Graph Stream Algorithms: A Survey*

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 finding applications in other areas including data structures for dynamic graphs, approximation algorithms, and distributed and parallel computation. 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 can be challenging given the sheer size of the graphs. For example, both the web graph and models of the human brain would use around 10^{10} nodes and IPv6 supports 2^{128} possible addresses.

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. For example, the stream could consist of the edges of the graph. Algorithms in this model must process the input stream in the order it arrives while using only 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. Related questions that arise include how to trade-off size and accuracy when constructing data summaries and how to quickly update these 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 the focus of much of the early work was 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 graph problems was the influential by paper by Henzinger et al. [36] which considered problems related to following paths in directed graphs and connectivity. Most of the work on graph streams has occurred in the last decade and focuses on the *semi-streaming* model [27, 52]. In this model the data stream algorithm is permitted $O(n \operatorname{polylog} n)$ space where n is the number of nodes in the graph. This is because most problems are provably intractable if the available space is sub-linear in n, whereas many problems become feasible once there is memory roughly proportional to the number of nodes in the graph.

In this document we will 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. Throughout, we will present various simple algorithms that illustrate basic ideas and would be suitable for teaching in an undergraduate or graduate classroom setting.

Notation. Throughout this document we will use n and

*Database Principles Column. Column editor: Pablo Barceló. Department of Computer Science, University of Chile, Santiago, Chile. E-mail: pbarcelo@dcc.uchile.cl

†Support. The author is supported in part by NSF awards CCF-0953754, IIS-1251110, and CCF-1320719 and a Google Research Award.

	Insert-Only	Insert-Delete	Sliding Window (width w)
Connectivity	Deterministic [27]	Randomized [5]	Deterministic [22]
Bipartiteness	Deterministic [27]	Randomized [5]	Deterministic [22]
Cut Sparsifier	Deterministic [2, 8]	Randomized [6, 31]	Randomized [22]
Spectral Sparsifier	Deterministic [8, 46]	Randomized $\tilde{O}(n^{5/3})$ space [7]	Randomized $\tilde{O}(n^{5/3})$ space [22]
$(2t-1)$-Spanners	$O(n^{1+1/t})$ space [11, 23]	Only multiple pass results known [6]	$O(\sqrt{wn^{(1+1/t)}})$ space [22]
Min. Spanning Tree	Exact [27]	$(1+\epsilon)$-approx. [5] Exact in $O(\log n)$ passes [5]	$(1+\epsilon)$-approx. [22]
Unweighted Matching	2-approx. [27] 1.58 lower bound [42]	Only multiple pass results known [3, 4]	$(3+\epsilon)$-approx. [22]
Weighted Matching	4.911-approx. [25]	Only multiple pass results known [3, 4]	9.027-approx. [22]

Table 1: Single-Pass, Semi-Streaming Results: Algorithms use $O(n \operatorname{polylog} n)$ space unless noted otherwise. Results for approximating the frequency of subgraphs discussed in Section 2.3.

m to denote the number of nodes and edges in the graph under consideration. For any natural number k, we use $[k]$ to denote the set $\{1, 2, \ldots, k\}$. We write $a = b \pm c$ to denote $b - c \leq a \leq b + c$. Many of the algorithms are randomized and we refer to events occurring *with high probability* if the probability of the event is at least $1 - 1/\operatorname{poly}(n)$. We use $\tilde{O}(\cdot)$ to indicate that logarithmic factors have been omitted.

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

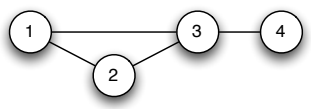

Figure 1: The graph on four nodes defined by the stream $S = \langle \{1,2\}, \{2,3\}, \{1,3\}, \{3,4\} \rangle$. It will be used to illustrate various definitions in later sections.

[1]Although many of the algorithms discussed immediately extend to the multigraph setting. Other problems such as estimating the number of triangles or distinct paths of length two require new ideas when edges have multiplicity [21, 38].

2.1 Connectivity, Trees, and Spanners

One of early motivations for considering the semi-streaming model 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: we maintain a set of edges H and add the next edge in the stream $\{u, v\}$ to H if there is currently no path from u to v in H.

Spanners. A simple extension of the above algorithm also allows us to approximate the distance between any two nodes by constructing a *spanner*.

