine this approximation procedure with an error correction procedure for handling edges where our estimate isn't sufficiently accurate.

CCS CONCEPTS

• **Theory of computation → Generating random combinatorial structures**; *Continuous optimization*; *Divide and conquer*; • **Mathematics of computing →** *Trees*;

*This material is based on work supported by ONR Award N00014-16-1-2374
†This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374 and by the National Science Foundation under Grant No. 1065125.

STOC'17, Montreal, Canada

978-1-4503-4528-6/17/06…$15.00
DOI: 10.1145/3055399.3055463

KEYWORDS

Sampling Algorithm, Graph Sparsification, Approximate Schur Complement, Random Spanning Trees

ACM Reference format:
David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. 2017. Sampling Random Spanning Trees Faster Than Matrix Multiplication. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing, Montreal, Canada, June 2017 (STOC'17),* 13 pages.
DOI: 10.1145/3055399.3055463

1 INTRODUCTION

Random spanning trees are one of the most well-studied probabilistic structures in graphs. Their history goes back to the classic matrix-tree theorem due to Kirchhoff in 1840s that connects the spanning tree distribution to matrix determinants [14]. The task of algorithmically sampling random spanning trees has been studied extensively [1, 3, 6, 9, 10, 12, 17, 22].

Over the past decade, sampling random spanning trees have found a few surprising applications in theoretical computer science – they were at the core of the breakthroughs in approximating the traveling salesman problem in both the symmetric [7] and the asymmetric case [2]. Goyal et al. [8] showed that one could construct a cut sparsifier by sampling random spanning trees.

Given an undirected, weighted graph $G(V, E, w)$, the algorithmic task is to sample a tree with a probability that is proportional to the product of the weights of the edges in the tree. We give an algorithm for this problem, that, for a given $\delta > 0$, outputs a random spanning tree from this distribution with probability $1 - \delta$ in expected time $\widetilde{O}(n^{5/3}m^{1/3}\log^4 1/\delta)$[1].

For weighted graphs, a series of works building on the connection between matrix trees and determinants, culminated in an algorithm due to Colbourn, Myrvold, and Neufeld [6] that generates a random spanning tree in matrix multiplication time ($O(n^{2.37\cdots})$ [25]). Our result is the first improvement on this bound for more than twenty years! It should be emphasized that the applications to traveling salesman problem [2, 7] require sampling trees on graphs with arbitrary weights.

A beautiful connection, independently discovered by Broder [3] and [1] proved that one could sample a random spanning tree, by simply taking a random walk in the graph until it covers all

[1]The $\widetilde{O}(\cdot)$ notation hides poly$(\log n)$ factors

nodes, and only keeping the first incoming edge at each vertex. For graphs with unit-weight edges, this results in an $O(mn)$ algorithm. The work of Kelner-Madry [12] and Madry et al. [22] are based on trying to speed up these walks. These works together give a previously best running time of $\tilde{O}(\min\{n^\omega, m\sqrt{n}, m^{4/3}\})$ for unit-weighted graphs. Our algorithm is an improvement for all graphs with $m \gtrsim n^{7/4}$.

The above works based on random walks seem challenging to generalize to weighted graphs. The key challenge being that, in weighted graphs, random walks can take a very long time to cover the graph. We take an approach based on another intimate and beautiful connection; one between random spanning trees and Laplacians.

Random Spanning Trees and the Laplacian Paradigm. A by-now well-known but beautiful fact states that the marginal probability of an edge being in a random spanning tree is exactly equal to the product of the edge weight and the effective resistance of the edge (see Fact 3.7). Our algorithm will be roughly based on estimating these marginals, and sampling edges accordingly. The key challenge we overcome here, is that these sampling probabilities change every time we condition on an edge being present or absent in the tree.

Taking this approach of computing marginals allows us to utilize fast Laplacian solvers and the extensive tools developed therein [4, 13, 15, 16, 18–20, 24]. As part of our algorithm for generating random spanning trees, we give a procedure to estimate all pairwise effective resistances in the graph without using the Johnson-Lindenstrauss lemma. Our procedure is also faster if we only want to compute effective resistances for a smaller subset of pairs.

Our procedure for estimating effective resistances is recursive. If we focus on a small subset of the vertices, for the purpose of computing effective resistances, we can eliminate the remaining vertices, and compute the resulting *Schur complement*. Computing the Schur complement exactly is costly and results in an $O(n^\omega)$ algorithm (similar to [10]). Instead, we develop a fast algorithm for approximating the Schur complement. Starting from a graph with m edges, we can compute a Schur complement onto k vertices with at most ϵ error (in the spectral sense), in $\widetilde{O}(m + n\epsilon^{-2})$ time. The resulting approximation has only $\widetilde{O}(k\epsilon^{-2})$ edges.

We hope that faster generation of random spanning trees and the tools we develop here will find further applications, and become an integral part of the *Laplacian paradigm*.

1.1 Prior Work

One of the first major results in the study of spanning trees was Kirchoff's matrix-tree theorem, which states that the total number of spanning trees for general edge weighted graphs is equal to any cofactor of the associated graph Laplacian [14].

Much of the earlier algorithmic study of random spanning trees heavily utilized these determinant calculations by taking a random integer between 1 and the total number of trees, then efficiently mapping the integer to a unique tree. This general technique was originally used in [9, 17] to give an $O(mn^3)$-time algorithm, and ultimately was improved to an $O(n^\omega)$-time algorithm by [6], where m, n are the numbers of edges and vertices in the graph, respectively, and $\omega \approx 2.373$ is the matrix multiplication exponent [25]. These determinant-based algorithms have the advantage that they can handle edge-weighted graphs, where the weight of a tree is defined as the product of its edge weights.[2] Despite further improvements for unweighted graphs, no algorithm prior to our work improved upon this $O(n^\omega)$ runtime in the general weighted case in over 20 years since this work. Even for unweighted graphs, nothing faster than $O(n^\omega)$ was known for dense graphs with $m \geq n^{1.78}$.

We now give a brief overview of the improvements for unweighted graphs along with a recent alternative $O(n^\omega)$ algorithm for weighted graphs.

Around the same time as the $O(n^\omega)$-time algorithm was discovered, Broder and Aldous independently showed that spanning trees could be randomly generated with random walks, where each time a new vertex is visited, the edge used to reach that vertex is added to the tree [1, 3]. Accordingly, this results in an algorithm for generating random spanning trees that runs in the amount of time proportional to the time it takes for a random walk to cover the graph. For unweighted this *cover time* is $O(mn)$ in expectation is better than $O(n^\omega)$ in sufficiently sparse graphs and worse in dense ones. However, in the more general case of edge-weighted graphs, the cover time can be exponential in the number of bits used to describe the weights. Thus, this algorithm does not yield any improvement in worst-case runtime for weighted graphs.

Kelner and Madry improved upon this result by showing how to simulate this random walk more efficiently. They observed that one does not need to simulate the portions of the walk that only visit previously visited vertices. Then, they use a low diameter decomposition of the graph to partition the graph into components that are covered quickly by the random walk and do pre-computation to avoid explicitly simulating the random walk on each of these components after each is initially covered. This is done by calculating the probability that a random walk entering a component at each particular vertex exits on each particular vertex, which can be determined by solving Laplacian linear systems. This approach yields an expected runtime of $\widetilde{O}(m\sqrt{n})$ for unweighted graphs [12].

This was subsequently improved for sufficiently sparse graphs with an algorithm that also uses shortcutting procedures to obtain an expected runtime of $\widetilde{O}(m^{4/3})$ in unweighted graphs [22]. Their algorithm uses a new partition scheme based on effective resistance and additional shortcutting done by recursively finding trees on smaller graphs that correspond to random forests in the original graph, allowing the contraction and deletion of many edges.

Recently, Harvey and Xu [10] gave a simpler deterministic $O(n^\omega)$ time algorithm that uses conditional effective resistances to decide whether each edge is in the tree, contracting the edge in the graph if the edge will be in the tree and deleting the edge from the graph if the edge will not.[3] Updating the effective resistance of each edge is done quickly by using recursive techniques similar to those in [5] and via an extension of the Sherman-Morrison formula.

[2] To see why this definition is natural, note that this corresponds precisely to thinking of an edge with weight k as representing k parallel edges and then associating all spanning trees that differ only in which parallel edges they use.

[3] Note that for any edge e, there is a bijection between spanning trees of the graph in which e is contracted and spanning trees of the original graph that contain e. Similarly, there is a bijection between spanning trees of the graph in which e is deleted and spanning trees of the original graph that do not contain e.

2 OUR RESULTS

2.1 Random Spanning Trees

Theorem 2.1. *For any $0 < \delta < 1$, the routine GenerateSpanningTree (Algorithm 1) outputs a random spanning tree from the $\mathbf{w}$-uniform distribution with probability at least $1 - \delta$ and takes expected time $\widetilde{O}(n^{5/3}m^{1/3}\log^4 1/\delta)$.*

Our algorithm samples edges according to their conditional effective resistance as in [5, 6, 10]. We repeatedly use the well known fact that the effective resistance multiplied by the edge weight, which we will refer to as the *leverage score* of the edge, is equal to the probability that the edge belongs to a randomly generated spanning tree. To generate a uniformly random spanning tree, one can sample edges in an iterative fashion. In every iteration, the edge being considered is added to the spanning tree with probability exactly equal to its leverage score. If it is added to the tree, the graph is updated by contracting that edge, otherwise, the edge is removed from the graph. Though using fast Laplacian solvers [24] one can compute the leverage score of a single edge in $\widetilde{O}(m)$ time, since one needs to potentially do this m times (and the graph keeps changing every iteration), this can take $\widetilde{O}(m^2)$ time if done in a naive way. It therefore becomes necessary to compute the leverage scores in a more clever manner.

The algorithms in [5, 6, 10] get a speed up by a clever recursive structure which enables one to work with much smaller graphs to compute leverage scores at the cost of building such a structure. This kind of recursion will be the starting point of our algorithm which will randomly partition the vertices into two equally sized sets, and compute Schur complements onto each set. It crucially uses the fact that Schur complement, which can be viewed as block Gaussian elimination, preserves effective resistances of all the edges whose incident vertices are not eliminated. It first recursively samples edges contained in both these sets, contracting or deleting every edge along the way, and then the edges that go across the partition are sampled. This starting point to our algorithm is then a very close variant of the algorithms in [6, 10], and they prove that it takes $O(n^\omega)$.

In order to improve the running time, the main workhorse we use is derived from the recent paper [19] on fast Laplacian solvers which provided an almost linear time algorithm for performing an approximate Gaussian elimination of Laplacians. We generalize the statement in [19] to show that one can compute an approximate Schur complement of a set of vertices in a Laplacian quickly. Accordingly, one of our primary results, discussed in Section 2.2, will be that a spectrally approximate Schur complement can be efficiently computed, and we will leverage this result to achieve a faster algorithm for generating random spanning trees.

Since we compute approximate Schur complements, the leverage scores of edges are preserved only approximately. But we set the error parameter such that we can get a better estimate of the leverage score if we move up the recursion tree, at the cost of paying more for the computing leverage score of an edge in a bigger graph. We give a sampling procedure that samples edges into the random spanning tree from the true distribution by showing that approximate leverage score can be used to make the right decisions most of the times.

Subsequently, we are presented with a natural trade-off for our error parameter choice in the ApproxSchur routine: larger errors speed up the runtime of ApproxSchur, but smaller errors make moving up the recursion to obtain a more exact effective resistance estimate less likely. Furthermore, the recursive construction will cause the total vertices across each level to double making small error parameters even more costly as we recurse down. Our choice of the error parameter will balance these trade-offs to optimize running time.

