A Decade of Graph Streams and Sketches

Andrew McGregor*
University of Massachusetts
mcgregor@cs.umass.edu

ABSTRACT

Over the last decade, there has been considerable interest in designing algorithms for processing massive graphs in the data stream model. The original motivation was two-fold: a) in many applications, the dynamic graphs that arise are too large to be stored in the main memory of a single machine and b) considering graph problems yields new insights into the complexity of stream computation. However, the techniques developed in this area are now also finding applications in other areas including data structures for dynamic graphs, approximation algorithms, and distributed and parallel computation. In this paper we survey the state-of-the-art results; identify general techniques; and highlight some simple algorithms that illustrate basic ideas.

1. INTRODUCTION

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; neurons and synapses; papers and citations; IP addresses and network flows; people and their friendships. Graphs have also become the de facto standard for representing many types of highly-structured data. However, analyzing these graphs via classical algorithms is often infeasible given the sheer size of the graphs.

One approach to handling such graphs is to process them in the *data stream model* where the input is defined by a stream of data, e.g., the stream consists of the edges of the graph. The model is defined by the constraints that the input stream must be processed in the order it arrives by an algorithm that only uses a limited amount memory. These constraints capture various challenges that arise when processing massive data sets, e.g., monitoring network traffic in real time or ensuring I/O efficiency when processing data that does not fit in main memory. Designing algorithms in this model also raises related questions such as the best way to trade-off size and accuracy when constructing data summaries. Techniques that have been developed to the reduce the space use have also been useful in reducing communication in distributed systems. The model also has deep connections with a variety of areas in theoretical computer science including communication complexity, metric embeddings, compressed sensing, and approximation algorithms.

The data stream model has become increasingly popular over the last twenty years although much of the focus has been on processing numerical data such as estimating quantiles, heavy hitters, or the number of distinct elements in the stream. The earliest work to explicitly consider graphs was by Henzinger et al. [28] who considered problems related to following paths in directed graphs and connectivity. Most of the work on graph streams has occurred in the last decade since has focused on the *semi-streaming* model [22, 41] in which the data stream algorithm is permitted $O(n \operatorname{polylog} n)$ space where n is the number of nodes in the graph. It can be shown that with less space, there are very few problems that can be solved whereas many problems become feasible once we have memory roughly proportional to the number of nodes in the graph.

In this paper we survey the results known for processing graph streams. In doing so, there are numerous goals including identifying the state-of-the-art results for a variety of popular problems and identifying general algorithmic techniques. It will also be natural to discuss some important summary data structures for graphs, such as spanners and sparsifiers. We will also discuss some simple algorithms, some of which may not be optimal, that illustrate basic ideas and would be suitable for teaching in an undergraduate or graduate clas classroom setting.

Notation. Throughout the document we use n and m to denote the total number of edges in the graph under consideration. For any natural number k, we use $[k]$ to denote the set $\{1, 2, \ldots, k\}$. Many of the algorithms are randomized and we refer to events occurring "with high probability" if the probability of the event is at least $1 - 1/n^{\Omega(1)}$.

*Supported in part by NSF awards CCF-0953754 and CCF-1320719 and a Google Research Award.

	Insert-Only	Insert-Delete	Sliding Window
Connectivity	Deterministic [22]	Randomized [4]	Deterministic
Bipartiteness	Deterministic [22]	Randomized [4]	Deterministic
$(1+\epsilon)$-Sparsifier	Deterministic [1]	Randomized [5,25]	Randomized
$(2t-1)$-Spanners	$O(n^{1+1/t})$ space [8,18]	None	$O(w^{1/2}n^{(1+1/t)/2})$ space
Min. Spanning Tree	Exact [22]	$(1+\epsilon)$-approx. [4]	$(1+\epsilon)$-approx.
Unweighted Matching	2-approx. [22]	None	$(3+\epsilon)$-approx.
Weighted Matching	4.911-approx. [20]	None	9.027-approx.

Table 1: Single-Pass, Semi-Streaming Results: All the above algorithms use $O(n \operatorname{polylog} n)$ space with the exception of the spanner constructions.

2. INSERT-ONLY STREAMS

In this section, we consider streams consisting of a sequence of unordered pairs $e = \{u, v\}$ where $u, v \in [n]$. Such a stream,

$$S = \langle e_1, e_2, \ldots, e_m \rangle$$

naturally defines an undirected graph $G = (V, E)$ where $V = [n]$ and $E = \{e_1, \ldots, e_m\}$. For simplicity we will assume that all stream elements are distinct and the resulting graph is not a multigraph[1]. We can also consider weighted, undirected graphs $G = (V, E, w)$ where now each element of the stream, (e, w_e), defines both an edge of the graph and its weight.

2.1 Connectivity, Trees, and Spanners

One of early motivations for considering the semi-streaming model is that is that $\tilde{\Theta}(n)$ space is necessary and sufficient to determine whether a graph is connected. The sufficiency follows from the following simple algorithm that constructs a spanning forest, i.e., a maximal acyclic subgraph. We maintain a set of edges H and add an edge $\{u, v\}$ from the stream to H if there is currently no path from u to v in H. A simple extension of the above algorithm also allows us to approximate the distance between any two nodes by constructing a *spanner*.

DEFINITION 1 (SPANNERS). *Given a graph $G = (V, E)$, we say that a subgraph $H = (V, E')$ is an α-spanner for G if for all $u, v \in V$,*

$$d_G(u, v) \le d_H(u, v) \le \alpha \cdot d_G(u, v) \ .$$

where $d_G(\cdot, \cdot)$ and $d_H(\cdot, \cdot)$ are lengths of the shortest paths in G and H respectively.