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

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

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 added an edge if it did not complete a cycle, the algorithm for constructing a spanner will add an edge if it does not complete a short cycle.

Algorithm 1: Spanner

```
H ← ∅;
for each {u, v} ∈ S do
    If d_H(u, v) > α then H ← H ∪ {{u, v}};
return H
```

The fact that the 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 corresponding 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 has at most $O(n^{1+1/t})$ edges [15]. A naive implementation of the above algorithm would be slow and more recent work has focused on developing faster algorithms [11, 23]. Other work [24] has considered constructing (α, β)-spanners where H is required to satisfy

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

Minimum Spanning Tree. 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

```
H ← ∅;
for each {u, v} ∈ S do
    H ← H ∪ {(u, v)};
    If H includes a cycle, remove the largest weight
    edge in the cycle from H.
return H
```

By using the appropriate data structures, the above algorithm can be implemented such that each update takes $O(\log n)$ time [60].

2.2 Graph Sparsification

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. We will be interested in different types of sparsifier. First, Benczür and Karger [14] introduced the notion of cut sparsification.

DEFINITION 2 (CUT SPARSIFICATION). *We say that a weighted subgraph H is a $(1+\epsilon)$ 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 [59] 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, let $w(i, j) = 0$.

DEFINITION 4 (SPECTRAL SPARSIFICATION). *We say that a weighted subgraph H is a $(1+\epsilon)$ 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 replace $\forall x \in \mathbb{R}^n$ in Equation 2 by $\forall x \in \{0, 1\}^n$ then we recover Equation 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 various 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 there exists a $(1+\epsilon)$ spectral sparsifier with at most $O(\epsilon^{-2} n)$ edges [12].

A Simple "Merge and Reduce" Approach. Not only do small spectral sparsifiers exist but they can also be constructed in the semi-streaming model [2, 46]. 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 [8].

The following algorithm uses, as a black box, any existing algorithm that returns a $(1+\gamma)$ spectral 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 sparsifier. 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:* Suppose H_1 and H_2 are α spectral sparsifiers of two graphs G_1 and G_2 on the same set of nodes. Then $H_1 \cup H_2$ is an α spectral sparsifier of $G_1 \cup G_2$.
- *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 i-th 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 weighted subgraph H_i^j using the sparsification algorithm $\mathcal{A}$ 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. 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 at any given time. This is because, as soon as we have constructed H_i^j, we can forget H_{2i-1}^{j-1} and H_{2i}^{j-1}. Hence, at any given time we will only need to store H_i^j for at most two values of i for each j.

2.3 Counting Subgraphs

Another problem that has received significant attention is counting the number of triangles, T_3, in a graph. This is closely related to 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 for any constant approximation [17]. Hence, research has focused on designing algorithms whose space will depend on a given lower bound $t \leq T_3$.

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 $\mathbf{x}$ indexed by subsets T of $[n]$ of size three. Each T represents a triple of nodes and the entry corresponding to T is defined to be,

$$\mathbf{x}_T = |\{e \in S : e \subset T\}| \ .$$

For example, in the stream S corresponding to the graph in Figure 1, the entries of the vector are:

$$\mathbf{x}_{\{1,2,3\}} = 3, \ \mathbf{x}_{\{1,2,4\}} = 1, \ \mathbf{x}_{\{1,3,4\}} = 2, \ \mathbf{x}_{\{2,3,4\}} = 2 \ .$$

Note that the number of triangles T_3 in G equals the number entries of $\mathbf{x}$ that equal 3. Bar-Yossef et al. [10] presented an algorithm based on the following relationship between T_3 and the frequency moments of $\mathbf{x}$, i.e., $F_k = \sum_T \mathbf{x}_T^k$.

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

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,

$$|\tilde{T}_3 - T_3| < \gamma \left(F_0 + 1.5F_1 + 0.5F_2\right) \leq 8\gamma mn$$

where the last inequality follows since

$$\max(F_0, F_2/9) \leq F_1 = m(n-2) \ .$$

It is possible to $(1+\gamma)$-approximate each of these frequency moments in $\tilde{O}(\gamma^{-2})$ space and so, by setting $\gamma = \epsilon/(8mn)$, this implies a $(1+\epsilon)$-approximation algorithm using $\tilde{O}(\epsilon^{-2}(mn/t)^2)$ space.

