Fast and Space Efficient Spectral Sparsification in Dynamic Streams*

Michael Kapralov
EPFL

Aida Mousavifar
EPFL

Cameron Musco
UMass Amherst

Christopher Musco
New York University

Navid Nouri
EPFL

Aaron Sidford
Stanford University

Jakab Tardos
EPFL

Abstract

In this paper, we resolve the complexity problem of spectral graph sparcification in dynamic streams up to polylogarithmic factors. Using a *linear sketch* we design a streaming algorithm that uses $\tilde{O}(n)$ space, and with high probability, recovers a spectral sparsifier from the sketch in $\tilde{O}(n)$ time.[1] Prior results either achieved near optimal $\tilde{O}(n)$ space, but $\Omega(n^2)$ recovery time [Kapralov et al. '14], or ran in $o(n^2)$ time, but used polynomially suboptimal space [Ahn et al '13].

Our main technical contribution is a novel method for recovering graph edges with high effective resistance from a linear sketch. We show how to do so in nearly linear time by 'bucketing' vertices of the input graph into clusters using a coarse approximation to the graph's effective resistance metric.

A second main contribution is a new pseudorandom generator (PRG) for linear sketching algorithms. Constructed from a *locally computable randomness extractor*, our PRG stretches a seed of $\tilde{O}(n)$ random bits polynomially in length with just $\log^{O(1)} n$ runtime cost per evaluation. This improves on Nisan's commonly used PRG, which in our setting would require $\tilde{O}(n)$ time per evaluation. Our faster PRG is essential to simultaneously achieving near optimal space and time complexity.

*This paper merges a subset of results in https://arxiv.org/abs/1903.12165 and https://arxiv.org/abs/1903.12150.

[1] We use $\tilde{O}(t)$ to denote $O(t \log^c n)$ for fixed constant c that is independent of n

1 Introduction

The dynamic streaming setting has emerged as a popular model for studying algorithms for processing massive graphs. In this model, we receive a stream of edge insertions or deletions to a graph with n-nodes. The goal is to maintain a small space (typically $\ll n^2$) "sketch" of the information received, and at the end of the stream, use the sketch to compute some property of the streamed graph (its minimum spanning tree, number of connected components, etc.) [McG14].

Many basic graph problems have been studied in the dynamic streaming setting – we review a subset of this work in Section 1.3. Allowing for edge deletions (in contrast to "insertion only" streams) makes the dynamic model particularly difficult, and theoretical work has lead to a powerful toolkit of algorithmic techniques for large scale and distributed graph processing [McG17].

Within the broad literature on dynamic graph streaming, *graph sparsification* has emerged as a central challenge problem. In the offline setting, it is known that any n-node, undirected graph G can be well approximated by a sparse graph [ST11, BSS12]. In particular, even though G may have $\Theta(n^2)$ edges, it is always possible to find a graph G' with $O(n/\epsilon^2)$ edges so that, letting L and L' denote the graph Laplacians of

G and G', respectively,

$$(1-\epsilon)L \preceq L' \preceq (1+\epsilon)L. \tag{1.1}$$

Here $\preceq$ stands for the positive semidefinite ordering of matrices. Any G' satisfying (1.1) is called a *spectral sparsifier* for G. L' is called a spectral approximation to L. This is a strong notion of graph approximation, implying e.g., that all cut values and effective resistances in G are preserved to within a $(1 \pm \epsilon)$ factor in G' [SS11]. If G' can be computed in a dynamic graph stream, it can be used as a general proxy for answering a variety questions about G, which makes spectral sparsification a natural and important goal.

1.1 Streaming graph sparsification. The first progress on computing graph sparsifiers in dynamic streams focused on the weaker approximation goal of cut sparsification due to [Kar94, BK96]. Ahn, Guha, and McGregor [AGM12a] gave a $O(\log n)$-pass sparsification algorithm for dynamic streaming cut sparsifiers, before later developing a single-pass algorithm using nearly optimal $\widetilde{\Omega}(n\epsilon^{-2})$ space (as for spectral sparsifiers, outputting G' requires $O(n\epsilon^{-2})$ space) [AGM12b]. Ahn, Guha, and McGregror's methods are based on *oblivous linear sketching*: they sketch G by multiplying its edge-vertex incidence matrix by a random matrix, an operation that is implemented easily in the dynamic streaming setting. The random matrix is chosen *independently* of the input data (i.e., "obliviously") and a cut sparsifier is extracted directly from the resulting matrix product i.e., the graph sketch.

Like algorithms for many other streaming graph problems [AGM12a], the cut sparsification method of [AGM12b] specifically exploits linear sketches for ℓ_0 sparse recovery [JST11, KNP+17]. One ℓ_0 sketch is stored for each node of the graph, maintaining local neighborhood information. At the end of the graph stream, this local information is used to extract global information about G via a node contraction procedure reminiscent of Boruvka's algorithm. This procedure allows the algorithm to find edges crossing small cuts in G, an essential step to computing a cut sparsifier.

Sparse recovery sketches are also used to address the more challenging problem of spectral sparsification in [AGM13], but lead to a solution with suboptimal $\tilde{O}(n^{5/3}\epsilon^{-2})$ space and runtime. The open problem of achieving near optimal $\tilde{O}(n\epsilon^{-2})$ space for streaming spectral sparsifiers was eventually resolved in [KLM+17]. This result, which was also surveyed in [Woo14], is based on *edge sampling*. In particular, a well-known approach for computing spectral sparsifiers in the offline setting is to subsample edges from G with non-uniform probabilities proportional to each edge's *effective resistance* [SS11]. Edges with high effective resistance are more "important" to constructing a spectral sparsifier. The key technique in [KLM+17] is an algorithmic primitive for recovering high effective resistance edges of G from a collection of ℓ_2 sparse recovery sketches, departing from prior methods based on ℓ_0 sketches. As in the case of cut sparsifiers, one sketch is maintained for each node.

1.2 Our contributions. While nearly optimal in terms of space, the algorithm from [KLM+17] is slow – it requires $\Omega(n^2)$ time to extract a spectral sparsifier G' from the maintained linear graph sketch. As small time and space complexity are both critical in large scale graph processing, a quadratic dependence on n is prohibitively expensive. Thus, the central open question raised by [KLM+17] was whether or not an $\tilde{O}(n\epsilon^{-2})$ space algorithm could also be designed to run in $\tilde{O}(n\epsilon^{-2})$ time.

Our main contribution is a resolution of this question:

THEOREM 1.1. *There is an algorithm which processes, in a single pass, a list of edge insertions and deletions to an unweighted graph G and, for any $\epsilon \leq 1$, maintains a linear sketch of this input in $\widetilde{O}(n\epsilon^{-2})$ space. With high probability, the algorithm computes in $\tilde{O}(n\epsilon^{-2})$ time*

a weighted subgraph G' with $O(\epsilon^{-2} n \log n)$ edges, such that G' is a $(1 \pm \epsilon)$-spectral sparsifier of G – i.e., G' satisfies (1.1).

Remarks. Theorem 1.1 is easily extended to weighted graphs using a standard reduction to the unweighted problem [AGM12b, KNP+17]. This reduction incurs a logarithmic dependence on the ratio of the maximum and minimum edge weight in G, both in space and runtime. Theorem 1.1 can also be modified to return a spectral sparsifier of optimal, $O(n\epsilon^{-2})$ size. In particular, we can apply the offline algorithm of [LS17] to compute an optimally sized $(1 \pm \epsilon)$ sparsifier G'' for G' and simply note that G'' is a $(1 \pm \epsilon)^2$ spectral sparifier for G.

Proving Theorem 1.1 requires two key technical contributions:

Efficient edge recovery. As in the algorithm of [KLM+17], our method is based on designing a primitive for recovering high effective resistance edges from a dynamically streamed graph G. To do so in $\tilde{O}(n)$ instead of $\tilde{O}(n^2)$ time requires a substantial departure from prior techniques. In particular, [KLM+17] requires exhaustively testing all pairs of nodes in G to determine if they are linked by a high effective resistance edge. This incurs an $\Omega(n^2)$ runtime cost.

We more efficiently take advantage of information in local node sketches by showing that any high effective resistance edge must have high energy in the electrical flow between one of the edge's end points and an arbitrary nearby node in the effective resistance metric. All high energy edges in this flow can be identified with a single ℓ_2 sparse recovery sketch, allowing us to perform $\tilde{O}(n)$ tests overall (one for each node in G) in contrast to the $\tilde{O}(n^2)$ tests performed in [KLM+17].

Applying the above technique requires a novel clustering method which "buckets" nodes based on their distance in G's effective resistance metric and tests flows within each bucket. Inspired by methods for locality sensitive hashing, our bucketing method can be implemented in a dynamic stream using linear sketches for approximating ℓ_2 distances.

This approach can be viewed as the first efficient 'ℓ_2-graph sketching' result, using an analogy to compressed sensing for vector data. Nearly linear time compressed sensing algorithms usually operate by hashing the input vector into buckets so as to isolate dominant entries, which can then be recovered efficiently. Our work provides an analogous 'bucketing scheme' for graphs, which allows for nearly linear time recovery and may be of independent interest.

Faster pseudorandomness for linear sketching. In addition to a better understanding of edge recovery, obtaining nearly optimal sketching methods for sparsification requires solving a largely orthogonal issue. In particular, existing dynamic streaming algorithms for sparsification are developed with the assumption that we have access to a large number of uniformly random hash functions. To ensure that the algorithms actually run in small space, a common technique is to compress these hash functions using a pseudorandom number generator (PRG) for small space computation [Ind06]. Unfortunately, existing PRGs run slowly in our setting, creating another $\Omega(n^2)$ runtime bottleneck for computing a sparsifier [Nis92].

