Dynamic Graphs in the Sliding-Window Model*

Michael S. Crouch, Andrew McGregor, and Daniel Stubbs

University of Massachusetts Amherst
140 Governors Drive, Amherst, MA 01003
{mcc,mcgregor,dstubbs}@cs.umass.edu

Abstract. We present the first algorithms for processing graphs in the sliding-window model. The sliding window model, introduced by Datar et al. (SICOMP 2002), has become a popular model for processing infinite data streams in small space when older data items (i.e., those that predate a sliding window containing the most recent data items) are considered "stale" and should implicitly be ignored. While processing massive graph streams is an active area of research, it was hitherto unknown whether it was possible to analyze graphs in the sliding-window model. We present an extensive set of positive results including algorithms for constructing basic graph synopses like combinatorial sparsifiers and spanners as well as approximating classic graph properties such as the size of a graph matching or minimum spanning tree.

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; papers and citations; IP addresses and network flows; phone numbers and phone calls; Tweeters and their followers. Graphs have become the de facto standard for representing many types of highly-structured data. Furthermore, many interesting graphs are dynamic, e.g., hyperlinks are added and removed, citations hopefully accrue over time, and the volume of network traffic between two IP addresses may vary depending on the time of day.

Consequently there is a growing body of work on designing algorithms for analyzing dynamic graphs. This includes both traditional data structures where the goal is to enable fast updates and queries [16, 22–24, 29] and data stream algorithms where the primary goal is to design randomized data structures of *sublinear size* that can answer queries with high probability [2, 3, 17, 18, 25, 27, 30]. The paper focuses on the latter: specifically, processing graphs using sublinear space in the sliding-window model. Although our focus isn't on update time, many of our algorithms can be made fast by using standard data structures.

Dynamic Graph Streams. Almost all of the existing work on processing graph streams considers what is sometimes referred to as the *partially-dynamic* case where the stream consists of a sequence of edges $\langle e_1, e_2, e_3, \ldots \rangle$ and the graph being monitored consists of the set of edges that have arrived so far. In other words, the graph is formed by a sequence of edge insertions. Over the last decade, it has been shown that many interesting

* Supported by NSF CAREER Award CCF-0953754 and associated REU supplement.

problems can be solved using $O(n \operatorname{polylog} n)$ space, where n is the number of nodes in the graph. This is referred to as the *semi-streaming* space restriction [18]. It is only in the last year that semi-streaming algorithms for the *fully-dynamic* case, where edges can be inserted and deleted, have been discovered [2, 3]. A useful example illustrating why the fully-dynamic case is significantly more challenging is testing whether a graph is connected. If there are only insertions, it suffices to track which nodes are in which connected component since these components only merge over time. In the dynamic case, connected components may also subdivide if bridge edges are deleted.

Sliding-Window Model. The sliding-window model, introduced by Datar et al. [14], has become a popular model for processing infinite data streams in small space when the goal is to compute properties of data that has arrived in the last window of time. Specifically, given an infinite stream of data $\langle a_1, a_2, \dots \rangle$ and a function f, at time t we need to return an estimate of $f(a_{t-L+1}, a_{t-L+2}, \dots, a_t)$. We refer to $\langle a_{t-L+1}, a_{t-L+2}, \dots, a_t \rangle$ as the *active* window where L is length of this window. The length of the window could correspond to hours, days, or years depending on the application. The motivation is that by ignoring data prior to the active window, we focus on the "freshest" data and can therefore detect anomalies and trends more quickly. Existing work has considered estimating various numerical statistics and geometric problems in this model [5–8,11,12,19], as well developing useful techniques such as the *exponential histogram* [14] and *smooth histogram* data structures [11, 12].

1.1 Our Contributions

The paper initiates the study of processing graphs in the sliding-window model where the goal is to monitor the graph described by the last L entries of a stream of inserted edges. Note the following differences between this model and fully-dynamic model. In the sliding-window model the edge deletions are *implicit*, in the sense that when an edge leaves the active window it is effectively deleted but we may not know the identity of the deleted edge unless we store the entire window. In the case of fully-dynamic graph streams, the identity of the deleted edge is *explicit* but the edge could correspond to any of the edges already inserted but not deleted.