The routine ApproxSchur produced an approximate Schur complement only with high probability. We are not aware of a way to certify that a graph sparsifier is good quickly. Therefore, we condition on the event that the ApproxSchur produces correct output on all the calls, and show ultimately show that it is true with high probability.

Our algorithm for approximately generating random spanning trees, along with a proof of Theorem 2.1 is given in Section 4 and 4.2.1

2.2 Approximating the Schur Complement

Theorem 2.2. *Given a connected undirected multi-graph $G = (V, E)$, with positive edges weights $w : E \to \mathbb{R}_+$, and associated Laplacian $\mathbf{L}$, a set vertices $C \subset V$, and scalars $0 < \epsilon \leq 1/2$, $0 < \delta < 1$, the algorithm ApproxSchur$(\mathbf{L}, C, \epsilon, \delta)$ returns a Laplacian matrix $\widetilde{\mathbf{S}}$. With probability $\geq 1 - \delta$, the following statements all hold: $\widetilde{\mathbf{S}} \approx_\epsilon \mathbf{S}$, where $\mathbf{S}$ is the Schur complement of $\mathbf{L}$ w.r.t elimination of $F = V - C$. $\widetilde{\mathbf{S}}$ is a Laplacian matrix whose edges are supported on C. Let $k = |C| = n - |F|$. The total number of non-zero entries $\widetilde{\mathbf{S}}$ is $O(k\epsilon^{-2}\log(n/\delta))$. The total running time is bounded by $O((m\log n\log^2(n/\delta) + n\epsilon^{-2}\log n\log^4(n/\delta))\,\mathrm{polyloglog}(n))$.*

The proof of this appears in Section 5.

As indicated earlier, the algorithm ApproxSchur builds on the tools developed in [19]. Roughly speaking, the algorithm in [19] produces an ϵ-approximation to a Cholesky decomposition of the Laplacian in $\widetilde{O}(\frac{m}{\epsilon^2})$ time. Our algorithm for approximating Schur complements is based on three key modifications to the algorithm from [19]: Firstly, we show that the algorithm can be used to eliminate an arbitrary subset $U \subset V$ of the vertices, giving a approximate partial Cholesky decomposition. Part of this decomposition is an approximate Schur complement w.r.t. elimination of the set of vertices U. Secondly, we show that although the spectral approximation quality of this decomposition is measured in terms of the whole Laplacian, in fact it implies a seemingly stronger guarantee on the approximate Schur complement: Its quadratic form resembles the true Schur complement up to a small multiplicative error. Thirdly, we show that the algorithm from [19] can utilize leverage score estimates (constant-factor approximations) to produce an approximation in only $\widetilde{O}(m + n\epsilon^{-2})$ time. Additionally, we also sparsify the output to ensure that the final approximation has only $\widetilde{O}((n-|U|)\epsilon^{-2})$ edges. The leverage score estimates can be obtained by combining a Laplacian solver with Johnson-Lindenstrauss projection. It is worth noting that the Laplacian solver from [18] is also based on approximating Schur complements. However, their algorithm can only approximate the Schur complement obtained

by eliminating very special subsets of vertices. The above theorem, in contrast, applies to an arbitrary set of vertices. This algorithm also had a much worse dependence on ϵ^{-1}, making it unsuitable for our applications where ϵ^{-1} is $\Omega(n^c)$ for some small constant c.

2.3 Computing Effective Resistance

Our techniques also enable us to develop a novel algorithm for computing the effective resistances of pairs of vertices. In contrast with prior work, our algorithm does not rely on the Johnson-Lindenstrauss lemma, and it achieves asymptotically faster running times in certain parameter regimes.

Theorem 2.3. *When given a graph G, a set S of pairs of vertices, and an error parameter ϵ, the function EstimateR$_{eff}$ (Algorithm 5 in Section 6) returns $e^{\pm\epsilon}$-multiplicative estimates of the effective resistance of each of the pairs in S in time $\widetilde{O}\left(m + \frac{n+|S|}{\epsilon^2}\right)$ with high probability.*

By $e^{\pm\epsilon}$-multiplicative estimates here, we mean that $e^{-\epsilon}R_{eff}(u,v) \leq \tilde{R}_{eff}(u,v) \leq e^{\epsilon}R_{eff}(u,v)$, where $R_{eff}(u,v)$ and $\tilde{R}_{eff}(u,v)$ are the actual and estimated effective resistance of (u,v), respectively. One can compare this runtime with what can be obtained using (now standard) linear system solving machinery introduced in [24]. Using such machinery, one obtains an algorithm for this same problem with runtime[4] $\widetilde{O}\left((\min\left(\frac{m+|S|}{\epsilon^2}, m + \frac{n}{\epsilon^4} + \frac{|S|}{\epsilon^2}\right)\right)$. When the number of pairs $|S|$ and the error parameter ϵ are both small, the runtime of our algorithm is asymptotically smaller than this existing work.

3 NOTATION

3.1 Graphs

We assume we are given a weighted undirected graph $G = (V, E, \boldsymbol{w})$, with the vertices are labelled $V = \{1, 2, ..., n\}$. Let $\boldsymbol{A}_G$ be its adjacency matrix. The (i,j)'th entry of the adjacency matrix $\boldsymbol{A}_G(i,j) = \boldsymbol{w}_{i,j}$ is the weight of the edge between the vertices i and j. Let $\boldsymbol{D}_G$ is the diagonal matrix consisting of weighted degrees of the vertices, i.e., $\boldsymbol{D}_G(i,i) = \sum_{j\sim i} \boldsymbol{w}_{i,j}$. The Laplacian matrix is defined as $\boldsymbol{L}_G = \boldsymbol{D}_G - \boldsymbol{A}_G$. We drop the subscript G when the underlying graph is clear from the discussion.

Definition 3.1 (Induced Graph). Given a graph $G = (V,E)$ and a set of vertices $V_1 \subseteq V$, we use the notation $G(V_1)$ to mean the induced graph on V_1.

Definition 3.2. Given a set of edges E on vertices V, and $V_1, V_2 \subseteq V$, we use the notation $E \cap (V_1, V_2)$ to mean the set of all edges in E with one end point in V_1 and the other in V_2.

Definition 3.3 (Contraction and Deletion). Given a graph $G = (V,E)$ and a set of edges $E_1 \subset E$, we use the notation $G \backslash E_1$ to denote the graph obtained by deleting the edges in E_1 from G and G/E_1 to denote the graph obtained by contracting the edges in E_1 within G and deleting all the self loops.

[4]The first runtime in the min expression comes from applying JL with the original Laplacian. The second runtime comes from sparsifying the Laplacian first and then applying JL.

3.2 Spanning Trees

Let $\mathcal{T}_G$ denote the set of all spanning subtrees of G. We now define a probability distribution on these trees.

Definition 3.4 ($\boldsymbol{w}$-uniform distribution on trees). Let $\mathcal{D}_G$ be a probability distribution on $\mathcal{T}_G$ such that

$$\Pr\left(X = T \mid X \sim \mathcal{D}_G\right) \propto \Pi_{e\in T}\, \boldsymbol{w}_e.$$

We refer to $\mathcal{D}_G$ as the $\boldsymbol{w}$-uniform distribution on $\mathcal{T}_G$. When the graph G is unweighted, this corresponds to the uniform distribution on $\mathcal{T}_G$.

Definition 3.5 (Effective Resistance). The *effective resistance* of a pair of vertices $u, v \in V_G$ is defined as

$$R_{eff}(u,v) = \boldsymbol{b}_{u,v}^T \boldsymbol{L}^{\dagger} \boldsymbol{b}_{u,v}.$$

where $\boldsymbol{b}_{u,v}$ is an all zero vector corresponding to V_G, except for entries of 1 at u and v

Definition 3.6 (Leverage Score). The *statistical leverage score*, which we will abbreviate to *leverage score*, of an edge $e = (u,v) \in E_G$ is defined as

$$l_e = \boldsymbol{w}_e R_{eff}(u,v).$$

Fact 3.7 (Spanning Tree Marginals). *The probability $\Pr(e)$ that an edge $e \in E_G$ appears in a tree sampled $\boldsymbol{w}$-uniformly randomly from $\mathcal{T}_G$ is given by*

$$\Pr(e) = l_e,$$

where l_e is the leverage score of the edge e (see for eg. Cor 4.4 in [21]).

3.3 Schur Complement

Definition 3.8 (Schur Complement). Let $\boldsymbol{M}$ be a block matrix

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{B}^T & \boldsymbol{C} \end{bmatrix}. \tag{1}$$

We use $\textsc{Schur}(\boldsymbol{M}, \boldsymbol{A})$ to denote the Schur complement of $\boldsymbol{C}$ onto $\boldsymbol{A}$ in $\boldsymbol{M}$; ie.,

$$\textsc{Schur}(\boldsymbol{M}, \boldsymbol{A}) = \boldsymbol{A} - \boldsymbol{B}\boldsymbol{C}^{-1}\boldsymbol{B}^T.$$

Equivalently, this is simply the result of running Gaussian elimination of the block $\boldsymbol{C}$.

When the matrix $\boldsymbol{M} = \boldsymbol{L}$ is a Laplacian of a graph $G = (V,E)$ and $V_1 \subseteq V$ is a set of vertices, we abuse the notation and use $\textsc{Schur}(\boldsymbol{L}, V_1)$ or $\textsc{Schur}(G, V_1)$ to denote the Schur complement of $\boldsymbol{L}$ onto the submatrix of $\boldsymbol{L}$ corresponding to V_1; i.e., onto the submatrix of $\boldsymbol{L}$ consisting of all entries whose coordinates (i,j) satisfy $i, j \in V_1$.

Fact 3.9. *Let $G = (V,E)$ be a graph and $V = V_1 \cup V_2$ be a partition of the vertices. Then $\textsc{Schur}(G, V_1)$ is Laplacian matrix of a graph on vertices in V_1.*

This means that Schur complement in a graph $G = (V,E)$ onto a set of vertices V_1 can be viewed as a graph on V_1. Furthermore, we can view this as a multigraph obtained by adding (potentially parallel) edges to $G(V_1)$, the induced graph on V_1. We take this view in this paper: whenever we talk about Schur complements, we separate out the edges of the original graph from the ones created during Schur complement operation.

We now provide some basic facts about how Schur complements relate to spanning trees. This first lemma says that edge deletions and contractions commute with taking Schur complements.