We address this issue by describing a much faster, "locally computable" pseudorandom generator based on a construction of Nisan and Zuckerman [NZ96] and a locally computable randomness extractor of De and Vidick [DV10]. We believe this result will be more widely useful in designing faster sketching algorithms for graph problems and other applications.

1.3 Related work.

The study of algorithms for processing streaming graph data began with work of Feigenbaum et al. [FKM+05], which considered streams of edge *insertions* only. For the more challenging *dynamic* setting, where

both edge insertions and deletions may appear in the stream, few algorithms were known until the seminal work of Ahn, Guha and McGregor [AGM12a]. Amongst other problems, [AGM12a] presents a streaming algorithm for graph connectivity with $O(n \log^3 n)$ space complexity (which is now known to be optimal [NY19]). This algorithm was the first to use oblivious linear sketches for streaming graphs.

Since [AGM12a], oblivious linear sketching has turned out to be a powerful technique for progress on dynamic streaming graph problems.[2] Sketching had lead to efficient algorithms for dense subgraph detection [MTVV15, BHNT15], spanner construction [AGM12b, KW14], matching and matching size approximation [AKL17], sketching the Laplacian [ACK+16, JS18], and a wide variety of other problems. We refer the reader to the surveys of [McG14, McG17] for a more complete review of work on graph sketching. We note that algorithmic techniques developed through this work have been influential in developing dynamic and distributed graph algorithms as well [KKM13, AKLY16, AKLY16].

The effective resistance metric. The *effective resistance metric* induced by an undirected graph plays a central role in spectral graph theory and has been at the heart of numerous algorithmic breakthroughs over the past decade. Aside from its role in spectral sparsification, it is used in constructing vertex sparsifiers [KLP+16b], sparsifiers of the random walk Laplacian [JKPS17], and subspace sparsifiers [LS18]. It has played a key role in many advances in solving Laplacian systems [ST04, KMP10, KMP11, PS14, CKM+14, KLP+16b, KS16] and algorithms for maximum flow and minimum cost flow [LS14]. Given their utility, the computation of effective resistances has itself become an area of active research [DKP+17, JS18, CGP+18, LS18].

[2] In fact, it is now known that linear sketching is essentially a universal approach for designing dynamic streaming algorithms [LNW14, AHLW16].

2 Preliminaries

General Notation. For $s \in \mathbb{R}$, let $s^+ := \max\{0, s\}$. Let $G = (V, E)$ be an unweighted, undirected graph with n vertices and m edges. For any vertex $v \in V$, let $\chi_v \in \mathbb{R}^n$ be an indicator vector with a one at position v and zeros elsewhere. Let $B_n \in \mathbb{R}^{\binom{n}{2} \times n}$ denote the vertex edge incidence matrix of an unweighted and undirected complete graph. For any edge $e = (u, v) \in \binom{n}{2}$, B_n's e^{th} row is equal to $\mathbf{b}_e := \mathbf{b}_{uv} := \chi_u - \chi_v$. Let $B \in \mathbb{R}^{\binom{n}{2} \times n}$ denote the vertex edge incidence matrix of $G = (V, E)$. B is obtained by zeroing out any rows of B_n corresponding to $(u, v) \notin E$.[3] Denote a weighted graph as $G = (V, E, w)$, where $w : E \to \mathbb{R}_+$ gives the edge weights. Let $W \in \mathbb{R}^{\binom{n}{2} \times \binom{n}{2}}$ be a diagonal matrix where $W(e, e) = w(e)$ for $e \in E$ and $W(e, e) = 0$ otherwise. $L_G = B^\top W B = B_n^T W B_n$, is the Laplacian matrix of G. Let L_G^+ denote the Moore-Penrose pseudoinverse of L_G. When the graph G is clear from context we sometimes drop the subscript from L_G.

We require the following basic fact about unweighed, undirected graphs:

FACT 2.1. (LEMMA 6.1, [ST14]) *Let L be the Laplacian of an unweighted, undirected graph. L's non-zero eigenvalues are lower bounded by $\lambda_\ell = \frac{1}{8n^2}$ and upper bounded by $\lambda_u = 2n$. Let $d := \lceil \log_2 \frac{\lambda_u}{\lambda_\ell} \rceil$. Note that $d = \Theta(\log n)$.*

DEFINITION 2.1. *For any unweighted graph $G = (V, E)$ and $\ell \in [0, d+1]$ we define L_ℓ as follows:*

$$L_\ell = \begin{cases} L_G + \frac{\lambda_u}{2^\ell} I & \text{if } 0 \le \ell \le d \\ L_G & \text{if } \ell = d+1. \end{cases}$$

where d and λ_u are as defined in Fact 2.1.

REMARK 2.1. *Our main algorithm employs a chain of 'coarse sparsifiers' of $L_0, \ldots, L_{d+1}$ (thus*

[3] This definition differs from others where B is an $n \times m$ matrix with rows not in G removed altogether.

the last is a sparsifier for L_G itself). Since L_ℓ and $L_{\ell+1}$ are constant factor spectral approximations to each other, once we have obtained a sparsifier for L_ℓ we can use it to approximate effective resistances and compute a sparsifier for $L_{\ell+1}$. For details see Lemma A.1.

Effective Resistance. We associate any weighted graph $G = (V, E, w)$ with an electric circuit: the vertices are junctions and each edge e is a resistor of resistance $1/w(e)$. Now suppose in this circuit we inject one unit of current at vertex u, extract one from vertex v, and let $\mathbf{f}_{uv} \in \mathbb{R}^{\binom{n}{2}}$ denote the the currents induced on the edges. By Kirchhoff's current law, except for the source u and the sink v, the sum of the currents entering and exiting any vertex is zero. Hence, we have $\mathbf{b}_{uv} = B^\top \mathbf{f}_{uv}$. Let $\varphi \in \mathbb{R}^n$ denote the voltage potentials induced at the vertices in the above setting. By Ohm's law we have $\mathbf{f}_{uv} = WB\varphi$. Putting these facts together:

$$\chi_u - \chi_v = B^\top W B \varphi = L\varphi.$$

Observe that $(\chi_u - \chi_v) \perp \ker(L)$, and hence $\varphi = L^+(\chi_u - \chi_v)$.

The *effective resistance* between vertices u and v, denoted by R_{uv} is defined as the voltage difference between u and v when a unit of current is injected into u and is extracted from v. We have:

$$R_{uv} = \mathbf{b}_{uv}^\top L^+ \mathbf{b}_{uv}. \tag{2.2}$$

For convenience, we let $R_{uu} := 0$ for all u. For any matrix $K \in \mathbb{R}^{n\times n}$, we let $R^K_{uv} := \mathbf{b}_{uv}^\top K^+ \mathbf{b}_{uv}$.

For any pair of vertices (x, y), the potential difference induced on this pair when sending a unit of flow from u to v can be calculated as:

$$\varphi(x) - \varphi(y) = \mathbf{b}_{xy}^\top L^+ \mathbf{b}_{uv}. \tag{2.3}$$

Furthermore, if the graph is unweighted, the flow on edge (x, y) is

$$\mathbf{f}_{uv}(xy) = \mathbf{b}_{xy}^\top L^+ \mathbf{b}_{uv}. \tag{2.4}$$

We frequently use the following simple fact.

FACT 2.2. (SEE E.G. [KLM+17], LEMMA 3) *For any $\gamma > 0$, a vertex set V and any Laplacian matrix $L \in \mathbb{R}^{|V|\times|V|}$, let $K = L + \gamma I$. Then, for any pair of vertices $(u, v), (x, y) \in V \times V$,*

$$|\mathbf{b}_{xy}^\top K^+ \mathbf{b}_{uv}| \le \mathbf{b}_{uv}^\top K^+ \mathbf{b}_{uv}.$$

Proof. Let $\varphi = K^+ \mathbf{b}_{uv}$. Suppose that for some $x \in V \setminus \{u\}$, $\varphi(x) > \varphi(u)$. Then, since $K = L + \gamma I$ is a full rank and diagonally dominant matrix, one can easily see that we should have $\mathbf{b}_{uv}(x) > 0$, which is a contradiction. So, $\varphi(u) \ge \varphi(x)$ for any $x \in V \setminus \{u\}$. In a similar way, we can argue that $\varphi(v) \le \varphi(y)$ for any $y \in V \setminus \{v\}$. So, the claim holds. ∎

Spectral Approximation. For matrices $C, D \in \mathbb{R}^{p\times p}$, we write $C \preceq D$, if $\forall x \in \mathbb{R}^p$, $x^\top C x \le x^\top D x$. We say that $\widetilde{C}$ is $(1\pm\epsilon)$–spectral sparsifier of C, and we write it as $\widetilde{C} \approx_\epsilon C$, if $(1-\epsilon)C \preceq \widetilde{C} \preceq (1+\epsilon)C$. Graph $\widetilde{G}$ is $(1\pm\epsilon)$–spectral sparsifier of graph G if, $L_{\widetilde{G}} \approx_\epsilon L_G$. We sometimes use a slightly weaker notation $(1-\epsilon)C \preceq_r \widetilde{C} \preceq_r (1+\epsilon)C$, to indicate that $(1-\epsilon)x^\top C x \le x^\top \widetilde{C} x \le (1+\epsilon)x^\top C x$, for any x in the row span of C.