We present semi-streaming algorithms in the sliding-window model for various classic graph problems including testing connectivity, constructing minimum spanning trees, and approximating the size of graph matchings. We also present algorithms for constructing graph synopses including *sparsifiers* and *spanners*. We say a subgraph H of G is a $(2t-1)$-spanner if:

$$\forall u, v \in V: \quad d_G(u,v) \leq d_H(u,v) \leq (2t-1) d_G(u,v)$$

where $d_G(u,v)$ and $d_H(u,v)$ denote the distance between nodes u and v in G and H respectively. We say a weighted subgraph H of G is a $(1+\epsilon)$ sparsifier if

$$\forall U \subset V: \quad (1-\epsilon)\lambda_G(U) \leq \lambda_H(u,v) \leq (1+\epsilon)\lambda_G(U)$$

where $\lambda_G(U)$ and $\lambda_H(U)$ denote the weight of the cut $(U, V \setminus U)$ in G and H respectively. A summary of our results can be seen in Table 1 along with the state-of-the-art results for these problems in the insert-only and insert/delete models.

	Insert-Only	Insert-Delete	Sliding Window (this paper)
Connectivity	Deterministic [18]	Randomized [2]	Deterministic
Bipartiteness	Deterministic [18]	Randomized [2]	Deterministic
$(1+\epsilon)$-Sparsifier	Deterministic [1]	Randomized [3, 21]	Randomized
$(2t-1)$-Spanners	$O(n^{1+1/t})$ space [9, 15]	None	$O(L^{1/2}n^{(1+1/t)/2})$ space
Min. Spanning Tree	Exact [18]	$(1+\epsilon)$-approx. [2]	$(1+\epsilon)$-approx.
Unweighted Matching	2-approx. [18]	None	$(3+\epsilon)$-approx.
Weighted Matching	4.911-approx. [17]	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 Connectivity and Graph Sparsification

We first consider the problem of 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 $\operatorname{toa}(e)$ for each $e \in F$ where F satisfies the following property:

– *Recent Edges Property.* For every cut $(U, V \setminus U)$, the stored edges F contain the most recent $\min(k, \lambda(U))$ edges across the cut where $\lambda(U)$ denotes the total number of edges from $\{e_1, e_2, \ldots, e_t\}$ that cross the cut.

Then, we can easily tell whether the graph on the active edges, $\langle e_{t-L+1}, e_{t-L+2}, \ldots, e_t \rangle$, is k-connected by checking whether F would be k-connected once we remove all edges $e \in F$ where $\operatorname{toa}(e) \leq t - L$. This follows because if there are k or more edges among the last L edges across a cut, F will include the k most recent of them.

Algorithm. The following simple algorithm maintains a set F with the above property. The algorithm maintains k disjoint sets of edges $F_1, F_2, \ldots, F_k$ where each F_i is acyclic. Initially, $F_1 = F_2 = \ldots = F_k = \emptyset$ and on seeing edge e in the stream, we update the sets as follows:

1. Define the sequence $f_0, f_1, f_2, f_3, \ldots$ where $f_0 = \{e\}$ and for each $i \geq 1$, f_i consists of the oldest edge in a cycle in $F_i \cup f_{i-1}$ if such a cycle exists and $f_i = \emptyset$ otherwise. Since each F_i is acyclic, there will be at most one cycle in each $F_i \cup f_{i-1}$.
2. For $i \in \{1, 2, \ldots, k\}$,
$$F_i \leftarrow (F_i \cup f_{i-1}) \setminus f_i$$

In other words, we add the new edge e to F_1. If it completes a cycle, we remove the oldest edge on this cycle and add that edge to F_2. If we now have a cycle in F_2, we remove the oldest edge on this cycle and add that edge to F_3. And so on. By using an existing data structure for constructing online minimum spanning trees [28], the above algorithm can be implemented with $O(k \log n)$ update time.

Lemma 1. *$F = F_1 \cup F_2 \cup \ldots \cup F_k$ satisfies the Recent Edges Property.*

Proof. Fix some $i \in [k]$ and a cut $(U, V \setminus U)$. Observe that the youngest edge $y \in F_i$ crossing a cut $(U, V \setminus U)$ is never removed from F_i since its removal would require it to be the oldest edge in some cycle C. This cannot be the case since there must be an even number of edges in C that cross the cut and so there is another edge $x \in C$ crossing the cut. This edge must have been older than y since y was the youngest.