While the connectivity algorithm only adds an edge that does not complete a cycle, the algorithm for constructing a spanner only adds an edge that does not complete a *short* cycle.

```
Algorithm 1: Spanner
1 $H \leftarrow \emptyset$;
2 for each $(u, v) \in S$ do
3     If $d_H(u, v) > \alpha$ then $H \leftarrow H \cup \{(u, v)\}$;
4 return $H$
```

The fact that resulting graph is an α-spanner follows because for each edge $(u, v) \in G \setminus H$, there must have already been a path of length at most α in H. Hence, for any path in G of length d, including a shortest path, there is a path of length at most αd in H. The algorithm needs to store at most $O(n^{1+1/t})$ edges when $\alpha = 2t-1$ for integral t. This follows because the shortest cycle in H has length $2t + 1$ and any such graph as at most $O(n^{1+1/t})$ edges [11]. A naive implementation of the above algorithm would be slow and more recent work has focused on developing faster algorithms [8,18]. Other work [19] has considered constructing (α, β)-spanners where H is required to satisfy

$$d_G(u, v) \le d_H(u, v) \le \alpha \cdot d_G(u, v) + \beta \ .$$

Another generalization of the basic connectivity algorithm, is to maintain a minimum spanning tree (or spanning forest if the graph is not connected).

```
Algorithm 2: Minimum Spanning Tree
1 $H \leftarrow \emptyset$;
2 for each $(u, v) \in S$ do
3     $H \leftarrow H \cup \{(u, v)\}$;
4     If $H$ includes a cycle, remove the largest weight
      edge in the cycle from $H$.
5 return $H$
```

2.2 Graph Sparsification

[1]Although many of the algorithms discussed immediately extend to the multigraph setting. Other problems such as estimating the number of distinct paths of length two require new ideas when edges have multiplicity [16].

We next consider constructing graph sparsifiers in the data stream model. Rather than just determining whether a graph is connected, these sparsifiers will allow us to estimate a richer set of connectivity properties such as the size of *all* cuts in the graph.

Different notions of sparsifier exist. First, Benczür and Karger [10] introduced the notion of cut sparsification.

DEFINITION 2 (CUT SPARSIFICATION). *We say that a weighted subgraph H is a cut sparsification of a graph G if*

$$\lambda_A(H) = (1 \pm \epsilon)\lambda_A(G) \ , \quad \forall A \subset V \ , \qquad (1)$$

where $\lambda_A(G)$ and $\lambda_A(H)$ is the weight of the cut $(A, V \setminus A)$ in G and H respectively.

Spielman and Teng [45] introduced the more general notion of spectral sparsification based on approximating the Laplacian of a graph.

DEFINITION 3 (LAPLACIAN). *The Laplacian of an undirected weighted graph $H = (V, E, w)$, is a matrix $L_H \in \mathbb{R}^{n \times n}$ where*

$$L_H(i,j) = \begin{cases} \sum_{\{i,k\} \in E} w(i,k) & \text{if } i = j \\ -w(i,j) & \text{otherwise} \end{cases}$$

and $w(i,j)$ is the weight of the edge between nodes i and j. If there is no such edge, we set $w(i,j) = 0$.

DEFINITION 4 (SPECTRAL SPARSIFICATION). *We say that a weighted subgraph H is a spectral sparsification of a graph G if,*

$$x^T L_H x = (1 \pm \epsilon) x^T L_G x \ , \ \ \forall x \in \mathbb{R}^n \ , \qquad (2)$$

where L_G and L_H are the Laplacians of H and G.

Note that if we relax the requirement that the inequality in Eq. (2) is satisfied by all $x \in \mathbb{R}^n$ to only needing satisfied for all $x \in \{0,1\}^n$ then we recover Eq. (1). Hence, given a spectral sparsification of G, we can approximate the weight of all cuts in G. We can also approximate other "spectral properties" of G including the eigenvalues (via the Courant-Fischer Theorem), the effective resistances in the analogous electrical network, and properties of random walks. Obviously, any graph G has a spectral sparsifier since G is a spectral sparsifier of itself. What is surprising is that, for any G, there exists a $1+\epsilon$ spectral sparsifier with at most $O(\epsilon^{-2}n)$ edges [9].

A Simple "Merge and Reduce" Approach. Not only do small sparsifiers exist but they can also be constructed in the semi-streaming model [1, 35]. In this section, we present a simple algorithm that demonstrates the useful "merge and reduce" framework that has been useful for other data stream problems, in particular .

The following algorithm uses, as a black box, any existing algorithm that returns a $1+\gamma$ sparsifier. Let $\mathcal{A}$ be such an algorithm and let $\text{size}(\gamma)$ be an upper bound on the number of edges in the resulting spanner. As mentioned above we may assume that $\text{size}(\gamma) = O(\gamma^{-2}n)$. We will also use the following easily verifiable properties of a spectral sparsifier:

- *Mergeable:* If H_1 and H_2 are $(1+\epsilon)$ spectral sparsifiers of G_1 and G_2 then $H_1 \cup H_2$ is a $(1+\epsilon)$ spectral sparsifier of $G_1 \cup G_2 = (V, E_1 \cup E_2)$.
- *Composable:* If H_3 is an α spectral sparsifier for H_2 and H_2 is a β spectral sparsifier for H_1 then H_3 is an $\alpha\beta$ spectral sparsifier for H_1.

The algorithm is based on a hierarchical partitioning of the stream. First we partition the input stream of edges into $t = m/\text{size}(\gamma)$ segments of length $\text{size}(\gamma)$. For simplicity assume that t is a power of two. Let G_i^0 be the graph corresponding to the ith segment of edges. For $i \in \{1, \ldots, \log_2 t\}$ and $j \in \{1, \ldots, t/2^i\}$, define

$$G_i^j = G_{2i-1}^{j-1} \cup G_{2i}^{j-1} \ .$$

For example, if $t = 4$, we have:

$$G_1^1 = G_1^0 \cup G_2^0 \ \ , \ \ G_2^1 = G_3^0 \cup G_4^0 \ ,$$

$$G_1^2 = G_1^1 \cup G_2^1 = G_1^0 \cup G_2^0 \cup G_3^0 \cup G_4^0 = G \ .$$

For each G_i^j, we define a graph H_i^j as follows:

$$H_i^0 = G_i^0 \ \text{ and } \ H_i^j = \mathcal{A}(H_{2i-1}^{j-1} \cup H_{2i}^{j-1}) \text{ for } j > 0 \ .$$

It follows from the mergeable and composeable properties that $H_1^{\log_2 t}$ is an $(1+\gamma)^{\log_2 t}$ sparsifier of G. If we set $\gamma = \epsilon/(2\log_2 t)$ then this is a $1+\epsilon$ sparsifier for $\epsilon \in (0,1)$. Furthermore, it is possible to compute $H_1^{\log_2 t}$ while only storing at most

$$2\,\text{size}(\gamma)\log_2 t = O(\epsilon^{-2} n \log^3 n)$$

edges. Specifically, as we read through the stream we only need to store H_i^j for at most two values of i for each $0 \le j \le \log_2 t$ since once we have constructed H_i^j we can forget H_{2i-1}^{j-1} and H_{2i}^{j-1}.

