2016 IEEE 57th Annual Symposium on Foundations of Computer Science

On Fully Dynamic Graph Sparsifiers

Ittai Abraham*, David Durfee†, Ioannis Koutis‡, Sebastian Krinninger§ and Richard Peng†
*VMware Research
Palo Alto, CA, USA
†Georgia Institute of Technology
Atlanta, GA, USA
‡University of Puerto Rico
Rio Piedras, Puerto Rico
§Max Planck Institute for Informatics
Saarbrücken, Germany

***Abstract*—We initiate the study of fast dynamic algorithms for graph sparsification problems and obtain fully dynamic algorithms, allowing both edge insertions and edge deletions, that take polylogarithmic time after each update in the graph. Our three main results are as follows. First, we give a fully dynamic algorithm for maintaining a $(1\pm\epsilon)$-spectral sparsifier with amortized update time $\mathrm{poly}(\log n, \epsilon^{-1})$. Second, we give a fully dynamic algorithm for maintaining a $(1\pm\epsilon)$-cut sparsifier with *worst-case* update time $\mathrm{poly}(\log n, \epsilon^{-1})$. Both sparsifiers have size $n \cdot \mathrm{poly}(\log n, \epsilon^{-1})$. Third, we apply our dynamic sparsifier algorithm to obtain a fully dynamic algorithm for maintaining a $(1-\epsilon)$-approximation to the value of the maximum flow in an unweighted, undirected, bipartite graph with amortized update time $\mathrm{poly}(\log n, \epsilon^{-1})$.**

***Keywords*-dynamic graph algorithms; sparsification;**

I. Introduction

Problems motivated by graph cuts are well studied in theory and practice. The prevalence of large graphs motivated sublinear time algorithms for cut based problems such as clustering [1]–[6]. In many cases such as social networks or road networks, these algorithms need to run on dynamically evolving graphs. In this paper, we study an approach for obtaining sublinear time algorithms for these problems based on dynamically maintaining graph sparsifiers.

Recent years have seen a surge of interest in dynamic graph algorithms. On the one hand, very efficient algorithms, with polylogarithmic running time per update in the graph, could be found for some key problems in the field [7]–[15]. On the other hand, there are polynomial conditional lower bounds for many basic graph problems [16]–[18]. This leads to the question which problems can be solved with polylogarithmic update time. Another relatively recent trend in graph algorithmics is graph sparsification where we reduce the size of graphs while approximately preserving key properties such as the sizes of cuts [19]. These routines and their extensions to the spectral setting [20], [21] play central roles in a number of recent algorithmic advances [22]–[28], often leading to graph algorithms that run in almost-linear time. In this paper, we study problems at the intersection of dynamic algorithms and graph sparsification, leveraging ideas from both fields.

Full version of this paper available at https://arxiv.org/abs/1604.02094.

At the core of our approach are data structures that dynamically maintain graph sparsifiers in $\mathrm{polylog}\, n$ time per edge insertion or deletion. They are motivated by the spanner based constructions of spectral sparsifiers of Koutis [29]. By modifying dynamic algorithms for spanners [14], we obtain data structures that spend amortized $\mathrm{polylog}\, n$ per update. Our main result for spectral sparsifiers is:

Theorem 1. *Given a graph with polynomially bounded edge weights, we can dynamically maintain a $(1\pm\epsilon)$-spectral sparsifier of size $n \cdot \mathrm{poly}(\log n, \epsilon^{-1})$ with amortized update time $\mathrm{poly}(\log n, \epsilon^{-1})$ per edge insertion / deletion.*

When used as a black box, this routine allows us to run cut algorithms on sparse graphs instead of the original, denser network. Its guarantees interact well with most routines that compute minimum cuts or solve linear systems in the graph Laplacian. Some of them include:

1) min-cuts, sparsest cuts, and separators [30],
2) eigenvector and heat kernel computations [31],
3) approximate Lipschitz learning on graphs [32] and a variety of matrix polynomials in the graph Laplacian [33].

In many applications the full power of spectral sparsifiers is not needed, and it suffices to work with a cut sparsifier. As spectral approximations imply cut approximations, research in recent years has focused spectral sparsification algorithms [34]–[39]. In the dynamic setting however we get a strictly stronger result for cut sparsifiers than for spectral sparsifiers: we can dynamically maintain cut sparsifiers with polylogarithmic *worst-case* update time after each insertion / deletion. We achieve this by generalizing Koutis' sparsification paradigm [29] and replacing spanners with approximate maximum spanning trees in the construction. While there are no non-trivial results for maintaining spanners with worst-case update time, spanning trees can be maintained with polylogarithmic worst-case update time by a recent

0272-5428/16 $31.00 © 2016 IEEE
DOI 10.1109/FOCS.2016.44

breakthrough result [9]. This allows us to obtain the following result for cut sparsifiers:

Theorem 2. *Given a graph with polynomially bounded edge weights, we can dynamically maintain a $(1\pm\epsilon)$-cut sparsifier of size $n \cdot \mathrm{poly}(\log n, \epsilon^{-1})$ with worst-case update time $\mathrm{poly}(\log n, \epsilon^{-1})$ per edge insertion / deletion.*

We then explore more sophisticated applications of dynamic graph sparsifiers. A key property of these sparsifiers is that they have arboricity $\mathrm{polylog}\, n$. This means the sparsifier is locally sparse, and can be represented as a union of spanning trees. This property is becoming increasingly important in recent works [11], [40]: Peleg and Solomon [40] gave data structures for maintaining approximate maximum matchings on fully dynamic graphs with amortized cost parameterized by the arboricity of the graphs. We demonstrate the applicability of our data structures for designing better data structures on the undirected variant of the problem. Through a two-stage application of graph sparsifiers, we obtain the first non-separator based approach for dynamically maintaining $(1-\epsilon)$-approximate maximum flow on fully dynamic graphs:

Theorem 3. *Given a dynamically changing unweighted, undirected, bipartite graph $G = (A, B, E)$ with demand -1 on every vertex in A and demand 1 on every vertex in B, we can maintain a $(1-\epsilon)$-approximation to the value of the maximum flow, as well as query access to the associated approximate minimum cut, with amortized update time $\mathrm{poly}(\log n, \epsilon^{-1})$ per edge insertion / deletion.*

To obtain this result we give stronger guarantees for vertex sparsification in bipartite graphs, identical to the terminal cut sparsifier question addressed by Andoni, Gupta, and Krauthgamer [41]. Our new analysis profits from the ideas we develop by going back and forth between combinatorial reductions and spectral sparsification. This allows us to analyze a vertex sampling process via a mirror edge sampling process, which is in turn much better understood.