It follows that F_1 always contains the youngest edge crossing any cut, and by induction on i, the ith youngest edge crossing any cut is contained in $\bigcup_{j=1}^{i} F_j$. This is true because this edge was initially added to $F_1 \subseteq \bigcup_{j=1}^{i} F_j$, and cannot leave $\bigcup_{j=1}^{i} F_j$. That is, for the ith youngest edge to leave F_i, there would have to be a younger crossing edge in F_i, but, inductively, any such edge is contained in $\bigcup_{j=1}^{i-1} F_j$. □

Theorem 1. *There exists a sliding-window algorithm for monitoring k-connectivity using $O(kn \log n)$ space.*

2.1 Applications: Bipartiteness and Graph Sparsification

Bipartiteness. To monitor whether a graph is bipartite, we run the connectivity tester on the input graph and also simulate the connectivity tester on the *cycle double cover* of the input graph. The cycle double cover $D(G)$ of a graph $G = (V, E)$ is formed by replacing each node $v \in V$ by two copies v_1 and v_2 and each edge $(u, v) \in E$ by the edges (u_1, v_2) and (u_2, v_1). Note that this transformation can be performed in a streaming fashion. Furthermore, $D(G)$ has exactly twice the number of connected components as G iff G is bipartite [2].

Theorem 2. *There exists a sliding-window algorithm for monitoring bipartiteness using $O(n \log n)$ space.*

Graph Sparsification. Using the k-connectivity tester as a black-box we can also construct a $(1 + \epsilon)$-sparsifier following the approach of Ahn et al. [3]. The approach is based upon a result by Fung et al. [20] that states that sampling each edge e with probability $p_e \geq \min\left\{253\lambda_e^{-1}\epsilon^{-2}\log^2 n, 1\right\}$, where λ_e is the size of the minimum cut that includes e, and weighting the sampled edges by $1/p_e$ results in a $(1+\epsilon)$ sparsifier with high probability. To emulate this sampling without knowing λ_e values, we subsample the graph stream to generate sub-streams that define $O(\log n)$ graphs $G_0, G_1, G_2, \ldots$ where each edge is in G_i with probability 2^{-i}. For each i, we store the set of edges $F(G_i)$ generated by the k-connectivity algorithm. If $k = \Theta(\epsilon^{-2}\log^2 n)$, then note that e is in some $F(G_i)$ with probability at least $\min\{\Omega(\lambda_e^{-1}\epsilon^{-2}\log^2 n), 1\}$ as required. See Ahn et al. [3] for further details.

Theorem 3. *There exists a sliding-window algorithm for maintaining a $(1+\epsilon)$ sparsifier using $O(\epsilon^{-2} n \operatorname{polylog} n)$ space.*

3 Matchings

We next consider the problem of finding large matchings in the sliding-window model. We first consider the unweighted case, maximum cardinality matching, and then generalize to the weighted case.

3.1 Maximum Cardinality Matching

Our approach for estimating the size of the maximum cardinality matching combines ideas from the powerful "smooth histograms" technique of Braverman and Ostrovsky [11, 12] with the fact that graph matchings are submodular and satisfy a "smooth-like" condition.

Smooth Histograms. The smooth histogram technique gives a general framework for maintaining an estimate of a function f on a sliding window provided that f satisfies a certain set of conditions. Among these conditions are:

1. *Smoothness:* For any $\alpha \in (0, 1)$ there exists $\beta \in (0, \alpha]$ such that

$$f(B) \geq (1-\beta)f(AB) \quad \text{implies} \quad f(BC) \geq (1-\alpha)f(ABC) \tag{1}$$

   where A, B, and C are disjoint segments of the stream and AB, BC, ABC denote concatenations of these segments.
2. *Approximability:* There exists a sublinear space stream algorithm that returns an estimate $\tilde{f}(A)$ for f evaluated on a (non-sliding-window) stream A, such that

$$(1-\beta/4)f(A) \leq \tilde{f}(A) \leq (1+\beta/4)f(A)$$

The basic idea behind smooth histograms is to approximate f on various suffixes $B_1, B_2, \ldots, B_k$ of the stream where $B_1 \supseteq W \supsetneq B_2 \supsetneq \cdots \supsetneq B_k$ and W is the active window. We refer to the B_i as "buckets." Roughly speaking, if we can ensure that $f(B_{i+1}) \approx (1-\epsilon)f(B_i)$ for each i then $f(B_2)$ is a good approximation for $f(W)$ and we will only need to consider a logarithmic number of suffixes. We will later present the relevant parts of the technique in more detail in the context of approximate matching.