3 Main result

We start by giving intuition and presenting the high level idea of our algorithm in Section 3.1 below. In Section 3.3 we formally state the algorithm and prove its correctness. In Section 3.4 we describe how the required sketches can be implemented using an efficient pseudorandom number generator. Finally in Section 3.5 we give the proof of Theorem 1.1.

3.1 Overview of the approach.

To illustrate our approach, consider the goal of finding all "heavy edges" – i.e., edges with effective resistance at least $\frac{1}{\log n}$ in a graph $G = (V, E)$. This task was studied in prior work [KLM+17]

and shown to be sufficient for solving the spectral sparsification problem. That work recovers heavy edges by running ℓ_2-heavy hitters on approximate flow vectors, computed using a coarse sparsifier of the graph. The number of test flow vectors used in [KLM+17] is quadratic in the number of vertices: they brute force on all pair of vertices to find the heavy edges. We start by asking:

> Can we find a near-linear number of test flow vectors that enable us to recover all heavy edges?

In this work, we answer this question in the affirmative and formally show that there exist a linear number of test flow vectors that suffice to find all heavy edges. This is the key technical contribution of this paper and generalizing this solution yields our main algorithmic results.

To illustrate our approach, suppose that one can compute a flow vector using the formula[4]

$$BL^{+}\mathbf{b}_{uv} = \mathbf{f}_{uv} \tag{3.5}$$

for any pair of vertices in polylogarithmic time (in our actual algorithms we will be unable to compute these flow vectors exactly). Note that

$$||\mathbf{f}_{uv}||_2^2 = \mathbf{b}_{uv}^{\top}L^{+}B^{\top}BL^{+}\mathbf{b}_{uv} = \mathbf{b}_{uv}^{\top}L^{+}\mathbf{b}_{uv} = R_{uv} \tag{3.6}$$

and

$$\mathbf{f}_{uv}(uv) = \mathbf{b}_{uv}^{\top}L^{+}\mathbf{b}_{uv} = R_{uv}. \tag{3.7}$$

This implies that, when $R_{uv} > \frac{1}{\log n}$, the contribution of uv coordinate of the flow vector to its ℓ_2 norm is substantial, and known ℓ_2-heavy hitters techniques can recover this edge from a polylog n sized linear sketch efficiently. In $O(n \operatorname{polylog} n)$ space we can store a ℓ_2-heavy hitters sketch of the full vertex edge incidence matrix B and by linearity can reconstruct a heavy-hitters sketch of any electrical flow vector. Testing all $O(n^2)$ flows $\mathbf{f}_{uv}$ was the main technique of [KLM+17].

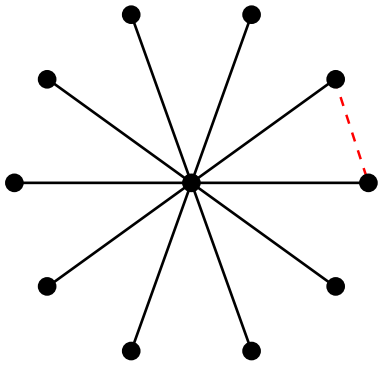
Since ℓ_2-heavy hitters in fact returns the set of all edges with a $\Omega\left(\frac{1}{\operatorname{polylog} n}\right)$ contribution to the ℓ_2 norm of the flow vector, a natural question is whether it is possible to recover a heavy edge without using its flow vector, but rather using other flow vectors. In this way we can test a smaller number of flow vectors overall. Consider the following example.

Example 3.1. (Star Graph Plus Edge) *Suppose that graph $G = (V, E)$ is a "star" with a center and n petals along with one additional edge that connects a pair of petals, i.e., $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{(v_1, v_2), (v_1, v_3), \cdot, (v_1, v_n)\} \cup \{(v_2, v_3)\}$ (see Figure 1a).*

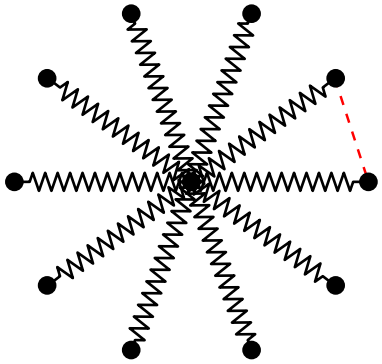
Note that for edge (v_2, v_3), $R_{v_2v_3} = \frac{2}{3}$. Suppose that we want to recover this edge by examining an electrical flow vector other than $\mathbf{f}_{v_2v_3}$. We can in fact pick an arbitrary vertex $x \in V \setminus v_2$ and send one unit of flow to v_2. Regardless of the choice of x, edge (v_2, v_3) contributes an $\Omega(1)$ fraction of the energy of the flow, and thus can be recovered by applying heavy hitters to $\mathbf{f}_{xv_2}$. Similarly, for any v_i, when one unit of flow is sent from x to v_i, at least a constant fraction of the energy is contributed by edge (x, v_i). So, all high effective resistance edges in this graph (all edges) can be recovered using $n - 1$ simple flow vectors, i.e., $\{\mathbf{f}_{xv_1}, \ldots, \mathbf{f}_{xv_n}\}$.

Of course, the graph in Example 3.1 has only n edges, and so could be stored explicitly in the streaming setting, without needing to recover edges from heavy hitter queries. However, we can give a similar example which is in fact dense.

[4] Note that in the actual algorithm we will use a sparsifier of the regularized graph L_ℓ when approximating flows in the less regularized graph $L_{\ell+1}$. See Remark 2.1. We use L in this overview of our techniques to simplify notation.



(a) Graph of Example 3.1



(b) Graph of Example 3.2

Figure 1: (a) Graph of Example 3.1. A star with n petals along with one additional edge. (b) Graph of Example 3.2. A star graph with $\Theta(n^{0.7})$ petals, along with one additional edge. Each zigzag represents a path of connected cliques with effective resistance diameter $O(1)$.

EXAMPLE 3.2. (THICK STAR PLUS EDGE) *Suppose that graph G is a dense version of the previous example as follows: it has a center and $\Theta(n^{0.7})$ petals. Each petal consists of a chain of $\Theta(n^{0.2})$ cliques of size $n^{0.1}$, where each pair of consecutive cliques is connected with a complete bipartite graph. One can verify that the effective resistance diameter of each petal is $\Theta(1)$. Now, we add an additional edge, e, that connects an arbitrary node in the leaf of one petal to a node in the leaf of another petal (see Figure 1b).*

As in Example 3.1, e is heavy, with $R_e = \Theta(1)$. In fact, it is the only heavy edge in the graph. One can verify that, similar to Example 3.1, if we let C_2 and C_3 denote the cliques that e connects, choosing an arbitrary vertex x and sending flow to any node in C_2 and then to any node in C_3, will give an electrical flow vector where e contributes an $\Omega(1)$ fraction of the energy. Thus, e can be recovered by applying heavy hitters to these vectors. Consequently, using n test flow vectors (sending flow from x to each other node in the graph) one can recover all heavy edges of this example.

Unfortunately, it is possible to give an example where the above simple procedure of checking the flow from an arbitrary vertex to all others fails.

EXAMPLE 3.3. (THICK LINE PLUS EDGE) *Suppose that graph $G = (V, E)$ is a thick line, consisting of $n^{0.9}$ set of points (clusters) where any two consecutive clusters form a complete bipartite graph. Formally, $V = \{v_1, v_2, \ldots, v_n\} = C_1 \cup C_2 \cup \cdots \cup C_{n^{0.9}}$, where C_i's are disjoint sets of size $n^{0.1}$ and*

$$E = \bigcup_{i=1}^{n^{0.9}-1} C_i \times C_{i+1}.$$

Also, add an edge $e = (u, v)$ such that $u \in C_1$ and $v \in C_{n^{0.2}}$ (see Figure 2).

One can verify that $R_e = \Omega(1)$. However, if one picks an arbitrary vertex $x \in V$ and sends one unit of flow each other vertex, running ℓ_2-heavy hitters on each of these flows will not recover edge e if x is far from u and v in the thick path. Any flow that must cross (u, v) will have very large energy due to the fact that it must travel a long distance to the clusters containing these vertices, so e will not contribute a non-trivial fraction.

Fortunately, the failure of our recovery method in Example 3.3 is due to a simple fact: the effective resistance diameter of the graph is large. When the effective resistance diameter is small (as in Examples 3.1 and 3.2) the strategy always suffices. This follows from the following simple observation:

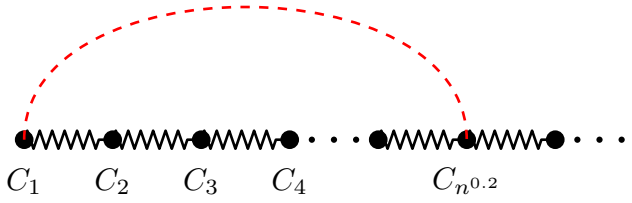


Figure 2: Graph of Example 3.3. Each C_i represents a cluster with $n^{0.1}$ vertices (with no internal edges) and each zigzag represents the edges of a complete bipartite graph between consecutive C_i's.

OBSERVATION 3.1. *For a graph $G = (V, E)$, suppose that for an edge $e = (u, v) \in E$, one has $R_e \geq \beta$. Then, for any $x \in V$, in at least one of these settings, edge e carries at least $\beta/2$ units of flow: either* **(1)** *one unit of flow is sent from x to u or* **(2)** *one unit of flow is sent from x to v.*

This observation follows formally from the following simple lemma.

