Graph Sketches: Sparsification, Spanners, and Subgraphs

Kook Jin Ahn[*]
University of Pennsylvania
kookjin@cis.upenn.edu

Sudipto Guha[*]
University of Pennsylvania
sudipto@cis.upenn.edu

Andrew McGregor[†]
University of Massachusetts
mcgregor@cs.umass.edu

ABSTRACT

When processing massive data sets, a core task is to construct *synopses* of the data. To be useful, a synopsis data structure should be easy to construct while also yielding good approximations of the relevant properties of the data set. A particularly useful class of synopses are *sketches*, i.e., those based on linear projections of the data. These are applicable in many models including various parallel, stream, and compressed sensing settings. A rich body of analytic and empirical work exists for sketching numerical data such as the frequencies of a set of entities. Our work investigates *graph sketching* where the graphs of interest encode the relationships between these entities. The main challenge is to capture this richer structure and build the necessary synopses with only linear measurements.

In this paper we consider properties of graphs including the size of the cuts, the distances between nodes, and the prevalence of dense sub-graphs. Our main result is a sketch-based sparsifier construction: we show that $\tilde{O}(n\epsilon^{-2})$ random linear projections of a graph on n nodes suffice to $(1+\epsilon)$ approximate *all* cut values. Similarly, we show that $O(\epsilon^{-2})$ linear projections suffice for (additively) approximating the fraction of induced sub-graphs that match a given pattern such as a small clique. Finally, for distance estimation we present sketch-based spanner constructions. In this last result the sketches are adaptive, i.e., the linear projections are performed in a small number of batches where each projection may be chosen dependent on the outcome of earlier sketches. All of the above results immediately give rise to data stream algorithms that also apply to dynamic graph streams where edges are both inserted and deleted. The non-adaptive sketches, such as those for sparsification and subgraphs, give us single-pass algorithms for distributed data streams with insertion and deletions. The adaptive sketches can be used to analyze MapReduce algorithms that use a small number of rounds.

[*]Supported by NSF Awards CCF-0644119 and IIS-0713267 and a gift from Google.

[†]Supported by NSF CAREER Award CCF-0953754.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms & Problem Complexity**]

General Terms

Algorithms, Theory

Keywords

data streams, graph sketching, sparsification, spanners, subgraphs

1. INTRODUCTION

When processing massive data sets, a core task is to construct *synopses* of the data. To be useful, a synopsis data structure should be easy to construct while also yielding good approximations of the relevant properties of the data set. A particularly useful class of synopses are *sketches*, i.e., those based on linear projections of the data. These are applicable in many settings including various parallel, stream, and compressed sensing models. There is a large body of work on sketching numerical data, e.g., finding heavy hitters and quantiles [10, 13]; estimating norms and support sizes [32, 33]; and constructing histograms and low-dimensional approximations [11, 26]. See Cormode [12] for a survey. In this paper, we design and analyze sketches for graph data.

Massive graphs arise in any application where there is data about both basic entities and the relationships between these entities, e.g., web-pages and hyperlinks between web-pages, IP addresses and flows between addresses, people and their friendships. Properties of interest include the distances between nodes of the graph, natural partitions and the size of cuts, and the prevalence of dense sub-graphs. Applicable synopses for these properties include *spanners* and *sparsifisers*. These are sparse (weighted) subgraphs of the original graph from which properties of the original graph can be approximated. Both spanners and sparsifiers have been studied extensively [8, 22, 34]. Our work addresses the problem of constructing these synopses for massive graphs. Specifically, we show how to construct such synopses given only linear projections of the input graph.

Sketching is naturally connected to dimensionality reduction. For example, the classic tug-of-war sketch of Alon, Mattias, and Szegedy [5] is closely related to the Johnson-Lindenstrauss lemma for ℓ_2 metric embedding [29]. Our results can similarly be viewed as a form of linear dimensionality reduction for graphs. For example, a graph on n nodes is essentially an $O(n^2)$ dimensional object. However, our sparsification result shows that it is possible to linearly project the graph into a $O(\epsilon^{-2} \cdot n \cdot \mathrm{polylog}\, n)$ dimensional sketch space such that the size of every cut in the graph can still be approximated up to a $(1+\epsilon)$ factor from the sketch of the graph.

1.1 Applications of Sketches

One of the main motivations for our work was to design algorithms for processing *dynamic graph streams*. A dynamic graph stream consists of a sequence of updates to a graph, i.e., edges are added and removed. The goal is to compute properties of this evolving graph without storing the entire graph. Sketches are immediately applicable for this task since the linearity of the sketch ensures that the sketch is updatable with edge deletions canceling out previously insertions. One proviso is that linear measurements required in the sketch can themselves be implicitly stored in small space and constructed when required. The sketches we design have this property.

Our sketches are also applicable in the distributed stream model [15] where the stream is partitioned over multiple locations and communication between the sites should be minimized. Again this follows because the linearity of the sketches ensures that by adding together the sketches of the partial streams, we get the sketch of the entire stream. More generally, sketches can be applied in any situation where the data is partitioned between different locations, e.g., data partitioned between reducer nodes in a MapReduce job or between different data centers.

1.2 Related Work

There exists a growing body on processing graph streams. In this setting, an algorithm is presented with a stream of m edges on n nodes and the goal is to compute properties of the resulting graph given only sequential access to the stream and limited memory. The majority of work considers the *semi-streaming* model in which the algorithm is permitted $O(n \operatorname{polylog} n)$ memory [19, 38]. Recent results include algorithms for constructing graph sparsifiers [1, 35], spanners [16, 20], matchings [2, 3, 18, 36, 41], and counting small subgraphs such as triangles [6,9,30]. This includes both single-pass algorithms and algorithms that take multiple pass over the data. See McGregor [37] for an overview.

This paper builds upon our earlier work [4] in which we established the first results for processing dynamic graph in the semi-streaming model. In the previous paper we presented sketch-based algorithms for testing if a graph was connected, k-connected, bipartite, and for finding minimum spanning trees and sparsifiers. We also consider sparsifiers in this paper (in addition to estimating shortest path distances and the frequency of various subgraphs) however our earlier results required sketches that were adaptive and the resulting semi-streaming algorithm used multiple passes. In this paper we present a single-pass sparsification algorithm. No previous work on distance estimation addresses the case of edges being both inserted and deleted. The space/accuracy trade-off of our new algorithm for counting small subgraphs matches that of the state-of-the-art result for counting triangles in the insert-only case [9].