Matchings are Almost Smooth. Let $m(A)$ denote the size of the maximum matching on a set of edges A. Unfortunately, the function m does not satisfy the above smoothness condition and cannot be approximated to sufficient accuracy. It does however satisfy a "smooth-like" condition:

Lemma 2. *For disjoint segments of the stream A, B, and C and for any $\beta > 0$:*

$$m(B) \geq (1-\beta)m(AB) \quad \text{implies} \quad m(BC) \geq \frac{1}{2}(1-\beta)m(ABC) \tag{2}$$

Proof. $2m(BC) \geq m(B)+m(BC) \geq (1-\beta)m(AB)+m(BC) \geq (1-\beta)m(ABC)$. The last step follows since $m(AB) + m(BC) \geq m(A) + m(BC) \geq m(ABC)$. □

The best known semi-streaming algorithm for approximating m on a stream A is a 2-approximation and a lower bound 1.582 has recently been proved [25]. Specifically, let $\hat{m}(A)$ be the size of the greedy matching on A. Then it is easy to show that

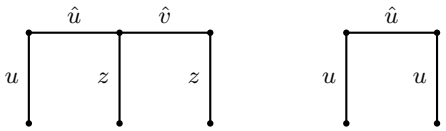
$$m(A) \geq \hat{m}(A) \geq m(A)/2 \tag{3}$$

Unfortunately, it is not possible to maintain a greedy matching over a sliding window.[1] However, by adjusting the analysis of [12], properties (2) and (3) suffice to show that smooth histograms can obtain an $(8+\epsilon)$-approximation of the maximum matching in the sliding-window model. However, by proving a modified smoothness condition that takes advantage of relationships between m and $\hat{m}$, and specifically the fact that $\hat{m}$ is maximal rather than just a 2-approximation, we will show that a smooth histograms-based approach can obtain a $(3+\epsilon)$-approximation.

Lemma 3. *Consider any disjoint segments A, B, C of a stream of edges and $\beta \in (0,1)$.*

$$\hat{m}(B) \geq (1-\beta)\hat{m}(AB) \quad \text{implies} \quad m(ABC) \leq \left(3 + \frac{2\beta}{1-\beta}\right)\hat{m}(BC)\ .$$

Note that it the size of the *maximum* matching on ABC that is being compared with the size of the greedy matching on BC. Also to see that the above lemma is tight for any $\beta \in (0,1)$ consider the following graph on $O(n)$ nodes:



and let A be the stream of the $\hat{u}$ edges (which are placed in greedy matching) followed the u edges; B are the $\hat{v}$ edges, and C are the z edges. Then $\hat{m}(AB) = n$, $\hat{m}(B) = (1-\beta)n = \hat{m}(BC)$, and $m(ABC) = (3-\beta)n$.

Proof (Lemma 3). Let $M(X)$ and $\hat{M}(X)$ be the set of edges in an optimal matching on X and a maximal matching on X. We say that an edge in a matching *covers* the two nodes which are its endpoints.

We first note that every edge in $M(ABC)$ covers at least one node which is covered by $\hat{M}(AB) \cup \hat{M}(BC)$; otherwise, the edge could have been added to $\hat{M}(AB)$ or $\hat{M}(BC)$ or both. Since $M(ABC)$ is a matching, no two of its edges can cover the same node. Thus $m(ABC)$ is at most the number of nodes covered by $\hat{M}(AB) \cup \hat{M}(BC)$.