Overall, our algorithms bring together a wide range of tools from data structures, spanners, and randomized algorithms. We will provide more details on our routines, as well as how they relate to existing combinatorial and probabilistic tools in Section III.

Proofs to all claims made in this paper can be found in the full version.

II. Background

A. Dynamic Graph Algorithms

In this paper we consider undirected graphs $G = (V, E)$ with n vertices and m edges that are either unweighted or have non-negative edge weights. We denote the weight of an edge $e = (u, v)$ in a graph G by $w_G(e)$ or $w_G(u, v)$ and the ratio between the largest and the smallest edge weight by W. The weight $w_G(F)$ of a set of edges $F \subseteq E$ is the sum of the individual edge weights. We will assume that all weights are polynomially bounded because there are standard reductions from the general case using minimum spanning trees (e.g. [42] Section 10.2., [43] Theorem 5.2). Also, these contraction schemes in the data structure setting introduces another layer of complexity akin to dynamic connectivity, which we believe is best studied separately.

A *dynamic algorithm* is a data structure for dynamically maintaining the result of a computation while the underlying input graph is updated periodically. We consider two types of updates: edge insertions and edge deletions. An *incremental* algorithm can handle only edge insertions, a *decremental* algorithm can handle only edge deletions, and a *fully dynamic* algorithm can handle both edge insertions and deletions. After every update in the graph, the dynamic algorithm is allowed to process the update to compute the new result. For the problem of maintaining a sparsifier, we want the algorithm to output the changes to the sparsifier (i.e., the edges to add to or remove from the sparsifier) after every update in the graph.

B. Running Times and Success Probabilities

The running time spent by the algorithm after every update is called *update time*. We distinguish between *amortized* and *worst-case* update time. A dynamic algorithm has amortized update time $T(m, n, W)$, if the total time spent after q updates in the graph is at most $qT(m, n, W)$. A dynamic algorithm has worst-case update time $T(m, n, W)$, if the total time spent after *each* update in the graph is at most $T(m, n, W)$. Here m refers to the maximum number of edges ever contained in the graph. All our algorithms are randomized.

The guarantees we report in this paper (quality and size of sparsifier, and update time) will hold *with high probability (w.h.p.)*, i.e. with probability at least $1 - 1/n^c$ for some arbitrarily chosen constant $c \geq 1$. These bounds are against an *oblivious adversary* who chooses its sequence of updates independently from the random choices made by the algorithm. Formally, the oblivious adversary chooses its sequence of updates before the algorithm starts. In particular, this means that the adversary is not allowed to see the current edges of the sparsifier. As our composition of routines involve $\mathrm{poly}(n)$ calls, we will assume the composability of these w.h.p. bounds.

Most of our update costs have the form $O(\log^{O(1)} n\epsilon^{-O(1)})$, where ϵ is the approximation error. We will often state these as $\mathrm{poly}(\log n, \epsilon^{-1})$.

C. Cuts and Laplacians

A *cut* $U \subseteq V$ of G is a subset of vertices whose removal makes G disconnected. We denote by $\partial_G(U)$ the edges crossing the cut U, i.e., the set of edges with one endpoint in U and one endpoint in $V \setminus U$. The weight of the cut U is $w_G(\partial_G(U))$. An *edge cut* $F \subseteq E$ of G is a a subset of

edges whose removal makes G disconnected and the weight of the edge cut F is $w_G(F)$. For every pair of vertices u and v, the *local edge connectivity* $\lambda_G(u,v)$ is the weight of the minimum edge cut separating u and v. If G is unweighted, then $\lambda_G(u,v)$ amounts to the number of edges that have to be removed from G to make u and v disconnected.

Assuming some arbitrary order $v_1, \dots v_n$ on the vertices, the *Laplacian matrix* $\mathcal{L}_G$ of an undirected graph G is the $n \times n$ matrix that in row i and column j contains the negated weight $-w_G(v_i, v_j)$ of the edge (v_i, v_j) and in the i-th diagonal entry contains the weighted degree $\sum_{j=1}^{n} w_G(v_i, v_j)$ of vertex v_i. Note that Laplacian matrices are symmetric. The matrix $\mathcal{L}_e$ of an edge e of G is the $n \times n$ Laplacian matrix of the subgraph of G containing only the edge e. It is 0 everywhere except for a 2×2 submatrix.

For studying the spectral properties of G we treat the graph as a resistor network. For every edge $e \in E$ we define the *resistance* of e as $r_G(e) = 1/w_G(e)$. The *effective resistance* $R_G(e)$ of an edge $e = (v,u)$ is defined as the potential difference that has to be applied to u and v to drive one unit of current through the network. A closed form expression of the effective resistance is $R_G(e) = b_{u,v}^{\top} \mathcal{L}_G^{\dagger} b_{u,v}$, where $\mathcal{L}_G^{\dagger}$ is the Moore-Penrose pseudo-inverse of the Laplacian matrix of G and $b_{u,v}$ is the n-dimensional vector that is 1 at position u, -1 at position v, and 0 otherwise.

D. Graph Approximations

The goal of graph sparsification is to find sparse subgraphs, or similar small objects, that approximately preserve certain metrics of the graph. We first define spectral sparsifiers where we require that Laplacian quadratic form of the graph is preserved approximately. Spectral sparsifiers play a pivotal role in fast algorithms for solving Laplacian systems, a special case of linear systems.

Definition 4. *A $(1 \pm \epsilon)$-spectral sparsifier H of a graph G is a subgraph of G with weights w_H such that for every vector $x \in \mathbb{R}^n$*

$$(1-\epsilon) x^{\top} \mathcal{L}_H x \leq x^{\top} \mathcal{L}_G x \leq (1+\epsilon) x^{\top} \mathcal{L}_H x \,.$$

Using the Loewner ordering on matrices this condition can also be written as $(1-\epsilon)\mathcal{L}_H \preceq \mathcal{L}_G \preceq (1+\epsilon)\mathcal{L}_H$. An $n \times n$ matrix $\mathcal{A}$ is *positive semi-definite*, written as $\mathcal{A} \succeq 0$, if $x^{\top} \mathcal{A} x \geq 0$ for all $x \in \mathbb{R}^n$. For two $n \times n$ matrices $\mathcal{A}$ and $\mathcal{B}$ we write $\mathcal{A} \succeq \mathcal{B}$ as an abbreviation for $\mathcal{A} - \mathcal{B} \succeq 0$.