This paper also uses several techniques which are standard in streaming such as hierarchical sampling [23, 28], ℓ_0 sampling [21, 31] and sparse recovery [24].

1.3 Our Results and Roadmap

We start in Section 2 with some preliminary definitions and lemmas. In the following three sections we present our results.

1. *Sparsifiers:* Our main result is a sketch-based sparsifier construction: we show that $O(\epsilon^{-2} n \operatorname{polylog} n)$ random linear projections of a graph on n nodes suffice to $1+\epsilon$ approximate *all* cut values including the minimum cut. This leads to a one-pass semi-streaming algorithm that constructs a graph sparsifier in the presence of both edge insertions and deletions. This result improves upon the previous algorithm that required $O(\log n)$ passes [4]. These results are presented in Section 3.

2. *Subgraphs:* We show that $O(\epsilon^{-2})$ linear projections suffice for approximating the fraction of non-empty sub-graphs that match a given pattern up to an ϵ additive term. This leads to a $\tilde{O}(\epsilon^{-2})$-space, single-pass algorithm for dynamic graph streams. In the special case of estimating the number of triangles, the space used by our algorithm matches that required for the state-of-the-art result in the insert-only data stream model [9]. We present this result in Section 4.

3. *Spanners:* In our final section, we consider adaptive sketches. We say that a sketches scheme is r-adaptive if the linear measurements are performed in r batches where measurements performed in a given batch may depend on the outcome of measurements performed in previous batches. We first show that a simple adaptation of an existing non-streaming algorithm gives rise to a k-adaptive sketch that uses $\tilde{O}(n^{1+1/k})$ linear measurements that can be used to approximate every graph distance up to a factor of $2k-1$. This naturally yields a k-pass, $\tilde{O}(n^{1+1/k})$-space algorithm. The main result of this section is our second algorithm in which we reduce the adaptivity/passes to $\log k$ at the expense of increasing the approximation factor to $k^{\log_2 5} - 1$. We present these results in Section 5.

2. PRELIMINARIES

2.1 Model Definitions

We start with the basic model definitions of a dynamic graph stream, sketches, and linear measurements.

DEFINITION 1 (DYNAMIC GRAPH STREAM). *A stream $S = \langle a_1, \ldots, a_t \rangle$ where $a_k \in [n] \times [n] \times \{-1, 1\}$ defines a multi-graph graph $G = (V, E)$ where $V = [n]$ and the multiplicity of an edge (i, j) equals*

$$A(i,j) = |\{k : a_k = (i,j,+)\}| - |\{k : a_k = (i,j,-)\}| \ .$$

We assume that the edge multiplicity is non-negative and that the graph has no self-loops.

DEFINITION 2 (LINEAR MEASUREMENTS AND SKETCHES). *A* linear measurement *of a graph is defined by a set of coefficients $c(i,j)$ for $1 \le i < j \le n$. Given a multi-graph $G = (V, E)$ where edge (i, j) has multiplicity $A(i, j)$, the evaluation of this measurement is $\sum_{1 \le i < j \le n} c(i,j) A(i,j)$. A* sketch *is a collection of linear measurements. An r-adaptive sketching scheme is a sequences of r sketches where the linear measurements performed in the rth sketch may be chosen based on the outcomes of earlier sketches.*

2.2 Graph Definitions and Notation

We denote the shortest path distance between two nodes u, v in graph $G = (V, E)$ by $d_G(u, v)$. We denote the minimum cut of G by $\lambda(G)$. For $u, v \in V$, let $\lambda_{u,v}(G)$ denote the minimum u-v cut in G. Finally, let $\lambda_A(G)$ denote the capacity of the cut $(A, V \setminus A)$.

DEFINITION 3 (SPANNERS). *Given a graph $G = (V, E)$, we say that a subgraph $H = (V, E')$ is an α-spanner for G if*

$$\forall u, v \in V, \quad d_G(u,v) \le d_H(u,v) \le \alpha \cdot d_G(u,v) \ .$$

DEFINITION 4 (SPARSIFICATION). *Given a graph $G = (V, E)$, we say that a weighted subgraph $H = (V, E', w)$ is an ϵ-sparsification for G if*

$$\forall A \subset V, \quad (1-\epsilon)\lambda_A(G) \le \lambda_A(H) \le (1+\epsilon)\lambda_A(G) \ .$$

2.3 Algorithmic Preliminaries

An important technique used throughout this paper is ℓ_0-sampling [14, 21, 31]. Consider a turnstile stream $S = \langle s_1, \dots, s_t \rangle$ where each $s_i \in (u_i, \Delta_i) \in [n] \times \mathbb{R}$ and the aggregate vector $\mathbf{x} \in \mathbb{R}^n$ defined by this stream, i.e., $x_i = \sum_{j:u_j=i} \Delta_i$. A δ-error ℓ_0-sampler for $\mathbf{x} \neq 0$ returns `FAIL` with probability at most δ and otherwise returns (i, x_i) where i is drawn uniformly at random from

$$\text{support}(\mathbf{x}) = \{i : x_i \neq 0\} .$$

The next lemma is due to Jowhari et al. [31].

THEOREM 2.1 (ℓ_0-SAMPLING). *There exists a sketch-based algorithm that performs ℓ_0 sampling using $O(\log^2 n \log \delta^{-1})$ space assuming access to a fully independent random hash function.*

While our final results will not make any assumptions about fully independent hash functions, it will be useful to state the previous results under this assumption and only address the assumption once the we have constructed the full algorithm. Another useful result will be that we can efficiently recover $\mathbf{x}$ exactly if $\mathbf{x}$ is sparse.

THEOREM 2.2 (SPARSE RECOVERY). *There exists a sketch-based algorithm, k-RECOVERY, that recovers $\mathbf{x}$ exactly with high probability if $\mathbf{x}$ has at most k non-zero entries and outputs `FAIL` otherwise. The algorithm uses $O(k \log n)$ space assuming access to a fully independent random hash function.*

In our previous paper [4], we presented an algorithm that tests k-connectivity of a graph. In addition to testing k-connectivity, the algorithm returns a "witness" which will be useful in Section 3.