The number of nodes covered by $\hat{M}(AB) \cup \hat{M}(BC)$ is clearly at most $2\hat{m}(AB) + 2\hat{m}(BC)$. But this over-counts edges in $\hat{M}(B)$. Every edge in $\hat{M}(B)$ is clearly in

[1] Maintaining the matching that would be generated by a greedy algorithm on the active window requires $\Omega(\min(n^2, L))$ space since it would always contain the oldest edge in the window and advancing the window allows us to recover all the edges. Similarly, it is not possible to construct the matching that would be returned by a greedy algorithm on reading the active window in reverse. This can be seen to require $\Omega(n^2)$ space even in the unbounded-stream model via reduction from INDEX. Alice considers the possible edges on an n-clique, and includes an edge iff the corresponding bit of her input is a 1. Bob then adds edges forming a perfect matching on all nodes except the endpoints of an edge of interest. The backwards greedy matching on the resulting graph consists of all of Bob's edges, plus one additional edge iff Alice's corresponding bit was a 1.

$\hat{M}(BC)$; also, every edge in $\hat{M}(B)$ shares at least one node with an edge in $\hat{M}(AB)$ since the construction was greedy. Thus we find

$$\begin{aligned} m(ABC) &\leq 2\hat{m}(BC) + 2\hat{m}(AB) - \hat{m}(B) \\ &\leq 2\hat{m}(BC) + \frac{2}{1-\beta}\hat{m}(B) - \hat{m}(B) \\ &= 2\hat{m}(BC) + \frac{1+\beta}{1-\beta}\hat{m}(B) \\ &\leq \left(3 + \frac{2\beta}{1-\beta}\right)\hat{m}(BC)\ . \end{aligned}$$

where the second inequality follows from the assumption $\hat{m}(B) \geq (1-\beta)\hat{m}(AB)$. □

Theorem 4. *There exists a sliding-window algorithm for maintaining a $(3+\epsilon)$ approximation of the maximum cardinality matching using $O(\epsilon^{-1} n \log^2 n)$ space.*

Proof. We now use the smooth histograms technique to estimate the maximum matching size. 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. Let W be the set of updates within the window. The buckets will always satisfy $B_1 \supseteq W \supsetneq B_2 \supsetneq \cdots \supsetneq B_k$, and thus $m(B_1) \geq m(W) \geq m(B_2)$.

Within each bucket B, we will keep a greedy matching whose size we denote by $\hat{m}(B)$. To achieve small space usage, whenever two nonadjacent buckets have greedy matchings of similar size, we will delete any buckets between them. Lemma 3 tells us that if the greedy matchings of two buckets have ever been close, then the smaller bucket's greedy matching is a good approximation of the size of the maximum matching on the larger bucket.

When a new edge e arrives, we update the buckets $B_1, \ldots, B_k$ and greedy matchings $\hat{m}(B_1), \ldots, \hat{m}(B_k)$ as follows where $\beta = \epsilon/4$:

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

Space Usage: Step 3 deletes "unnecessary" buckets and therefore ensures 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 $O(\epsilon^{-1} n \log^2 n)$.

Approximation Factor: We prove the invariant that for any $i < k$, either $\hat{m}(B_{i+1}) \geq m(B_i)/(3+\epsilon)$ or $|B_i| = |B_{i+1}| + 1$ (i.e., B_{i+1} includes all but the first edge of B_i) 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. But then Lemma 3 implies that 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 $m(W)$. □

3.2 Weighted Matching

We next consider the weighted case where every edge e in the stream is accompanied by a numerical value corresponding to its weight. We combine our algorithm for maximum cardinality matching with the approach of Epstein et al. [17] to give a 9.027 approximation. In this approach, we partition the set of edges into classes of geometrically increasing weights and construct a large cardinality matching in each weight class. We assume that the edge weights are polynomially bounded in n and hence there are $O(\log n)$ weight classes.

Geometrically Increasing Edge Weights. Initially, we assume that for some constants $\gamma > 1, \phi > 0$, every edge has weight $\gamma^i \phi$ for some $i \in \{0, 1, 2, \ldots\}$. Let E_i denote the set of edges with weight $\gamma^i \phi$. Our algorithm will proceed as follows:

1. For each i, use an instantiation of the maximum cardinality algorithm from the previous section to maintain a matching $A_i \subseteq E_i$ among the active edges.
2. Let R be the matching formed by greedily adding all possible edges from $A = \cup_i A_i$ in decreasing order of weight.