Fact 3.10. *(Lemma 4.1 of [5]) Given G with any vertex partition V_1, V_2, for any edge $e \in E \cap (V_1, V_1)$ we have:*

$$\textsc{Schur}(G \setminus e, V_1) = \textsc{Schur}(G, V_1) \setminus e$$
$$\textsc{Schur}(G/e, V_1) = \textsc{Schur}(G, V_1)/e$$

Fact 3.11. *Given G with any vertex partition V_1, V_2, for any edge $e \in E \cap (V_1, V_1)$, the leverage score of e in G is same as that in $\textsc{Schur}(G, V_1)$.*

Proof. Note that for $e \in (V_1, V_1)$, $b_e^T L_G^{-1} b_e = (b_e[V_1])^T (L_G^{-1}[V_1, V_1])(b_e[V_1])$. Here, $b_e[V_1]$ corresponds to a vector restricted to indices in V_1 and $L_G^{-1}[V_1, V_1]$ is similarly the minor of the matrix L_G^{-1} corresponding to indices in V_1. By a standard property of Schur complements (see [11]), we have $L_G^{-1}[V_1, V_1]) = (\textsc{Schur}(G, V_1))^{-1}$. This proves the fact. □

Spectral Approximation

Definition 3.12. Given two graphs G, H on identical vertex sets, and respective Laplacians L_G and L_H. We say $G \approx_\epsilon H$ if

$$\exp(-\epsilon) L_H \preceq L_G \preceq \exp(\epsilon) L_H.$$

Definition 3.13 (Approximate Schur Complement). Given a graph $G = (V, E)$ and vertex set $U \subset V$, let S_U be the Laplacian of $\textsc{Schur}(G, V \setminus U) - G(V \setminus U)$; ie., the set of edges added to the induced subgraph $G(V \setminus U)$ by the Schur complement operation. We call a matrix $\widetilde{S}_U$ an *ϵ-approximate Schur complement* if it satisfies

$$\widetilde{S}_U \approx_\epsilon S_U.$$

Furthermore, $\widetilde{S}_U$ is a Laplacian.

4 ALGORITHM FOR SAMPLING SPANNING TREES

It is well known that for any edge of a graph, the probability of that edge appearing in a random spanning tree is equal to it's leverage score. We can iteratively apply this fact to sample a $\boldsymbol{w}$-uniform random tree. We can consider the edges in an arbitrary sequential order, say $e_1, \ldots, e_m \in E$, and make decisions on whether they belong to tree. Having decided for edges $e_1, \ldots, e_i$, one computes the probability p_{i+1}, conditional on the previous decisions, that edge e_{i+1} belongs to the tree. Edge e_{i+1} is then added to the tree with probability p_{i+1}.

To estimate the probability that edge e_{i+1} belongs to the tree conditional on the decisions made on $e_1, \ldots, e_i$, we can use Fact 3.7. Let $E_T \subset \{e_1, \ldots, e_i\}$ be the set of edges that were included in the tree, and $F_T \subset \{e_1, \ldots, e_i\}$ the subset of edges that were not included. Then, p_{i+1} is equal to the leverage score of edge e_{i+1} in the graph $G^{(i+1)} := (G \setminus F_T)/E_T$ obtained by deleting edges E_T from G and then contracting edges E_{T^c}. In other words, we get $G^{(i+1)}$ from $G^{(i)}$ by either deleting the edge e_i or contracting it, depending on if e_i was not added to the tree or added to the tree, respectively. Note that as we move along the sequence, some of the original edges may no longer exist in the updated graph due to edge contractions. In that case, we just skip the edge and move to the next one.

There are known routines for computing leverage score of an edge, with ϵ multiplicative error, which require $\widetilde{O}(m \log 1/\epsilon)$ runtime. Since we potentially have to compute leverage score of every edge, this immediately gives a total runtime of $\widetilde{O}(m^2)$.

Algorithms in [5, 6, 10] will similarly make decisions on edges in a sequential order. Where they differ from the above algorithm is the graph they use to compute the leverage score of the edge. Instead of computing the leverage score of an edge in the original graph updated with appropriate contractions and deletions, they deal with potentially much smaller graphs containing the edge such that the effective resistance of the edge in the smaller graph is approximately same as in the original graph.

In this section, we first describe the close variant of these algorithms that our recursive structure will follow. We then invoke the routine described in Section 2.2 to obtain access to a cheap but approximate routine for computing the sampling probability. Finally, we give a procedure for dealing with the errors incurred from the approximate routine, and by careful analysis of these errors will achieve our desired running time.

4.1 Structure of the Recursion

We now describe the recursive structure of the algorithm given in Algorithm 1. The structure of the recursion is same as in [6, 10]. Let the input graph be $G = (V_G, E_G)$. Suppose at some stage of the algorithm, we have a graph $\widetilde{G}$. The task is to make decisions on edges in $E_G \cap E_{\widetilde{G}}$. We initially divide the vertex set into two equal sized sets $V_{\widetilde{G}} = V_1 \cup V_2$. Recursively, we first make decisions on edges in $\widetilde{G}(V_1) \cap E_G$, then make decisions on edges in $\widetilde{G}(V_2) \cap E_G$ and finally make decisions on the remaining edges. To make decisions on $\widetilde{G}(V_1) \cap E_G$, we use the fact that the effective resistance of edges are preserved under Schur complement. We work with the graph $G_1 = \textsc{ApproxSchur}(\widetilde{G}, V_1, \epsilon)$ and recursively make decisions on edges in $E_G \cap G(V_1)$. Having recursively made decisions on edges in $E_G \cap \widetilde{G}(V_1)$, let E_T be the set of tree edges from this set. We now need to update the graph $\widetilde{G}$ by contracting edges in E_T and deleting all the edges in $E_T^c \cap \widetilde{G}(V_1) \cap E_G$. Then we do the same for the edges in $E_G \cap \widetilde{G}(V_2)$.

Finally, we treat the edges $E_G \cap (V_1, V_2)$ that cross V_1, V_2 in a slightly different way, and is handled by the subroutine SampleAcross in the algorithm. If we just consider the edges in E_G, this is trivially a bipartite graph. This property is maintained in all the recursive calls by the routine SampleAcross. The routine SampleAcross works by dividing V_1, V_2 both into two equal sized sets $V_1 = L_1 \cup L_2$ and $V_2 = R_1 \cup R_2$ and making four recursive calls, one each for edges in $E_G \cap (L_i, R_j), i = 1, 2; j = 1, 2$. To make decisions on edges in $E_G \cap (L_i, R_j)$, it recursively calls SampleAcross on the graph $G_{ij} = \textsc{ApproxSchur}(\widetilde{G}, (L_i, R_j)^c, \epsilon)$ obtained by computing approximate Schur complement on to vertices in (L_i, R_j) of vertices outside it.

4.2 Approximate Schur Complement and Expected $\widetilde{O}(n^{5/3} m^{1/3})$ Time Algorithm

The recursion gives an $O(n^\omega)$ time algorithm as shown in [6, 10], which we speed up $O(n^\omega)$ algorithm by computing approximate Schur complements faster. Having access only to approximate Schur

complements, which preserves leverage score only approximately, introduces an issue with computing sampling probability. It is a-priori not clear how to make decisions on edges when we preserve leverage scores only approximately during the recursive calls. The key idea here is as follows. Suppose we want to decide if a particular edge e belongs to the tree. Tracing the recursion tree produced by Algorithm 1, we see that we have a sequence of graphs $G, G_1, G_2, \ldots, G_k$ all containing the edge e, starting from the original input graph G all the way down to G_k which has a constant number of vertices. We also have $V(G_i) \subset V(G_{i-1})$ for all $k \geq i \geq 1$, all of them being subsets of $V(G)$. We set the error parameter ϵ in ApproxSchur as follows. Let $n = |V(G)|, m = |E_G|$ be the number of vertices and edges in the input graph, then we define ϵ in terms of the level i as

$$\epsilon(i) = \begin{cases} \left(\frac{m}{n}\right)^{-2/3} & \text{for } i \in [0, t_1] \\ 4^{i-t_1}\left(\frac{m}{n}\right)^{-2/3} & \text{for } i \in [t_1, t_1 + t_2] \\ \log^{-2} n & \text{otherwise} \end{cases} \tag{2}$$

The two threshold values t_1 and t_2 are such that $2^{t_1} = \frac{n^2}{m}$ and $2^{t_2} = (\frac{m}{n})^{1/3}\frac{1}{\log n}$.

Our $\epsilon(\cdot)$ function will ensure for all $i, l_e(G) \in [(1-\epsilon_i)l_e(G_i), (1+\epsilon_i)l_e(G_i)]$ for an appropriate ϵ_i. We sample a uniform random number $r \in [0,1]$, and initially compute $l_e(G_k)$. If r lies outside the interval $[(1-\epsilon_i)l_e(G_k), (1+\epsilon_i)l_e(G_k)]$, then we can make a decision on the edge e. Otherwise, we estimate $l_e(G)$ to a higher accuracy by computing $l_e(G_{k-1})$. We continue this way, and if r lies inside the interval $[(1-\epsilon_i)l_e(G_i), (1+\epsilon_i)l_e(G_i)]$ for every i, then we compute $l_e(G)$ in the input graph G. In the next section we describe SampleEdge in more detail.

At this point, we find it important to mention that the spectral error guarantees from the ApproxSchur subroutine only hold with probability $\geq 1-\delta$. The explanation of the SampleEdge subroutine above relied on these spectral guarantees, and the error in our algorithm for generating random spanning trees will be entirely due to situations in which the sparsification routine does not give a spectrally similar Schur complement. For the time being we will work under the following assumption and later use the fact that it is true w.h.p. to bound the error of our algorithm.

Assumption 4.1. *Every call to ApproxSchur with error parameter ϵ always computes an ϵ-approximate Schur Complement*

Sampling Scheme: SampleEdge. In this section we describe the routine SampleEdge(e) for an edge $e \in G$ in the input graph. By keeping track of the recursion tree, we have $G_0, G_1, \ldots, G_k$ and $e \in G_i$ for all i.

Lemma 4.2. *For graph G and G_i, the respective conditional leverage scores l_e and $l_e^{(i)}$ for edge e are such that $l_e \in [(1-2\epsilon(i)\log n)l_e^{(i)}, (1+2\epsilon(i)\log n)l_e^{(i)}]$*

This will now allow us to set $\epsilon_i = 2\epsilon(i)\log n$. We will delay the proof of Lemma 4.2 until later in this section in favor of first giving the sampling procedure. The sampling procedure is as follows. We generate a uniform random number in $r \in [0,1]$. We want to sample edge e if $r \leq l_e(G)$. Instead, we use $l_e(G_k)$ as a proxy. Note that using fast Laplacian solvers, we can in $\widetilde{O}$(no. of edges) time compute leverage score of an edge up to a factor of $1 + 1/\text{poly}(n)$. Since $l_e(G) \in [(1-\epsilon_k)l_e(G_k), (1+\epsilon_k)l_e(G_k)]$, we include the edge in the tree if $r \leq (1-\epsilon_k)l_e(G_k)$, otherwise if $r > (1+\epsilon_k)l_e(G_k)$, we don't include it in the tree. If $r \in [(1-\epsilon_k)l_e(G_k), (1+\epsilon_k)l_e(G_k)]$, which happens with probability $2\epsilon_k l_e(G_k)$, we get a better estimate of $l_e(G)$ by computing $l_e(G_{k-1})$. We can make a decision as long as $r \notin [(1-\epsilon_{k-1})l_e(G_{k-1}), (1+\epsilon_{k-1})l_e(G_{k-1})]$, otherwise, we consider the bigger graph G_{k-2}. In general, if $r \notin [(1-\epsilon_i)l_e(G_i), (1+\epsilon_i)l_e(G_i)]$, then we can make a decision on e, otherwise we get a better approximation of $l_e(G)$ by computing $l_e(G_{i-1})$. If we can't make a decision in any of the k steps, which happens if $r \in [(1-\epsilon_i)l_e(G_i), (1+\epsilon_i)l_e(G_i)]$ for all i, then we compute

Algorithm 1: GenerateSpanningTree($\widetilde{G} = (E_{\widetilde{G}}, \widetilde{V})$) : Recurse using Schur Complement

Input: Graph $\widetilde{G}$. Let E_G, a global variable, denote the edges in the original (input) graph G.
Output: E_T is the set of edges in the sampled tree.

```
E_T ← SampleWithin(G) ;
return E_T;
Procedure SampleWithin(G̃)
    Set E_T ← {} ;
    if |Ṽ| = 1 then
        return;
    else
        Divide V into equal sets V = V_1 ∪ V_2. ;
        for i = 1,2 do
            Compute G_i = ApproxSchur(G̃, V_i, ε(level)) (see Equation 2) ;
            E_T ← E_T ∪ SampleWithin(G_i) ;
            Update G̃ by deleting edges in G̃(V_i) ∩ E_T^c and contracting edges in G̃(V_i) ∩ E_T. (Note the convention E_T^c := E_G \ E_T ) ;
        E_T ← E_T ∪ SampleAcross(G̃, (V_1, V_2)) ;
        return E_T ;
Procedure SampleAcross(G̃, (L, R))
    if |L| = |R| = 1 then
        E_T = SampleEdge(G̃, (L, R) ∩ E_G) ;
        return E_T ;
    Divide L, R into two equal sized sets: L = L_1 ∪ L_2, R = R_1 ∪ R_2. ;
    for i = 1,2 do
        for j = 1,2 do
            G̃_ij ← ApproxSchur(G̃, (L_i ∪ R_j), ε(level)) (see Equation 2) ;
            E_T ← E_T ∪ SampleAcross(G̃_ij, (L_i, R_j)) ;
            Update G̃ by contracting edges E_T and deleting edges in E_T^c ∩ (L_i, R_j) ;
    return E_T ;
```

the leverage score of e in G updated with edge deletions and contractions resulting from decisions made on all the edges that were considered before e.