THEOREM 2.3 (EDGE CONNECTIVITY). *There exists a sketch-based algorithm k-EDGECONNECT that returns a subgraph H with $O(kn)$ edges such that $e \in H$ if e belongs to a cut of size k or less in the input graph. Assuming access to a fully independent random hash function, the algorithm runs in $O(kn \log^2 n)$ space.*

3. SPARSIFICATION

In this section we design a linear sketch for graph sparsification. This yields a single-pass, semi-streaming algorithm for processing dynamic graphs.

Many sparsification algorithms are based on independently sampling edges based on their connectivity properties [8, 22, 34]. In particular, we will make use of the following recent result.

THEOREM 3.1 (FUNG ET AL. [22]). *Given an undirected unweighted graph G, let λ_e be the size of the minimum u-v cut for each edge $e = (u, v)$. If we sample each edge e with probability*

$$p_e \geq \min\{253\lambda_e^{-1}\epsilon^{-2}\log^2 n, 1\}$$

and assign weight $1/p_e$ to sampled edges, then the resulting graph is an ϵ-sparsification of G with high probability.

The challenges in performing such sampling in a dynamic graph stream are numerous. Even sampling a random edge is non-trivial since the selected edge may be subsequently removed from the graph. We solve this problem using random hash functions to ensure a consistent sampling process. However, there are two major complications that we need to overcome if we want our algorithm to run in a single pass and use small space.

- First, the sampling probability of an edge can be computed only after analyzing the entire graph stream. Unfortunately, at this point it is too late to actually sample the edges. To overcome this we develop an approach that will allow us to simultaneously sample edges and estimate sample properties. We present a basic version of our technique in Section 3.2. We then bootstrap the process to develop a more efficient construction in Section 3.3.
- Second, the random hash function being used for the consistent hashing needs to be stored in $\tilde{O}(n)$ space. However, such a random hash function cannot guarantee the full independence between random variables which is required for Lemma 3.1 and Theorem 3.1. We will use Nisan's pseudorandom generator [39] which produces a random bits that are indistinguishable to an algorithm that uses a small space, along the same lines as Indyk [27]. In the next three sections, we will assume a random oracle that facilitates full independence. In Section 3.4, we remove this assumption and detail the application of Nisan's pseudorandom generator.

3.1 Warm-up: Minimum Cut

To warm up, we start with a one-pass semi-streaming algorithm, MINCUT, for the minimum cut problem. This will introduce some the ideas used in the subsequent sections on sparsification. The algorithm is based on Karger's Uniform Sampling Lemma [34].

LEMMA 3.1 (UNIFORM SAMPLING). *Given an undirected unweighted graph G, let λ be the minimum cut value. If we sample each edge with probability*

$$p \geq \min\{6\lambda^{-1}\epsilon^{-2}\log n, 1\}$$

and assign weight $1/p$ to sampled edges, then the resulting graph is an ϵ-sparsification of G with high probability.

See Fig. 1 for our Minimum Cut Algorithm. The algorithm generates a sequence of graphs $G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \dots$ where G_i is formed by independently removing each edge in G_{i-1} with probability $1/2$. Simultaneously we use k-EDGECONNECT to construct a sequence of graphs $H_0, H_1, H_2, \dots$ where H_i contains all edges in G_i that participate in a cut of size k or less. The idea is that if i is not too large, $\lambda(G)$ can be approximated via $\lambda(G_i)$ and if $\lambda(G_i) \leq k$ then $\lambda(G_i)$ can be calculated from H_i.

THEOREM 3.2. *Assuming access to fully independent random hash functions, there exists a single-pass, $O(\epsilon^{-2} n \log^4 n)$-space algorithm that $(1+\epsilon)$-approximates the minimum cut in the dynamic graph stream model.*

PROOF. If a cut in G_i has less than k edges that cross the cut, the witness contains all such edges. On the other hand, if a cut value is larger than k, the witness contains at least k edges that cross the cut. Therefore, if G_i is not k-edge-connected, we can correctly find a minimum cut in G_i using the corresponding witness.

Let $\lambda(G)$ be the minimum cut size of G and let

$$i^* = \left\lfloor \log \max\left\{1, \frac{\lambda\epsilon^2}{6\log n}\right\}\right\rfloor .$$

For $i \leq i^*$, the edge weights in G_i are all 2^i and therefore G_i approximates all the cut values in G w.h.p. by Lemma 3.1. Therefore, if MINCUT returns a minimum cut from G_i with $i \leq i^*$, the returned cut is a $(1+\epsilon)$-approximation.

By Chernoff bound, the number of edges in G_{i^*} that crosses the minimum cut of G is $O(\epsilon^{-2}\log n) \leq k$ with high probability. Hence, MINCUT terminates at $i \leq i^*$ and returns a $(1+\epsilon)$-approximation minimum cut with high probability. $\square$

```
Algorithm MINCUT
  1. For $i \in \{1,\dots,2\log n\}$, let $h_i : E \to \{0,1\}$ be a uniform hash function.
  2. For $i \in \{0,1,\dots,2\log n\}$,
      (a) Let $G_i$ be the subgraph of $G$ containing edges $e$ such that $\prod_{j\le i} h_j(e)=1$.
      (b) Let $H_i \leftarrow k$-EDGECONNECT$(G_i)$ for $k=O(\epsilon^{-2}\log n)$
  3. Return $2^j\lambda(H_j)$ where $j=\min\{i:\lambda(H_i)<k\}$
```

Figure 1: Minimum Cut Algorithm. Steps 1 and 2 are performed together in a single pass. Step 3 is performed in post-processing.

```
Algorithm SIMPLE-SPARSIFICATION
  1. For $i \in \{1,\dots,2\log n\}$, let $h_i : E \to \{0,1\}$ be a uniform hash function.
  2. For $i \in \{0,1,\dots,2\log n\}$,
      (a) Let $G_i$ be the subgraph of $G$ containing edges $e$ such that $\prod_{j\le i} h_j(e)=1$.
      (b) Let $H_i \leftarrow k$-EDGECONNECT$(G_i)$ for $k=O(\epsilon^{-2}\log^2 n)$.
  3. For each edge $e=(u,v)$, find $j=\min\{i:\lambda_e(H_i)<k\}$. If $e\in H_j$, add $e$ to the sparsifier with weight $2^j$.
```

Figure 2: Simple Sparsification Algorithm. Steps 1 and 2 are performed in a single pass. Step 3 is performed in post-processing.