The next lemma bounds the total weight of edges in A in terms of the total weight of edges in R.

Lemma 4. $w(R)/w(A) \geq (\gamma - 1)/(\gamma + 1)$.

Note that the lemma is tight: consider the graph with a single edge of weight γ^k, itself adjacent to two edges of each smaller weight $\gamma^{k-1}, \gamma^{k-2}, \ldots$. If $A = E$, we have $w(R) = \gamma^k = (1 - 2/(\gamma+1))w(A)$.

Proof (Lemma 4). Consider the process of greedily constructing R. Call an edge $e \in A$ "chosen" when it is added to R, and "discarded" if some covering edge is added to R. Edges which have not yet been chosen or discarded are said to be "in play". Note that once edges are discarded they cannot be added to R, and that the greedy construction continues until no edges remain in play.

We bound the weight of edges discarded when an edge is chosen. For an edge to be chosen, it must be the heaviest edge in play. None of its in-play neighbors can be in the same weight class, because within each weight class we have a matching. Thus,

when an edge is chosen, the edges discarded are all in smaller weight classes; there are at most two edges discarded in each of these classes. If the edge $e \in A_i$ is chosen, it has weight $\gamma^i\phi$. For each $j < i$ there are at most two edges discarded with weight $\gamma^j\phi$. Let $T(e)$ be the set of edges discarded when $e \in A_i$ is chosen including e itself. Then,

$$\frac{w(e)}{w(T(e))} = \frac{\gamma^i\phi}{\gamma^i\phi + 2\sum_{j=0}^{i-1}\gamma^j\phi} = \frac{\gamma^i}{\gamma^i + 2\gamma^{i-1}\sum_{j=0}^{i-1}\gamma^{-j}} \geq \frac{1}{1+\frac{2}{\gamma-1}} = \frac{\gamma-1}{\gamma+1}$$

Since this holds for each chosen edge and all the edges appear in some $T(e)$, we conclude

$$w(R) = \sum_{e\in R} w(e) \geq \frac{\gamma-1}{\gamma+1}\sum_{e\in A} w(e) = \frac{\gamma-1}{\gamma+1}w(A)$$

as required. □

Let OPT be the maximum-weight matching on E. $w(\text{OPT})$ is clearly at most the sum of the optimum-weight matchings on each E_i. Thus we have

Corollary 1. *If each A_i is an $(3+\epsilon)$ approximation for the maximum cardinality matching on E_i then*

$$w(\text{OPT}) \leq (3+\epsilon)\frac{\gamma+1}{\gamma-1}w(R) \tag{4}$$

Arbitrary Edge Weights. We now reduce the case of arbitrary edge weights to the geometric case. Let OPT be the maximum-weight matching on $G = (V, E, w)$ and let OPT$'$ be the maximum weight matching on $G' = (V, E, w'_\phi)$ where $w'_\phi(e) = \gamma^i\phi$ for some $\gamma > 1, \phi > 0$ and i satisfies $\gamma^{i+1}\phi > w(e) \geq \gamma^i\phi$. This ensures that

$$w(\text{OPT}) < \gamma w'_\phi(\text{OPT}) \leq \gamma w'_\phi(\text{OPT}') \ .$$

However, Epstein et al. show that there exists $\phi \in \{\gamma^{0/q}, \gamma^{1/q}, \gamma^{2/q}, \ldots, \gamma^{1-1/q}\}$ where $q = O(\log_\gamma(1+\epsilon))$ such that

$$w(\text{OPT}) \leq \frac{(1+\epsilon)\gamma\ln\gamma}{\gamma-1}w'_\phi(\text{OPT}) \leq \frac{(1+\epsilon)\gamma\ln\gamma}{\gamma-1}w'_\phi(\text{OPT}') \ .$$

And so, if we run the above algorithm with respect to w'_ϕ in parallel for each choice of ϕ, we ensure that for some ϕ,

$$w(\text{OPT}) \leq \frac{(1+\epsilon)\gamma\ln\gamma}{\gamma-1}w'_\phi(\text{OPT}) \leq (3+\epsilon)\cdot\frac{(1+\epsilon)\gamma\ln\gamma}{\gamma-1}\cdot\frac{\gamma+1}{\gamma-1}w(R) \ ,$$

by appealing to the analysis for geometrically increasing weights (Corollary 1). This is minimized at $\gamma \approx 5.704$ to give an approximation ratio of less than 9.027 when we set ϵ to be some sufficiently small constant.