Note that $x^{\top} \mathcal{L}_G x = \sum_{(u,v) \in E} w(u,v)(x(u) - x(v))^2$ where the vector x is treated as a function on the vertices and $x(v)$ is the value of x for vertex v. A special case of such a function on the vertices is given by the binary indicator vector x_U associated with a cut U, where $x_U(v) = 1$ is $v \in U$ and 0 otherwise. If limited to such indicator vectors, the sparsifier approximately preserves the value of every cut.

Definition 5. *A $(1 \pm \epsilon)$-cut sparsifier H of a graph G is a subgraph of G with weights w_H such that for every subset $U \subseteq V$*

$$(1-\epsilon) w_H(\partial_H(U)) \leq w_G(\partial_G(U)) \leq (1+\epsilon) w_H(\partial_H(U)) \,.$$

E. Spanning Trees and Spanners

A *spanning forest* F of G is a forest (i.e., acyclic graph) on a subset of the edges of G such that every pair of vertices that is connected in G is also connected in F. A minimum/maximum spanning forest is a spanning forest of minimum/maximum total weight.

For every pair of vertices u and v we denote by $d_G(u,v)$ the distance between u and v (i.e., the length of the shortest path connecting u and v) in G with respect to the resistances. The graph sparsification concept also exists with respect to distances in the graph. Such sparse subgraphs that preserves distances approximately are called spanners.

Definition 6. *A* spanner of stretch α, *or short* α-spanner, *(where $\alpha \geq 1$) of an undirected (possibly weighted) graph G is a subgraph H of G such that, for every pair of vertices u and v, $d_H(u,v) \leq \alpha d_G(u,v)$.*

III. Overview and Related Work

A. Dynamic Spectral Sparsifier

We first develop a fully dynamic algorithm for maintaining a spectral sparsifier of a graph with polylogarithmic amortized update time.

Related Work: Spectral sparsifiers play important roles in fast numerical algorithms [21]. Spielman and Teng were the first to study these objects [20]. Their algorithm constructs a $(1 \pm \epsilon)$-spectral sparsifier of size $O(n \cdot \mathrm{poly}(\log n, \epsilon^{-1}))$ in nearly linear time. This result has seen several improvements in recent years [37], [42], [44], [45]. The state of the art in the sequential model is an algorithm by Lee and Sun [38] that computes a $(1 \pm \epsilon)$-spectral sparsifier of size $O(n\epsilon^{-2})$ in nearly linear time. Most closely related to the data structural question are streaming routines, both in one pass incremental [34], and turnstile [36], [46], [47].

A survey of spectral sparsifier constructions is given in [21]. Many of these methods rely on solving linear systems built on the graph, for which there approaches with a combinatorial flavor using low-stretch spanning trees [48], [49] and purely numerical solvers relying on sparsifiers [25] or recursive constructions [27]. We build on the spectral sparsifier obtained by a simple, combinatorial construction of Koutis [29], which initially was geared towards parallel and distributed implementations.

Sparsification Framework: In our framework we determine 'sampleable' edges by using spanners to compute a set of edges of bounded effective resistance. From these edges we then sample by coin flipping to obtain a (moderately sparser) spectral sparsifier in which the number of edges has been reduced by a constant fraction. This step can then be

iterated a small number of times in order to compute the final sparsifier.

Concretely, we define a t-bundle spanner $B = T_1 \cup \cdots \cup T_t$ (for a suitable, polylogarithmic, value of t) as a sequence of spanners $T_1, \ldots, T_t$ where the edges of each spanner are removed from the graph before computing the next spanner, i.e., T_1 is a spanner of G, T_2 is a spanner of $G \setminus T_1$, etc; here each spanner has stretch $O(\log n)$. We then sample each non-bundle edge in $G \setminus B$ with some constant probability p and scale the edge weights of the sampled edges proportionally. The t-bundle spanner serves as a certificate for small resistance of the non-bundle edges in $G \setminus B$ as it guarantees the presence of t disjoint paths of length at most the stretch of the spanner. Using this property one can apply matrix concentration bounds [50] to show the t-bundle together with the sampled edges is a moderately sparse spectral sparsifier. We repeat this process of 'peeling off' a t-bundle from the graph and sampling from the remaining edges until the graph is sparse enough (which happens after a logarithmic number of iterations). Our final sparsifier consists of all t-bundles together with the sampled edges of the last stage. Figure 1 gives an overview for this spectral sparsifier construction.

Towards a Dynamic Algorithm: To implement the spectral sparsification algorithm in the dynamic setting we need to dynamically maintain a t-bundle spanner. Our approach to this problem is to run t different instances of a dynamic spanner algorithm, in order to separately maintain a spanner T_i for each graph $G_i = G \setminus \bigcup_{j=1}^{i-1} T_j$, for $1 \leq i \leq t$.

Baswana, Khurana, and Sarkar [14] gave a fully dynamic algorithm for maintaining a spanner of stretch $O(\log n)$ and size $O(n \log^2 n)$ with polylogarithmic update time.[1] A natural first idea would be to use this algorithm in a black-box fashion in order to separately maintain each spanner of a t-bundle. However, we do not know how to do this because of the following obstacle. A single update in G might lead to several changes of edges in the spanner T_1, an average of $\Omega(\log n)$ according to the amortized upper bound. This means that the next instance of the fully dynamic spanner algorithm which is used for maintaining T_2, not only has to deal with the deletion in G but also the artificially created updates in $G_2 = G \setminus T_1$. This of course propagates to more updates in all graphs G_i. Observe also that any given update in G_t caused by an update in G, can be requested *repeatedly*, as a result of subsequent updates in G. Without further guarantees, it seems that with this approach we can only hope for an upper bound of $O(\log^{t-1} n)$ (on average) on the number of changes to be processed for updating G_t after a single update in G. That is too high because the sparsification algorithm requires us to take $t = \Omega(\log n)$. Our solution to this problem lies in a substantial modification of the dynamic spanner algorithm in [14] outlined below.

Dynamic Spanners with Monotonicity: The spanner algorithm of Baswana et al. [14] is at its core a decremental algorithm (i.e., allowing only edge deletions in G), which is subsequently leveraged into a fully dynamic algorithm by a black-box reduction. We follow the same approach by first designing a decremental algorithm for maintaining a t-bundle spanner. This is achieved by modifying the decremental spanner algorithm so that, in addition to its original guarantees, it has the following **monotonicity** property:

Every time an edge is added to the spanner T, it stays in T until it is deleted from G.