3.2 A Simple Sparsification

See Fig. 2 for a simple Sparsification Algorithm. The algorithm extends the Min-Cut Algorithm by taking into account the connectivity of different edges.

LEMMA 3.2. *Assuming access to fully independent random hash functions,* SIMPLE-SPARSIFICATION *uses $O(\epsilon^{-2}n\log^5 n)$ space and the number of edges in the sparsification is $O(\epsilon^{-2}n\log^3 n)$.*

PROOF. Each of the $O(\log n)$ instance of k-EDGECONNECT runs in $O(kn\log^2 n)$ space. Hence, the total space used by the algorithm is $O(\epsilon^{-2}n\log^5 n)$. Since the total number of edges returned is $O(kn\log n)$, the number of edges in the sparsification is also bounded by $O(\epsilon^{-2}n\log^3 n)$. □

As mentioned earlier, the analysis of our sparsification result uses a modification of Theorem 3.1 that arises from the fact that we will not be able to independently sample each edge. The proof of Theorem 3.1 is based on the following version of the Chernoff bound.

LEMMA 3.3 (FUNG ET AL. [22]). *Consider any subset C of edges of unweighted edges, where each edge $e\in C$ is sampled independently with probability p_e for some $p_e\in(0,1]$ and given weight $1/p_e$ if selected in the sample. Let the random variable X_e denote the weight of edges e in the sample; if e is not selected, then $X_e=0$. Then, for any $p\le p_e$ for all edges e, any $\epsilon\in(0,1]$, and any $N\ge|C|$, the following bound holds:*

$$\mathbb{P}\left[\left|\sum_{e\in C}X_e-|C|\right|\ge\epsilon N\right]<2\exp(-0.38\epsilon^2pN)\ .$$

We will need to prove an analogous lemma for our sampling procedure. Consider the SIMPLE-SPARSIFICATION algorithm as a sampling process that determines the edge weight in the sparsification. Initially, the edge weights are all 1. For each round $i=1,2,\dots$ if an edge e is not k-connected in G_{i-1}, we freeze the edge weight. For an edges e that is not frozen, we sample the edge with probability $1/2$. If the edge is sampled, we double the edge weight and otherwise, we assign weight 0 to the edge.

DEFINITION 5. *Let $X_{e,i}$ be random variables that represent the edge weight of e at round i and let X_e be the final edge weight of e. Let $p_e=\min\left\{253\lambda_e^{-1}\epsilon^{-2}\log^2 n,1\right\}$ where λ_e is the edge-connectivity of e and let $p'_e=\min\{4p_e,1\}$. Let B_e be the event that the edge weight of e is not frozen until round $\lfloor\log 1/p'_e\rfloor$ and let $B_C=\cup_{e\in C}B_e$ for a set C of edges.*

In the above process, freezing an edge weight at round i is equivalent to sampling an edge with probability $1/2^{i-1}$. We will use Azuma's inequality, which is an exponentially decaying tail inequality for dependent random process, instead of Lemma 3.3.

LEMMA 3.4 (AZUMA'S INEQUALITY). *A sequence of random variables $X_1,X_2,X_3,\dots$ is called* a martingale *if for all $i\ge1$,*

$$\mathbb{E}\left[X_{i+1}|X_i\right]=X_i.$$

If $|X_{i+1}-X_i|\le c_i$ almost surely for all i, then

$$\mathbb{P}\left[|X_n-X_1|\ge t\right]<2\exp(-t^2/2\sum_i c_i^2).$$

We prove the following lemma which is identical to Theorem 3.3 if no bad event B_e occurs.

LEMMA 3.5. *Let C be a set of edges. For any $p\le p_e$ for all $e\in C$ and any $N\ge|C|$, we have*

$$\mathbb{P}\left[\neg B_C \text{ and } \left|\sum_{e\in C}X_e-|C|\right|\ge\epsilon N\right]<2\exp(-0.38\epsilon^2pN)\ .$$

PROOF. Suppose that we sample edges one by one and let $Y_{i,j}$ be the total weight of edges in C after j steps at round i. If $Y_{i,0}\ge|C|+\epsilon N$ for any i, we stop the sampling process.

```
Algorithm SPARSIFICATION
  1. Using SIMPLE-SPARSIFICATION, construct a $(1 \pm 1/2)$-sparsification $H$.
  2. For $i \in \{1, \dots, 2\log n\}$, let $h_i : E \to \{0,1\}$ be a uniform hash function.
  3. For $i \in \{0, 1, \dots, 2\log n\}$,
     (a) Let $G_i$ be the subgraph of $G$ containing edges $e$ such that $\prod_{j \le i} h_j(e) = 1$.
     (b) For each $u \in V$, compute $k$-RECOVERY $(\mathbf{x}^{u,i})$ for $k = O(\epsilon^{-2}\log^2 n)$ where $\mathbf{x}^{u,i} \in \{-1,0,1\}^{\binom{V}{2}}$ with entries
         $\mathbf{x}^{u,i}[v,w] = \begin{cases} 1 & \text{if } u = v \text{ and } (v,w) \in G_i \\ -1 & \text{if } u = w \text{ and } (v,w) \in G_i \\ 0 & \text{otherwise} \end{cases}$ .   (1)
  4. Let $T = (V, E_T, w)$ be the Gomory-Hu tree of $H$ and for each edge $e \in E_T$,
     (a) Let $C$ be the cut induced by $e$ and let $w(e)$ be the weight of the cut.
     (b) Let $j = \lfloor \log(\max\{w(e)\epsilon^2/\log n, 1\}) \rfloor$.
     (c) $k$-RECOVERY $(\sum_{u \in A} \mathbf{x}^{u,j})$ returns all the edges in $G_j$ that cross $C$ with high probability.
     (d) Let $e = (u,v)$ be a returned edge and $f$ be the minimum weight edge in the $u$-$v$ path in the Gomory-Hu tree. If $f$
         induces $C$, include $e$ to the graph sparsification with edge weight $2^j$.
```

Figure 3: Better Sparsification Algorithm. Steps 1-3 are performed in a single pass. Step 4 is performed in post-processing.