A more space-efficient approach proposed by Ahn et al. [6], is to use the ℓ_0 sampling technique [40]. An algorithm for ℓ_0 sampling uses $O(\text{polylog}\, n)$ space and returns a random non-zero element from $\mathbf{x}$. Let $X \in \{1,2,3\}$ be determined by picking a random non-zero element of v and returning the associated value of this element. Let $Y = 1$ if $X = 3$ and $Y = 0$ otherwise. Note that $E[Y] = T_3/F_0$. By an application of the Chernoff bound, the mean of $\tilde{O}(\epsilon^{-2}(mn/t))$ independent copies of Y equals $(1 \pm \epsilon)T_3/F_0$ with high probability. Multiplying this by an approximation of F_0 yields a good estimate of T_3. Note that an earlier algorithm using similar space was presented by Buriol et al. [18] but the above algorithm has the advantage that it is also applicable in the setting (discussed in a later section) where edges can be inserted and deleted.

Extensions and Other Approaches. Pavan et al. [54] developed the approach of Buriol et al. such that the dependence on n in the space used became a dependence on the maximum degree in the graph, and a tighter analysis is possible. Pagh and Tsourakakis [53] presented an algorithm based on randomly coloring the nodes and counting the number of monochromatic triangles exactly. Algorithms have also been developed for the multi-pass model including a two-pass algorithm using $\tilde{O}(m/t^{1/3})$ space [17] and an $O(\log n)$-pass semi-streaming algorithm [13]. Kutzkov and Pagh [49] and Jha et al. [37] also designed algorithms for estimating the clustering and transitivity coefficients directly.

Extending an approach used by Jowhari and Ghodsi [39], another line of work [39, 41, 50] makes clever use of complex-valued hash functions for counting longer cycles and other subgraphs. Lower bounds for finding short cycles were proved by Feigenbaum et al. [28]. Other related work includes approximating the size of cliques [35], independent sets [34], and dense components [9].

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 including computing the matching of maximum cardinality or maximum total weight.

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

Algorithm 3: Greedy Matching

```
M ← ∅;
for each e ∈ S do
    If M ∪ {e} is a matching, M ← M ∪ {e};
return M
```

The fact that the algorithm returns a 2-approximation follows from the fact that for every edge $\{u, v\}$ in a maximum cardinality 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 [30,42]. Konrad et al. [47] considered a relaxation of the problem where the edges arrive in a random-order and, in this setting, they designed an algorithm that achieved a 1.98-approximation in expectation.

The greedy algorithm can easily be generalized to the weighted case as follows [27, 51]. 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 then remove these conflicting edges.

Algorithm 4: Greedy Weighted Matching

```
M ← ∅;
for each e ∈ S do
    Let C = {e' ∈ M : e' ∩ e ≠ ∅} ;
    If w(e)/w(C) ≥ (1 + γ) then M ← M ∪ {e} \ C;
return M
```

It would be 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

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

arriving 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

$$1+(n-1)\epsilon+1+(n-3)\epsilon+1+(n-5)\epsilon+\ldots > \frac{n-1}{2},$$

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, the total weight of edges in T_e satisfies,

$$w(T_e) \leq w(e)/\gamma .$$

By a careful charging scheme [51], the weight of the optimal solution can be bounded in terms of the weight of the final edges and the trails:

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

Substituting in the bound on $w(T_e)$ and optimizing over γ yields a 5.828-approximation. The analysis can be extended to sub-modular maximization problems [20].

The above algorithm is optimal among deterministic algorithms that only remember the edges of a valid matching at any time [61]. However, after a sequence of results [25, 26, 62] 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. No exact algorithm is known with these parameters but it is possible to find an arbitrarily good approximation. Ahn and Guha [3,4] 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 and a careful analysis of the intrinsic adaptivity of primal-dual algorithms. In the node arrival setting where edges are grouped by endpoint, Kapralov presented an algorithm that achieved a $1/(1-1/\sqrt{2\pi p}+o(1/p))$-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 is a random node from the set $\Gamma(v_{i-1})$, the neighbors of 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. [56] that returns a sample from $\mu_t(u)$. Note that it is trivial to sample from $\mu_t(u)$ with t passes; in the i-th 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 short walks of length $\sqrt{t}$. Each of these short walks 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 all the steps of the final 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 k 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 that 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 U. 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 $\tilde{O}(n)$. 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 edges can be both added and removed. The input is a sequence

$$S = \langle a_1, a_2, \ldots \rangle \quad \text{where} \quad a_i = (e_i, \Delta_i)$$

where 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 where $f_e \in \{0, 1\}$ for all edges e.