The algorithm is outlined in Figure 2

Recall that we initially want to maintain a t-bundle spanner $T_1, \ldots, T_t$ under edge deletions only. In general, whenever an edge is added to T_1, it will cause its deletion from the graph $G \setminus T_1$ for which the spanner T_2 is maintained. Similarly, removing an edge from T_1 causes its insertion into $G \setminus T_1$, *unless* the edge is deleted from G. This is precisely what the monotonicity property guarantees: that an edge will not be removed from T_1 unless deleted from G. The consequence is that no edge insertion can occur for $G_2 = G \setminus T_1$. Inductively, no edge is ever inserted into G_i, for each i. Therefore the algorithm for maintaining the spanner T_i only has to deal with edge deletions from the graph G_i, thus it becomes possible to run a different instance of the same decremental spanner algorithm for each G_i. A single deletion from G can still generate many updates in the bundle. But for each i, the instance of the dynamic spanner algorithm working on G_i can only delete each edge *once*. Furthermore, we only run a small number t of instances. So the total number of updates remains bounded, allowing us to claim the upper bound on the amortized update time.

In addition to the modification of the dynamic spanner algorithm, we have also deviated from Koutis' original scheme [29] in that we explicitly 'peel off' each iteration's bundle from the graph. In this way we avoid that the t-bundles from different iterations share any edges, which seems hard to handle in the decremental setting we ultimately want to restrict ourselves to.

The modified spanner algorithm now allows us to maintain t-bundles in polylogarithmic update time, which is the main building block of the sparsifier algorithm. The remaining parts of the algorithm, like sampling of the non-bundle edges by coin-flipping, can now be carried out in the straightforward way in polylogarithmic amortized update time. At any time, our modified spanner algorithm can work in a purely decremental setting. As mentioned above, the fully dynamic sparsifier algorithm is then obtained by a reduction from the decremental sparsifier algorithm.

[1] More precisely, they gave two fully dynamic algorithms for maintaing a $(2k-1)$-spanner for any integer $k \geq 2$: The first algorithm guarantees a spanner of expected size $O(kn^{1+1/k} \log n)$ and has expected amortized update time $O(k^2 \log^2 n)$ and the second algorithm guarantees a spanner of expected size $O(k^8 n^{1+1/k} \log^2 n)$ and has expected amortized update time $O(7^{k/2})$.

LIGHT-SPECTRAL-SPARSIFY (G, ϵ)

1) $t \leftarrow \lceil 12(c+1)\alpha\epsilon^{-2}\ln n \rceil$ for some absolute constant c.
2) let $B = \bigcup_{j=1}^{t} T_j$ be a t-bundle α-spanner of G
3) $H := B$
4) **for each** edge $e \in G \setminus B$
   a) with probability $1/4$: add e to H with $w_H(e) \leftarrow 4w_G(e)$
5) **return** (H, B)

SPECTRAL-SPARSIFY (G, c, ϵ)

1) $k \leftarrow \lceil \log \rho \rceil$
2) $G_0 \leftarrow G$
3) $B_0 \leftarrow (V, \emptyset)$
4) **for** $i = 1$ **to** k
   a) $(H_i, B_i) \leftarrow$ LIGHT-SPECTRAL-SPARSIFY$(G_{i-1}, c, \epsilon/(2k))$
   b) $G_i \leftarrow H_i \setminus B_i$
   c) **if** G_i has less than $(c+1)\ln n$ edges **then break** (* break loop *)
5) $H \leftarrow \bigcup_{1 \le j \le i} B_j \cup G_i$
6) **return** $(H, \{B_j\}_{j=1}^{i}, G_i)$

Figure 1. Our spectral sparsifier is obtained by running the procedures LIGHT-SPECTRAL-SPARSIFY and SPECTRAL-SPARSIFY. In our dynamic algorithm we follow the same scheme and implement both procedures dynamically. To implement LIGHT-SPECTRAL-SPARSIFY, we in particular dynamically maintain the t-bundle α-spanner B which results in a dynamically changing graph $G \setminus B$. To implement SPECTRAL-SPARSIFY, we dynamically maintain each H_i and B_i as the result of a dynamic implementation of LIGHT-SPECTRAL-SPARSIFY which results in dynamically changing graphs G_i.

DECREMENTAL-SPANNER

1) $k \leftarrow \lceil \log n \rceil$
2) Create sequence of sets $V = S_0 \supseteq S_1 \supseteq \ldots \supseteq S_k = \emptyset$: add each vertex of S_i to S_{i+1} independently with probability $n^{-1/k}$.
3) Pick random permutation σ of vertices in V
4) For every $1 \le i \le k$, maintain clustering C_i: Every vertex within distance i to vertices in S_i is assigned to cluster $C_i[s]$ of closest vertex $s \in S_i$; ties are broken according to permutation σ
5) For every $1 \le i \le k$, maintain forest F_i containing, for every $s \in S_i$, shortest path tree from s to all vertices in $C_i[s]$ such that every vertex v chooses parent that comes first in permutation σ among all candidates (i.e., among the vertices that are in the same cluster $C_i[s]$ as v and that are at distance $d(v,s) - 1$ from s)
6) Maintain spanner H containing following two types of edges
   a) For every $1 \le i \le k$, H contains all edges of forest F_i
   b) For every $1 \le i \le k$, every vertex v contained in some cluster $C_i[s]$, and every neighboring cluster $C_i[s']$ of v, H contains *one* edge to $C_i[s']$, i.e., one edge (v, v') such that $v' \in C_i[s']$
7) Every time an edge is added to H it is kept in H until deleted from G

Figure 2. Decremental algorithm for maintaining spanner with monotonicity property. We show in the full version of this paper that this algorithm has a total update time of $O(m \operatorname{polylog} n)$ over all deletions and maintains a spanner of stretch $O(\log n)$ and size $O(n \operatorname{polylog} n)$.

B. Dynamic Cut Sparsifier

We also give dynamic algorithms for maintaining a $(1 \pm \epsilon)$-cut sparsifier. We obtain a fully dynamic algorithm with polylogarithmic worst-case update time by leveraging a recent worst-case update time algorithm for dynamically maintaining a spanning tree of a graph [9]. As mentioned above, spectral sparsifiers are more general than cut sparsifiers. The big advantage of studying cut sparsification as a separate problem is that we can achieve polylogarithmic *worst-case* update time, where the update time guarantee holds for each individual update and is *not* amortized over a sequence of updates.