Recall that $\epsilon(i)$ is a function that is only changing for $i \in [t_1, t_1 + t_2]$, and it follows that the process above should only look a graph one level up if the estimate is better. Accordingly, we will only iterate this process from $i = t_1 + t_2$ to $i = t_1$.

Finally, note that in the final step, we can compute $l_e(G)$ up to an approximation factor of $1 + \rho$ in $\widetilde{O}(m \log 1/\rho)$. We can therefore start with $\delta_0 = 1/n$ and if $r \in [(1-\rho)\widetilde{l_e}(G), (1+\rho)\widetilde{l_e}(G)]$, we set $\rho = \rho_0/2$ and repeat. This terminates in $\widetilde{O}(m)$ expected (over randomness in r) time.

For our algorithm, assume that we have an efficient data structure that gives access to each graph $G_0, \ldots G_k$ in which e appears.

Algorithm 2: SampleEdge(e) : Sample an edge using conditional leverage score

Input: An edge e and access to graphs $G_0, \ldots G_k$ in which e appears
Output: Returns $\{e\}$ if edge belongs to the tree, and $\{\}$ if it doesn't

```
Generate a uniform random number r in [0,1] ;
l_e ← EstimateLeverageScore(e)
if r < l_e then
    return {e}
else
    return {}
Procedure EstimateLeverageScore(e)
    Compute l_e^(k) to error 1/n ;
    if isGood(l_e^(k), ε) then
        return l_e^(k)
    for i = t_1 + t_2 to t_1 do
        Compute l, an estimate for l_e^(i) with error 1/n;
        if isGood(l, ε(i)) then
            return l
    for i = 0 to ∞ do
        Compute l, an estimate for l_e^(0) with error 1/2^i n;
        if isGood(l, 1/2^i n) then
            return l
Procedure isGood(l_e, ε)
    if r < (1 − ε) l_e or r > (1 + ε) l_e then
        return True
    return False
```

Proof of Lemma 4.2. This edge sampling scheme relies upon the error in the leverage score estimates remaining small as we work our way down the subgraphs and remaining small when we contract and delete edges. Theorem 2.2 implies leverage score estimates will have small error between levels, so we will only have compounding of small errors. However, it does not imply that these errors remain small after edge contractions and deletions, which becomes necessary to prove in the following lemma.

Lemma 4.3. *Given a graph $G = (V, E)$, vertex partition V_1, V_2, and edges $e \in E \cap (V_1, V_1)$, then*

$$\textit{ApproxSchur}(G, V_1, \epsilon)/e \approx_\epsilon \textit{Schur}(G/e, V_1)$$
$$\textit{ApproxSchur}(G, V_1, \epsilon) \setminus e \approx_\epsilon \textit{Schur}(G \setminus e, V_1)$$

Proof. ApproxSchur$(G, V_1, \epsilon)/e \approx_\epsilon$ Schur$(G, V_1)/e$ because spectral approximations are maintained under contractions. Furthermore, ApproxSchur$(G, V_1, \epsilon) = L_{V_1} + \tilde{S_{V_2}}$ where L_{V_1} is the Laplacian of the edges in $E \cap (V_1, V_1)$. Similarly, write Schur$(G, V_1) = L_{V_1} + S_{V_2}$, and because $\tilde{S_{V_2}} \approx_\epsilon S_{V_2}$ then $L_{V_1} \setminus e + \tilde{S_{V_2}} \approx_\epsilon L_{V_1} \setminus e + S_{V_2}$. Combining these facts with Fact 3.10 gives the desired result. □

Proof. (of Lemma 4.2)
By construction, $\epsilon(i) \le \epsilon(k)$ for every $i \le k$. Iteratively applying Theorem 2.2 and Lemma 4.3, gives
$l_e \in [e^{-\epsilon(k)k} l_e^{(k)}, e^{\epsilon(k)k} l_e^{(k)}]$, and using $\epsilon(k) \le 1/\log^2 n$ for all k, and $k \le \log n$ finishes the proof □

Correctness. Under Assumption 4.1, we were able to prove Lemma 4.2. This, in turn, implies the correctness of our algorithm, which is to say that it generates a tree from a $\boldsymbol{w}$-uniform distribution on trees. We now remove Assumption 4.1, and prove the approximate correctness of our algorithm, and the first part of Theorem 2.1.

Theorem 2.1. *For any $0 < \delta < 1$, the routine GenerateSpanningTree (Algorithm 1) outputs a random spanning tree from the $\boldsymbol{w}$-uniform distribution with probability at least $1 - \delta$ and takes expected time $\widetilde{O}(n^{5/3} m^{1/3} \log^4 1/\delta)$.*

Proof. Each subgraph makes at most 6 calls to ApproxSchur, and there are $\log n$ recursive levels, so $O(n^3)$ total calls are made to ApproxSchur. Setting $\delta' = \frac{\delta}{O(n^3)}$ for each call to ApproxSchur, Assumption 4.1 holds with probability $(1 - \delta')^{O(n^3)} = 1 - \delta$, and $\log^4 \frac{O(n^3)}{\delta} = \tilde{O}(\log^4 1/\delta)$. Therefore, our algorithm will only fail to generate a random tree from the $\boldsymbol{w}$-uniform distribution on trees with probability at most δ □

4.2.1 Runtime Analysis. We will now analyze the runtime of the algorithm. Let $T(n)$ be the time taken by SampleWithin on input a graph $\widetilde{G}$ with n vertices and let $B(n)$ be the time taken by SampleAcross on a graph with n vertices. We recall that the recursive structure then gives $T(n) = 2T(n/2) + 4B(n/2)$ and $B(n/2) = 4B(n/4)$. To compute the total runtime, we separate out the work done in the leaves of the recursion tree from the rest. Note that SampleEdge is invoked only on the leaves.

First we bound the total number of nodes of the recursion tree as a function of the depth in the tree.

Lemma 4.4. *Level i of the recursion tree has at most $4^{i+1} - 2^i$ nodes, the number of vertices in the graphs at each of the nodes is at most $n/2^i$.*

Proof. It is clear that the size of the graph at a node at depth i is at most $n/2^i$. We will bound the number of nodes by induction. There are two types of nodes in the recursion tree due to the recurrence having two kinds of branches corresponding to $T(n), B(n)$. We will call the nodes corresponding to $T(n)$ as the first type and it is clear from the recurrence relation that there are 2^i such nodes. Let us call the other type of nodes the second type, and it is clear that every node (both first and second type) at depth $i-1$ branches into four type two nodes. Therefore, if a_i is the total number of nodes at level i, then $a_i = 4a_{i-1} + 2^i$. We will now prove by induction that $a_i \leq 4^{i+1} - 2^i$. Given $a_0 = 1$, the base case follows trivially. Suppose it is true for $i-1$, then we have $a_i = 4a_{i-1} + 2^i \leq 4(4^i - 2^{i-1}) + 2^i = 4^{i+1} - 2^i$, proving the lemma. □

Now we will compute the total work done at all levels other than the leaves. We recall the error parameter in ApproxSchur calls is a function of the depth in the tree, see Equation 2.

Lemma 4.5. *The total work done at all levels of the recursion tree excluding the leaves is bounded by* $\widetilde{O}(n^{5/3}m^{1/3}\log^4 1/\delta)$.

Proof. From Theorem 2.2 the work done in a node at depth i is $\widetilde{O}\left(((n/2^i)^2 + n/2^i\epsilon(i)^2)\log^4 1/\delta\right)$. The $\log^4 1/\delta$ factor is removed from the remaining analysis for simplicity. By Lemma 4.4, the total work done at depth i is $\widetilde{O}\left(n^2 + 2^i n/\epsilon(i)^2\right)$. Finally, bound for the total running time across all levels follows from

$$\begin{aligned}\sum_{i=0}^{\log n} n^2 + 2^i\frac{n}{\epsilon(i)^2} &= n^2\log n + \sum_{i=0}^{t_1} 2^i n\frac{m^{4/3}}{n^{4/3}} \\ &\quad + \sum_{i=t_1}^{t_1+t_2} 2^i n\frac{m^{4/3}}{n^{4/3}4^{i-t_1}} + \sum_{i=t_1+t_2}^{\log n} 2^i n\frac{1}{\log^4 n} \\ &= O(n^2\log n + n^{5/3}m^{1/3} + n^{5/3}m^{1/3} + n^2/\log^4 n).\end{aligned}$$

Therefore, the total work done is $\widetilde{O}(n^{5/3}m^{1/3})$. □

We will now analyze the total work done at the leaves of the recursion tree. We first state a lemma which gives the probability that approximate leverage score of an edge can be used to decide if the edge belongs to the tree.

Corollary 4.6. *If r is drawn uniformly randomly from $[0,1]$, then the probability that $r \in [1-\hat{\epsilon}\hat{l_e}\log^2 n, 1+\hat{\epsilon}\hat{l_e}\log^2 n]$ is $\widetilde{O}(\hat{\epsilon}l_e)$ w.h.p.*

Proof. The exact probability is $2\hat{\epsilon}\hat{l_e}\log^2 n$, and from Lemma 5.8, we know $l_e \leq 2\hat{l_e}$ w.h.p. □

We now consider the expected work done at a single leaf of the recursion tree.

Lemma 4.7. *Let l_e be the leverage score of an edge e in the G which is obtained by updating the input graph based on the decisions made on all the edges considered before e. The routine SampleEdge takes* $\widetilde{O}(1 + l_e n^{2/3}m^{1/3})$.