For each step in round i, we change the edge weight from 2^{i-1} to either 2^i or 0 with equal probability. The expectation of the edge weight is 2^{i-1} and therefore, $\mathbb{E}[Y_{i,j}|Y_{i,j-1}] = Y_{i,j-1}$. In addition, there are at most $\frac{|C|+\epsilon N}{2^{i-1}}$ random variables $Y_{i,j}$ at round i since otherwise, $Y_{i,0}$ has to be greater than $|C| + \epsilon N$ and we would have stopped the sampling process. So

$$\begin{aligned}\sum_{i'<i}\sum_j |Y_{i',j} - Y_{i',j-1}|^2 &\le \sum_{i'<i} \frac{|C| + \epsilon N}{2^{i'-1}} 2^{2(i'-1)} \\ &= \sum_{i'<i} 2^{i'-1}(|C| + \epsilon N) \le 2^{i+1} N \,.\end{aligned}$$

Now the following inequality follows from Azuma's inequality.

$$\mathbb{P}[|Y_{i,0} - |C|| \ge \epsilon N] < 2\exp\left(-\frac{\epsilon^2 N}{2^{i+2}}\right)$$

Let $i = \lfloor \log \max\{1/(4p), 1\} \rfloor$. If B_C does not occur, $Y_{i,0} = \sum_{e \in C} X_e$. From the definition of i, $i = 0$ or $2^{-(i+2)} \ge 0.38p$. If $i = 0$, obviously $Y_{i,0} = |C|$. If $2^{-(i+2)} \ge 0.38p$, we get the desired result: $\mathbb{P}[|Y_{i,0} - |C|| \ge \epsilon N] < 2\exp(-0.38\epsilon^2 pN)$. □

THEOREM 3.3. *Assuming access to fully independent random hash functions, there exists a single-pass, $O(\epsilon^{-2} n \log^5 n)$-space $(1+\epsilon)$-sparsification algorithm in the dynamic graph stream model.*

PROOF. By replacing Theorem 3.3 by Lemma 3.5, we can conclude that SPARSIFICATION produces a sparse graph that approximates every cut with high probability or for some edge e, B_e occurs. Consider an edge $e = (u,v)$ and some minimum u-v cut of cut value λ_e. For $i = \lfloor \log 1/p'_e \rfloor$, the expected number of edges in this cut is smaller than $k/2$ (assuming that we use a sufficiently large constant to decide k). By the Chernoff bound, e is not k-connected in G_i with high probability. By union bound, B_e do not occur for all e with high probability and we obtain the desired result. □

3.3 A Better Sparsification

In this section we present a more efficient implementation of SIMPLE-SPARSIFICATION. See Fig. 3. The idea is to first construct a less accurate "rough" sparsifier that we can use to estimate the connectivity of an edge. Then, rather than constructing all the H_i graphs via k-EDGECONNECT, we can use the more efficient sparse-recovery algorithm k-RECOVERY in combination with the Gomory-Hu data structure.

1. *Rough-Sparsification:* We construct a $(1\pm1/2)$-sparsification using the algorithm in the previous section. The goal is to compute the sampling probability of edges upto a constant factor.

2. *Final-Sparsification:* For each edge $e = (u,v)$, we find a $O(1)$-approximate minimum u-v cut C_e using the rough sparsification. Based on the cut value of C_e, we compute a sampling probability p_e of e. Let $i_e = \lfloor \log 1/p_e \rfloor$. We find all edges in G_{i_e} that cross C_e. If $e \in G_{i_e}$, assign weight 2^{i_e} to e and otherwise, assign weight 0 to e.

It is important to note that dividing the process into two steps is conceptual and that both steps are performed in a single pass over the stream.

We next discuss finding the cut C_e for each e. Note that the collection of C_e has to be efficiently computable and stored in a small space. Fortunately, Gomory-Hu tree [25] is such a data structure, and it can be computed efficiently [40].

DEFINITION 6. *A tree T is a* Gomory-Hu tree *of graph G if for every pair of vertices u and v in G, the minimum edge weight along the u-v path in T is equal to the cut value of the minimum u-v cut.*

Each edge in the Gomory-Hu tree induces a cut. It is a well-known fact that the cut value of such a cut is equal to the weight of the corresponding edge.

The method for finding the edges across a cut (line 4c) is based an ideas developed in our previous paper [4]. The definition of $\mathbf{x}^{u,i}$ in Eq. 1 ensures that for any cut $(A, V \setminus A)$,

$$\text{support}(\sum_{u\in A} \mathbf{x}^{u,i}) = E_{G_i}(A) \ ,$$

where $E_{G_i}(A)$ is the set of edges in G_i that cross the cut. Because k-RECOVERY is a linear sketch, to find $E_{G_i}(A)$ (on the assumption there are at most k edges crossing the cuts) it suffices to have computed k-RECOVERY $(\mathbf{x}^{u,i})$ because

$\sum_{u\in A}$ k-RECOVERY $(\mathbf{x}^{u,i})$= k-RECOVERY $(\sum_{u\in A} \mathbf{x}^{u,i})$.

THEOREM 3.4. *Assuming access to fully independent random hash functions, there exists a single-pass, $O(n(\log^5 n + \epsilon^{-2} \log^4 n))$-space ϵ-sparsification algorithm in the dynamic graph stream model.*

PROOF. The algorithm can be implemented in one pass. The sparse-recovery sketches do not require any knowledge of the Gomory-Hu tree and thus can be constructed in parallel with the rough sparsification. The rest of the algorithm is performed in post-processing.

The space required to construct a $(1 \pm 1/2)$-sparsification is $O(n \log^5 n)$. The space required for each sampler is $O(k \log n)$ which is $O(\epsilon^{-2} \log^3 n)$. Since there are n such samplers per G_i, the total space required for the samplers is $O(\epsilon^{-2} n \log^4 n)$. We obtain the desired space bound by summing up both terms. □

3.4 Derandomization

In this section, we prove that we can replace the uniform random hash function with Nisan's pseudorandom generator [39]. This can be viewed as a limited independence style analysis, however this construction yields the basic result cleanly. Nisan's pseudorandom generator has the following property.