Linear Sketches. An important type of data stream algorithms are *linear sketches*. Such algorithms maintain a random linear projection, or "sketch", of the input. We want to be able to a) 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:

$$\mathcal{A}(\mathbf{f}) \leftarrow \mathcal{A}(\mathbf{f}) + \Delta \cdot \mathcal{A}(\mathbf{i}^e)$$

where $\mathbf{i}^e$ is the vector whose only non-zero entry is a "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 $\mathbf{f}^v \in \{0, 1\}^{n-1}$ be the vector $\mathbf{f}$ restricted

to coordinates that involve node v. Then, the sketches 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 the projections $\mathcal{A}_i(\mathbf{f}^{v_i})$. 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

We start with a simple algorithm for finding a spanning forest of a graph and then show how to emulate this algorithm via sketches.

Basic Non-Sketch Algorithm. The algorithm is based on the following simple $O(\log n)$ stage process. In the first stage, we find an arbitrary incident edge for 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. From this we can obviously deduce whether the graph is connected.

Emulation via Sketches. There are two main steps to constructing the sketches for the 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 $\sum_{i \in S} \mathbf{a}^i$ correspond exactly to the edges across the cut $(S, V \setminus S)$.

2. *ℓ_0-Sampling via Linear Sketches:* As mentioned earlier, the goal of ℓ_0-sampling is to take a non-zero vector $\mathbf{x} \in \mathbb{R}^d$ and 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 [40] on ℓ_0 sampling is that it can be performed via linear projections, i.e., for any string r there exists $M_r \in \mathbb{R}^{k \times d}$ such that the sample can be reconstructed from $M_r\mathbf{x}$. For the process to be successful with constant probability $k = O(\log^2 n)$ suffices. Consequently, given $M_r\mathbf{x}$ and $M_r\mathbf{y}$ we have enough information to determine a random sample from the set $\{i : x_i + y_i \neq 0\}$ since

$$M_r(\mathbf{x} + \mathbf{y}) = M_r\mathbf{x} + M_r\mathbf{y} .$$

For example, for the graph in Figure 1, we have

$$\begin{array}{rcl} \mathbf{a}^1 & = & (\ 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0\ ) \\ \mathbf{a}^2 & = & (\ -1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0\ ) \\ \mathbf{a}^3 & = & (\ 0 \quad -1 \quad 0 \quad -1 \quad 0 \quad 1\ ) \\ \mathbf{a}^4 & = & (\ 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad -1\ ) \end{array}$$

where the entries correspond to the sets $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{2,3\}$, $\{2,4\}$, $\{3,4\}$ in that order. Note that the non-zero entries of

$$\mathbf{a}^1 + \mathbf{a}^2 \ = \ (\ 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0\ )$$

correspond to $\{1,3\}$ and $\{2,3\}$ which are exactly the edges across the cut $(\{1,2\}, \{3,4\})$.

The resulting algorithm for connectivity is relatively simple but makes use of linearity in an essential way:

1. *In a single pass, compute the sketches:* Choose $t = O(\log n)$ random strings $r_1, \ldots, r_t$ and construct the ℓ_0-sampling projections $M_{r_j}\mathbf{a}^i$ for $i \in [n]$, $j \in [t]$. Then,

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

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 $\sum_{i \in S} M_{r_j}\mathbf{a}^i = M_{r_j}(\sum_{i \in S} \mathbf{a}^i)$ to sample an edge between S and another supernode. Collapse the connected supernodes to form a new set of supernodes.

Since each sketch $\mathcal{A}_i$ has dimension $O(\text{polylog}\, n)$ and there are n such sketches to be computed, the final connectivity algorithm uses $O(n \cdot \text{polylog}\, n)$ space.

Extensions and Further Work. Note that the above algorithm has $O(\text{polylog}\, n)$ update time but a connectivity query may take $\Omega(n)$ time. This was addressed in subsequent work by Kapron et al. [44].

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 formed 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 that G is bipartite iff the number of connected components in the double cover is exactly twice the number of connected components in G.