2.3 Counting Subgraphs

Another problem that has received a significant amount of attention is counting the number of triangles, T_3, in a graph. This is closely related the *transitivity coefficient*, the fraction of paths of length two that form a triangle, and the *clustering coefficient*, i.e.,

$$\frac{1}{n}\sum_v \frac{T_3(v)}{\binom{\deg(v)}{2}}$$

where $T_3(v)$ is the number of triangles that include the node v. Both statistics play an important role in the analysis of social networks. Unfortunately, it can be shown that determining whether a graph is triangle-free requires $\Omega(n^2)$ space even with a constant number of passes and more generally, $\Omega(m/T_3)$ space is required to for any constant approximation [13]. Hence, research has focused finding additive approximations or have expressed the space use of an algorithm in terms of a given lower bound t on T_3.

Vector-Based Approach. A number of approaches for estimating the number of triangles have been based on reducing the problem to a problem about vectors. Consider a vector $v \in \mathbb{R}$ indexed by subsets S of $[n]$ of size three. The entry corresponding to S is defined

$$v_S = |\{e \in S : e \subset S\}| \ .$$

Note that the number of triangles T_3 in G equals the number entries of v that equal 3. Bar-Yossef et al. [7] presented an algorithm based on the following relationship between T_3 and the frequency moments of v, i.e., $F_k = \sum_S v_S^k$.

LEMMA 2.1. $T_3 = F_0 - 1.5F_1 + 0.5F_2$.

PROOF. Let T_i be the number of triples of nodes such that the induced graph on these nodes have exactly i edges. Note that $F_k = T_1 + 2^k T_2 + 3^k T_3$ for any k. Solving the equations for $k = 0, 1,$ and 2 gives the required relationship. □

Let $\tilde{T}_3$ be the estimate of T_3 that results by combining $(1+\gamma)$ approximations of the relevant frequency moments with the above lemma. Then.

$$|\tilde{T}_3 - T_3| < \gamma\,(F_0 + 1.5F_1 2 - 0.5F_2)$$

which is less than $8\gamma mn$ because $F_0 \le F_1 \le F_2/9$ and $F_1 = m(n-2)$. It is possible to $(1+\gamma)$ approximate each of these frequency moments in $O(\gamma^{-2} \operatorname{polylog} n)$ space and this leads to a $\tilde{O}(\epsilon^{-2}(mn/t)^2)$ space $(1+\epsilon)$-approximation by setting γ appropriately.

A better approach proposed by Ahn et al. [5] is to use the ℓ_0 sampling technique. An algorithm for ℓ_0 sampling uses $O(\operatorname{polylog} n)$ space and returns a random non-zero element from v. Let $X \in \{1, 2, 3\}$ be determined by picking a random non-zero element of v and returning the value of this element. This can be achieved using the ℓ_0 sampling technique discussed in the next section. Note that $E[X] = T_3/F_0$. By an application of the Chernoff bound, returning the mean $\tilde{X}$, of $O(\epsilon^{-2}(mn/t)\log n)$ samples ensures that with high probability

$$\tilde{X} = (1 \pm \epsilon/3)\frac{T_3}{F_0} \ .$$

It is possible to construct this many samples in space $O(\epsilon^{-2}(mn/t) \operatorname{polylog} n)$. In parallel we can compute a $(1 \pm \epsilon/3)$ approximation $\tilde{F}_0$ of F_0. Hence, we have constructed $\tilde{X}\tilde{F}_0 = (1 \pm \epsilon)T_3$. An earlier algorithm using similar space was presented by Buriol et al. [14] but the above algorithm has the advantage that it is also applicable in the setting (discussed in the next section) where edges can be inserted and deleted.

Extensions and Other Approaches. Algorithms have also been developed for the multi-pass model including a two-pass algorithm that returns a constant approximation using $\tilde{O}(m/T_3^{1/3})$ [13]. Inspired by the approach of Jowhari and Ghodsi [29], another line of work [29, 31, 39] has made clever use of complex-valued hash functions for counting and other longer cycles and other subgraphs. Pagh and Tsourakakis [42] considered a technique based on randomly coloring nodes and considering the induced graphs on monochromatic sets of nodes. Kutzkov and Pagh [38] also considered the problem of estimating the clustering coefficient directly. Other related work include approximating the size of cliques and independent sets [26, 27].

2.4 Matchings

A matching in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in M share an endpoint. Well-studied problems include finding the matching of maximum cardinality and, in the case the edges are weighted, the matching of largest total weight.

Greedy Single-Pass Algorithms. A simple semi-streaming algorithm that returns a 2-approximation for the unweighted problem is the following greedy algorithm.

Algorithm 3: Greedy Matching

```
1 $M \leftarrow \emptyset$;
2 for each $e \in S$ do
3     If $M \cup \{e\}$ is a matching, $M \leftarrow M \cup \{e\}$;
4 return $M$
```

The well-known fact that the algorithm is a 2-approximation follows from the fact that for every edge $\{u, v\}$ in the largest matching, M must include an edge with at least one of u or v as an endpoint. At present this is the best approximation known for the problem! The strongest known lower bound is $e/(e-1) \approx 1.58$ which also applies when edges are grouped by endpoint [24, 32]. Konrad et al. [36] considered a relaxation of the problem where the edges arrive in a random-order and, in this setting, they showed an algorithm that achieved a 1.98-approximation in expectation.

The greedy algorithm can be generalized to the weighted case as follows [22,40]. Rather than only adding an edge if there are no "conflicting" edges, we also add the edge if its weight is at least some factor larger than the weight of the (at most two) conflicting edges and remove these conflicting edges.