THEOREM 3.5 (NISAN [39]). *Any randomized algorithm that runs in S space and using one way access to R random bits may be converted to an algorithm that uses $O(S \log R)$ random bits and runs in $O(S \log R)$ space using a pseudorandom generator.*

A pseudorandom generator is different from a hash function that only one-way read is allowed. If a random bit has been read, it cannot be read again. So Theorem 3.5 does not apply to the graph sparsification algorithm as it is. Instead, we rearrange the input data so that the algorithm read each random bit only once. The argument was used first in Indyk [27].

Assume that the data stream is sorted, i.e., insertion and deletion operations of the same edge appear consecutively. For each edge, we generate necessary random bits (which are $O(\text{polylog}\, n)$ in number) and remember them until all the operations on the edge are read. In this way, we read each random bit only once and the algorithm still runs in $S = \tilde{O}(n)$ space and R is at most polynomial in n. We apply Theorem 3.5 to the algorithm with the sorted input stream. The graph sparsification algorithm (with the pseudorandom generator) succeeds with high probability.

Now note that because the algorithm is sketch-based, the algorithm's behavior does not change even if we change the order of the data stream. Therefore, the algorithm succeeds with high probability. The same argument also applies to the minimum cut algorithm. We have the following theorems.

THEOREM 3.6 (VARIANT OF THEOREM 3.2). *There exists a single-pass, $O(\epsilon^{-2} n \log^5 n)$-space algorithm that $(1+\epsilon)$-approximates the minimum cut in the dynamic graph stream model.*

THEOREM 3.7 (VARIANT OF THEOREM 3.4). *There exists a single-pass, $O(n(\log^6 n + \epsilon^{-2} \log^5 n))$-space ϵ-sparsification algorithm in the dynamic graph stream model.*

3.5 Sparsifying a Weighted Graph

LEMMA 3.6. *Let C be a set of edges such that edge weights are in $[1, L]$. For any $p \leq p_e$ for all $e \in C$ and any $N \geq |C|$, we have*

$$\mathbb{P}\left[\neg B_C \text{ and } \left|\sum_{e\in C} X_e - \sum_{e\in C} w_e\right| \geq \epsilon N L\right] < 2\exp(-0.38\epsilon^2 p N)$$

Lemma 3.6 is a variant of Lemma 3.5 where we have a weighted graph with edge weights in $[1, L]$ rather than an unweighted graph. The proof of Lemma 3.6 is identical to Lemma 3.5. Lemma 3.6 implies that by increasing sampling probability of edges by factor L (or equivalently, increasing k by factor L), we have a sparsification algorithm for a weighted graph with edge weights in $[1, L]$. This increases the space requirement and the number of edges in the graph sparsification.

LEMMA 3.7. *There is a semi-streaming sparsification algorithm that runs in a single pass, $O(nL(\log^6 n + \epsilon^{-2} \log^5 n))$ space, and polynomial time in the dynamic graph stream model where edge weights are in $[1, L]$.*

For graphs with polynomial edge weights, we will partition the input graph into $O(\log n)$ subgraphs where edge weights are in range $[1, 2), [2, 4), \ldots$. We construct a graph sparsification for each subgraph and merge the graph sparsifications. The merged graph is a graph sparsification for the input graph. Summarizing, we have the following theorem:

THEOREM 3.8. *There is a semi-streaming sparsification algorithm that runs in a single pass, $O(n(\log^7 n + \epsilon^{-2} \log^6 n))$ space, and polynomial time in the dynamic graph stream model where edge weights are $O(\text{poly}\, n)$.*

4. SMALL SUBGRAPHS

In this section, we present sketches for estimating the number of subgraphs of a graph G that are isomorphic to a given pattern graph H with k nodes. Specifically we are interested in estimating the fraction of non-empty induced subgraphs that match H. We denote this quantity by

$$\gamma_H(G) := \frac{\text{Number of induced subgraphs in } G \text{ isomorphic to } H}{\text{Number of non-empty subgraphs in } G \text{ of order } |H|} \ .$$

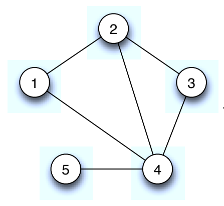
Our result is as follows:

THEOREM 4.1. *For a given order-k graph H and an order-n graph G determined by a dynamic graph stream, it is possible to approximate $\gamma_H(G)$ up to an additive ϵ term with probability $1 - \delta$ using $\tilde{O}(\epsilon^{-2} \log \delta^{-1})$ space.*

We assume k is a small constant. In the case when H is a triangle, i.e., a size-3 clique, the above result matches the parameters of the best known[1] algorithm for the insert-only case [9].

The algorithm uses a simple extension of ℓ_0 sampling. Given a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the goal of ℓ_0 sampling is to return a pair (i, x_i) for an i that is chosen uniformly from $\{i : x_i \neq 0\}$,

[1]Note that Buriol et al. [9] state their result in terms of approximating the number of triangles T_3 up to a $(1 + \epsilon)$ factor with $\tilde{O}(\epsilon^{-2}(T_1 + T_2 + T_3)/T_3)$ space but the result can equivalently be stated as an additive ϵ approximation to $T_3/(nm)$ using the fact that $T_1 + T_2 + T_3 = \Theta(mn)$. Note that nm is an upper bound on the number of non-empty induced subgraphs of size 3.



$$\longrightarrow X = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix} \longrightarrow \mathrm{Squash}(X) = \begin{pmatrix} 5 & 7 & 1 & 6 & 0 & 5 & 7 & 1 & 5 & 5 \end{pmatrix}$$

Figure 4: Linearly Encoding Small Subgraphs. See text for description.

i.e., the support set of $\mathbf{x}$. For our application we will consider a $a \times b$ binary matrix X with columns $x^1, \ldots, x^b$ and the goal is to return (i, x^i) where i is chosen uniformly from$\{i : x^i \neq 0\}$, i.e., we're picking a column of X uniformly from the set of non-zero columns.

This can easily be achieved with the machinery of ℓ_0 sampling. To do this, we encode the binary matrix X as a vector

$$\mathrm{squash}(X) \in \{0, 1, 2, \ldots, 2^{a-1}\}^b \ .$$

Specifically, adding 1 to the (i, j)th entry of X corresponds to adding 2^i to the j entry of $\mathrm{squash}(X)$. Then performing ℓ_0 sampling of $\mathrm{squash}(X)$ returns the encoding of a column picked uniformly from the set of all non-zero columns.