Theorem 5. *There exists a sliding-window algorithm for maintaining a 9.027 approximation for the maximum weighted matching using $O(n\log^3 n)$ space.*

4 Minimum Spanning Tree

We next consider the problem of maintaining a minimum spanning forest in the sliding-window model. We show that it is possible to maintain a spanning forest that is at most a factor $(1+\epsilon)$ from optimal but that maintaining the exact minimum spanning tree requires $\Omega(\max(n^2, L))$ space where L is the length of the sliding window.

The approximation algorithm is based on an idea of Chazelle et al. [13] where the problem is reduced to finding maximal acyclic subgraphs, i.e., spanning forests, among edges with similar weights. If each edge weight is rounded to the nearest power of $(1+\epsilon)$, it can be shown that the minimum spanning tree in the union of these subgraphs is a $(1+\epsilon)$ approximation of the minimum spanning tree of the original graph. The acyclic subgraphs can be found in the sliding-window model using the connectivity algorithm we presented earlier. The proof of the next theorem is almost identical to those in [2, Lemma 3.4].

Theorem 6. *There exists a sliding-window algorithm for maintaining a $(1+\epsilon)$ approximation for the minimum spanning tree using $O(\epsilon^{-1} n \log^2 n)$ bits of space.*

In the unbounded stream model, it was possible to compute the exact minimum spanning tree via a simple algorithm: 1) add the latest edge to an acyclic subgraph that is being maintained, 2) if this results in a cycle, remove the heaviest weight edge in the cycle. However, the next theorem shows that maintaining an exact minimum spanning tree in the sliding-window model is not possible in sublinear space.

Theorem 7. *Maintaining an exact minimum spanning forest in the sliding-window model requires $\Omega(\min(L, n^2))$ space.*

Proof. Let $p = \min(L, n^2/4)$. The proof is by a reduction from the communication complexity of the two-party communication problem INDEX(p) where Alice holds a binary string $\boldsymbol{a} = a_1 a_2 \ldots a_p$ and Bob has an index $k \in [p]$. If Alice sends a single message to Bob that enables Bob to output a_k with probability at least $2/3$, then Alice's message must contain at $\Omega(p)$ bits [26].

Alice encodes her bits on the edges of a complete bipartite graph, writing in order the edges (u_1, v_1), (u_1, v_2), (u_1, v_3), $\ldots$, $(u_1, v_{\sqrt{p}})$, (u_2, v_1), $\ldots$, $(u_2, v_{\sqrt{p}})$, $\ldots$, $(u_{\sqrt{p}}, v_{\sqrt{p}})$ where the ith edge weight $2i + a_i$. Note that all these edges are in the current active window. Suppose she runs a sliding-window algorithm for exact MST on this graph and sends the memory state to Bob. Bob continues running the algorithm on an arbitrary set of $L-p+k-1$ edges each of weight $2p+2$. At this point any minimum spanning forest in the active window must contain the edge of weight $2k+a_k$ since it is the lowest-cost edge in the graph. Bob can thus learn a_k and hence the algorithm must have used $\Omega(p)$ bits of memory. Note that if Bob can only determine *what* the MST edges are, but not their weights, he can add an alternative path of weight $2k + 1/2$ to the node in question. □

5 Graph Spanners

In the unbounded stream model, the following greedy algorithm constructs a $2t-1$ stretch spanner with $O(n^{1+1/t})$ edges [4, 18]. We process the stream of edges in order;

when seeing each edge (u, v), we add it to the spanner if there is not already a path from u to v of length $2t - 1$ or less. Any path in the original graph then increases by a factor of at most $2t-1$, so the resulting graph is a $(2t-1)$-spanner. The resulting graph has girth at least $2t + 1$, so it has at most $O(n^{1+1/t})$ edges [10].

For graphs G_1, G_2 on the same set of nodes, let $G_1 \cup G_2$ denote the graph with the union of edges from G_1 and G_2. We will need the following simple lemma.