Related Work: In the static setting, Benczúr and Karger [19] developed an algorithm for computing a $(1 \pm \epsilon)$-cut sparsifier of size $O(n \cdot \operatorname{poly}(\log n, \epsilon^{-1}))$ in nearly linear

time. Their approach is to first compute a value called *strength* for each edge and then sampling each edge with probability proportional to its strength. Their proof uses a cut-counting argument that shows that the majority of cuts are large, and therefore less likely to deviate from their expectation. A union bound over these (highly skewed) probabilities then gives the overall w.h.p. success bound. This approach was refined by Fung et al. [51] who show that a cut sparsifier can also be obtained by sampling each edge with probability inversely proportional to its (approximate) local edge connectivity, giving slightly better guarantees on the sparsifier. The work of Kapron, King, and Mountjoy [9] contains a fully dynamic approximate "cut oracle" with worst-case update time $O(\log^2 n)$. Given a set $U \subseteq V$ as the input of a query, it returns a 2-approximation to the number of edges in $U \times V \setminus U$ in time $O(|U| \log^2 n)$. The cut sparsifier question has also been studied in the (dynamic) streaming model [52]–[54].

Our Framework: The algorithm is based on the observation that the spectral sparsification scheme outlined above in Section III-A. becomes a cut sparsification algorithm if we simply replace spanners by maximum weight spanning trees (MSTs). This is inspired by sampling according to edge connectivities; the role of the MSTs is to certify lower bounds on the edge connectivities. We observe that the framework does not require us to use exact MSTs. For our t-bundles we can use a relaxed, approximate concept that we call α-MST that. Roughly speaking, an α-MST guarantees a 'stretch' of α in the infinity norm and, as long as it is sparse, does not necessarily have to be a tree.

Similarly to before, we define a t-bundle α-MST B as the union of a sequence of α-MSTs $T_1, \ldots T_t$ where the edges of each tree are removed from the graph before computing the next α-MST. The role of α-MST is to certify uniform lower bounds on the connectivity of edges; these bounds are sufficiently large to allow uniform sampling with a fixed probability.

This process of peeling and sampling is repeated sufficiently often and our cut sparsifier then is the union of all the t-bundle α-MSTs and the non-bundle edges remaining after taking out the last bundle. Thus, the cut sparsifier consists of a polylogarithmic number of α-MSTs and a few (polylogarithmic) additional edges. This means that for α-MSTs based on spanning trees, our cut sparsifiers are not only sparse, but also have polylogarithmic *arboricity*, which is the minimum number of forests into which a graph can be partitioned. Figure 3 gives an overview for this cut sparsifier construction.

Simple Fully Dynamic Algorithm: Our approach immediately yields a fully dynamic algorithm by using a fully dynamic algorithm for maintaining a spanning forest. Here we basically have two choices. Either we use the randomized algorithm of Kapron, King, and Mountjoy [9] with polylogarithmic worst-case update time. Or we use the deterministic algorithm of Holm, de Lichtenberg, and Thorup [8] with polylogarithmic amortized update time. The latter algorithm is slightly faster, at the cost of providing only amortized update-time guarantees. A t-bundle 2-MST can be maintained fully dynamically by running, for each of the $\log W$ weight classes of the graph, t instances of the dynamic spanning tree algorithm in a 'chain'.

An important observation about the spanning forest algorithm is that with every update in the graph, at most one edge is changed in the spanning forest: If for example an edge is deleted from the spanning forest, it is replaced by another edge, but no other changes are added to the tree. Therefore a single update in G can only cause one update for each graph $G_i = G \setminus \bigcup_{j=1}^{i-1} T_j$ and T_i. This means that each instance of the spanning forest algorithm creates at most one 'artificial' update that the next instance has to deal with. In this way, each dynamic spanning forest instance used for the t-bundle has polylogarithmic update time. As $t = \operatorname{polylog} n$, the update time for maintaining a t-bundle is also polylogarithmic. The remaining steps of the algorithm can be carried out dynamically in the straightforward way and overall give us polylogarithmic worst-case or amortized update time.

A technical detail of our algorithm is that the high-probability correctness achieved by the Chernoff bounds only holds for a polynomial number of updates in the graph. We thus have to restart the algorithm periodically. This is trivial when we are shooting for an amortized update time. For a worst-case guarantee we can neither completely restart the algorithm nor change all edges of the sparsifier in one time step. We therefore keep two instances of our algorithm that maintain two sparsifiers of two alternately growing and shrinking subgraphs that at any time partition the graph. This allows us to take a blend of these two subgraph sparsifiers as our end result and take turns in periodically restarting the two instances of the algorithm.

C. $(1-\epsilon)$-Approximate Undirected Bipartite Flow

We then study ways of utilizing our sparsifier constructions to give routines with truly sublinear update times. The problem that we work with will be maintaining an approximate maximum flow problem on a bipartite graph $G_{A,B} = (A, B, E)$ with demand -1 and 1 on each vertex in A and B, respectively. All edges are unit weight and we dynamically insert and delete edges. The maximum flow minimum cut theorem states that the objective here equals to the minimum $s-t$ cut or maximum $s-t$ flow in G, which will be $G_{A,B}$ where we add vertices s and t, and connect each vertex in A to s and each vertex in B to t. The only dynamic changes in this graph will be in edges between A and B. As our algorithms builds upon cut sparsifiers, and flow sparsifiers [24] are more involved, we will focus on only finding cuts.

LIGHT-CUT-SPARSIFY (G, c, ϵ)

1) $t \leftarrow C_\xi c\alpha \log W \log^2 n/\epsilon^2$
2) Let B be a t-bundle α-MST of G
3) $H := B$
4) **For each** edge $e \in G \setminus B$
   a) With probability $1/4$ add e to H with $4w_H(e) \leftarrow w_G(e)$
5) **Return** (H, B)

CUT-SPARSIFY (G, c, ϵ)

1) $k \leftarrow \lceil \log \rho \rceil$
2) $G_0 \leftarrow G$
3) $B_0 \leftarrow (V, \emptyset)$
4) **for** $i = 1$ **to** k
   a) $(H_i, B_i) \leftarrow$ LIGHT-CUT-SPARSIFY$(G, c + 1, \epsilon/(2k))$
   b) $G_{i+1} \leftarrow H_i \setminus B_i$
   c) **if** G_{i+1} has less than $(c + 2) \ln n$ edges **then break** (* break loop *)
5) $H \leftarrow \bigcup_{1 \le j \le i} B_j \cup G_{i+1}$
6) **return** $(H, \{B_j\}_{j=1}^{i}, G_{i+1})$

Figure 3. Our cut sparsifier is obtained by running the procedures LIGHT-CUT-SPARSIFY and CUT-SPARSIFY. In our dynamic algorithm we follow the same scheme and implement both procedures dynamically. To implement LIGHT-CUT-SPARSIFY, we in particular dynamically maintain the t-bundle α-MST B which results in a dynamically changing graph $G \setminus B$. To implement SPECTRAL-SPARSIFY, we dynamically maintain each H_i and B_i as the result of a dynamic implementation of LIGHT-CUT-SPARSIFY which results in dynamically changing graphs G_i.