LEMMA 3.1. *Consider a vertex set V and any PSD matrix $D \in \mathbb{R}^{|V|\times|V|}$. For any pair of vertices $(u, v) \in V \times V$ and any vertex $x \in V \setminus \{u, v\}$, $\max\{|\mathbf{b}_{xu}^\top D\mathbf{b}_{uv}|, |\mathbf{b}_{xv}^\top D\mathbf{b}_{uv}|\} \geq \frac{\mathbf{b}_{uv}^\top D\mathbf{b}_{uv}}{2}$.*

Proof. Note that $\mathbf{b}_{uv}^\top D\mathbf{b}_{uv} = \left(\mathbf{b}_{ux}^\top + \mathbf{b}_{xv}^\top\right) D\mathbf{b}_{uv} = \mathbf{b}_{ux}^\top D\mathbf{b}_{uv} + \mathbf{b}_{xv}^\top D\mathbf{b}_{uv}$, and hence, the claim holds. ∎

Consider the setting where $\beta = \frac{1}{\log n}$. The observation guarantees that edge e contributes at least $\frac{1}{4\log^2 n}$ energy to either flow $\mathbf{f}_{xv}$ or $\mathbf{f}_{xu}$. Thus, we can recover this edge via ℓ_2 heavy hitters, as long as the total energy $\|\mathbf{f}_{xv}\|_2^2$ or $\|\mathbf{f}_{xu}\|_2^2$ is not too large. Note that this energy is just equal to the effective resistance R_{xv} between x and v (respectively x and u). Thus it is bounded if the effective resistance diameter is small, demonstrating that our simple recovery procedure always succeeds in this setting. For example, if the diameter is $O(1)$, both $\|\mathbf{f}_{xv}\|_2^2 = O(1)$ and $\|\mathbf{f}_{xu}\|_2^2 = O(1)$, and so by Observation 3.1, edge $e = (u, v)$ contributes at least a $\Theta\left(\frac{1}{\log^2 n}\right)$ fraction of the energy of at least one of these flows.

We next explain how to extend this procedure to handle general graphs, like that of Example 3.3.

Ball carving in effective resistance metric. When the effective resistance diameter of G is large, if we attempt to recover e using ℓ_2-heavy hitters on the flow vectors $\mathbf{f}_{xu}$ and $\mathbf{f}_{xv}$, for an arbitrary chosen $x \in V$, we may fail if the effective resistance distance between x and v or u ($\|\mathbf{f}_{xv}\|_2^2$ or $\|\mathbf{f}_{xu}\|_2^2$) is large. This is exactly what we saw in Example 3.3.

However, using the fact that $||\mathbf{f}_{xv}||_2^2 = R_{xv}$, our test will succeed if we find a vertex x, which is close to u and v in the effective resistance metric. This suggests that we should partition the vertices into cells of fairly small effective resistance diameter, ensuring that both endpoints of an edge (u, v) that we would like to recover fall in the same cell with nontrivially large probability. This is exactly what standard metric decomposition techniques achieve through a ball-carving approach, which we use, as described next.

Partitioning the graph into low effective resistance diameter sets. We first embed the vertices of the graph into Euclidian space such that the Euclidian distance squared between two vertices corresponds to the effective resistance between those vertices. This can be done using known techniques. Indeed, recall that

$$R_{uv} = ||BL^+ b_{uv}||_2^2 = ||BL^+\chi_u - BL^+\chi_v||_2^2.$$

Therefore, the embedding $v \mapsto BL^+\chi_v$ suffices. This embedding is into $\mathbb{R}^{\binom{n}{2}}$, however, using Johnson-Lindenstrauss (JL) dimension reduction, one can instead embed the vertices into $\mathbb{R}^q$, for $q = O(\log n)$, such that the Euclidean distances squared correspond to constant factor multiplicative approximations to the effective resistances.

We then partition $\mathbb{R}^q$ into ℓ_∞ balls centered at points of a randomly shifted infinite q-dimensional grid with side length $w > 0$, essentially defining a hash function that maps every point in $\mathbb{R}^q$ to the nearest point on the randomly shifted grid. We can bound the maximum effective resistance of pair of vertices in the same cell (see Claim 3.2), and show how an appropriate choice of the width w ensures that any u and v that have an edge between them (and thus are reasonably close in the effective resistance metric) belong to the same cell, with a probability no less than $1/2$ (see Claim 3.1). This ensures that in at least one of $O(\log n)$ independent repetitions of this process with high probability, u and v fall into the same cell. We note that the parameters of our partitioning scheme can be improved somewhat using Locality Sensitive Hashing techniques (e.g., [IM98, DIIM04, AI06, AINR14, AR15]). More precisely, LSH techniques would improve the space complexity by polylogarithmic factors at the expense of slightly higher runtime (the best improvement in space complexity would result from Euclidean LSH [AI06, AR15], at the cost of an $n^{o(1)}$ additional factor in runtime). However, since the resulting space complexity does not quite match the lower bound of $\Omega(n \log^3 n)$ due to [NY19], we leave the problem of fine-tuning the parameters of the space partitioning scheme as an exciting direction for further work.

Sampling edges with probability proportional to effective resistances. The above techniques can actually be extended to recover edges of any specific target effective resistance. Broadly speaking, if we aim to capture edges of effective resistance about R, we can afford to lower our grid cell size proportionally to R. Unfortunately, these edges don't contribute enough to the flow vector to be recoverable. Thus, we will also subsample the edges of the graph at rate approximately proportional to R to allow us to detect the target edges while also subsampling them.

3.2 Reducing the cost of randomness.

Beyond the new algorithmic ideas discussed above, obtaining a near linear time recovery procedure for streaming graph sparsification requires solving a second mostly orthogonal problem: we need a faster way to generate pseudorandom bits for use in our randomized sketching algorithms.

Nisan's pseudorandom number generator. Like most streaming algorithms, our methods depend heavily on randomness. We compute sketches of the form $\Pi \cdot B$, where Π is a *randomly constructed matrix* with $\binom{n}{2}$ columns and $s = \text{polylog}(n)$ rows. Naively, storing Π after random initialization would take $\Omega(n^2)$ space, dominating the space complexity of our algorithms. Accordingly, to obtain truly space efficient methods, we need to find a more compact way of representing the random matrix Π. This is not a challenge unique to graph sketching – essentially all linear sketching algorithms require efficient ways of representing the sketch matrix Π.

Amongst several techniques for doing so (e.g., many algorithms build Π using low-independence hash functions), one powerful approach is to generate Π using a *pseudorandom number generator* (PRG) with a small seed. Indyk first applied this idea for streaming estimation of vector norms [Ind06]. He showed that any PRG than can "fool" a small space algorithm can also fool any linear sketching algorithm with a small sketch size (i.e., with few rows in Π).

Instantiating Indyk's result with Nisan's well known PRG [Nis92] allows Π to be generated from a seed of just $\tilde{O}(N \log R)$ random bits, as long as $\Pi \cdot x$, or in our case, $\Pi \cdot B$ can be stored in N space and Π can be generated from R random bits. Instead of storing Π, the streaming algorithm just needs to store this small random seed and any entry of Π can be generated "on-the-fly" as needed. This is a powerful result: since Indyk's original application, Nisan's generator has become a central tool in streaming algorithm de-

sign.

However, when runtime is a concern, Nisan's PRG is a costly option for graph streaming algorithms. If R random bits are required to generate Π, Nisan's generator requires $O(N \log R)$ time to generate even a single random bit from its $O(N \log R)$ length seed. In our setting R is polynomial in N, but upwards of $O(n)$ random bits need to be accessed during our sparsifier recovery algorithm. Generating these bits on-the-fly would immediately imply an $\Omega(nN) \geq \Omega(n^2)$ runtime bottleneck.

A faster pseudorandom generator. To deal with the cost of generating Π in a pseudorandom way, in Section 4 we present a pseudorandom generator that is much faster than Nisan's. In particular, we show that, at least when R is polynomial in N, it is possible to construct a generator that can produce any pseudorandom output bit in $\operatorname{polylog} N$ time. At the same time, our generator still uses a seed of just $O(N \operatorname{polylog} N)$ random bits (only a $\operatorname{polylog} N$ factor more than Nisan's).

To understand how to achieve a faster generator, notice why Nisan's pseudorandom generator requires $O(N \log N)$ time per output bit: every output bit in Nisan's construction depends on *every seed bit* in its $O(N \log N)$ length seed. To avoid this cost, we need a generator that is inherently *local*, with each pseudorandom bit only depending on $\operatorname{polylog} N$ seed bits.

While "locally computable" pseudorandom number generators have not been studied directly, there do exist locally computable constructions of *extractors*, a closely related object [Vad04, Lu02, DPVR12]. The goal of an extractor is to extract a small string of nearly uniform random bits from a long stream of *weakly* random bits. In certain cryptographic settings, it is desirable to do so in a way that only bases each output bit on a relatively small number of input bits.

Furthermore, it is actually possible to construct a pseudorandom number generator using an algorithm for randomness extraction. In particular, by plugging a locally computable extractor from [DV10] into a pseudorandom generator of Nisan and Zuckerman [NZ96], we obtain a generator that can compute each pseudorandom bit using just $O(\operatorname{polylog} N)$ pseudorandom bits. Naively, this construction can output up to N^2 pseudorandom bits using a seed of $\tilde{O}(N)$. We describe an iterative process which further exands the output to N^c pseudorandom bits, while still maintaining a generation time of $\operatorname{polylog} N$ whenever c is constant.