Algorithm 4: Greedy Weighted Matching

```
1 $M \leftarrow \emptyset$;
2 for each $e \in S$ do
3     Let $C = \{e' \in M : e' \cap e \neq \emptyset\}$ ;
4     If $\frac{w(e)}{w(C)} \geq (1+\gamma)$ then $M \leftarrow M \cup \{e\} \setminus C$;
5 return $M$
```

It is reasonable to ask why we shouldn't add e if $w(e) \geq w(C)$, i.e., set $\gamma = 0$. However, consider what would happen if the stream consisted of edges

$$\{1,2\}, \{2,3\}, \{3,4\}, \ldots, \{n-1, n\}$$

arrived in that order where the weight of edge $\{i, i+1\}$ is $1 + i\epsilon$ for some small value $\epsilon > 0$. The above algorithm would return the last edge with a total weight of $1 + (n-1)\epsilon$ whereas the optimal solution has weight

$$1+(n-1)\epsilon+1+(n-3)\epsilon+1+(n-3)\epsilon+\ldots > \frac{n-1}{2} ,$$

and hence decreasing ϵ makes the approximation factor arbitrarily large.

Roughly speaking, the problem with setting $\gamma = 0$ is that the weight of the "trail" of edges that are inserted into M but subsequently removed can be much larger than the weight of the final edges in M. By setting $\gamma > 0$, we ensure the weights in this trail are geometrically increasing. Specifically, let $T_e = C_1 \cup C_2 \cup \ldots$ where C_1 is the set of edges removed when e was added to M and C_{i+1} is the set of edges removed when an edge in C_i was added to M. Then, it is easy to show that for any edge e in the final matching M we have $w(T_e) \leq w(e)/\gamma$. By a careful charging scheme [40], the weight of the optimal solution can be bounded in terms of the weight of the final edges and the trails:

$$w(\mathsf{OPT}) \leq (1+\gamma) \sum_{e \in M^*} w(T_e) + 2w(e))$$

Putting together the two equations and picking the optimal value of γ yields a 5.828 approximation. The analysis can be extended to sub-modular maximization problems [15].

The above algorithm is optimal if we only consider deterministic algorithm that stores a valid matching at all times [46]. However, after a sequence of results [20, 21, 47] it is now known how to achieve a 4.91 approximation.

Multiple-Pass Algorithms. The above algorithm can be extended to a multiple-pass algorithm that achieves a $2 + \epsilon$ approximation for weighted matchings. We simply set $\gamma = O(\epsilon)$ and take $O(\epsilon^{-3})$ passes over the data where, at the start of a pass, M is initiated to the matching returned at the end of the previous pass.

Guruswami and Onak showed that finding the size of the maximum cardinality matching exactly given p passes requires $n^{1+\Omega(1/p)}/p^{O(1)}$ space. While there is no matching algorithm, it is possible to achieve arbitrarily good approximations in a similar number of passes. Ahn and Guha [2, 3] showed that a $1 - \epsilon$ approximation is possible using $O(n^{1+1/p})$ space and $O(p/\epsilon)$ passes. They also show a similar result for weighted matching if the graph is bipartite. Their results are based on adapting linear programming techniques, in particular the *multiplicative weights update method*, for data stream computation. In the vertex arrive setting, Kapralov presented an algorithm that achieved a $1 - 1/\sqrt{2\pi p} + o(1/p)$ approximation ratio given p passes. This is achieved by a fractional load balancing approach.

2.5 Random Walks

A random walk in an unweighted graph from a node $u \in V$ is a random sequence of nodes $v_0, v_1, v_2, \ldots$ where $v_0 = u$ and $v_i \in_R \Gamma(v_{i-1})$. For any fixed positive integer t, we can consider the distribution of $v_t \in V$. Call this distribution $\mu_t(u)$.

In this section, we present a semi-streaming algorithm by Das Sarma et al. [43] that returns a sample from μ_t. Note that it is trivial to sample from μ_t with t passes; in the ith pass we randomly select v_i from the neighbors of the node v_{i-1} determined in the previous pass. Das Sarma et al. show that it is possible to reduce the number of passes to $O(\sqrt{t})$. They also present algorithms that use less space at the expense of increasing the number of passes.

Algorithm. As noted above, it is trivial to perform length t walks in t passes. The main idea of the algorithm to build up a length t walk by "stitching" together length $\sqrt{t}$ walks, each of which can be constructed in parallel in $\sqrt{t}$ passes and $O(n \log n)$ space. However, we will need to be careful to ensure that the all the steps of the random walk are independent. Specifically, the algorithm starts as follows:

1. Let $T(v)$ be a node sampled from $\mu_{\sqrt{t}}(v)$.
2. Let $v = T^k(u) = T(\ldots T(T(u))\ldots)$ where ℓ is maximal values such that the nodes in
$$U = \{u, T(u), T^2(u), \ldots, T^{k-1}(u)\}$$
are all distinct and $k \leq \sqrt{t}$.

The reason we insist that the nodes in U are disjoint is because otherwise, the next steps of the random walk will not be independent of the previous steps. So far we have generated a sample v from $\mu_\ell(u)$ where $\ell = k\sqrt{t}$. We then enter the following loop:

3. While $\ell \leq \sqrt{t}$
   (a) If $v \notin U$, let $v \leftarrow T(v), \ell \leftarrow \sqrt{t} + \ell, U \leftarrow U \cup \{v\}$
   (b) Otherwise, sample $\sqrt{t}$ edges with replacement incident on each node in U. Find the maximal path from v such that on the i-th visit to node x, we take the i-th edge that was sampled for node x. The path terminates either when a node in U is visited more than $\sqrt{t}$ times or we reach a node that is not in S. Reset v to be the final node of this path and increase ℓ by the length of the path. If we complete the length t random walk during this process we may terminate at this point and return the current node.
4. Perform the remaining $O(\sqrt{t})$ steps of the walk using the trivial algorithm.

Analysis. First note that $|U|$ is never larger than $\sqrt{t}$ because $|U|$ is only incremented when ℓ increases by at least $\sqrt{t}$ and we know that $\ell \leq t$. The total space required to store the vertices T is $O(n \log n)$. When we sample $\sqrt{t}$ edges incident on each node in U, this requires $\tilde{O}(|U|\sqrt{t}) = \tilde{O}(t)$ space. Hence the total space is $\tilde{O}(n+t)$. For the number of passes, note that when we need to take a pass to sample edges incident on U, we make $O(\sqrt{t})$ hops of progress because either we reach a node with an unused short walk or the walk uses $\Omega(\sqrt{t})$ samples edges. Hence, including the $O(\sqrt{t})$ passes used at the start and end of the algorithm, the total number of passes is $O(\sqrt{t})$.