This problem is motivated by the dynamic approximate maximum matching problem, which differs in that the edges are directed, and oriented from A to B. This problem has received much attention recently [10]–[12], [40], [55], [56], and led to the key definition of low arboricity graphs [11], [40]. On the other hand, bipartite graphs are known to be difficult to sparsify: the directed reachability matrix from A to B can encode $\Theta(n^2)$ bits of information. As a result, we study the undirected variant of this problem instead, with the hope that this framework can motivate other definitions of sparsification suitable for wider classes of graphs.

Another related line of work are fully dynamic algorithm for maintaining the global minimum cut [57], [58] with update time $O(\sqrt{n}\,\mathrm{polylog}\, n)$. As there are significant differences between approximating global minimum cuts and st-minimum cuts in the static setting [59], we believe that there are some challenges to adapting these techniques for this problem. The data structure by Thorup [57] can either maintain global edge connectivity up to $\mathrm{polylog}\, n$ exactly or, with high probability, arbitrary global edge connectivity with an approximation of $1 + o(1)$. The algorithms also maintain concrete (approximate) minimum cuts, where in the latter algorithm the update time increases to $O(\sqrt{m}\,\mathrm{polylog}\, n)$ (and cut edges can be listed in time $O(\log n)$ per edge). Thorup's result was preceded by a randomized algorithm with worse approximation ratio for the global edge connectivity by Thorup and Karger [58] with update time $O(\sqrt{n}\,\mathrm{polylog}\, n)$.

The problem we have formulated above is shown in the full paper to be different from matching. On the other hand, our incorporation of sparsifiers for maintaining solutions to this problem relies on several properties that hold in a variety of other settings:

1) The static version can be efficiently approximated.
2) The objective can be approximated via graph sparsifiers.
3) A small answer (for which the algorithm's current approximation may quickly become sub-optimal) means the graph also has a small vertex cover.
4) The objective does not change much per each edge update.

As with algorithms for maintaining high quality matchings [40], [55], our approach aims to get a small amortized cost by keeping the same minimum $s - t$ cut for many consecutive dynamic steps. Specifically, if we have a minimum $s - t$ cut of size $(2 + \frac{\epsilon}{2})OPT$, then we know this cut will remain $(2 + \epsilon)$ approximately optimal for $\frac{\epsilon}{2}OPT$ dynamic steps. This allows us to only compute a new minimum $s - t$ cut every $\frac{\epsilon}{2}OPT$ dynamic steps.

As checking for no edges would be an easy boundary case, we will assume throughout all the analysis that $OPT > 0$. To obtain an amortized $O(\mathrm{poly}(\log n, \epsilon^{-1}))$ update cost, it suffices for this computation to take $O(OPT \cdot \mathrm{poly}(\log n, \epsilon^{-1}))$ time. In other words, we need to solve approximate maximum flow on a graph of size $O(OPT \cdot \mathrm{poly}(\log n, \epsilon^{-1}))$. Here we

incorporate sparsifiers using the other crucial property used in matching data structures [10], [40], [55]: if OPT is small, G also has a small vertex cover.

Lemma 7. *The minimum vertex cover in G has size at most $OPT + 2$ where OPT is the size of the minimum $s - t$ cut in G.*

We utilize the low arboricity of our sparsifiers to find a small vertex cover with the additional property that all non-cover vertices have small degree. We will denote this (much) smaller set of vertices as VC. In a manner similar to eliminating vertices in numerical algorithms [27], the graph can be reduced to only edges on VC at the cost of a $(2 + \epsilon)$-approximation. Maintaining a sparsifier of this routine again leads to an overall routine that maintains a $(2 + \epsilon)$-approximation in $\text{polylog}\, n$ time per update.

Sparsifying vertices instead of edges inherently implies that an approximation of all cut values cannot be maintained. Instead, the sparsifier, which will be referred to as a *terminal-cut-sparsifier*, maintains an approximation of all minimum cuts between any two terminal vertices, where the vertex cover is the terminal vertex set for our purposes. More specifically, given a minimum cut between two terminal vertices on the sparsified graph, by adding each independent vertex from the original graph to the cut set it is more connected to, an approximate minimum cut on the original graph is achieved. This concept of *terminal-cut-sparsifier* will be equivalent to that in [41].

The large approximation ratio motivated us to reexamine the sparsification routines, namely the one of reducing the graph to one whose size is proportional to $|VC|$. This is directly related to the terminal cut sparsifiers studied in [41], [60]. However, for an update time of $\text{poly}(\log n, \epsilon^{-1})$, it is crucial for the vertex sparsifier to have size $O(|VC| \,\text{poly}(\log n, \epsilon^{-1}))$. As a result, instead of doing a direct union bound over all $2^{|VC|}$ cuts to get a size of $\text{poly}(|VC|)$ as in [41], we need to invoke cut counting as with cut sparsifier constructions. This necessitates the use of objects similar to t-bundles to identify edges with small connectivity. This leads to a sampling process motivated by the $(2+\epsilon)$-approximate routine, but works on vertices instead of edges.

By relating the processes, we are able to absorb the factor 2 error into the sparsifier size. Here a major technical challenge compared to analyses of cut sparsifiers [51] is that the natural scheme of bucketing by edge weights is difficult to analyze because a sampled vertex could have non-zero degree in multiple buckets. We work around this issue via a pre-processing scheme on G that creates an approximation so that all vertices outside of VC have degree $\text{polylog}\, n$. This scheme is motivated in part by the weighted expanders constructions from [27]. Bucketing after this processing step ensures that each vertex belongs to a unique bucket. In terms of a static sparsifier on terminals, the result that is most comparable to results from previous works is:

Corollary 8. *Given any graph $G = (V, E)$, and a vertex cover VC of G, where $X = V \setminus VC$, with error ϵ, we can build an ϵ-approximate terminal-cut-sparsifier H with $O(|VC| \,\text{poly}(\log n, \epsilon^{-1}))$ vertices in $O(m \cdot \text{poly}(\log n, \epsilon^{-1}))$ work.*

Turning this into a dynamic routine leads to the result described in Theorem 3: a $(1 + \epsilon)$-approximate solution that can be maintained in time $\text{polylog}(n)$ per update. It is important to note that Theorem 2 plays an integral role in extending Corollary 8 to a dynamic routine, particularly the low arboricity property that allows us to maintain a small vertex cover such that all non-cover vertices have low degree.