Proof. It takes $O(1)$ time to compute the leverage score at a leaf of the recursion tree. The routine SampleEdge successively climbs up the recursion tree to compute the leverage score if the leverage score estimation at the current level is not sufficient. The probability that the outcome of r is such that we cannot make a decision at level i is $\widetilde{O}(\epsilon(i)l_e)$. The time required to compute the leverage score of edge e in the graph at a node at depth i in the recursion tree is $\widetilde{O}((n/2^i)^2)$. Finally, with probability $\left(\frac{m}{n}\right)^{-2/3}$, we need to compute the leverage score in the input graph and the expected running time is $\widetilde{O}(m)$. Therefore, the total expected running time is

$$\widetilde{O}\left(\left(\frac{m}{n}\right)^{-2/3} ml_e + l_e\sum_{i=t_1}^{i=t_1+t_2}\epsilon(i)\frac{n^2}{4^i}\right) = \widetilde{O}\left(1 + l_e n^{2/3}m^{1/3}\right).$$

□

We now want to give the runtime cost over all edges. Let us label the edges $e_1, \ldots, e_m$ in the order in which the decisions are made on them. In the following, when we talk about leverage score l_{e_i} of an edge e_i, we mean the leverage score of the edge e_i in the graph obtained by updating G based on the decisions made on $e_1, \ldots, e_{i-1}$.

Lemma 4.8. *Let e_i be the first edge sampled to be in the tree, and $X = l_{e_1} + l_{e_2} + l_{e_3} + \ldots + l_{e_i}$ be a random variable. Then,*

$$\Pr(X > C) \leq e^{-C}.$$

Proof. Let $p_j = l_{e_j}$, we have $0 \leq p_j \leq 1$. If $\sum_j p_j \geq C$, then the probability that the edges $e_1, \ldots, e_{i-1}$ is deleted is

$$\prod_{j=1}^{i}(1-p_j) \leq \left(1 - \frac{C}{i}\right)^i \leq e^{-C}.$$

□

We thus have $E(X) = O(1)$, and also, with probability at least $1 - 1/\text{poly}(n)$ we have $X = O(\log n)$. Applying this iteratively until $n-1$ edges are sampled to be in the tree, we have that the expected sum of conditional leverage scores is $O(n)$, and is $O(n\log n)$ with probability $1 - 1/\text{poly}(n)$.

Corollary 4.9. *The total expected work done over all the leaves of the recursion tree is $\widetilde{O}(n^{5/3}m^{1/3})$.*

Proof. This immediately follows from Lemma 4.7 by plugging in $\sum_e l_e = O(n\log n)$, which holds with probability at least $1 - 1/\text{poly}(n)$, and observing that the work done at the leaves is $\text{poly}(n)$ in the worst case. □

5 SCHUR COMPLEMENT APPROXIMATION

In this section, we give an algorithm for spectral approximation of the Schur complement of a Laplacian matrix. Our approach closely follows that in [19], with the main distinction being: We show that if their algorithm is used to eliminate only part of the original set of vertices, then the remaining matrix is a good spectral approximation of the Schur complement. This requires showing the somewhat surprising fact that spectral approximation between a Laplacian and an approximate partial factorization of the Laplacian implies spectral approximation between their Schur complements.

We also improve the performance of their algorithm by combining it with additional leverage score estimation and sparsification to produce a sparser output.

5.1 Preliminaries

This subsection is mostly replicated from [19] for the sake of completeness. We start by introducing Cholesky factorizations and Schur complements. Conventionally, these matrix operations are understood in terms of factorizations into lower triangular matrices. We will instead present an an equivalent view where the Schur complement is obtained by iteratively subtracting rank one terms from a matrix. Let $\boldsymbol{L}$ be the Laplacian of a connected graph. Let $\boldsymbol{L}(:,i)$ denote the i^{th} column of $\boldsymbol{L}$.

$$\boldsymbol{S}^{(1)} \stackrel{\text{def}}{=} \boldsymbol{L} - \frac{1}{\boldsymbol{L}(1,1)}\boldsymbol{L}(:,1)\boldsymbol{L}(:,1)^\top,$$

is called the *Schur complement* of $\boldsymbol{L}$ with respect to vertex 1. $\boldsymbol{S}^{(1)}$ are identically 0, and thus this is effectively a system in the remaining $n-1$ indices.

More generally, we can compute the Schur complement w.r.t. any single vertex (row and column index) of $\boldsymbol{L}$. Suppose we want the Schur complement w.r.t. vertex v_1. Letting $\alpha_1 \stackrel{\text{def}}{=} \boldsymbol{L}(v_1,v_1), \boldsymbol{c}_1 \stackrel{\text{def}}{=} \frac{1}{\alpha_1}\boldsymbol{L}(:,v_1)$, we have $\boldsymbol{L} = \boldsymbol{S}^{(1)} + \alpha_1 \boldsymbol{c}_1 \boldsymbol{c}_1^\top$.

We can also perform a sequence of eliminations, where in the i^{th} step, we select a vertex $v_i \in V \setminus \{v_1,\ldots,v_{i-1}\}$ and eliminate the vertex v_i. We define

$$\begin{aligned}\alpha_i &= \boldsymbol{S}^{(i-1)}(v_i,v_i)\\ \boldsymbol{c}_i &= \frac{1}{\alpha_i}\boldsymbol{S}^{(i-1)}(:,v_i)\\ \boldsymbol{S}^{(i)} &= \boldsymbol{S}^{(i-1)} - \alpha_i \boldsymbol{c}_i \boldsymbol{c}_i^\top.\end{aligned}$$

If at some step i, $\boldsymbol{S}^{(i-1)}(v_i,v_i) = 0$, then we define $\alpha_i = 0$, and $\boldsymbol{c}_i = 0$. However, when the original matrix is the Laplacian of a connected graph, it can be shown that every choice of v_i gives a non-zero α_i, and that the resulting matrix $\boldsymbol{S}^{(i)}$ is always the Laplacian of a connected graph.

While it does not follow immediately from the above, it is a well-known fact that the Schur complement $\boldsymbol{S}^{(i)}$ w.r.t. a sequence of variables $v_1,\ldots,v_i$ does not depend on the order in which the vertices are eliminated (but the $\boldsymbol{c}_i$ and α_i do depend on the order). Consequently it makes sense to define $\boldsymbol{S}^{(i)}$ as the Schur complement w.r.t. elimination of the *set* of vertices $\{v_1,\ldots,v_i\}$ (see Fact 5.3).

Suppose we eliminate a sequence of vertices $v_1,\ldots,v_j$ Let $\boldsymbol{\mathcal{L}}$ be the $n \times j$ matrix with $\boldsymbol{c}_i$ as its i^{th} column, and $\boldsymbol{\mathcal{D}}$ be the $n \times j$ diagonal matrix $\boldsymbol{\mathcal{D}}(i,i) = \alpha_i$, then

$$\boldsymbol{L} = \boldsymbol{S}^{(j)} + \sum_{i=1}^{j} \alpha_i \boldsymbol{c}_i \boldsymbol{c}_i^\top = \boldsymbol{S}^{(j)} + \boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{D}}\boldsymbol{\mathcal{L}}^\top.$$

This decomposition is known a partial Cholesky factorization. Let us write $F = \{v_1,\ldots,v_j\}$, and $C = V - F$. We can then write $\boldsymbol{\mathcal{L}} = \begin{pmatrix}\boldsymbol{\mathcal{L}}_{FF}\\ \boldsymbol{\mathcal{L}}_{CF}\end{pmatrix}$. If we abuse notation and also identify $\boldsymbol{S}^{(j)}$ if the matrix restricted to its non-zero support C, then we can also write

$$\boldsymbol{L} = \begin{pmatrix}\boldsymbol{\mathcal{L}}_{FF} & \boldsymbol{0}\\ \boldsymbol{\mathcal{L}}_{CF} & \boldsymbol{I}_{CC}\end{pmatrix}\begin{pmatrix}\boldsymbol{\mathcal{D}} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{S}^{(j)}\end{pmatrix}\begin{pmatrix}\boldsymbol{\mathcal{L}}_{FF} & \boldsymbol{0}\\ \boldsymbol{\mathcal{L}}_{CF} & \boldsymbol{I}_{CC}\end{pmatrix}^\top \tag{3}$$

Clique Structure of the Schur Complement. Given a Laplacian $\boldsymbol{L}$, let $(\boldsymbol{L})_v \in \mathbb{R}^{n\times n}$ denote the Laplacian corresponding to the edges incident on vertex v, i.e.

$$(\boldsymbol{L})_v \stackrel{\text{def}}{=} \sum_{e\in E: e\ni v} w(e)\boldsymbol{b}_e\boldsymbol{b}_e^\top. \tag{4}$$

For example, we denote the first column of $\boldsymbol{L}$ by $\begin{pmatrix}d\\ -\boldsymbol{a}\end{pmatrix}$, then $(\boldsymbol{L})_1 = \begin{bmatrix}d & -\boldsymbol{a}^\top\\ -\boldsymbol{a} & \text{diag}(\boldsymbol{a})\end{bmatrix}$. We can write the Schur complement $\boldsymbol{S}^{(1)}$ w.r.t. a vertex v_1 as $\boldsymbol{S}^{(1)} = \boldsymbol{L} - (\boldsymbol{L})_{v_1} + (\boldsymbol{L})_{v_1} - \frac{1}{\boldsymbol{L}(v_1,v_1)}\boldsymbol{L}(:,v_1)\boldsymbol{L}(:,v_1)^\top$. It is immediate that $L - (L)_{v_1}$ is a Laplacian matrix, since $\boldsymbol{L} - (\boldsymbol{L})_{v_1} = \sum_{e\in E: e\not\ni v_1} w(e)\boldsymbol{b}_e\boldsymbol{b}_e^\top$. A more surprising (but well-known) fact is that

$$C_{v_1}(\boldsymbol{L}) \stackrel{\text{def}}{=} (\boldsymbol{L})_{v_1} - \frac{1}{\boldsymbol{L}(v_1,v_1)}\boldsymbol{L}(:,v_1)\boldsymbol{L}(:,v_1)^\top \tag{5}$$

is also a Laplacian, and its edges form a clique on the neighbors of v_1. It suffices to show it for $v_1 = 1$. We write $i \sim j$ to denote $(i,j) \in E$. Then

$$\begin{aligned}C_1(\boldsymbol{L}) &= \boldsymbol{L}_1 - \frac{1}{\boldsymbol{L}(1,1)}\boldsymbol{L}(:,1)\boldsymbol{L}(:,1)^\top\\ &= \begin{bmatrix}\boldsymbol{0} & \boldsymbol{0}^\top\\ \boldsymbol{0} & \text{diag}(\boldsymbol{a}) - \frac{\boldsymbol{a}\boldsymbol{a}^\top}{d}\end{bmatrix} = \sum_{i\sim 1}\sum_{j\sim 1}\frac{w(1,i)w(1,j)}{d}\boldsymbol{b}_{(i,j)}\boldsymbol{b}_{(i,j)}^\top.\end{aligned}$$

Thus $\boldsymbol{S}^{(1)}$ is a Laplacian since it is a sum of two Laplacians. By induction, for all k, $\boldsymbol{S}^{(k)}$ is a Laplacian. Thus:

Fact 5.1. *The Schur complement of a Laplacian w.r.t. vertices $v_1,\ldots,v_k$ is a Laplacian.*

5.2 Further Properties of the Schur Complement and Other Factorizations

Consider a general PSD matrix of the form

$$\boldsymbol{M} = \begin{pmatrix}\boldsymbol{A} & \boldsymbol{0}\\ \boldsymbol{B} & \boldsymbol{I}\end{pmatrix}\begin{pmatrix}\boldsymbol{R} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{A} & \boldsymbol{0}\\ \boldsymbol{B} & \boldsymbol{I}\end{pmatrix}^\top \tag{6}$$

where $\boldsymbol{A}$ is invertible and $\boldsymbol{I}$ is the identity matrix on a subset of the indices of $\boldsymbol{M}$. It is easy to show the following well-known fact:

Fact 5.2. *Suppose $\boldsymbol{X}$ is a non-singular matrix and $\boldsymbol{A}$ is a symmetric matrix, and $\boldsymbol{P}$ is the orthogonal projection to the complement of the null space of $\boldsymbol{X}\boldsymbol{A}\boldsymbol{X}^\top$. Then $(\boldsymbol{X}\boldsymbol{A}\boldsymbol{X}^\top)^+ = P\boldsymbol{X}^{-1}\boldsymbol{A}^+\boldsymbol{X}^{-\top}P$.*

Based on Fact 5.2 for vectors orthogonal to null space of $\boldsymbol{M}$ we have

$$\boldsymbol{x}^\top\boldsymbol{M}^+\boldsymbol{x} = \boldsymbol{x}^\top\begin{pmatrix}\boldsymbol{A} & \boldsymbol{0}\\ \boldsymbol{B} & \boldsymbol{I}\end{pmatrix}^{-\top}\begin{pmatrix}\boldsymbol{R}^+ & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{T}^+\end{pmatrix}\begin{pmatrix}\boldsymbol{A} & \boldsymbol{0}\\ \boldsymbol{B} & \boldsymbol{I}\end{pmatrix}^{-1}\boldsymbol{x}$$

By applying the general formula for blockwise inversion and simplifying, we get

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{B} & \boldsymbol{I} \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{A}^{-1} & \boldsymbol{0} \\ -\boldsymbol{B}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix}$$

So

$$\boldsymbol{x}^\top \boldsymbol{M}^+ \boldsymbol{x} = \boldsymbol{x}^\top \begin{pmatrix} \boldsymbol{A}^{-1} & \boldsymbol{0} \\ -\boldsymbol{B}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{R}^+ & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}^+ \end{pmatrix} \begin{pmatrix} \boldsymbol{A}^{-1} & \boldsymbol{0} \\ -\boldsymbol{B}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix} \boldsymbol{x}.$$

Suppose $\boldsymbol{x} = \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{y} \end{pmatrix}$, and again $\boldsymbol{x}$ is orthogonal to the null space of $\boldsymbol{M}$. Then

$$\begin{aligned} &\boldsymbol{x}^\top \boldsymbol{M}^+ \boldsymbol{x} \\ &= \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{y} \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{A}^{-1} & \boldsymbol{0} \\ -\boldsymbol{B}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{R}^+ & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}^+ \end{pmatrix} \begin{pmatrix} \boldsymbol{A}^{-1} & \boldsymbol{0} \\ -\boldsymbol{B}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{y} \end{pmatrix} \\ &= \boldsymbol{y}^\top \boldsymbol{T}^+ \boldsymbol{y}. \end{aligned} \tag{7}$$

Consider a partial Cholesky decomposition of a connected Laplacian $\boldsymbol{L}$ w.r.t. elimination of the sequence of vertices $v_1, \ldots, v_j$, where we write $F = \{v_1, \ldots, v_j\}$ and $C = V - F$. Recall that the resulting Schur complement $\boldsymbol{S}$ is another Laplacian. We can write $\boldsymbol{L}$ in terms of the Schur complement $\boldsymbol{S}$ as

$$\boldsymbol{L} = \begin{pmatrix} \boldsymbol{\mathcal{L}}_{FF} & \boldsymbol{0} \\ \boldsymbol{\mathcal{L}}_{CF} & \boldsymbol{I}_{CC} \end{pmatrix} \begin{pmatrix} \boldsymbol{\mathcal{D}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{\mathcal{L}}_{FF} & \boldsymbol{0} \\ \boldsymbol{\mathcal{L}}_{CF} & \boldsymbol{I}_{CC} \end{pmatrix}^\top \tag{8}$$

Note that as $\boldsymbol{S}$ is a connected Laplacian on a subset of the vertices of $\boldsymbol{L}$. In the full version of the paper, we prove the two standard facts stated below.

Fact 5.3. *The Schur complement of a connected Laplacian $\boldsymbol{L}$ w.r.t. to a sequence of vertices $v_1, \ldots, v_j$ does not depend on the order of elimination of these vertices. Let $C = V - \{v_1, \ldots, v_j\}$, then the Schur complement is equivalent to the Schur complement $\textsc{Schur}(G, C)$ as stated in Definition 3.8.*

Fact 5.4. *Consider a connected Laplacian $\boldsymbol{L}$ and a subset $F \subseteq V$ of its vertices, and let $\boldsymbol{S}$ be the Schur complement of $\boldsymbol{L}$ w.r.t. elimination of F. Let $C = V - F$. Suppose $\boldsymbol{x} = \begin{pmatrix} \boldsymbol{x}_F \\ \boldsymbol{x}_C \end{pmatrix}$ is a vector orthogonal to the null space of $\boldsymbol{L}$, and $\boldsymbol{x}_F = \boldsymbol{0}$.*

Then $\boldsymbol{x}^\top \boldsymbol{L}^+ \boldsymbol{x} = \boldsymbol{x}_C^\top \boldsymbol{S}^+ \boldsymbol{x}_C$.

5.3 Spectral Approximation of the Schur Complement

Theorem 2.2, stated below, characterizes the performance of our algorithm ApproxSchur. This algorithm computes a spectral approximation of the Schur complement of a Laplacian w.r.t elimination of a set of vertices $F = V - C$. The algorithm relies on three procedures:

- LevScoreEst, which computes approximate leverage scores of all edges in a graph; The guarantees of LevScoreEst are given in Lemma 5.5.
- GraphSparsify, which sparsifies a graph. GraphSparsify is characterized in Lemma 5.5.
- CliqueSample which returns a sparse Laplacian matrix approximating a clique created by elimination (see [19], Algorithm 2).

The pseudocode for ApproxSchur is given in Algorithm 3.

Theorem 2.2. *Given a connected undirected multi-graph $G = (V, E)$, with positive edges weights $w : E \to \mathbb{R}_+$, and associated Laplacian $\boldsymbol{L}$, a set vertices $C \subset V$, and scalars $0 < \epsilon \leq 1/2$, $0 < \delta < 1$, the algorithm $\textsc{ApproxSchur}(\boldsymbol{L}, C, \epsilon, \delta)$ returns a Laplacian matrix $\widetilde{\boldsymbol{S}}$. With probability $\geq 1 - \delta$, the following statements all hold: $\widetilde{\boldsymbol{S}} \approx_\epsilon \boldsymbol{S}$, where $\boldsymbol{S}$ is the Schur complement of $\boldsymbol{L}$ w.r.t elimination of $F = V - C$. $\widetilde{\boldsymbol{S}}$ is a Laplacian matrix whose edges are supported on C. Let $k = |C| = n - |F|$. The total number of non-zero entries $\widetilde{\boldsymbol{S}}$ is $O(k\epsilon^{-2} \log(n/\delta))$. The total running time is bounded by $O((m \log n \log^2(n/\delta) + n\epsilon^{-2} \log n \log^4(n/\delta)) \operatorname{polyloglog}(n))$.*

Lemma 5.5 stated below follows immediately from using the Laplacian solver of [16] in the effective resistance estimation procedure of [23].

Lemma 5.5. *Given a connected undirected multi-graph $G = (V, E)$, with positive edges weights $w : E \to \mathbb{R}_+$, and associated Laplacian $\boldsymbol{L}$, and a scalar $0 < \delta < 1$ the algorithm $\textsc{LevScoreEst}(\boldsymbol{L}, \delta)$ returns estimates $\widehat{\tau}_e$ for all the edges such that with probability $\geq 1 - \delta$*

(1) For each edge e, we have $\tau_e \leq \widehat{\tau}_e \leq 1$ where τ_e is the true leverage score of e in G.
(2) $\sum_e \widehat{\tau}_e \leq 2n$.

The algorithm runs in time $O(m \log^2(n/\delta) \operatorname{polyloglog}(n))$.

Lemma 5.5 stated below follows immediately from using the Laplacian solver of [16] in the sparsification routine of [23].

Lemma 5.6. *Given a connected undirected multi-graph $G = (V, E)$, with positive edges weights $w : E \to \mathbb{R}_+$, and associated Laplacian $\boldsymbol{L}$, and scalars $0 < \epsilon \leq 1/2$, $0 < \delta < 1$, $\textsc{GraphSparsify}(\boldsymbol{L}, \epsilon, \delta)$ returns a Laplacian $\tilde{\boldsymbol{L}}$ s.t. with probability $\geq 1 - \delta$ it holds that $\tilde{\boldsymbol{L}} \approx_\epsilon \boldsymbol{L}$ and $\tilde{\boldsymbol{L}}$ has $O(n\epsilon^{-2} \log(n/\delta))$ edges. The algorithm runs in time $O(m \log^2(n/\delta) \operatorname{polyloglog}(n) + n\epsilon^{-2} \log(n/\delta))$.*

Our proof of Theorem 2.2 relies on the following lemma which provides a similar, but seemingly weaker guarantee about the output of the algorithm ApxPartialCholesky. Its pseudo-code is given in Figure 4.

Lemma 5.7. *Given a connected undirected multi-graph $G = (V, E)$, with positive edges weights $w : E \to \mathbb{R}_+$, and associated Laplacian $\boldsymbol{L}$, a set vertices $C \subset V$, and scalars $0 < \delta < 1$, $0 < \epsilon \leq 1/2$, the algorithm*

$\textsc{ApxPartialCholesky}(\boldsymbol{L}, C, \epsilon)$ returns a decomposition $(\widetilde{\boldsymbol{\mathcal{L}}}, \widetilde{\boldsymbol{\mathcal{D}}}, \widetilde{\boldsymbol{S}})$. With probability $\geq 1 - \delta$, the following statements all hold:

$$\boldsymbol{L} \approx_\epsilon \tilde{\boldsymbol{L}} \tag{9}$$

where $F = V - C$ and

$$\tilde{\boldsymbol{L}} = \begin{pmatrix} \widetilde{\boldsymbol{\mathcal{L}}}_{FF} \\ \widetilde{\boldsymbol{\mathcal{L}}}_{CF} \end{pmatrix} \widetilde{\boldsymbol{\mathcal{D}}} \begin{pmatrix} \widetilde{\boldsymbol{\mathcal{L}}}_{FF} \\ \widetilde{\boldsymbol{\mathcal{L}}}_{CF} \end{pmatrix}^\top + \begin{pmatrix} \boldsymbol{0}_{FF} & \boldsymbol{0}_{FC} \\ \boldsymbol{0}_{CF} & \widetilde{\boldsymbol{S}} \end{pmatrix}.$$

Here $\widetilde{\boldsymbol{S}}$ is a Laplacian matrix whose edges are supported on C. Let $k = |C| = n - |F|$. The total number of non-zero entries $\widetilde{\boldsymbol{S}}$ is $O(k\epsilon^{-2} \log(n/\delta))$. $\widetilde{\boldsymbol{\mathcal{L}}}_{FF}$ is an invertible matrix. The total number of non-zero entries in

Algorithm 3: ApproxSchur(L, C, ϵ, δ)

Call LevScoreEst$(\boldsymbol{L}, \delta/3)$ to compute leverage score estimates $\widehat{\tau}_e$ for every edge e;
for *every edge* e **do**
  $\widetilde{S}^{(0)} \leftarrow \tilde{\boldsymbol{L}}$ with multi-edges split into $\rho_e = \left\lceil \widehat{\tau}_e \cdot 12 \left(\frac{\epsilon}{2}\right)^{-2} \ln^2(3n/\delta) \right\rceil$ copies with $1/\rho_e$ of the original weight;
Let $F = V - C$;
Label the vertices in F by $\{1, \ldots, |F|\}$ and the remaining vertices C by $\{|F|+1, \ldots, n\}$;
Let π be a uniformly random permutation on $\{1, \ldots, |F|\}$;
for $i = 1$ *to* $|F|$ **do**
  $\widetilde{C}_i \leftarrow$ CliqueSample$(\widetilde{S}^{(i-1)}, \pi(i))$;
  $\widetilde{S}^{(i)} \leftarrow \widetilde{S}^{(i-1)} - \left(\widetilde{S}^{(i-1)}\right)_{\pi(i)} + \widetilde{C}_i$;
$\widetilde{S} \leftarrow$ GraphSparsify$(\widetilde{S}^{(|F|)}, \epsilon, \delta/3)$;
return $\widetilde{S}$;

$\widetilde{\boldsymbol{\mathcal{L}}}_{FF}$ *and* $\widetilde{\boldsymbol{\mathcal{L}}}_{FC}$ *is* $O(m+n\epsilon^{-2}\log n \log(n/\delta)^2)$. *The total running time is bounded by* $O((m \log n \log^2(n/\delta) + n\epsilon^{-2} \log n \log^4(n/\delta))\,\mathrm{polyloglog}(n))$.