3. GRAPH SKETCHES

In this section, we consider dynamic graph streams where edge can be both added and removed. The input is a sequence $S = \langle a_1, a_2, \ldots \rangle$ where $a_i = (e_i, \Delta_i)$ and e_i encodes an undirected edge as before and $\Delta_i \in \{-1, 1\}$. The multiplicity of an edge e is defined as $f_e = \sum_{i:e_i=e} \Delta_i$. For simplicity, we restrict our attention to the case when for all edges e we have $f_e \in \{0, 1\}$.

Linear Sketches. An important type of data stream algorithms are *linear sketches*. Such algorithms maintain a random linear projection, or "sketch", of the input. To be interesting, we want to a) be able infer relevant properties of the input from the sketch and b) maintain the sketch in small space. The second property follows from the linearity of the sketch if the dimensionality of the projection is small. Specifically, suppose $\mathbf{f} \in \{0,1\}^{\binom{n}{2}}$ is the vector with entries equalling the current values of f_e and let $\mathcal{A}(\mathbf{f}) \in \mathbb{R}^d$ be the sketch of this vector where we call d the dimensionality of the sketch. Then, when (e, Δ) arrives we can simple update $\mathcal{A}(\mathbf{f})$ as follows:

$$\mathcal{A}(\mathbf{f}) \leftarrow \mathcal{A}(\mathbf{f}) + \Delta\mathcal{A}(I_e)$$

where I_e is the vector whose only non-zero entry is an "1" in the position corresponding to e. Hence, it suffices to store the current sketch and any random bits needed to compute the projection. The main challenge is therefore to design low dimensional sketches.

Homomorphic Sketches. Many of the graph sketches that have been designed so far are built up from sketches of the rows of the adjacency matrix for the graph G. Specifically, let $\mathbf{f}^v \in \{0,1\}^n$ be the vector $\mathbf{f}$ restricted to coordinates that involve node v. Then, the many of the sketches that have been developed are formed by concatenating sketches of each $\mathbf{f}^v$, i.e.,

$$\mathcal{A}(\mathbf{f}) = \mathcal{A}_1(\mathbf{f}^{v_1}) \circ \mathcal{A}_2(\mathbf{f}^{v_2}) \circ \ldots \circ \mathcal{A}_n(\mathbf{f}^{v_n}) \ .$$

Note that the random projections for different $\mathcal{A}_i$ need not be independent but that these sketches can still be updated as before.

The algorithms discussed in subsequent sections all fit the following template. First, we consider a basic algorithm for the graph problem in question. Second, we design sketches $\mathcal{A}_i$ such that it is possible to emulate the basic algorithm given only each sketch $\mathcal{A}_i(\mathbf{f}^{v_i})$. The challenge is to ensure that the sketches are homomorphic with respect to the operations of the basic algorithm, i.e., for each operation on the original graph, there is a corresponding operation on the sketches.

3.1 Connectivity

Basic Non-Sketch Algorithm. The algorithm is based on the following simple $O(\log n)$ stage process. In the first stage, we find any incident edge on each node. We then collapse each of the resulting connected components into a "supernode". In each subsequent stage, we find an edge from every supernode to another supernode (if one exists) and collapse the connected components into new supernodes. It is not hard to argue that this process terminates after $O(\log n)$ stages and that the set of edges used to connect supernodes in the different stages include a spanning forest of the graph.

Emulation via Sketches. There are two main steps to constructing the sketches for connectivity algorithm:

1. *An Appropriate Graph Representation.* For each node $v_i \in V$, define a vector $\mathbf{a}_i \in \{-1,0,1\}^{\binom{n}{2}}$:

$$\mathbf{a}^i_{\{j,k\}} = \begin{cases} 1 & \text{if } i = j < k \text{ and } \{v_j, v_k\} \in E \\ -1 & \text{if } j < k = i \text{ and } \{v_j, v_k\} \in E \\ 0 & \text{otherwise} \end{cases}$$

These vectors then have the useful property that for any subset of nodes $\{v_i\}_{i \in S}$, the non-zero entries of $\sum_{i \in S} \mathbf{a}^i$ equals $\{\{i,j\} : \{v_i, v_j\} \in E, i \in S, j \notin S\}$, the edges across the cut $(S, V \setminus S)$.

2. *ℓ_0-Sampling via Linear Measurements:* ℓ_0-Sampling takes as input a non-zero vector $\mathbf{x} \in \mathbb{R}^d$ and a random binary string r. The goal is to return a sample j where

$$\Pr_r[\text{sample equals } j] = \begin{cases} \frac{1}{|F_0(\mathbf{x})|} & \text{if } \mathbf{x}_j \neq 0 \\ 0 & \text{if } \mathbf{x}_j = 0 \end{cases} .$$

A useful feature of existing work [30] on ℓ_0 sampling is that it can be performed via linear projections, i.e., for any string r there exists $S_r \in \mathbb{R}^{k \times d}$ such that the sample can be reconstructed from $S_r\mathbf{x}$. For the process to be successful with constant probability $k = O(\log^2 n)$ suffices. Consequently given $S_r\mathbf{x}$ and $S_r\mathbf{y}$ we have enough information to determine a random sample from the set $\{i : x_i + y_i \neq 0\}$ since $S_r(\mathbf{x}+\mathbf{y}) = S_r\mathbf{x} + S_r\mathbf{y}$.

The algorithm for connectivity is clean and simple but makes use of linearity in a clever way:

1. *Simultaneously perform all the linear measurements:* Choose $t = O(\log n)$ random strings $r_1, \ldots, r_t$ and construct the ℓ_0-sampling projections $S_{r_j}\mathbf{a}^i$ for $i \in [n]$, $j \in [t]$. Then,

$$\mathcal{A}_i(\mathbf{f}^{v_i}) = \left(S_{r_1}\mathbf{a}^i\right) \circ \left(S_{r_2}\mathbf{a}^i\right) \ldots \circ \left(S_{r_t}\mathbf{a}^i\right) .$$

2. *In post-processing emulate the original algorithm:*

   (a) Let $\hat{V} = V$ be the initial set of "supernodes".

   (b) For $i = 1, \ldots, t$: for each supernode $s \in \hat{V}$, use $\sum_{i \in s} S_{r_j}\mathbf{a}^i = S_{r_j}(\sum_{i \in s} \mathbf{a}^i)$ to sample an edge between s and another supernode. Collapse the connected supernodes to form a new set of supernodes.