IV. Discussion

Graph Sparsification: We use a sparsification framework in which we 'peel off' bundles of sparse subgraphs to determine 'sampleable' edges, from which we then sample by coin flipping. This leads to combinatorial and surprisingly straightforward algorithms for maintaining graph sparsifiers. Additionally, this gives us low-arboricity sparsifiers; a property that we exploit for our main application.

Although spectral sparsification is more general than cut sparsification. Our treatment of cut sparsification has two motivations. First, we can obtain stronger running time guarantees. Second, our sparsifier for the $(1+\epsilon)$-approximate maximum flow algorithm on bipartite graphs hinges upon improved routines for vertex sparsification, a concept which leads to different objects in the spectral setting.

Dynamic Graph Algorithms: In our sparsification framework we sequentially remove bundles of sparse subgraphs to determine 'sampleable' edges. This leads to 'chains' of dynamic algorithms where the output performed by one algorithm might result in updates to the input of the next algorithm. This motivates a more fine-grained view on of dynamic algorithms with the goal of obtaining strong bounds on the number of changes to the output.

Future Work: The problem whether spectral sparsifiers can be maintained with polylogarithmic *worst-case* update time remains open. Our construction goes via spanners and therefore a natural question is whether spanners can be maintained with worst-case update time. Maybe there are also other more direct ways of maintaining the sparsifier. A more general question is whether we can find more dynamic algorithms for numerical problems.

Our dynamic algorithms cannot avoid storing the original graph, which is undesirable in terms of space consumption. Can we get space-efficient dynamic algorithms without sacrificing fast update time?

The sparsification framework for peeling off subgraphs and uniformly sampling from the remaining edges is very general. Are there other sparse subgraphs we could start with in the peeling process? Which properties do the sparsifiers obtained

in this way have? In particular, it would be interesting to see whether our techniques can be generalized to flow sparsifiers [24], [41].

The combination of sparsifiers with density-sensitive approaches for dynamic graph data structures [11], [40] provides an approach for obtaining $\mathrm{poly}(\log, \epsilon^{-1})$ update times. We believe this approach can be generalized to other graph cut problems. In particular, the flow networks solved for balanced cuts and graph partitioning are also bipartite and undirected, and therefore natural directions for future work.

REFERENCES

[1] D. A. Spielman and S.-H. Teng, "A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning," *SIAM Journal on Computing*, vol. 42, no. 1, pp. 1–26, 2013.

[2] C. Borgs, M. Brautbar, J. Chayes, and S.-H. Teng, "A sublinear time algorithm for PageRank computations," in *Algorithms and Models for the Web Graph*. Springer, 2012, pp. 41–53.

[3] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using PageRank vectors," in *FOCS*, ser. FOCS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 475–486.

[4] R. Andersen and Y. Peres, "Finding sparse cuts locally using evolving sets," in *STOC*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 235–244.

[5] L. Orecchia and N. K. Vishnoi, "Towards an SDP-based approach to spectral methods: a nearly-linear-time algorithm for graph partitioning and decomposition," in *SODA*, ser. SODA '11. SIAM, 2011, pp. 532–545.

[6] S. O. Gharan and L. Trevisan, "Approximating the expansion profile and almost optimal local graph clustering," in *FOCS*. IEEE, 2012, pp. 187–196.

[7] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *Journal of the ACM*, vol. 46, no. 4, pp. 502–516, 1999, announced at STOC'95.

[8] J. Holm, K. Lichtenberg, and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *Journal of the ACM*, vol. 48, no. 4, pp. 723–760, 2001, announced at STOC'98.

[9] B. M. Kapron, V. King, and B. Mountjoy, "Dynamic graph connectivity in polylogarithmic worst case time," in *SODA*, 2013, pp. 1131–1142.

[10] K. Onak and R. Rubinfeld, "Maintaining a large matching and a small vertex cover," in *STOC*, 2010, pp. 457–464.

[11] O. Neiman and S. Solomon, "Simple deterministic algorithms for fully dynamic maximal matching," in *STOC*, 2013, pp. 745–754.

[12] S. Baswana, M. Gupta, and S. Sen, "Fully dynamic maximal matching in $O(\log n$ update time," *SIAM Journal on Computing*, vol. 44, no. 1, pp. 88–113, 2015, announced at FOCS'11.

[13] S. Bhattacharya, M. Henzinger, and G. F. Italiano, "Deterministic fully dynamic data structures for vertex cover and matching," in *SODA*, 2015, pp. 785–804.

[14] S. Baswana, S. Khurana, and S. Sarkar, "Fully dynamic randomized algorithms for graph spanners," *ACM Transactions on Algorithms*, vol. 8, no. 4, pp. 35:1–35:51, 2012, announced at ESA'06 and SODA'08.

[15] I. Abraham, S. Chechik, D. Delling, A. V. Goldberg, and R. F. Werneck, "On dynamic approximate shortest paths for planar graphs with worst-case costs," in *SODA*, 2016, pp. 740–753.

[16] M. Patrascu, "Towards polynomial lower bounds for dynamic problems," in *STOC*, 2010, pp. 603–610.

[17] A. Abboud and V. Vassilevska Williams, "Popular conjectures imply strong lower bounds for dynamic problems," in *FOCS*, 2014, pp. 434–443.

[18] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak, "Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture," in *STOC*, 2015, pp. 21–30.

[19] A. A. Benczúr and D. R. Karger, "Randomized approximation schemes for cuts and flows in capacitated graphs," *SIAM Journal on Computing*, vol. 44, no. 2, pp. 290–319, 2015.

[20] D. Spielman and S. Teng, "Spectral sparsification of graphs," *SIAM Journal on Computing*, vol. 40, no. 4, pp. 981–1025, 2011, announced at STOC'04.

[21] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, "Spectral sparsification of graphs: theory and algorithms," *Communications of the ACM*, vol. 56, no. 8, pp. 87–94, Aug. 2013.

[22] A. Madry, "Fast approximation algorithms for cut-based problems in undirected graphs," in *FOCS*. IEEE, 2010, pp. 245–254.