There are likely many possible improvements to our basic construction. We hope that bringing a broader set of tools from the pseudorandomness literature to the streaming algorithms community, we can initiate an exploration of these improvements, which could lead to fast linear sketching.

3.3 Our algorithm and proof of main result.

As discussed, our algorithm consists of two phases. In the first, the algorithm maintains sketches of the stream (in particular the vertex edge incidence matrix B), updating the sketches at each edge addition or deletion. Then, in the second phase, the algorithm recovers a spectral sparsifier of the graph from these sketches. In the following lines, we give a brief overview of each phase:

Updating sketches in the dynamic stream. Our algorithm maintains a set of sketches ΠB, of size $O(n \operatorname{polylog}(n) \cdot \epsilon^{-2})$, and updates them each time it receives an edge insertion or deletion in the stream. ΠB consists of multiple sketches $(\Pi_s^\ell B_s^\ell)_{\ell,s}$ where B_s^ℓ is a subsampling of the edges in B at rate 2^{-s} and Π_s^ℓ is an ℓ_2 heavy hitters sketch. In Section 3.4 we discuss these sketches in more detail and we show that the update time for each edge insertion or deletion is $O(\operatorname{polylog}(n) \cdot \epsilon^{-2})$.

Recursive sparsification. After receiving the updates in the form of a dynamic stream, our algorithm uses the maintained sketches to recover

a spectral sparsifier of the graph. This is done recursively, and relies on the idea of a chain of coarse sparsifiers described in Remark 2.1 and Lemma A.1. For a regularization parameter ℓ between 0 and $d = O(\log n)$ the task of SPARSIFY$(\Pi^{\leq \ell} B, \ell, \epsilon)$ is to output a spectral sparsifier to matrix L_ℓ, which was defined in Def. 2.1 as:

$$L_\ell = \begin{cases} L_G + \frac{\lambda_u}{2^\ell} I & \text{if } 0 \leq \ell \leq d \\ L_G & \text{if } \ell = d+1, \end{cases}$$

where $d = \lceil \log_2 \frac{\lambda_u}{\lambda_\ell} \rceil$ (see Lemma A.1 for more details about the chain of coarse sparsifiers).

Note that the call receives a collection of sketches $\Pi^{\leq \ell} B$ as input that suffices for all recursive calls with smaller values of ℓ. So, to compute a sparsifier of the graph we invoke SPARSIFY$(\Pi^{\leq d+1} B, d+1, \epsilon)$, which receives all the sketches maintained throughout the stream and passes the required sketches to the recursive calls in line 6 of Algorithm 1. The algorithm first invokes itself recursively to recover $\tilde{K}$, a spectral approximation for $L_{\ell-1}$ (or uses the trivial approximation $\lambda_u I$ when $\ell = 0$). The effective resistance metric induced by $\tilde{K}$ is then approximated using the Johnson-Lindenstrauss lemma (JL). Finally, the procedure RECOVEREDGES (i.e. Algorithm 2) uses this metric and the heavy hitters sketches $(\Pi_s^\ell B_s^\ell)_s$ to recover a sparsifier for $L_{\ell-1}$ using the techniques described in Section 3.1. We formally state our algorithm, Algorithm 1 below.

Algorithm 2 (the RECOVEREDGES primitive) is the core of Algorithm 1. It receives a parameter s as input, and its task is to recover edges with effective resistance $\approx \frac{\epsilon^2}{\log n} 2^{-s}$ from a sample at rate $\min(1, O(2^{-s}))$ from an appropriate sketch. It is convenient to let s range from $-O(\log(\log n/\epsilon^2))$ to $O(\log n)$, so that the smallest value of s corresponds to edges of constant effective resistance. That way the sampling level corresponding to s is simply equal to $s^+ := \max(0, s)$. Therefore, Algorithm 2 takes as input a heavy hitters sketch $\Pi_{s^+}^\ell B_{s^+}^\ell$ of $B_{s^+}^\ell$, the edge incidence matrix of L_ℓ sampled at rate 2^{-s^+}, an approximate effective resistance embedding M, the target sampling probability 2^{-s}, the dimension q of the embedding, and the target accuracy ϵ. This procedure then performs the previously described random grid hashing of the points using the effective resistance embedding and queries the heavy hitters sketch to find the edges sampled at the appropriate rate.

Algorithm 1 SPARSIFY$(\Pi^{\leq \ell} B, \ell, \epsilon)$

1: **procedure** SPARSIFY$(\Pi^{\leq \ell} B, \ell, \epsilon)$
2: $W \leftarrow 0^{n \times n}$
3: **if** $\ell = 0$ **then**
4: $\widetilde{K} \leftarrow \lambda_u I$
5: **else**
6: $\widetilde{K} \leftarrow \frac{1/2}{1+\epsilon}$SPARSIFY$(\Pi^{\leq \ell - 1} B, \ell - 1, \epsilon)$
7: $\widetilde{B} \leftarrow$ the edge vertex incident matrix of $\widetilde{K}$
8: $\widetilde{W} \leftarrow$ diagonal weightmatrix of edges of $\widetilde{K}$
9: $Q \leftarrow q \times \left(\binom{n}{2} + n\right)$ is a random ± 1 matrix
10: for $q \leftarrow 1000 \log n$
11: $M \leftarrow \frac{1}{\sqrt{q}} Q \widetilde{W}^{1/2} \widetilde{B} \widetilde{K}^+$ ▷ *M is such that*
12: $R_{uv}^{\widetilde{K}} \leq \frac{5}{4} ||M(\chi_u - \chi_v)||_2^2 \leq \frac{3}{2} R_{uv}^{\widetilde{K}}$
13: **for** $s \in [-\log\left(3c_2 \log n/\epsilon^2\right), 10 \log n]$ **do**
14: $E_s \leftarrow$ RECOVEREDGES$(\Pi_{s^+}^\ell B_{s^+}^\ell, M,$
15: $\widetilde{K}^+, s, q, \epsilon)$
16: ▷ *We use $s^+ = max(0, s)$*
17: **for** $e \in E_s$ **do** $W(e,e) \leftarrow 2^{-(s^+)}$
18: **if** $\ell = \left\lceil \log_2 \frac{\lambda_u}{\lambda_\ell} \right\rceil + 1$ **then** $\gamma \leftarrow 0$
19: **else** $\gamma \leftarrow \frac{\lambda_u}{2^\ell}$
20: **return** $B_n^\top W B_n + \gamma I$.

The development and analysis of RECOVEREDGES (Algorithm 2) is the main technical contribution of our paper. In the rest of the section we prove the correctness this algorithm (Lemma 3.2, our main technical lemma), and then provide a correctness proof for Algorithm 1, establishing Theorem 3.1. We put these results together with runtime and space complexity bounds to prove Theorem 1.1.

Specifically, Lemma 3.2 proves that if Algorithm 1 successfully executes all lines before

line 13, then each edge is sampled and weighted properly (as required by Theorem A.1), in the remaining steps.

LEMMA 3.2. (EDGE RECOVERY) *Consider an invocation of* RECOVEREDGES$(\Pi^{\ell}_{s^+}B^{\ell}_{s^+}, M, \widetilde{K}, s, q, \epsilon)$ *of Algorithm 2, where* $\Pi^{\ell}_{s^+}B^{\ell}_{s^+}$ *is a sketch of the edge incidence matrix* B *of the input graph* G *as described in Section 3.4,* s *is some integer, and* $\epsilon \in (0, 1/5)$*. Suppose further that* $\widetilde{K}$ *and* M *satisfy the following guarantees:*

(A) $\widetilde{K}$ *is such that* $\frac{1}{3} \cdot L_\ell \preceq_r \widetilde{K} \preceq_r L_\ell$ *(see lines 4 and 6 of Algorithm 1)*

(B) M *is such that for any pair of vertices* u *and* v, $R^{\widetilde{K}}_{uv} \leq \frac{5}{4}||M(\chi_u - \chi_v)||_2^2 \leq \frac{3}{2}R^{\widetilde{K}}_{uv}$ *($R^{\widetilde{K}}$ is the effective resistance metric in $\widetilde{K}$; see line 12 of Algorithm 1)*

Then, with high probability, for every edge e, RECOVEREDGES$(\Pi^{\ell}_{s^+}B^{\ell}_{s^+}, M, \widetilde{K}, s, q, \epsilon)$ *will recover* e *if and only if:*

(1) $\frac{5}{4} \cdot c_2 \cdot ||Mb_e||_2^2 \cdot \log(n)/\epsilon^2 \in (2^{-s-1}, 2^{-s}]$ *where* c_2 *is the oversampling constant of Theorem A.1 (see lines 23 and 24 of Algorithm 2), and*

(2) *edge* e *is sampled in* $B^{\ell}_{s^+}$.

The proof of Lemma 3.2 relies on the following two claims regarding the hashing scheme of Algorithm 2. First, Claim 3.1 shows that the endpoints of an edge of effective resistance bounded by a threshold most likely get mapped to the same grid point in the random hashing step in line 12 of Algorithm 2.