Analyzing this algorithm leads to the following theorem.

THEOREM 3.1. *$O(n \cdot \text{polylog}\, n)$ linear measurements suffice to determine whether a graph is connected and to construct a spanning forest.*

An easy corollary of the above the result is that it is also possible to test whether a graph is bipartite. This follows by running the connectivity algorithm on the both G and the "bipartite double cover" of G. The bipartite double cover of a graph is form by making two copies u_1, u_2 of every node u of G and adding edges $\{u_1, v_2\}, \{u_2, v_1\}$ for every edge $\{u, v\}$ of G. It can be shown the G is bipartite iff the number of connected components in the double cover is exactly the number of connected components in G.

3.2 k-Connectivity

We next present an extension for k-edge-connectivity. This algorithm builds upon ideas in the previous section and exploits the linearity of the measurements to an even greater extent.

Basic Non-Sketch Algorithm. The starting point for the algorithm is the following basic k phase algorithm:

1. For $i = 1$ to k: Let F_i be a spanning forest of $(V, E \setminus \bigcup_{j=1}^{i-1} F_j)$
2. Then $(V, F_1 \cup F_2 \cup \ldots \cup F_k)$ is k-edge-connected iff $G = (V, E)$ is at least k-edge-connected.

The correctness is simple to show by arguing that for any cut, either F_i contains an edge across this cut or $F_1 \cup \ldots \cup F_{i-1}$ contains *all* the edges across the cut. Hence, for any cut, if $F_1 \cup F_2 \cup \ldots \cup F_k$ does not contain all the edges across the cut, it contains at least k of them. We call a set of edges with this property a *k-skeleton*.

Emulation via Sketches. As with the connectivity algorithm, we performs the entire set of sketches and then emulate the algorithm on the compressed form. The important observation is that if we have computed a sketch $\mathcal{A}(G)$, but subsequently need the sketch $\mathcal{A}(G - F)$ for some set of edges F we have discovered, then this can be computed as $\mathcal{A}(G - F) = \mathcal{A}(G) - \mathcal{A}(F)$.

1. *Simultaneously compute all sketches:* Let $\mathcal{A}^1(G)$, $\mathcal{A}^2(G), \ldots, \mathcal{A}^k(G)$ be k independent sketches for finding a spanning forest.

2. *In post-processing emulate the original algorithm:* For $i \in [k]$, construct a spanning forest F_i of $(V, E \setminus F_1 \cup \ldots \cup F_{i-1})$ using

$$\mathcal{A}^i(G - F_1 - F_2 \ldots - F_{i-1}) = \mathcal{A}^i(G) - \sum_{j=1}^{i-1} \mathcal{A}^i(F_j) .$$

This leads to the following theorem.

THEOREM 3.2. *$O(k \cdot n \cdot \text{polylog}\, n)$ linear measurements suffice to determine k-edge connectivity.*

3.3 Min-Cut and Sparsification

In this section we revisit graph sparsification in the context of dynamic graphs. To do this we will need to discuss the offline algorithms for sparsification in more detail.

Sparsification via Sampling. The results in this section are based on the following generic sampling algorithm:

It is obvious that the size of any cut is preserved in expectation by the above process. However, if p_e is sufficiently large it can be shown that various properties,

Algorithm 5: Generic Sparsification Algorithm

```
1 Sample edge $e$ with probability $p_e$;
2 Weight sampled edge $e$ by $1/p_e$;
```

including the size of cuts are preserved with high probability. In particular, Karger [34] showed that for some constant c_1, if

$$p_e \geq q := \min\left\{1, c_1 \lambda^{-1} \epsilon^{-2} \log n\right\}$$

where λ is the size of the minimum cut of the graph then the resulting graph is a cut sparsifier with high probability. Fung et al. [23] strengthened this to show that the sampling probability need only scale with λ_e^{-1} where λ_e is the size of the minimum cut that separates the end points of e. Specifically, they showed that for some constant c_2, if

$$p_e \geq \min\left\{1, c_2 \lambda_e^{-1} \epsilon^{-2} \log^2 n\right\}$$

then the resulting graph is a cut sparsifier with high probability. Spielman and Srivistava [44] showed[2] that the resulting graph is a spectral sparsifier if

$$p_e \geq \min\left\{1, c_3 r_e \epsilon^{-2} \log n\right\}$$

for some constant c_3 where r_e is the effective resistance of e are equals the voltage difference that would need to be applied between u and v for 1 amp to flow between u and v in the electrical network formed by replacing each edge by a 1 ohm resister. The effective resistance r_e is a more nuanced quantity than λ_e in the sense that λ_e only depends on the number of edge-disjoint paths between the endpoints of e whereas the lengths of these paths are also relevant when calculating the effective resistance r_e. However, the two quantities can be related by the following inequality proved by Ahn et al. [6],

$$\lambda_e^{-1} \leq r_e = O(n^{2/3} \lambda_e^{-1}) \ .$$

Minimum Cut. As a warm-up, we show how to estimate the minimum cut λ of a dynamic graph [5]. To do this we will use the algorithm for constructing k-skeletons in the previous section. In addition to computing skeletons on the entire graph, we also construct skeletons for subsampled graphs. Specifically, let G_i be the graph formed from G by including each edge with probability $1/2^i$ and let

$$H_i = \text{skeleton}_k(G_i) \ ,$$

where $k = 3\epsilon^{-2} \log n$. Then, for

$$j = \min\{i : \text{mincut}(H_i) < k\} \ ,$$

[2]Note that their result is actually proved for a slightly different sampling with replacement procedure

we claim that

$$2^j \,\text{mincut}(H_j) = (1 \pm \epsilon)\lambda \ . \qquad (3)$$

For $i \leq \lfloor \log_2 1/q \rfloor$, Karger's result implies that all cuts are approximately preserved and, in particular,

$$2^i \cdot \text{mincut}(H_i) = (1 \pm \epsilon)\,\text{mincut}(G_i) \ .$$

However, for $i = \lfloor \log_2 1/q \rceil$,

$$E\left[\text{mincut}(H_i)\right] = 2^{-i}\lambda \leq 2q\lambda \leq \epsilon^{-2} \log n$$

and hence, by an application of the Chernoff bound, we have that $\text{mincut}(H_i) < k$ with high probability. Hence, $j \leq \lfloor \log_2 1/q \rfloor$ with high probability and Eq. 3 follows.