A proof of Lemma 5.7 appears in the full version.

Algorithm 4: ApxPartialCholesky$(\boldsymbol{L}, C, \epsilon, \delta)$

Call LevScoreEst$(\boldsymbol{L}, \delta/3)$ to compute leverage score estimates $\widehat{\tau}_e$ for every edge e;
for *every edge* e **do**
  $\widetilde{S}^{(0)} \leftarrow \tilde{\boldsymbol{L}}$ with multi-edges split into $\rho_e = \left\lceil \widehat{\tau}_e \cdot 12 \left(\frac{\epsilon}{2}\right)^{-2} \ln^2(3n/\delta) \right\rceil$ copies with $1/\rho_e$ of the original weight;
Let $F = V - C$;
Define the diagonal matrix $\widetilde{\boldsymbol{\mathcal{D}}} \leftarrow \boldsymbol{0}_{|F|\times|F|}$;
Label the vertices in F by $\{1, \ldots, |F|\}$ and the remaining vertices C by $\{|F|+1, \ldots, n\}$;
Let π be a uniformly random permutation on $\{1, \ldots, |F|\}$;
for $i = 1$ *to* $|F|$ **do**
  $\widetilde{\boldsymbol{\mathcal{D}}}(i,i) \leftarrow (\pi(i), \pi(i))$ entry of $\widetilde{S}^{(i-1)}$;
  $\boldsymbol{e}_i \leftarrow \pi(i)^{\text{th}}$ column of $\widetilde{S}^{(i-1)}$ divided by $\widetilde{\boldsymbol{\mathcal{D}}}(i,i)$ if $\widetilde{\boldsymbol{\mathcal{D}}}(i,i) \neq 0$, or zero otherwise;
  $\widetilde{C}_i \leftarrow$ CliqueSample$(\widetilde{S}^{(i-1)}, \pi(i))$;
  $\widetilde{S}^{(i)} \leftarrow \widetilde{S}^{(i-1)} - \left(\widetilde{S}^{(i-1)}\right)_{\pi(i)} + \widetilde{C}_i$;
$\widetilde{\boldsymbol{\mathcal{L}}} \leftarrow \begin{pmatrix} \boldsymbol{c}_1 & \boldsymbol{c}_2 & \ldots & \boldsymbol{c}_{|F|} \end{pmatrix}$;
$\widetilde{S} \leftarrow$ GraphSparsify$(\widetilde{S}^{(|F|)}, \epsilon, \delta/3)$;
return $(\widetilde{\boldsymbol{\mathcal{L}}}, \widetilde{\boldsymbol{\mathcal{D}}}, \widetilde{S})$;

Proof. (of Theorem 2.2) Note, given the elimination ordering $\pi(v_1), \ldots, \pi(v_{|F|})$ we can do a partial Cholesky factorization of $\boldsymbol{L}$ as

$$\boldsymbol{L} = \begin{pmatrix} \boldsymbol{\mathcal{L}}_{FF} & \boldsymbol{0} \\ \boldsymbol{\mathcal{L}}_{CF} & \boldsymbol{I}_{CC} \end{pmatrix} \begin{pmatrix} \boldsymbol{\mathcal{D}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{\mathcal{L}}_{FF} & \boldsymbol{0} \\ \boldsymbol{\mathcal{L}}_{CF} & \boldsymbol{I}_{CC} \end{pmatrix}^\top \tag{10}$$

where $\boldsymbol{S}$ is the Schur complement of $\boldsymbol{L}$ w.r.t. F.

We note that ApproxSchur and ApxPartialCholesky perform exactly the same computations, with the exception that ApxPartialCholesky records the values $\widetilde{\boldsymbol{\mathcal{L}}}$ and $\widetilde{\boldsymbol{\mathcal{D}}}$. This means we can establish a simple coupling between the algorithms by considering them executing based on the same source of randomness: They must then return the same matrix $\widetilde{S}$. Thus, if we can show for the matrix $\widetilde{S}$ returned by ApxPartialCholesky that $\widetilde{S} \approx_\epsilon \boldsymbol{S}$, then the same must be true for the $\widetilde{S}$ returned by ApproxSchur.

We can write the matrix $\tilde{\boldsymbol{L}}$ constructed from the output of ApxPartialCholesky as

$$\tilde{\boldsymbol{L}} = \begin{pmatrix} \widetilde{\boldsymbol{\mathcal{L}}}_{FF} & \boldsymbol{0} \\ \widetilde{\boldsymbol{\mathcal{L}}}_{CF} & \boldsymbol{I}_{CC} \end{pmatrix} \begin{pmatrix} \widetilde{\boldsymbol{\mathcal{D}}} & \boldsymbol{0} \\ \boldsymbol{0} & \widetilde{S} \end{pmatrix} \begin{pmatrix} \widetilde{\boldsymbol{\mathcal{L}}}_{FF} & \boldsymbol{0} \\ \widetilde{\boldsymbol{\mathcal{L}}}_{CF} & \boldsymbol{I}_{CC} \end{pmatrix}^\top \tag{11}$$

We now suppose that ApxPartialCholesky succeeds and returns $\tilde{\boldsymbol{L}} \approx_\epsilon \boldsymbol{L}$. These two matrices must have the same null space, namely the span of $\boldsymbol{1}$. Consider $\boldsymbol{x} = \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{y} \end{pmatrix}$, where $\boldsymbol{y}$ is orthogonal to $\boldsymbol{1}_C$ and hence $\boldsymbol{x}$ is orthogonal to $\boldsymbol{1}$. By Equation (7), $\boldsymbol{x}^\top \tilde{\boldsymbol{L}}^+ \boldsymbol{x} = \boldsymbol{y}^\top \widetilde{\boldsymbol{S}}^+ \boldsymbol{y}$, and $\boldsymbol{x}^\top \boldsymbol{L}^+ \boldsymbol{x} = \boldsymbol{y}^\top \boldsymbol{S}^+ \boldsymbol{y}$. $\tilde{\boldsymbol{L}} \approx_\epsilon \boldsymbol{L}$ implies $\tilde{\boldsymbol{L}}^+ \approx_\epsilon \boldsymbol{L}^+$, and so

$$\exp(-\epsilon)\boldsymbol{y}^\top \boldsymbol{S}^+ \boldsymbol{y} \leq \boldsymbol{y}^\top \widetilde{\boldsymbol{S}}^+ \boldsymbol{y} \leq \exp(\epsilon)\boldsymbol{y}^\top \widetilde{\boldsymbol{S}}^+ \boldsymbol{y}. \tag{12}$$

Furthermore, both $\boldsymbol{S}$ and $\widetilde{S}$ have a null space that is exactly the span of $\boldsymbol{1}_C$. We can see this in two steps: Firstly, both are Laplacian matrices, so their null spaces must include the span of $\boldsymbol{1}_C$. Secondly, from the product forms in Equations (10) and (11), if either had null space of rank strictly larger than 1, then the rank of $\boldsymbol{L}$ or $\tilde{\boldsymbol{L}}$ would be strictly less than 1, which is false. So by contradiction, both $\boldsymbol{S}$ and $\widetilde{S}$ have a null space that is exactly the span of $\boldsymbol{1}_C$. From this and Equation (12), which holds for all $\boldsymbol{y}$ orthogonal to $\boldsymbol{1}_C$, we conclude $\widetilde{\boldsymbol{S}}^+ \approx_\epsilon \boldsymbol{S}^+$. This in turn implies $\widetilde{S} \approx_\epsilon \boldsymbol{S}$.

The guarantees of success probability, running time and sparsity of $\widetilde{S}$ for ApproxSchur now follow from the guarantees for ApxPartialCholesky given in Lemma 5.7. □

6 EFFECTIVE RESISTANCE ESTIMATION

Recalling the statement of Theorem 2.3, we will give our algorithm and show the following.

Theorem 2.3. *When given a graph* G, *a set* S *of pairs of vertices, and an error parameter* ϵ, *the function* EstimateR$_{\text{eff}}$ *(Algorithm 5 in Section 6) returns* $e^{\pm\epsilon}$*-multiplicative estimates of the effective resistance of each of the pairs in* S *in time* $\widetilde{O}\left(m + \frac{n+|S|}{\epsilon^2}\right)$ *with high probability.*

First, we give our algorithm for estimating the effective resistance of a set of pairs S that achieves an improved running time (ignoring $\log(n)$ factors) over algorithms that are based on the Johnson-Lindenstrauss Lemma, for a sufficiently small set of pairs and error parameter. The algorithm EstimateR$_{\text{eff}}$ (Algorithm 5) is given below. The main tool it uses is the ability to quickly compute

a sparse spectral approximation of the Schur complement of a graph onto a subset of its vertices, along with the following observations:

(1) An approximate Schur complement of a graph onto a subset V_i of the vertices approximately preserves effective resistances between elements of V_i
(2) If the number of vertex pairs we wish to compute the effective resistances of is much smaller than the number of vertices, then there must be a large number of vertices that are not part of any pair, and these vertices can be removed by taking a Schur complement, shrinking the size of the graph.

Our proof of Theorem 2.3 relies on Theorem 6.1, which gives guarantees for the purely combinatorial Schur complement approximation algorithm CombApproxSchur that are almost as strong as the guarantees for the ApproxSchur algorithm given in Theorem 2.2. We prove Theorem 6.1 in the full version of this paper.

Theorem 6.1. *Given a connected undirected multi-graph $G = (V,E)$, with positive edges weights $w : E \to \mathbb{R}_+$, and associated Laplacian $\mathbf{L}$, a set vertices $C \subset V$, and and scalars $0 < \epsilon \leq 1/2$, $0 < \delta < 1$, the algorithm CombApproxSchur$(\mathbf{L}, C, \epsilon, \delta)$ returns a Laplacian matrix $\widetilde{\mathbf{S}}$. With probability $\geq 1-\delta$ the following statements hold: $\widetilde{\mathbf{S}} \approx_\epsilon \mathbf{S}$, where $\mathbf{S}$ is the Schur complement of $\mathbf{L}$ w.r.t elimination of $F = V - C$. $\widetilde{\mathbf{S}}$ is a Laplacian matrix whose edges are supported on C. Let $k = |C| = n - |F|$. The total number of non-zero entries $\widetilde{\mathbf{S}}$ is $O(k\epsilon^{-2}\,\mathrm{polylog}(n/\delta))$. The total running time is bounded by $O((m + n\epsilon^{-2})\,\mathrm{polylog}(n/\delta))$.*

Algorithm 5: EstimateR$_{\text{eff}}(G = (V,E), S, \epsilon)$

Input: A graph $G = (V,E)$, a set $S \subseteq V \times V$ of vertex pairs, and an error tolerance $0 < \epsilon \leq 1$
Output: Estimates of the effective resistances of each of the pairs in S accurate to within a factor of $e^{\pm\epsilon}$ with high probability

$\epsilon' \leftarrow \frac{\epsilon}{\log_2 n}$.
return HelpEstimateR$_{\text{eff}}(G, S, \epsilon')$

We now prove that this algorithm quickly computes effective resistances. In doing this analysis, we did not try to optimize log factors, and we believe that at least some of them can likely be eliminated through a more careful martingale analysis.