CLAIM 3.1. (HASH COLLISION PROBABILITY) *Let* q *be a positive integer and let the function* $\mathcal{G} : \mathbb{R}^q \to \mathbb{Z}^q$ *define a hashing with width* $w > 0$ *as follows:* $\forall i \in [q]$, $\mathcal{G}(u)_i = \lfloor \frac{u_i - s_i}{w} \rfloor$, *where* $s_i \sim \text{Unif}[0, w]$, *as per line 12 of Algorithm 2. If for a pair of points* $x, y \in \mathbb{R}^q$, $||x - y||_2 \leq w_0$ *and* $w \geq 2w_0 q$, *then* $\mathcal{G}(x) = \mathcal{G}(y)$ *with probability at least* $1/2$.

Proof of Claim 3.1 is deferred to [KNST19].

Algorithm 2 RECOVEREDGES$(\Pi^{\ell}_{s^+}B^{\ell}_{s^+}, M, \widetilde{K}^+, s, q, \epsilon)$

1: **procedure** RECOVEREDGES$(\Pi^{\ell}_{s^+}B^{\ell}_{s^+}, M, \widetilde{K}^+, s, q, \epsilon)$
3: $E' \leftarrow \emptyset$.
4: $C \leftarrow$ is as in the proof of Lemma 3.2
5: $c_2 \leftarrow$ the constant of Theorem A.1
6: $w \leftarrow 2q \cdot \sqrt{\frac{\epsilon^2}{c_2 \cdot 2^s \cdot \log n}}$.
7: **for** $j \in [10 \log n]$ **do**
8: $\forall i \in [q]$, choose $s_i \sim \text{Unif}([0, w])$.
9: $H \leftarrow \emptyset$ (an empty hash table)
10: **for** $u \in V$ **do** ▷ *Hash vertices to points*
11: *on randomly shifted grid*
12: $\forall i \in [q]$, $\mathcal{G}(u)_i \leftarrow \left\lfloor \frac{(M\chi_u)_i - s_i}{w} \right\rfloor$.
13: Insert u into H with key $\mathcal{G}(u)$. ▷ $\mathcal{G}(u)$
14: *indexes a point on a randomly shifted grid*
15: **for** $b \in \text{keys}(H)$ **do**
16: $x \leftarrow$arbitrary vertex in $H^{-1}(b)$
17: **for** $v \in H^{-1}(b) \setminus \{x\}$ **do**
18: $F \leftarrow$ HEAVYHITTER $\Big(\Pi^{\ell}_{s^+}B^{\ell}_{s^+}\widetilde{K}^+\mathbf{b}_{xv},$
19:
20: $\frac{1}{2} \cdot \frac{1}{C \cdot q^3} \cdot \sqrt{\frac{\epsilon^2}{\log n}}\Big)$.
21: ▷ *As per Lemma A.2*
22: **for** $e \in F$ **do**
23: $p'_e \leftarrow \frac{5}{4} \cdot c_2 \cdot ||Mb_e||_2^2 \cdot \log n/\epsilon^2$
24: **if** $p'_e \in (2^{-s-1}, 2^{-s}]$ **then**
25: $E' \leftarrow E' \cup \{e\}$.
26: **return** E'.

The next claim, Claim 3.2 bounds the effective resistance diameter of buckets in the hash table constructed in line 13 of Algorithm 2.

CLAIM 3.2. (HASH BUCKET DIAMETER) *Let the function* $\mathcal{G} : \mathbb{R}^q \to \mathbb{Z}^q$, *for some integer* q, *define a hashing with width* $w > 0$ *as follows:* $\forall i \in [q]$, $\mathcal{G}(u)_i = \lfloor \frac{u_i - s_i}{w} \rfloor$, *where* $s_i \sim \text{Unif}[0, w]$, *as per line 12 of Algorithm 2. For any pair of*

points $u, v \in \mathbb{R}^q$, such that $\mathcal{G}(u) = \mathcal{G}(v)$, one has $||u - v||_2 \leq w \cdot \sqrt{q}$.

Using Claim 3.1 and Claim 3.2 one can prove Lemma 3.2, however we defer the proof to [KNST19].

THEOREM 3.1. *(Correctness of Algorithm 1) Algorithm* SPARSIFY$(\Pi^{\leq \ell} B, \ell, \epsilon)$, *for* $\ell = d + 1 = \lceil \log_2 \frac{\lambda_u}{\lambda_\ell} \rceil + 1$ *(see Lemma A.1), any* $\epsilon \in (0, 1/5)$ *and sketches* $\Pi^{\leq \ell} B$ *of graph* G *as described in Section 3.4, returns a graph* H *with* $O(n \cdot \text{polylog}\, n \cdot \epsilon^{-2})$ *weighted edges, with Laplacian matrix* L_H, *such that* $L_H \approx_\epsilon L_G$, *with high probability.*

The proof of this theorem is deferred to [KNST19].

3.4 Maintenance of sketches. Note that Algorithm 2 takes sketch ΠB as input. More precisely, Π is a concatenation of HEAVYHITTER sketch matrices composed with sampling matrices, indexed by sampling rate s and regularization level ℓ. In particular, for all s and ℓ let B_s^ℓ be a row-sampled version of B at rate 2^{-s}. Then Π_s^ℓ is a HEAVYHITTER sketch drawn from the distribution from Lemma A.2 with parameter $\eta = \frac{1}{2} \cdot \frac{1}{C \cdot q^3} \cdot \sqrt{\frac{\epsilon^2}{\log n}}$. Note that the matrices $\left(\Pi_s^\ell\right)_{s,\ell}$ are independent and identically distributed. We then maintain $\Pi_s^\ell B_s^\ell$ for all s and ℓ. We define

$$\Pi^\ell B = \Pi_0^\ell B_0^\ell \oplus \ldots \oplus \Pi_{10 \log n}^\ell B_{10 \log n}^\ell,$$

where $\oplus$ denotes concatenation of rows. We let $\Pi^{\leq \ell}$ denote $\Pi^0 \oplus \ldots \oplus \Pi^\ell$, and let Π denote $\Pi^{\leq d+1}$ to simplify notation. Thus, the algorithm maintains ΠB throughout the stream. We maintain ΠB by maintaining each $\Pi_s^\ell B_s^\ell$ individually. To this end we have for each s and ℓ an independent hash function h_s^ℓ mapping $\binom{V}{2}$ to $\{0, 1\}$ independently such that $\mathbb{P}(h_s^\ell(u, v) = 1) = 2^{-s}$. Then when an edge insertion or deletion, $\pm(u, v)$, arrives in the stream, we update $\Pi_s^\ell B_s^\ell$ by $\pm \Pi_s^\ell \cdot b_{uv} \cdot h_s^\ell(u, v)$.

Overall, the number of random bits needed for all the matrices in an invocation of Algorithm 2 is at most $R = \widetilde{O}(n^2)$, in addition to the random bits needed for the recursive calls. To generate matrix Π we use the fast pseudo random numbers generator from Theorem 4.1.

Observe that the space used by Algorithm 2 is $s = \widetilde{O}(n)$ in addition to the space used by the recursive calls. Since $R = O(n^2)$, we have $R = O(s^2)$. Therefore, by Theorem 4.1 we can generate seed of $O(s \cdot \text{poly}(\log s))$ random bits in $O(s \cdot \text{poly}(\log s))$ time that can simulate our randomized algorithm.

Also, note that the random matrix $Q \in \mathbb{R}^{\Theta(\log n) \times \binom{n}{2}}$ for JL (line 9 of Algorithm 2) can be generated using $O(\log n)$-wise independent hash functions.

3.5 Proof of Theorem 1.1 Correctness of Algorithm 2 is proved in Theorem 3.1. It remains to prove space and runtime bounds.

Proof of Theorem 1.1:

Run-time and space analysis. We will prove that one call of SPARSIFY in Algorithm 1 requires $\widetilde{O}(n \cdot \epsilon^{-2})$ time and space, discounting the recursive call, where n is the size of the vertex set of the input graph. Consider first lines 9 and 12, and note that the random matrix $Q \in \mathbb{R}^{\Theta(\log n) \times \binom{n}{2}}$ for JL (line 9 of Algorithm 2) can be generated using $O(\log n)$-wise independent hash functions, resulting in $\text{poly}(\log n)$ time to generate an entry of Q and $O(\log n)$ space. We then multiply $Q\widetilde{W}^{1/2}\widetilde{B}$ by $\widetilde{K}^+$ which amounts to solving $\Theta(\log n)$ Laplacian systems and can be done in $O(n \,\text{polylog}\, n \cdot \epsilon^{-2})$ time, since $\widetilde{K}$ is $O(n \,\text{polylog}(n) \cdot \epsilon^{-2})$ sparse, using any of a variety of algorithms in the long line of improvements in solving Laplacian systems [ST04, KMP10, KMP11, KOSA13, LS13, PS14, CKM+14, KLP16a, KLP+16b, KS16]. The resulting matrix, M, is again $\Theta(\log n \times n)$ and can be stored in $n \,\text{polylog}\, n$ space. We note that the aforementioned Laplacian solvers provide ap-

proximate solutions with inverse polynomial precision, which is sufficient for application of the HeavyHitter sketch.

The for loops in both line 13 and line 7 iterate over only $\Theta(\log n)$ values. For all non-empty cells we iterate over all vertices in that cell, so overall we iterate n times. The HeavyHitters subroutine called with parameter $\eta = \epsilon/\operatorname{polylog} n$ returns by definition at most $O(\operatorname{polylog} n \cdot \epsilon^{-2})$ elements, so the for loop in line 22 is over $O(\operatorname{polylog} n \cdot \epsilon^{-2})$ iterations. In total this is $O(n \operatorname{polylog} n \cdot \epsilon^{-2})$ time and space as claimed.