The application to finding small subgraphs is as follows. For a graph G, define the matrix $X_G \in \{0,1\}^{a \times b}$ where $a = \binom{k}{2}$ and $b = \binom{n}{k}$. The columns of X_G correspond to size-k subsets of the nodes of G and the entries in the column encode the set of edges in the induced subgraph on the size-k subset.

See Fig. 4 for an example where $n = 5$ and $k = 3$. The first column of X corresponds to the subset of nodes $\{1, 2, 3\}$ and the top entry is 1 because the graph G has an edge between node 1 and 2. The non-zero entries in $\mathrm{squash}(X_G)$ correspond to the number to the number non-empty induced subgraphs of G. In the case of triangles, the entries equal to 7 correspond to the induced subgraphs which are triangles. More generally, the pattern graph H will correspond to multiple values A_H since each we are interested in induced subgraphs that are isomorphic to H are there may be multiple isomorphisms. Therefore, estimating $\gamma_H(G)$ is equivalent to estimating the fraction of non-zero entries that are in A_H. By an application of the Chernoff bound, this can be estimated up to an additive ϵ using $O(\epsilon^{-2} \log \delta^{-1})$ samples from the non-zero entries, i.e., ℓ_0-samples from $\mathrm{squash}(X_G)$.

5. SPANNERS

In this section, we consider the problem of approximating graph distances via the construction of graph spanners. Several papers have investigated spanner construction in an insertion-only graph stream [7, 17, 20]. The best result constructs a $(2k-1)$-spanner using $O(n^{1+1/k})$ space in a single pass and it is known that this accuracy/space tradeoff is optimal. All these algorithms are based on growing shallow trees from a set of randomly-selected nodes. Unfortunately, this emulating this process is hard in the dynamic graph setting if we only are permitted one pass over the data.

However, if we may take multiple passes over the stream, it is straight-forward to emulate these algorithms via the ℓ_0-sampling and sparse-recovery primitives from Section 2. For example, the Baswana-Sen construction [7] leads to an $O(k)$-pass $(2k-1)$-spanner construction using $O(n^{1+1/k})$ space in a dynamic graph streams. Their construction operates as follows:

- **Part 1: Growing Trees.** This part consists of $k-1$ phases where at the end of phase i we have constructed a set of rooted vertex-disjoint trees $T_i[v]$ where v is the root of the tree and the set of roots is going to be denoted by S_i. Each $T_i[v]$ will have the property that the distance between a leaf and v is at most i. At the end of phase i there may be many vertices that are not in a tree.
  - *First phase:* Pick each vertex with probability $n^{-1/k}$. Call the selected vertices S_1. We will start growing trees around the selected vertices where the selected vertices will be the roots of their respective trees. Specifically, if vertex u is adjacent to a selected vertex v add (u, v) to the tree $T_1[v]$. If u is adjacent to multiple selected vertex, add (u, v) to one of the trees arbitrarily. If a vertex u is not adjacent to any selected vertex, we remember the set of incident edges $L(u)$.
  - *i-th phase:* Construct S_i from S_{i-1} by sampling each vertex with probability $n^{-1/k}$. For each $v \in S_i$ initialize $T_i[v] = T_{i-1}[v]$. If u is adjacent to a vertex w in some tree $T_i[v]$ add (u, w) to $T_i[v]$. If u is adjacent to multiple trees, just add u to one of the trees (doesn't matter which). Again if a vertex is not adjacent to any selected tree, then remember the set of incident edges $L(u)$ where you only store one edge to vertices in the same T_{i-1} tree.
- **Part 2: Final Clean Up.** Once we have defined $T_{k-1}[v]$ for $v \in S_{k-1}$ (and deleted all vertices not in these trees) let V' be the set of vertices in the T_{k-1} trees. For each $u \in V'$ add a single edge to a vertex in some $T_{k-1}[v]$ if such an edge exists.

See [7] for a proof of correctness. Note that each phase requires selecting $O(n^{1/k})$ edges incident on each node and this can be performed via either sparse recovery of ℓ_0 sampling.

5.1 Pass-Efficient Recursive Contraction

The above application of the Baswana-Sen construction gave an optimum trade-off between space $\tilde{O}(n^{1+1/k})$ and approximation $2k-1$, but used $O(k)$ passes which is less desirable. For example, to achieve a semi-streaming space bound, the number of passes will need to be $\Omega(\log n / \log \log n)$. While this is interesting, it is natural to ask whether we can produce a spanner in fewer passes. In what follows, we answer the question in the affirmative and provide an algorithm that uses $\log k$ passes at the expense of a worse approximation factor.

The idea behind the pass reduction is as follows. In the Baswana-Sen algorithm we were growing regions of small diameter (at various granularities) and in each pass we are growing the radius at most one. Thus the growth of the regions is slow. Moreover in each of these steps we are using $O(n)$ space (if the graph is dense).

Yet the space allowed for the vertex is $\tilde{O}(n^{1/k})$ and we expect the extra space to matter precisely when the graphs are dense! But if we are growing BFS trees, the extra edges are simply not useful. We will therefore relax the BFS constraint — this will allow us to grow the regions faster. The algorithm RECURSECONNECT is as follows.

1. The algorithm proceeds in phases which correspond to passes over the stream. In pass i, we construct a graph $\tilde{G}_i$ which corresponds to a contraction of the graph $G = \tilde{G}_0$; that is, subsets of vertices of the G have been merged into super-vertices. This process will proceed recursively and we will maintain the invariant
$$|\tilde{G}_i| \leq n^{1-(2^i-1)/k} .$$
After $\log k$ passes we have a graph of size $\sqrt{n}$ and we can remember the connectivity between every pair of vertices in $O(n)$ space. We next describe how to construct $\tilde{G}_{i+1}$ from $\tilde{G}_i$.

2. For each vertex in $\tilde{G}_i$ we sample $n^{2^i/k}$ distinct neighbors.[2] To do this, for each vertex in $\tilde{G}_i$, we independently partition the vertex set of $\tilde{G}_i$ into $\tilde{O}(n^{2^i/k})$ subsets, and use an ℓ_0-sampler for each partition. This can be achieved in $\tilde{O}(n^{1/k})$ space per vertex and in total $\tilde{O}(n^{1+1/k})$ space, using the hypotheses $|\tilde{G}_i| \leq n^{1-(2^i-1)/k}$. Using sparse recovery we can also find all vertices in $\tilde{G}_i$ whose degree is at most $n^{2^i/k}$.