Proof. (of Theorem 2.3) First we prove correctness. In any recursive call of HelpEstimateR$_{\text{eff}}$ (Algorithm 6), let $\boldsymbol{L}$ denote the Schur complement of the graph onto (say) V_1. Fact 5.4 says that the Schur complement of a graph onto a subset of its vertices V_1 exactly preserves effective resistances between vertices in V_1. However, the algorithm we are analyzing does not take an exact Schur complement. Instead, it takes an approximate Schur complement $\widetilde{\boldsymbol{L}}$ which by Theorem 6.1, satisfies $e^{-\epsilon'}\boldsymbol{L} \preceq \widetilde{\boldsymbol{L}} \preceq e^{\epsilon'}\boldsymbol{L}$. We also know that the effective resistance between i and j in the approximate Schur complement is given by $(\vec{1}_i - \vec{1}_j)^\intercal \widetilde{\boldsymbol{L}}^\dagger (\vec{1}_i - \vec{1}_j)$, where $\vec{1}_z$ is the zth standard basis vector. These two facts imply that the effective resistance between i and j in $\widetilde{\boldsymbol{L}}$ is within an $e^{\pm\epsilon'}$ factor of what it was before taking the approximate Schur complement. Applying this inductively over the depth of the recursion, we get that the approximate effective resistances $\widetilde{R}_{\text{eff}}$ returned by the algorithm satisfy

$$e^{-\epsilon'(\lceil \log_2 n \rceil - 1)} R_{\text{eff}} \leq \widetilde{R}_{\text{eff}} \leq e^{\epsilon'(\lceil \log_2 n \rceil - 1)} R_{\text{eff}}$$
$$e^{-\epsilon} R_{\text{eff}} \leq \widetilde{R}_{\text{eff}} \leq e^{\epsilon} R_{\text{eff}}$$

For runtime, let n, m be the number of vertices and edges in the original graph, before any recursion is done. Consider any recursive call c. Let n_c be the number of vertices of the graph G that is given to c as an argument, before any modifications within c have been done. Let s_c denote the number of pairs in the argument S passed to the recursive call c. Finally, let n'_c denote the number of vertices in G after G has been replaced with its Schur complement onto V_0 in the call. By Theorem 6.1, the actual amount of work done in a recursive call of HelpEstimateR$_{\text{eff}}$ (other than the top level call) is $\widetilde{O}(n_c/\epsilon^2)$. Here and for the rest of this proof, $\widetilde{O}$ hides factors

Algorithm 6: HelpEstimateR$_{\text{eff}}(G = (V,E), S, \epsilon)$

Input: A graph $G = (V,E)$, a set $S \subseteq V \times V$ of vertex pairs, and an error tolerance $0 < \epsilon \leq 1$
Output: Estimates of the effective resistances of each of the pairs in S accurate to within a factor of $e^{\pm\epsilon \log_2 n}$ with high probability

if $S = \emptyset$ **then**
  return $\emptyset$

Let V_0 denote the set of all vertices that are part of at least one pair in S.
$G \leftarrow$ CombApproxSchur $(G, V_0, \epsilon,$ with high probability) (see Theorem 6.1) $V \leftarrow V_0$

if $|S| = 1$ *(or equivalently,* $|V| = 2$*)* **then**
  Let z denote the pair in S or equivalently, the only two vertices in the graph.
  return *the estimate* $1/w_z$*, where* w_z *is the weight of the only edge in* G.

Partition V into V_1, V_2 with $|V_1| = \lfloor n/2 \rfloor$ and $|V_2| = \lfloor n/2 \rfloor$.
Partition S into subsets S_1, S_2, S_3 with:
  $S_1 \leftarrow$ pairs with both elements in V_1
  $S_2 \leftarrow$ pairs with both elements in V_2
  $S_3 \leftarrow$ pairs with one element in V_1 and the other in V_2.
Let $G_1 \leftarrow$ CombApproxSchur $(G, V_1, \epsilon,$ with high probability).
Let $G_2 \leftarrow$ CombApproxSchur $(G, V_2, \epsilon,$ with high probability).
Concatenate and return the estimates given by:
  HelpEstimateR$_{\text{eff}}(G_1, S_1, \epsilon)$
  HelpEstimateR$_{\text{eff}}(G_2, S_2, \epsilon)$
  HelpEstimateR$_{\text{eff}}(G, S_3, \epsilon)$

polylogarithmic in n, but does not hide anything that explicitly depends on on n'_c or ϵ.

We claim that with proper amortization, the amount of work done in each recursive call is $\widetilde{O}(n'_c/\epsilon^2)$. To show this, define a potential function ϕ_c which is $\widetilde{\Theta}(n'_c/\epsilon^2)$. Then define the amortized cost of a recursive call as its true cost plus $(\phi_c - \phi_{\text{parent}(c)}/3)$. Since the recursion tree has branching factor 3, the sum of the amortized costs of the calls upper bounds the total true cost.

Then we have that the amortized cost of a call c is

$$\widetilde{O}(n_c/\epsilon^2) + (\phi_c - \phi_{\text{parent}(c)}/3) =$$
$$\widetilde{O}(n_c/\epsilon^2) + (\phi_c - \phi_{\text{parent}(c)})/3 + (2/3)\phi_c \leq \widetilde{O}(n'_c/\epsilon^2).$$

Recall that n'_c is the number of vertices in the graph given to the call that are part of at least one pair in S. Thus, $n'_c \leq 2s_c$. Putting this all together, we get that the total amortized work done in the first level of HelpEstimateR_{eff} is $\widetilde{O}(m + n/\epsilon^2)$, and for any subsequent level, it is given by $\sum_{\text{calls } c \text{ in the level}} \widetilde{O}(n_c/\epsilon^2) + (\phi_c - \phi_{\text{parent}(c)}/3) \leq \sum_{\text{calls } c \text{ in the level}} \widetilde{O}(s_c/\epsilon^2) \leq \widetilde{O}(|S|/\epsilon^2)$. Summing over all levels gives the claimed bound of $\widetilde{O}\left(m + \frac{n+|S|}{\epsilon^2}\right)$. □

REFERENCES

[1] David Aldous. 1990. The Random Walk Construction of Uniform Spanning Trees and Uniform Labelled Trees. In *SIAM Journal on Discrete Mathematics*. 450–465.

[2] Arash Asadpour, Michel X. Goemans, Aleksander Mądry, Shayan Oveis Gharan, and Amin Saberi. 2010. An O(Log N/ Log Log N)-approximation Algorithm for the Asymmetric Traveling Salesman Problem. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. 379–389.

[3] Andrei Broder. 1989. Generating Random Spanning Trees. In *Proceedings of the 30th annual Symposium on Foundations of Computer Science, FOCS 1989*. 442–447.

[4] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup Rao, and Shen Chen Xu. 2014. Solving SDD linear systems in nearly $m \log^{1/2} n$ time. In *STOC*. 343–352.

[5] Charles J Colbourn, Robert PJ Day, and Louis D Nel. 1989. Unranking and ranking spanning trees of a graph. *Journal of Algorithms* 10, 2 (1989), 271–286.

[6] Charles J Colbourn, Wendy J Myrvold, and Eugene Neufeld. 1996. Two algorithms for unranking arborescences. *Journal of Algorithms* 20, 2 (1996), 268–281.

[7] Shayan Oveis Gharan, Amin Saberi, and Mohit Singh. 2011. A Randomized Rounding Approach to the Traveling Salesman Problem. In *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science (FOCS '11)*. 550–559.

[8] Navin Goyal, Luis Rademacher, and Santosh Vempala. 2009. Expanders via Random Spanning Trees. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*. 576–585.

[9] Alain Guenoche. 1983. Random spanning tree. *Journal of Algorithms* 4, 3 (1983), 214–220.

[10] Nicholas J. A. Harvey and Keyulu Xu. 2016. Generating Random Spanning Trees via Fast Matrix Multiplication. In *LATIN 2016: Theoretical Informatics*, Vol. 9644. 522–535.

[11] Roger A Horn and Charles R Johnson. 2012. *Matrix analysis*. Cambridge university press.

[12] Jonathan Kelner and Aleksander Madry. 2009. Faster Generation of Random Spanning Trees. In *Proceedings of the 50th annual Symposium on Foundations of Computer Science, FOCS 2009*. 13–21. Available at https://arxiv.org/abs/0908.1448.

[13] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. 2013. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Proceedings of the 45th Annual Symposium on Theory of Computing (STOC '13)*. ACM, New York, NY, USA, 911–920. Available at http://arxiv.org/abs/1301.6628.

[14] Gustav Kirchhoff. 1847. Über die Auflösung der Gliechungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme geführt wird. In *Poggendorgs Ann. Phys. Chem.* 497–508.

[15] I. Koutis, G. Miller, and R. Peng. 2014. Approaching Optimality for Solving SDD Linear Systems. *SIAM J. Comput.* 43, 1 (2014), 337–354.

[16] Ioannis Koutis, Gary L. Miller, and Richard Peng. 2011. A Nearly-m log n Time Solver for SDD Linear Systems. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS '11)*. 590–598.

[17] Vidyadhar G. Kulkarni. 1990. Generating random combinatorial objects. *Journal of Algorithms* 11, 2 (1990), 185–207.

[18] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A Spielman. 2016. Sparsified Cholesky and multigrid solvers for connection laplacians. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 842–850. Available at http://arxiv.org/abs/1512.01892.

[19] Rasmus Kyng and Sushant Sachdeva. 2016. Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple. In *Proceedings of the 57th annual Symposium on Foundations of Computer Science, FOCS 2016*. Available at https://arxiv.org/pdf/1605.02353v1.pdf.

[20] Yin Tat Lee and Aaron Sidford. 2013. Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 147–156.

[21] Russell Lyons and Yuval Peres. 2016. *Probability on trees and networks*. Vol. 42. Cambridge University Press.

[22] Aleksander Madry, Damian Straszak, and Jakub Tarnawski. 2015. Fast Generation of Random Spanning Trees and the Effective Resistance Metric. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*. 2019–2036. Available at http://arxiv.org/pdf/1501.00267v1.pdf.

[23] Daniel A. Spielman and Nikhil Srivastava. 2011. Graph Sparsification by Effective Resistances. *SIAM J. Comput.* 40, 6 (2011), 1913–1926.

[24] Daniel A. Spielman and Shang-Hua Teng. 2014. Nearly Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems. *SIAM J. Matrix Anal. Appl.* 35, 3 (2014), 835–885.

[25] Virginia Vassilevska Williams. 2012. Multiplying Matrices Faster Than Coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing (STOC '12)*. ACM, New York, NY, USA, 887–898.