To get an ϵ-sparsifier of the input graph G, we need only to run Sparsify$(\Pi^{\leq d+1}B, d+1, \epsilon)$. Therefore, the chain of recursive calls will be $\Theta(\log(n))$ long, and the total run time will still be $\widetilde{O}(n\epsilon^{-2})$. □

4 Faster pseudorandom numbers for sketching algorithms

Like many sketching and streaming algorithms, our algorithm results rely crucially on randomness. In particular, they use many more random bits than they have space to store. Moreover, most of these random bits are not used in a "read once" way. For example, many are used to initialize persistent random hash functions, which must access the *same set of random bits* every time a particular edge is updated in our graph sketch.

Naively, after random initialization, we need to store each of these persistent hash functions. Doing so, however, would require $\tilde{O}(n^2)$ space for a graph with n nodes, which would dominate the space complexity of our methods. To cope with this issue, we need a more compact way of representing persistent random hash functions, a challenge arising in the design of most randomized streaming algorithms, both for graph problems and other applications.

There are several techniques to deal with the issue. One approach is to prove that an algorithm can be implemented with *limited independence* hash functions, which can take exponentially fewer bits to represent than fully random hash functions [CW79]. However, proving that limited independence hashing still gives a correct algorithm can be a significant burden. For example, Indyk's well known streaming algorithm for ℓ_p norm estimation [Ind06] was only shown to work with limited independence a decade after its introduction [KNW10]. Moreover, many streaming algorithms for graph problems are not known to work with limited independence (see e.g. [AGM12b, KLM+17]) and this same challenge carry's over to a variety of other problems [RU10, BZ16, CGK16, BBC+17].

In these cases, a more powerful 'black box' technique is needed to reduce the costly requirement of "storing randomness". For algorithms based on linear sketching (like those presented in this paper) one such technique is the application of *pseudorandom number generators* [Nis92, NZ96], which have been widely used in streaming algorithms since Indyk's original application to streaming norm estimation [Ind06]. A pseudorandom number generator can obviate the need to persistently store random hash functions altogether.

4.1 Simulating small space randomized algorithms.

The goal of a pseudorandom number generator (PRG) is to deterministically generate a large string of *pseudorandom* bits from a much smaller seed of *truly random* bits. For certain algorithms, including those that use bounded memory, it is possible to show that using these pseudorandom bits instead of a full set of truly random bits leads to very little degradation in performance.

While we are not interested in reducing the total number of random bits used by our algorithms, PRGs offer an additional advantage: they can reduce the space required to store randomness when random bits need to be accessed repeatedly. In particular, we only need to store the

PRG's small random seed and can then generate pseudorandom bits "on-the-fly", as they are needed.

Towards this goal, a pseudorandom number generator designed by Nisan has been especially popular in streaming applications [Nis92]. For any algorithm that use no more than S bits of space, Nisan's PRG generates R pseudorandom bits from a seed of $O(S \log R)$ truly random bits. If these pseudorandom bits are used to simulate truly random bits, the algorithm's failure probability increases by at most 2^{-s}. Furthermore, space to store the pseudorandom bits only increases the algorithm's space complexity from S to $O(S \log R)$.

This PRG can be used to reduce the randomness requirements of any *linear sketching* algorithm[5] that 1) does not access more than S random bits on every sketch update and 2) does not use more than S space beyond what is required to store randomness. This claim is not immediate: naively, streaming algorithms that use a large number of persistent random bits do not run in small space. However, it can be proven via a reordering argument from [Ind06], which is discussed further in [KMM+19].

4.2 The computation cost of PRGs. Since our results use linear sketching, we can apply Nisan's PRG to eliminate the assumption that hash functions and other random bits are chosen truly at random. In particular, for graphs on n nodes, our algorithms use $R = \text{poly}(n)$ random bits and $S = \text{poly}(n)$ space, beyond what is required to store randomness. So Nisan's PRG allows for implementations in *total space* $O(S \log S)$.

However, in contrast to prior work on streaming spectral sparsification [KLM+17], we are also interested in the *time complexity* of our sketching methods, both in terms of update and recovery time. When runtime is a concern, Nisan's PRG provides an unsatisfying solution: for our application it is prohibitively slow.

In particular, using a seed of $O(S \log R)$ random bits, Nisan's PRG requires $O(S \log R)$ time to generate any specific pseudorandom bit. This runtime is good when S is logarithmic in the natural problem parameters. For example, for many streaming problems involving length n vectors, $S = \text{polylog}(n)$ and $R = \text{poly}(n)$. In this setting, the total generation time for Nisan's PRG is just $\text{polylog}(n)$ per bit. This is very good considering that, if $\text{poly}(n)$ random bits are stored persistently, it takes $O(\log n)$ time just to specify which bit we would like the PRG to generate.

However, our algorithms and other graph streaming algorithms, use significantly more than $\text{polylog}(n)$ space [FKM+05, McG14]. Specifically, in our setting, both S and R are polynomial in n for graphs on n nodes, so Nisan's PRG requires $O(S \log S)$ time per random bit. Since $\text{polylog}(S)$ random bits are accessed on every edge update, this leads to an update time of $\tilde{O}(S)$. We would like to reduce the PRG's cost to $\text{polylog}(S)$ time per bit, which would improve our update time to $\text{polylog}(n)$.

4.3 Main Result. The goal of this section is to demonstrate that this significant runtime improvement can be achieved using a different pseudorandom generator than Nisan's. In particular, we prove:

THEOREM 4.1. *For any constants $q, c > 0$, there is an explicit PRG that draws on a seed of $O(S \,\text{polylog}(S))$ random bits and can simulate any randomized algorithm running in space S and using $R = O(S^q)$ random bits. This PRG can output any pseudorandom bit in $O(\log^{O(q)} S)$ time and the simulated algorithm fails with probability at most S^{-c} higher than the original.*

[5] This is a broad class: all known turnstile streaming algorithms (i.e. those handling insertions and deletions) are based on linear sketching [LNW14].

The proof of this theorem is deferred to [KMM+19].

Theorem 4.1 implies a method for reducing the randomness required by a large class of linear sketching algorithms (including those presented in this paper). A formal statement appears as Theorem 13. [KMM+19]. In short, as long as the sketching algorithm uses S space and, for any update to a particular entry (i.e. an edge in our case, or a vector entry in a vector streaming algorithm) the algorithm only accesses at most S persistent random bits, then it can be simulated using a PRG with seed $O(S \operatorname{polylog} S)$. As in Theorem 4.1, this PRG can produce a single pseudorandom bit in just $O(\operatorname{polylog} S)$ time.

Ackowledgements

This work was completed while Cameron Musco was employed at Microsoft Research New England. Michael Kapralov, Navid Nouri and Jakab Tardos are supported by ERC Starting Grant 759471. Aaron Sidford is supported by NSF CAREER Award CCF-1844855.

References

[Ach03] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.

[ACK+16] Alexandr Andoni, Jiecao Chen, Robert Krauthgamer, Bo Qin, David P. Woodruff, and Qin Zhang. On sketching quadratic forms. *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 311–319, 2016.

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st Symposium on Principles of Database Systems (PODS)*, pages 5–14, 2012.

[AGM13] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 1–10. Springer, 2013.

[AHLW16] Yuqing Ai, Wei Hu, Yi Li, and David P. Woodruff. New characterizations in turnstile streams with applications. In *Proceedings of the 31st Conference on Computational Complexity*, CCC '16, pages 20:1–20:22, 2016.

[AI06] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 459–468, 2006.

[AINR14] Alexandr Andoni, Piotr Indyk, Huy L. Nguyen, and Ilya P. Razenshteyn. Beyond locality-sensitive hashing. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1018–1028, 2014.

[AKL17] Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1723–1742, 2017.

[AKLY16] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1345–1364, 2016.

[AR15] Alexandr Andoni and Ilya P. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 793–801, 2015.

[BBC+17] Jaroslaw Blasiok, Vladimir Braverman, Stephen R. Chestnut, Robert Krauthgamer, and Lin F. Yang. Streaming symmetric norms via measure concentration. In *49th Annual Symposium on Theory of Computing*, 2017.

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 173–182, 2015.

[BK96] András A. Benczúr and David R. Karger. Approximating S-T minimum cuts in $\tilde{O}(n^2)$ time. In *38th Annual Symposium on Theory of Computing*, pages 47–55, 1996.

[BSS12] Joshua Batson, Daniel A. Spielman, and Nikhil Srivastava. Twice-Ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.

[BZ16] Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In *IEEE 57th Annual Symposium on Foundations of Computer Science.*, pages 51–60, 2016.

[CGK16] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Forty-eighth Annual ACM Symposium on Theory of Computing*, pages 712–725, 2016.

[CGP+18] Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. Graph sparsification, spectral sketches, and faster resistance computation, via short cycle decompositions. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 361–372, 2018.

[CKM+14] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m\log^{1/2}n$ time. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 343–352, 2014.

[CW79] J. Lawrence Carter and Mark N. Wegman. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 253–262, 2004.

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

[DPVR12] Anandiya De, Christopher Portmann, Thomas Vidick, and Renato Renner. Trevisan's extractor in the presence of quantum side information. *SIAM Journal on Computing*, 41(4):915–940, 2012.

[DV10] Anindya De and Thomas Vidick. Near-optimal extractors against quantum storage. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 161–170, 2010.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, 2005.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 604–613, 1998.