Sparsification. To construct a sparsifier, the basic idea is to sample edges with probability $q_e = \min\{1, t/\lambda_e\}$ for some value of t. Then, by the previous discussion, if $t = \Theta(\epsilon^{-2} \log^2 n)$ then the sampled graph is a combinatorial sparsifier whereas if $t = \Theta(\epsilon^{-2} n^{2/3} \log^2 n)$ then it is a spectral sparsifier. In this section, we outline how to perform such sampling. We refer the reader to Ahn et al. [5, 6] for details regarding independence issues and how to reweight the edges.

The challenge is that we do not know the values of λ_e ahead of time. To get around this we take a very similar approach to that used above for estimating the minimum cut. Specifically, let G_i be defined as above and let $H_i = \text{skeleton}_{3t}(G_i)$. For simplicity, we assume $\lambda_e \geq t$. We claim that

$$\Pr\left[e \in H_0 \cup H_1 \cup \ldots \cup H_{2\log n}\right] \geq t/\lambda_e \ .$$

This follows because the probability is at least $\Pr\left[e \in H_j\right]$ for $j = \lfloor \log \lambda_e/t \rfloor$. But the expected size of the minimum cut separating $e = \{u, v\}$ in H_j is at most $2t$ and appealing to the Chernoff bound, it has size at most $3t$ with high probability. Hence, $\Pr\left[e \in H_j\right] \approx \Pr\left[e \in G_j\right]$ since H_j was a $3t$-skeleton. The claim follows since $\Pr\left[e \in G_j\right] \geq t/\lambda$.

4. SLIDING WINDOW

In this section, we consider processing graphs in the sliding window model. In this model we consider an *infinite* stream of edges $\langle e_1, e_2, \ldots \rangle$ but at time t we only consider the graph whose edge set consists of the last w edges,

$$W = \{e_{t-w+1}, \ldots, e_t\} \ .$$

We call these the *active* edges. The results in this section were proved by Crouch et al. [17].

4.1 Connectivity

We first consider testing whether the graph is k-edge connected for a given $k \in \{1, 2, 3, \ldots\}$. Note that $k = 1$

corresponds to testing connectivity. To do this, it is sufficient to maintain a set of edges $F \subseteq \{e_1, e_2, \ldots, e_t\}$ along with the time-of-arrival $\mathrm{toa}(e)$ for each $e \in F$ such that for any cut, F contains the most recent k edges across the cut (or all the edges across the cut if there are less than k of them). Then, we can easily tell whether the graph on the active edges is k-connected by checking whether F would be k-connected once we remove all edges $e \in F$ where $\mathrm{toa}(e) \leq t - w$. This follows because if there are k or more edges among the last w edges across a cut, F will include the k most recent of them.

The following simple algorithm maintains a set

$$F = F_1 \cup F_2 \cup \ldots \cup F_k$$

with this property where the F_i are disjoint and each is cyclic. We add the new edge e to F_1. If it completes a cycle, we remove the oldest edge in this cycle and add that edge to F_2. If we now have a cycle in F_2, we remove the oldest edge in this cycle and add that edge to F_3. And so on.

THEOREM 4.1. *There exists a sliding-window algorithm for k-connectivity using $O(kn \log n)$ space.*

Applications. By reducing other problems to k-connectivity, as discussed in the previous sections, we have the following corollary:

COROLLARY 4.1. *There exists a sliding-window algorithm for:*

1. *Testing bipartiteness in $\tilde{O}(n)$ space.*
2. *Maintaining a cut sparsifier in $\tilde{O}(\epsilon^{-2}n)$ space.*
3. *Maintaining a spectral sparsifier in $\tilde{O}(\epsilon^{-2}n^{5/3})$ space.*

4.2 Matchings

We next consider the problem of finding large matchings in the sliding-window model. We will focus on the unweighted case and describe a $3+\epsilon$ approximation. However, by combining this algorithm with randomized rounding technique by Epstein et al. [20], it is possible to extend this to a 9.027 approximation.

Algorithm. The approach for estimating the size of the maximum cardinality matching is based on the "smooth histograms" technique of Braverman and Ostrovsky [12]. The algorithm maintains maximal matchings over various "buckets" $B_1, \ldots, B_k$ where each bucket comprises of the edges in some suffix of the stream that has arrived so far. The buckets will always satisfy

$$B_1 \supseteq W \supsetneq B_2 \supsetneq \cdots \supsetneq B_k \tag{4}$$

where W is the set of active edges. If we denote the size of a maximum matching on edge set B by $m(B)$, then Eq. 4 implies that

$$m(B_1) \geq m(W) \geq m(B_2) \geq \ldots \geq m(B_k) .$$

Within each bucket B, we construct keep a greedy matching $\hat{M}(B)$ whose size we denote by $\hat{m}(B)$. There is potentially a bucket for each of the w suffixes and keeping a matching for each suffix would use too much space. To reduce the space use, whenever two nonadjacent buckets have greedy matchings within a factor $1-\beta$ where $\beta = \epsilon/4$, we will delete any buckets between them. Specifically, when a new edge e arrives, we update the buckets and matchings as follows:

Algorithm 6: Update Buckets

```
1 Create a new empty bucket $B_{k+1}$;
2 Add $e$ to each $\hat{M}(B_i)$ if possible;
3 for $i = 1 \ldots k-2$ do
4     Find the largest $j > i$ such that
          $\hat{m}(B_j) \geq (1-\beta)\hat{m}(B_i)$
      and delete $B_t$ for any $i < t < j$ and renumber
      the buckets;
5 If $B_2$ contains the entire active window, delete $B_1$
  and renumber the buckets;
```

Analysis. We will prove the invariant that for any $i < k$, either $\hat{m}(B_{i+1}) \geq m(B_i)/(3+\epsilon)$ or $|B_i| = |B_{i+1}| + 1$ or both. If $|B_i| \neq |B_{i+1}| + 1$, then we must have deleted some bucket B which $B_i \subsetneq B \subsetneq B_{i+1}$. For this to have happened it must have been the case that $\hat{m}(B_{i+1}) \geq (1-\beta)\hat{m}(B_i)$ at the time. The next lemma shows that the optimal matching on the extension of the maximal matching B_{i+1}.