Lemma 5. *Let G_1 and G_2 be graphs on the same set of nodes, and let H_1 and H_2 be α-spanners of G_1 and G_2 respectively. Then $H_1 \cup H_2$ is an α-spanner of $G_1 \cup G_2$.*

Proof. Let $G = G_1 \cup G_2$ and $H = H_1 \cup H_2$. For arbitrary nodes u, v, consider a path of length $d_G(u, v)$. Each edge in this path is in G_1 or G_2 (or both). There is thus a path between the edge's endpoints in the corresponding H_1 or H_2 which is of length at most α. Thus, there is a path of length at most $\alpha d_G(u, v)$ in $H = H_1 \cup H_2$.

Theorem 8. *There exists a sliding-window algorithm for maintaining a $(2t - 1)$ spanner using $O(\sqrt{Ln^{1+1/t}})$ space.*

Proof. We batch the stream into blocks $E_1, E_2, E_3, \ldots$, where each consists of s edges. We buffer the edges in each block until it has been read completely, marking each edge with its arrival time. We then run the greedy spanner construction on each block in reverse order, obtaining a spanner S_i. By Lemma 5, the union of the spanners S_i and the edges in the current block, restricted to the active edges, is a spanner of the edges in the active window. This algorithm requires space s to track the edges in the current block. Each spanner S_i has $O(n^{1+1/t})$ edges, and at most L/s past blocks are within the window. The total number of edges stored by the algorithm is thus $s+(L/s)O(n^{1+1/t})$. Setting $s = \sqrt{Ln^{1+1/t}}$ gives $O(\sqrt{Ln^{1+1/t}})$ edges. □

6 Conclusions

We initiate the study of graph problems in the well-studied sliding-window model. We present algorithms for a wide range of problems including testing connectivity and constructing combinatorial sparsifiers; constructing minimum spanning trees; approximating weighted and unweighted matchings; and estimating graph distances via the construction of spanners. Open problems include reducing the space required to construct graph spanners and improving the approximation ratio when estimating matching size.

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, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *SODA*, pages 459–467, 2012.
3. K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14, 2012.
4. I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.

5. A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.
6. A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004.
7. B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.
8. B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *PODS*, pages 234–243, 2003.
9. S. Baswana. Streaming algorithm for graph spanners—single pass and constant processing time per edge. *Information Processing Letters*, 2008.
10. B. Bollobás. *Extremal graph theory*. Academic Press, New York, 1978.
11. V. Braverman, R. Gelles, and R. Ostrovsky. How to catch ℓ_2-heavy-hitters on sliding windows. In *COCOON*, pages 638–650, 2013.
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. B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, 2005.
14. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. *SIAM Journal on Computing*, 31(6):1794, 2002.
15. M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):1–17, Mar. 2011.
16. D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig. Sparsificationa technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.
17. L. Epstein, A. Levin, J. Mestre, and D. Segev. Improved Approximation Guarantees for Weighted Matching in the Semi-streaming Model. *SIAM Journal on Discrete Mathematics*, 25(3):1251–1265, Jan. 2011.
18. 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, Dec. 2005.
19. J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Algorithmica*, 41(1):25–41, 2004.
20. W. S. Fung, R. Hariharan, N. J. A. Harvey, and D. Panigrahi. A general framework for graph sparsification. In *STOC*, pages 71–80, 2011.
21. A. Goel, M. Kapralov, and I. Post. Single pass sparsification in the streaming model with edge deletions. *CoRR*, abs/1203.4900, 2012.
22. M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.
23. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.
24. G. Italiano, D. Eppstein, and Z. Galil. Dynamic graph algorithms. *Algorithms and Theory of Computation Handbook, CRC Press*, 1999.
25. M. Kapralov. Better bounds for matchings in the streaming model. In *SODA*, pages 1679–1697, 2013.
26. E. Kushilevitz and N. Nisan. *Communication Complexity*, volume 2006. Cambridge University Press, 1997.
27. A. McGregor. Finding graph matchings in data streams. *APPROX-RANDOM*, 2005.
28. R. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, 1983.
29. M. Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC*, pages 343–350, 2000.
30. M. Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012.