3.2 k-Connectivity

We next present an extension to test k-connectivity, i.e, determining whether every cut in the graph includes at least k edges. This algorithm builds upon ideas in the previous section and exploits the linearity of the sketches 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 easy to show by arguing that for any cut, every F_i contains an edge across this cut or

$$F_1 \cup \ldots \cup F_{i-1}$$

already contains *all* the edges across the cut. Hence, if $F_1 \cup F_2 \cup \ldots \cup F_k$ does not contain all the edges across the cut, it includes 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 compute 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. *In a single pass, compute the 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

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

Since computing each spanning forest sketch used $O(n \cdot \text{polylog}\, n)$, the total space used by the algorithm for k-connectivity is $O(k \cdot n \cdot \text{polylog}\, n)$.

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:

Algorithm 5: Generic Sparsification Algorithm

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

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 a range of properties, including the size of cuts are approximately preserved with high probability. In particular, Karger [45] 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. [29] 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 [58] 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. The effective resistance of an edge $\{u, v\}$ is 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 are related by the following inequality [7],

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

Minimum Cut. As a warm-up, we show how to estimate the minimum cut λ of a dynamic graph [6]. To do this we use the algorithm for constructing k-skeletons described in the previous section in conjunction with Karger's sampling result. In addition to computing a

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

skeleton 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

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

be a k-skeleton of G_i where $k = 3c_1\epsilon^{-2}\log n$. Then, for

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

we claim that

$$2^j \, \text{mincut}(H_j) = (1 \pm \epsilon)\lambda \ . \tag{4}$$

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 $i = \lfloor \log_2 1/q \rfloor$,

$$E\left[\text{mincut}(H_i)\right] \leq 2^{-i}\lambda \leq 2q\lambda \leq 2c_1\epsilon^{-2}\log n$$

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 Equation 4 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. If $t = \Theta(\epsilon^{-2}\log^2 n)$ then the resulting graph is a combinatorial sparsifier by appealing to the aforementioned result of Fung et al. [29]. If $t = \Theta(\epsilon^{-2}n^{2/3}\log n)$ then the resulting graph can be shown to be a spectral sparsifier by combining Equation 3 with the aforementioned sampling result of Spielman and Srivistava [58]. In this section, we briefly outline how to perform such sampling. We refer the reader to Ahn et al. [6, 7] 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 above 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 $\Pr\left[e \in G_j\right] \geq t/\lambda_e$.

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 and we will consider the case where $w \geq n$. The results in this section were proved by Crouch et al. [22]. Note that some of sampling-based algorithms for counting small subgraphs are also applicable in this model.

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 $\text{toa}(e)$ 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 of active edges is k-connected by checking whether F would be k-connected once we remove all edges $e \in F$ where $\text{toa}(e) \leq t - w$. 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 these edges.

The following simple algorithm maintains the set

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

where the F_i are disjoint and each is acyclic. 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 forth.

Therefore, it is possible to test k-connectivity in the sliding window model using $O(kn\log n)$ space. Furthermore, by reducing other problems to k-connectivity, as discussed in the previous sections, this also implies the existence of algorithms for testing bipartiteness and constructing sparsifiers.

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. It is also possible to get a 9.027-approximation for the weighted case by combing this algorithm with a randomized rounding technique by Epstein et al. [25].

Algorithm. The approach for estimating the size of the maximum cardinality matching is based on the "smooth histograms" technique of Braverman and Ostrovsky [16].

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 have arrived so far. The buckets will always satisfy

$$B_1 \supseteq W \supsetneq B_2 \supsetneq \cdots \supsetneq B_k \tag{5}$$

where W is the set of active edges. Equation 5 implies that

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

where $m(\cdot)$ denotes the size of the maximum matching on a sequence of edges.

Within each bucket B, we construct 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 usage, whenever two non-adjacent buckets have greedy matchings whose matching size is within a factor of $1-\beta$ where $\beta = \epsilon/4$, we will delete the intermediate buckets. Specifically, when a new edge e arrives, we update the buckets and matchings as follows:

Algorithm 6: Procedure for Updating 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
$$\hat{m}(B_j) \geq (1-\beta)\hat{m}(B_i)$$
Discard intermediate buckets and renumber;
5 If $B_2 = W$, discard B_1. Renumber the buckets;

Analysis. We will prove the invariant that for any $i < k$,

$$\hat{m}(B_{i+1}) \geq m(B_i)/(3+\epsilon)$$

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 such that $B_i \subsetneq B \subsetneq B_{i+1}$. 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 sequence of edges starting with B_i is not significantly larger than the greedy matching we find if we start with only start with B_{i+1}.