[Ind06] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006. Preliminary version in the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2000.

[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 2017, August 16-18, 2017, Berkeley, CA, USA*, pages 14:1–14:17, 2017.

[JS18] Arun Jambulapati and Aaron Sidford. Ef-

ficient $\widetilde{O}(n/\epsilon)$ spectral sketches for the laplacian and its pseudoinverse. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2487–2503. SIAM, 2018.

[JST11] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for Lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 49–58. ACM, 2011.

[Kar94] David R. Karger. Random sampling in cut, flow, and network design problems. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 648–657, 1994.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142, 2013.

[KLM$^+$17] Michael Kapralov, Yin Tat Lee, CN Musco, CP Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. *SIAM Journal on Computing*, 46(1):456–477, 2017. Preliminary version in FOCS 2014.

[KLP16a] Ioannis Koutis, Alex Levin, and Richard Peng. Faster spectral sparsification and numerical algorithms for SDD matrices. *ACM Trans. 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 *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 842–850, 2016.

[KMM$^+$19] Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, and Navid Nouri. Faster spectral sparsification in dynamic streams. *CoRR*, abs/1903.12165, 2019.

[KMP10] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 235–244, 2010.

[KMP11] Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly-m log n time solver for SDD linear systems. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 590–598, 2011.

[KNP$^+$17] Michael Kapralov, Jelani Nelson, Jakub Pachocki, Zhengyu Wang, David P. Woodruff, and Mobin Yahyazadeh. Optimal lower bounds for universal relation, and for samplers and finding duplicates in streams. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 475–486, 2017.

[KNST19] Michael Kapralov, Navid Nouri, Aaron Sidford, and Jakab Tardos. Dynamic streaming spectral sparsification in nearly linear time and space. *CoRR*, abs/1903.12150, 2019.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. On the exact space complexity of sketching and streaming small norms. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1161–1178, 2010.

[KOSA13] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 911–920, 2013.

[KS16] Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians - fast, sparse, and simple. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 573–582, 2016.

[KW14] Michael Kapralov and David P. Woodruff. Spanners and sparsifiers in dynamic streams. *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 272–281, 2014.

[LMP13] Mu Li, Gary L. Miller, and Richard Peng. Iterative row sampling. In *54th Annual IEEE*

Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA, pages 127–136, 2013.

[LNW14] Yi Li, Huy L. Nguyen, and David P. Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 174–183, 2014.

[LS13] Yin Tat Lee and Aaron Sidford. Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 147–156, 2013.

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ(vrank) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433, 2014.

[LS17] Yin Tat Lee and He Sun. An sdp-based algorithm for linear-sized spectral sparsification. In *Proceedings of the 49th Annual Symposium on Theory of Computing*, pages 678–687, 2017.

[LS18] Huan Li and Aaron Schild. Spectral subspace sparsification. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 385–396, 2018.

[Lu02] Chi-Jen Lu. Hyper-encryption against space-bounded adversaries from on-line strong extractors. In *Advances in Cryptology-CRYTO 2002*, pages 257–271. Springer, 2002.

[McG14] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, 2014. Preliminary version in the 31st International Colloquium on Automata, Languages and Programming (ICALP), 2004.

[McG17] Andrew McGregor. Graph sketching and streaming: New approaches for analyzing massive graphs. In *Computer Science - Theory and Applications - 12th International Computer Science Symposium in Russia, CSR 2017, Kazan, Russia, June 8-12, 2017, Proceedings*, pages 20–24, 2017.

[MTVV15] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T Vu. Densest subgraph in dynamic graph streams. In *International Symposium on Mathematical Foundations of Computer Science*, pages 472–482. Springer, 2015.

[Nis92] Noam Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992. Preliminary version in the 22nd Annual ACM Symposium on Theory of Computing (STOC), 1990.

[NY19] Jelani Nelson and Huacheng Yu. Optimal lower bounds for distributed and streaming spanning forest computation. *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1844–1860, 2019.

[NZ96] Noam Nisan and David Zuckerman. Randomness is linear in space. *J. Comput. Syst. Sci.*, 52(1):43–52, 1996. Preliminary version in the 25th Annual ACM Symposium on Theory of Computing (STOC), 1993.

[PS14] Richard Peng and Daniel A. Spielman. An efficient parallel solver for SDD linear systems. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 333–342, 2014.

[RU10] Atri Rudra and Steve Uurtamo. Data stream algorithms for codeword testing. In *International Colloquium on Automata, Languages and Programming*, pages 629–640, 2010.

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

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 81–90, 2004.

[ST11] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

[ST14] Daniel A. Spielman and Shang-Hua 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.

[Vad04] Salil P. Vadhan. Constructing locally computable extractors and cryptosystems in the bounded-storage model. *J. Cryptol.*, 17(1):43–77, 2004.

[Woo14] David P. Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends in Theoretical Computer Science*, 10(1–2):1–157, 2014.

A Supplementary material

We will need Lemma A.1 that we use in the correctness proof of our algorithm.

LEMMA A.1. ([LMP13, KLM+17]) *Consider any PSD matrix K with maximum eigenvalue bounded from above by λ_u and minimum nonzero eigenvalue bounded from below by λ_ℓ. Let $d = \lceil \log_2 \frac{\lambda_u}{\lambda_\ell} \rceil$. For $\ell \in \{0, 1, 2, \ldots, d\}$, define: $\gamma(\ell) = \frac{\lambda_u}{2^\ell}$. So $\gamma(d) \leq \lambda_\ell$, and $\gamma(0) = \lambda_u$. Then the chain of PSD matrices, $[K_0, K_1, \ldots, K_d]$ with $K_\ell = K + \gamma(\ell)I$ satisfies the following relations:*

1. $K \preceq_r K_d \preceq_r 2 \cdot K$,
2. $K_\ell \preceq K_{\ell-1} \preceq 2 \cdot K_\ell$ *for all* $\ell \in \{1, \ldots, d\}$,
3. $K_0 \preceq 2 \cdot \gamma(0) \cdot I \preceq 2 \cdot K_0$.

COROLLARY A.1. *By Fact 2.1, the statements of Lemma A.1 holds when K is the Laplacian of an unweighted graph and $d = \Theta(\log n)$.*

We will need Theorem A.1 that we use in the proof of correctness of the main algorithm. It is well known that by sampling the edges of B according to their effective resistance, it is possible to obtain a weighted edge vertex incident matrix $\widetilde{B}$ such that $(1-\epsilon)B^\top B \preceq \widetilde{B}^\top \widetilde{B} \preceq (1+\epsilon)B^\top B$ with high probability (see Lemma A.1).

THEOREM A.1. ([SS11]) *Let $B \in \mathbb{R}^{\binom{n}{2} \times n}$, $K = B^\top B$, and let $\widetilde{\tau}$ be a vector of leverage score overestimates for B's rows, i.e. $\widetilde{\tau}_y \geq \mathbf{b}_y^\top K^+ \mathbf{b}_y$ for all $y \in [m]$. For $\epsilon \in (0, 1)$ and fixed constant c, define the sampling probability for row $\mathbf{b}_y$ to be $p_y = \min\{1, c \cdot \epsilon^{-2} \log n \cdot \widetilde{\tau}_y\}$. Define a diagonal sampling matrix W with $W(y, y) = \frac{1}{p_y}$ with probability p_y and $W(y, y) = 0$ otherwise. With high probability, $\widetilde{K} = B^\top W B \approx_\epsilon K$. Furthermore W has $O(||\widetilde{\tau}||_1 \cdot \epsilon^{-2} \log n)$ non-zeros with high probability.*

LEMMA A.2. (ℓ_2 HEAVY HITTERS) *For any $\eta > 0$, there is a decoding algorithm denoted by* HEAVYHITTER *and a distribution on matrices S^h in $\mathbb{R}^{O(\eta^{-2} \operatorname{polylog}(N)) \times N}$ such that, for any $x \in \mathbb{R}^N$, given $S^h x$, the algorithm* HEAVYHITTER$(S^h x, \eta)$ *returns a list $F \subseteq [N]$ such that $|F| = O(\eta^{-2} \operatorname{polylog}(N))$ with probability $1 - \frac{1}{poly(N)}$ over the choice of S^h one has*

(1) *for every $i \in [N]$ such that $|x_i| \geq \eta ||x||_2$ one has $i \in F$ and* **(2)** *for every $i \in F$ one has $|x_i| \geq (\eta/2)||x||_2$. The sketch $S^h x$ can be maintained and decoded in $O(\eta^{-2} \operatorname{polylog}(N))$ time and space.*

LEMMA A.3. (BINARY JL LEMMA [ACH03]) *Let P be an arbitrary set of points in $\mathbb{R}^d$, represented by a $d \times n$ matrix A, such that the j^{th} point is $A\chi_j$. Given ϵ, $\beta > 0$ and $q \geq \frac{4+2\beta}{\epsilon^2/2 - \epsilon^3/3} \log n$. Let Q be a random $q \times d$ matrix $(q_{ij})_{ij}$ where q_{ij}'s are independent identically distributed variables taking 1 and -1 each with probability $1/2$. Then, if $M = \frac{1}{\sqrt{q}} QA$, then with probability at least $1 - n^{-\beta}$, for all $u, v \in [n]$: $(1-\epsilon)||A\chi_u - A\chi_v||_2^2 \leq ||M\chi_u - M\chi_v||_2^2 \leq (1+\epsilon)||A\chi_u - A\chi_v||_2^2$.*