3. The set of sampled edges in $\tilde{G}_i$ gives us a graph H_i. We now choose a clustering of H_i where the centers of the clusters are denoted by C_i. Consider the subset S_i of vertices of H_i which have degree at least $n^{2^i/k}$. We will ensure that C_i is a maximal subset of S_i which is independent in H_i^2. This is a standard construction used for the approximate k-center problem: We start from the set C_i^0 being an arbitrary vertex in H_i. We repeatedly augment C_i^j to C_i^{j+1} by adding vertices which are (i) at distance at least 3 (as measured in number of hops in H_i) from each vertex in C_i^j. and (ii) have degree at least $n^{2^i/k}$. Denote the final C_i^j, when we cannot add any more vertices, as C_i. Observe that
$$|C_i| \leq |\tilde{G}_i|/n^{2^i/k} \leq n^{1-(2^{(i+1)}-1)/k} .$$

4. For each vertex $p \in C_i$ all neighbors of p in H_i are assigned to p. For each vertex q with degree at least $n^{2^i/k}$ in $\tilde{G}_i$, if it is not chosen in C_i, we have a center p in C_i within 2 hops of q in H_i; then q is assigned to p as well.

5. We now collapse all the vertices assigned to $p \in C_i$ into a single vertex and these $|C_i|$ vertices define $\tilde{G}_{i+1}$.

We now analyze the approximation guarantee of the above algorithm.

LEMMA 5.1. *The distance between any pair of adjacent nodes $u, v \in G$ is at most $k^{\log_2 5} - 1$.*

PROOF. Define the maximum distance between any u, v which are in the same collapsed set in $\tilde{G}_i$ as a_i. Note that $a_1 \leq 4$ since the clustering C_1 has radius 2, and therefore any collapsed pair are at a distance at most 4. For $i > 1$ observe that $a_{i+1} \leq 5a_i + 4$ and the result follows. □

[2]Note that nodes in $\tilde{G}_i$ are subsets of the original vertex set. Vertices p, q in $\tilde{G}_i$ are neighbors in $\tilde{G}_i$ if there exists an edge $(u, v) \in G$ such that $u \in p$ and $v \in q$.

THEOREM 5.1. RECURSECONNECT *constructs a $(k^{\log_2 5} - 1)$-spanner in $\log k$ passes and $\tilde{O}(n^{1+1/k})$ space.*

Acknowledgments

We thank Graham Cormode, Atri Rudra, and David Woodruff for helpful discussions.

6. REFERENCES

[1] K. J. Ahn and S. Guha. Graph sparsification in the semi-streaming model. In *ICALP (2)*, pages 328–338, 2009.
[2] K. J. Ahn and S. Guha. Laminar families and metric embeddings: Non-bipartite maximum matching problem in the semi-streaming model. *Manuscript, available at http://arxiv.org/abs/1104.4058*, 2011.
[3] K. J. Ahn and S. Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. In *ICALP (2)*, pages 526–538, 2011.
[4] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *SODA*, 2012.
[5] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.
[6] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. of SODA*, pages 623–632, 2002.
[7] S. Baswana and S. Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.
[8] A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *STOC*, pages 47–55, 1996.
[9] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.
[10] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.
[11] K. L. Clarkson and D. P. Woodruff. Numerical linear algebra in the streaming model. In *STOC*, pages 205–214, 2009.
[12] G. Cormode. Sketch techniques for approximate query processing. In G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine, editors, *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches*, Foundations and Trends in Databases. NOW publishers, 2011.
[13] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
[14] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *VLDB*, pages 25–36, 2005.
[15] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Optimal sampling from distributed streams. In *PODS*, pages 77–86, 2010.
[16] M. Elkin. A near-optimal fully dynamic distributed algorithm for maintaining sparse spanners, 2006.
[17] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):20, 2011.
[18] L. Epstein, A. Levin, J. Mestre, and D. Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *CoRR*, abs/00907.0305, 2000.

[19] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, 2005.

[20] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38(5):1709–1727, 2008.

[21] G. Frahling, P. Indyk, and C. Sohler. Sampling in dynamic data streams and applications. In *Symposium on Computational Geometry*, pages 142–149, 2005.

[22] W. S. Fung, R. Hariharan, N. J. A. Harvey, and D. Panigrahi. A general framework for graph sparsification. In *STOC*, pages 71–80, 2011.

[23] S. Ganguly and L. Bhuvanagiri. Hierarchical sampling from sketches: Estimating functions over data streams. *Algorithmica*, 53(4):549–582, 2009.

[24] A. Gilbert and P. Indyk. Sparse recovery using sparse matrices. *Proceedings of the IEEE*, 98(6):937 –947, june 2010.

[25] R. E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[26] S. Guha, N. Koudas, and K. Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Trans. Database Syst.*, 31(1):396–438, 2006.

[27] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. *J. ACM*, 53(3):307–323, 2006.

[28] P. Indyk and D. Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 202–208. ACM New York, NY, USA, 2005.

[29] W. B. Johnson and J. Lindenstrauss. Extensions of Lipshitz mapping into Hilbert Space. *Contemporary Mathematics, Vol 26*, pages 189–206, May 1984.

[30] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *COCOON*, pages 710–716, 2005.

[31] H. Jowhari, M. Saglam, and G. Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *PODS*, pages 49–58, 2011.

[32] D. M. Kane, J. Nelson, E. Porat, and D. P. Woodruff. Fast moment estimation in data streams in optimal space. In *STOC*, pages 745–754, 2011.

[33] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52, 2010.

[34] D. R. Karger. Random sampling in cut, flow, and network design problems. In *STOC*, pages 648–657, 1994.

[35] J. A. Kelner and A. Levin. Spectral sparsification in the semi-streaming setting. In *STACS*, pages 440–451, 2011.

[36] A. McGregor. Finding graph matchings in data streams. *APPROX-RANDOM*, pages 170–181, 2005.

[37] A. McGregor. Graph mining on streams. In *Encyclopedia of Database Systems*, pages 1271–1275, 2009.

[38] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2006.

[39] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.

[40] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer, 2003.

[41] M. Zelke. Weighted matching in the semi-streaming model. *Algorithmica DOI: 10.1007/s00453-010-9438-5*, 2010.