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

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

where $B_i C$ is the concatenation of B_i and 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 $i \leq k-2$ we have $\hat{m}(B_{i+2}) < (1-\beta)\hat{m}(B_i)$. 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. CONCLUSIONS AND DIRECTIONS

There is now a large body of work on the design and analysis of algorithms for processing graphs in the data stream model. Problems that have received considerable attention include estimating connectivity properties, approximating graph distances, finding approximate matching, and counting the frequency of sub-graphs. The resulting algorithms combine existing data stream techniques with ideas from approximation algorithms and graph theory. By both identifying the state-of-the-art results and illustrating some of the techniques behind these results, it is hoped that this survey will be useful to both researchers that may want to use existing algorithms and to those that want to develop new algorithms for different problems.

There are numerous possible directions for future research. Naturally, it would be interesting to improve existing results. For example, does there exist a semi-streaming algorithm for constructing a spectral sparsifier when there are both edge insertions and deletions? What is the optimal approximation ratio for estimating the size and weight of maximum matchings? Other specific questions can be found at the wiki,

`sublinear.info` .

More general, open-ended research directions include:

1. *Directed Graphs.* Relatively little is known about processing directed graphs and yet many natural graphs are directed. For example, it is known that any semi-streaming algorithm testing s-t connectivity requires $\Omega(\log n)$ passes [33] but is this number of passes sufficient? If we could estimate the size of flows in directed graphs, this could lead to better algorithms for approximating the size of bipartite matchings.

2. *Communication Complexity.* The recent results on graph sketching imply surprisingly efficient communication protocols; if the rows of an adjacency

matrix are partitioned between n players, then the connectivity properties of the graph can be inferred from a $O(\text{polylog}\, n)$ bit message from each player. In contrast, if the partition of the entries is arbitrary, the players needs to send $\tilde{\Omega}(n)$ on average [55]. What other graph problems can be solved using only short messages? What if each player also knows the neighbors of the neighbors of a node? From a different perspective, establishing reductions from communication complexity problems is a popular approach for proving lower bounds in the data stream model. But less is known about graph stream lower bounds because it is often harder to decompose graph problems into multiple simpler "independent" problems and use existing communication complexity techniques.

3. *Stream Ordering.* The analysis of stream algorithms has traditionally been "doubly worst case" in the sense that the contents of the stream *and* the ordering of the stream are both chosen adversarially. If we relax the second assumption and assume that the stream is ordered randomly (or that the stream is stochastically generated), can we design algorithms that take advantage of this? Some recent work is already considering this direction [19, 43, 47]. Alternatively, it may be interesting to further explore the complexity of various graph problems under specific edge orderings, e.g., sorted-by-weight, grouped-by-endpoint, or orderings tailored to the problem at hand [57].

4. *More or Less Space.* Research focusing on the semi-streaming model has been very fruitful and many interesting techniques have been developed that have had applications beyond stream computation. However, the model itself is not suited to process sparse graphs where $m = \tilde{O}(n)$. While many basic problems require $\Omega(n)$ space, this does not preclude smaller-space algorithms if we may make assumptions about the input or only need to "property test" the input [32], i.e., we just need to distinguish graphs with a given property from graphs that are "far" from having the property. Do such algorithms exist? Alternatively, what if we are permitted *more* than $\tilde{O}(n\, \text{polylog}\, n)$ space? Various lower bounds, such as those for approximate unweighted matching, are very sensitive to the exact amount of space available and nothing is known if we may use $O(n^{1.1})$ space for example.

Acknowledgements. Thanks to Graham Cormode, Sagar Kale, Hoa Vu, and an anonymous reviewer for numerous helpful comments.

6. REFERENCES

[1] K. J. Ahn. *Analyzing massive graphs in the semi-streaming model*. PhD thesis, University of Pennsylvania, Philadelphia, Pennsylvania, Jan. 2013.