[23] J. Sherman, "Nearly maximum flows in nearly linear time," in *FOCS*, 2013, pp. 263–269.

[24] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, "An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations," in *SODA*, 2014, pp. 217–226.

[25] R. Peng and D. A. Spielman, "An efficient parallel solver for SDD linear systems," in *STOC*, ser. STOC '14. New York, NY, USA: ACM, 2014, pp. 333–342.

[26] D. Spielman and S. Teng, "Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 3, pp. 835–885, 2014.

[27] R. Kyng, Y. T. Lee, R. Peng, S. Sachdeva, and D. A. Spielman, "Sparsified cholesky and multigrid solvers for connection Laplacians," in *STOC*, ser. STOC 2016. New York, NY, USA: ACM, 2016, pp. 842–850.

[28] R. Peng, "Approximate undirected maximum flows in $O(m \operatorname{polylog} n)$ time," in *SODA*, 2016, pp. 1862–1867.

[29] I. Koutis, "Simple parallel and distributed algorithms for spectral graph sparsification," in *SPAA*, 2014, pp. 61–66.

[30] J. Sherman, "Breaking the multicommodity flow barrier for $O(\sqrt{\log n})$-approximations to sparsest cut," in *FOCS*, ser. FOCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 363–372.

[31] L. Orecchia, S. Sachdeva, and N. K. Vishnoi, "Approximating the exponential, the Lanczos method and an $\tilde{O}(m)$-time spectral algorithm for balanced separator," in *STOC*, ser. STOC '12. New York, NY, USA: ACM, 2012, pp. 1141–1160.

[32] R. Kyng, A. Rao, S. Sachdeva, and D. A. Spielman, "Algorithms for Lipschitz learning on graphs," in *COLT*, 2015, pp. 1190–1223.

[33] D. Cheng, Y. Cheng, Y. Liu, R. Peng, and S. Teng, "Efficient sampling for gaussian graphical models via spectral sparsification," in *COLT*, 2015, pp. 364–390.

[34] J. A. Kelner and A. Levin, "Spectral sparsification in the semi-streaming setting," *Theory of Computing Systems*, vol. 53, no. 2, pp. 243–262, 2013, announced at STACS'11.

[35] I. Koutis, A. Levin, and R. Peng, "Improved spectral sparsification and numerical algorithms for SDD matrices," in *STACS 2012*, vol. 14, 2012, pp. 266–277.

[36] M. Kapralov, Y. T. Lee, C. Musco, C. Musco, and A. Sidford, "Single pass spectral sparsification in dynamic streams," in *FOCS*, 2014, pp. 561–570.

[37] Z. A. Zhu, Z. Liao, and L. Orecchia, "Spectral sparsification and regret minimization beyond matrix multiplicative updates," in *STOC*, 2015, pp. 237–245.

[38] Y. T. Lee and H. Sun, "Constructing linear-sized spectral sparsification in almost-linear time," in *FOCS*, 2015, pp. 250–269.

[39] G. Jindal and P. Kolev, "Faster spectral sparsification of Laplacian and SDDM matrix polynomials," *CoRR*, vol. abs/1507.07497, 2015.

[40] D. Peleg and S. Solomon, "Dynamic $(1+\epsilon)$-approximate matchings: A density-sensitive approach," in *SODA*, 2016, pp. 712–729.

[41] A. Andoni, A. Gupta, and R. Krauthgamer, "Towards (1+ ε)-approximate flow sparsifiers," in *SODA*. SIAM, 2014, pp. 279–293.

[42] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011, announced at STOC'08.

[43] M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng, "Lower-stretch spanning trees," *SIAM Journal on Computing*, vol. 38, no. 2, pp. 608–628, 2008, announced at STOC'05.

[44] M. K. de Carli Silva, N. J. A. Harvey, and C. M. Sato, "Sparse sums of positive semidefinite matrices," *ACM Transactions on Algorithms*, vol. 12, no. 1, p. 9, 2016.

[45] A. Zouzias, "A matrix hyperbolic cosine algorithm and applications," in *ICALP*, 2012, pp. 846–858.

[46] K. J. Ahn, S. Guha, and A. McGregor, "Spectral sparsification in dynamic graph streams," in *APPROX*, 2013, pp. 1–10.

[47] M. Kapralov and D. P. Woodruff, "Spanners and sparsifiers in dynamic streams," in *PODC*, 2014, pp. 272–281.

[48] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, "A simple, combinatorial algorithm for solving SDD systems in nearly-linear time," in *STOC*, ser. STOC '13. New York, NY, USA: ACM, 2013, pp. 911–920.

[49] Y. T. Lee and A. Sidford, "Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems," in *FOCS*, ser. FOCS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 147–156.

[50] J. A. Tropp, "User-friendly tail bounds for sums of random matrices," *Foundations of Computational Mathematics*, vol. 12, no. 4, pp. 389–434, 2012.

[51] W. S. Fung, R. Hariharan, N. J. A. Harvey, and D. Panigrahi, "A general framework for graph sparsification," in *Symposium on Theory of Computing (STOC)*, 2011, pp. 71–80.

[52] K. J. Ahn and S. Guha, "Graph sparsification in the semi-streaming model," in *ICALP*, 2009, pp. 328–338.

[53] K. J. Ahn, S. Guha, and A. McGregor, "Analyzing graph structure via linear measurements," in *SODA*, 2012, pp. 459–467.

[54] ——, "Graph sketches: sparsification, spanners, and subgraphs," in *PODS*, 2012, pp. 5–14.

[55] M. Gupta and R. Peng, "Fully dynamic $(1+\epsilon)$-approximate matchings," in *FOCS*, 2013, pp. 548–557.

[56] A. Bernstein and C. Stein, "Faster fully dynamic matchings with small approximation ratios," in *SODA*. SIAM, 2016, pp. 692–711.

[57] M. Thorup, "Fully-dynamic min-cut," *Combinatorica*, vol. 27, no. 1, pp. 91–127, 2007, announced at STOC'01.

[58] M. Thorup and D. R. Karger, "Dynamic graph algorithms with applications," in *Scandinavian Workshop on Algorithm Theory (SWAT)*, 2000, pp. 1–9.

[59] D. R. Karger, "Minimum cuts in near-linear time," *Journal of the ACM*, vol. 47, no. 1, pp. 46–76, Jan. 2000, announced at STOC'96.

[60] D. Kogan and R. Krauthgamer, "Sketching cuts in graphs and hypergraphs," in *ITCS*. ACM, 2015, pp. 367–376.