LEMMA 4.1. *For any set of edges C,*

$$m(B_i C) \leq \left(3 + \frac{2\beta}{1-\beta}\right) \hat{m}(B_{i+1} C) .$$

And hence we currently satisfy:

$$m(B_i) \leq \left(3 + \frac{2\beta}{1-\beta}\right) \hat{m}(B_{i+1}) \leq (3+\epsilon)\hat{m}(B_{i+1}) .$$

Therefore, either $W = B_1$ and $\hat{m}(B_1)$ is a 2 approximation for $m(W)$, or we have

$$m(B_1) \geq m(W) \geq m(B_2) \geq \hat{m}(B_2) \geq \frac{m(B_1)}{3+\epsilon}$$

and thus $\hat{m}(B_2)$ is a $(3+\epsilon)$-approximation of $m(W)$.

The fact that the algorithm does not use too much space follows from the way that the algorithm deletes buckets. Specifically, we ensure that for all $i \leq k-2$

then $\hat{m}(B_{i+2}) < (1-\beta)\hat{m}(B_i)$. Since the maximum matching has size at most n, this ensures that the number of buckets is $O(\epsilon^{-1}\log n)$. Hence, the total number of bits used to maintain all k greedy matchings is $O(\epsilon^{-1}n\log^2 n)$.

5. 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. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *CoRR*, abs/1307.4359, 2013.

[3] K. J. Ahn and S. Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. *Inf. Comput.*, 222:59–79, 2013.

[4] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 459–467, 2012.

[5] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *ACM Symposium on Principles of Database Systems*, pages 5–14, 2012.

[6] K. J. Ahn, S. Guha, and A. McGregor. Spectral sparsification of dynamic graph streams. In *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, 2013.

[7] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, pages 623–632, 2002.

[8] S. Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Inf. Process. Lett.*, 106(3):110–114, 2008.

[9] J. D. Batson, D. A. Spielman, and N. Srivastava. Twice-ramanujan sparsifiers. *SIAM J. Comput.*, 41(6):1704–1721, 2012.

[10] A. A. Benczúr and D. R. Karger. Approximating *s-t* minimum cuts in $\tilde{o}(n^2)$ time. In *ACM Symposium on Theory of Computing*, pages 47–55, 1996.

[11] B. Bollobás. *Extremal Graph Theory*. Academic Press, New York, 1978.

[12] V. Braverman and R. Ostrovsky. Smooth Histograms for Sliding Windows. *48th Annual IEEE Symposium on Foundations of Computer Science*, pages 283–293, Oct. 2007.

[13] V. Braverman, R. Ostrovsky, and D. Vilenchik. How hard is counting triangles in the streaming model? In *ICALP (1)*, pages 244–254, 2013.

[14] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[15] A. Chakrabarti and S. Kale. Submodular maximization meets streaming: Matchings, matroids, and more. *CoRR*, arXiv:1309.2038, 2013.

[16] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *PODS*, pages 271–282, 2005.

[17] M. S. Crouch, A. McGregor, and D. Stubbs. Dynamic graphs in the sliding-window model. In *ESA*, pages 337–348, 2013.

[18] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):20, 2011.

[19] M. Elkin and J. Zhang. Efficient algorithms for constructing $(1+\epsilon,\beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18(5):375–385, 2006.

[20] L. Epstein, A. Levin, J. Mestre, and D. Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM J. Discrete Math.*, 25(3):1251–1265, 2011.

[21] L. Epstein, A. Levin, D. Segev, and O. Weimann. Improved bounds for online preemptive matching. In *STACS*, pages 389–399, 2013.

[22] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[23] W. S. Fung, R. Hariharan, N. J. A. Harvey, and D. Panigrahi. A general framework for graph sparsification. In *ACM Symposium on Theory of Computing*, pages 71–80, 2011.

[24] A. Goel, M. Kapralov, and S. Khanna. On the communication and streaming complexity of maximum bipartite matching. In *SODA*, pages 468–485, 2012.

[25] A. Goel, M. Kapralov, and I. Post. Single pass sparsification in the streaming model with edge deletions. *CoRR*, abs/1203.4900, 2012.

[26] B. V. Halldórsson, M. M. Halldórsson, E. Losievskaja, and M. Szegedy. Streaming algorithms for independent sets. In *ICALP (1)*, pages 641–652, 2010.

[27] M. M. Halldórsson, X. Sun, M. Szegedy, and C. Wang. Streaming and communication complexity of clique approximation. In *ICALP (1)*, pages 449–460, 2012.

[28] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *External memory algorithms*, pages 107–118, 1999.

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

[30] 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.

[31] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun. Counting arbitrary subgraphs in data streams. In *ICALP (2)*, pages 598–609, 2012.

[32] M. Kapralov. Better bounds for matchings in the streaming model. In *SODA*, pages 1679–1697, 2013.

[33] M. Kapralov, S. Khanna, and M. Sudan. Approximating matching size from random streams. In *SODA*, 2014.

[34] D. R. Karger. Random sampling in cut, flow, and network design problems. In *ACM Symposium on Theory of Computing*, pages 648–657, 1994.

[35] J. A. Kelner and A. Levin. Spectral sparsification in the semi-streaming setting. *Theory Comput. Syst.*, 53(2):243–262, 2013.

[36] C. Konrad, F. Magniez, and C. Mathieu. Maximum matching in semi-streaming with few passes. In *APPROX-RANDOM*, pages 231–242, 2012.

[37] C. Konrad and A. Rosén. Approximating semi-matchings in streaming and in two-party communication. In *ICALP (1)*, pages 637–649, 2013.

[38] K. Kutzkov and R. Pagh. On the streaming complexity of computing local clustering coefficients. In *WSDM*, pages 677–686, 2013.

[39] M. Manjunath, K. Mehlhorn, K. Panagiotou, and H. Sun. Approximate counting of cycles in streams. In *ESA*, pages 677–688, 2011.

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

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

[42] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Inf. Process. Lett.*, 112(7):277–281, 2012.

[43] A. D. Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. *J. ACM*, 58(3):13, 2011.

[44] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6):1913–1926, 2011.

[45] D. A. Spielman and S.-H. Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011.

[46] A. B. Varadaraja. Buyback problem - approximate matroid intersection with cancellation costs. In *ICALP (1)*, pages 379–390, 2011.

[47] M. Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012.