[2] K. J. Ahn and S. Guha. Graph sparsification in the semi-streaming model. In *International Colloquium on Automata, Languages and Programming*, pages 328–338, 2009.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] M. Badoiu, A. Sidiropoulos, and V. Vaikuntanathan. Computing s-t min-cuts in a semi-streaming model. *Manuscript*.

[9] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.

[10] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.

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

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

[13] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient algorithms for large-scale local triangle counting. *TKDD*, 4(3), 2010.

[14] 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.

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

[16] V. Braverman and R. Ostrovsky. Smooth histograms for sliding windows. In *IEEE Symposium on Foundations of Computer Science*, pages 283–293, 2007.

[17] V. Braverman, R. Ostrovsky, and D. Vilenchik. How hard is counting triangles in the streaming model? In *International Colloquium on Automata, Languages and Programming*, pages 244–254, 2013.

[18] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *ACM Symposium on Principles of Database Systems*, pages 253–262, 2006.

[19] A. Chakrabarti, G. Cormode, and A. McGregor. Robust lower bounds for communication and stream computation. In *ACM Symposium on Theory of Computing*, pages 641–650, 2008.

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

[21] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *ACM Symposium on Principles of Database Systems*, pages 271–282, 2005.

[22] M. S. Crouch, A. McGregor, and D. Stubbs. Dynamic graphs in the sliding-window model. In *European Symposium on Algorithms*, pages 337–348, 2013.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] A. Goel, M. Kapralov, and S. Khanna. On the communication and streaming complexity of maximum bipartite matching. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 468–485, 2012.

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

[32] O. Goldreich. Introduction to testing graph properties. In O. Goldreich, editor, *Studies in Complexity and Cryptography*, volume 6650 of *Lecture Notes in Computer Science*, pages 470–506. Springer, 2011.

[33] V. Guruswami and K. Onak. Superlinear lower bounds for multipass graph processing. In *IEEE Conference on Computational Complexity*, pages 287–298, 2013.

[34] B. V. Halldórsson, M. M. Halldórsson, E. Losievskaja, and M. Szegedy. Streaming algorithms for independent sets. In *International Colloquium on Automata, Languages and Programming*, pages 641–652, 2010.

[35] M. M. Halldórsson, X. Sun, M. Szegedy, and C. Wang. Streaming and communication complexity of clique approximation. In *International Colloquium on Automata, Languages and Programming*, pages 449–460, 2012.

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

[37] M. Jha, C. Seshadhri, and A. Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *KDD*, pages 589–597, 2013.

[38] M. Jha, C. Seshadhri, and A. Pinar. When a graph is not so simple: Counting triangles in multigraph streams. *CoRR*, arXiv:1310.7665, 2013.

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

[40] H. Jowhari, M. Saglam, and G. Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *ACM Symposium on Principles of Database Systems*, pages 49–58, 2011.

[41] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun. Counting arbitrary subgraphs in data streams. In *International Colloquium on Automata, Languages and Programming*, pages 598–609, 2012.

[42] M. Kapralov. Better bounds for matchings in the streaming model. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1679–1697, 2013.

[43] M. Kapralov, S. Khanna, and M. Sudan. Approximating matching size from random streams. In *ACM-SIAM Symposium on Discrete Algorithms*, 2014.

[44] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1131–1142, 2013.

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

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

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

[48] C. Konrad and A. Rosén. Approximating semi-matchings in streaming and in two-party communication. In *International Colloquium on Automata, Languages and Programming*, pages 637–649, 2013.

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

[50] M. Manjunath, K. Mehlhorn, K. Panagiotou, and H. Sun. Approximate counting of cycles in streams. In *European Symposium on Algorithms*, pages 677–688, 2011.

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

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

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

[54] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. In *International Conference on Very Large Data Bases*, 2013.

[55] J. M. Phillips, E. Verbin, and Q. Zhang. Lower bounds for number-in-hand multiparty communication complexity, made easy. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 486–501, 2012.

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

[57] A. D. Sarma, R. J. Lipton, and D. Nanongkai. Best-order streaming model. *Theor. Comput. Sci.*, 412(23):2544–2555, 2011.

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

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

[60] R. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, 1983.

[61] A. B. Varadaraja. Buyback problem - approximate matroid intersection with cancellation costs. In *International Colloquium on Automata, Languages and Programming*, pages 379–390, 2011.

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