Fully-Dynamic Graph Sparsifiers Against an Adaptive Adversary

Aaron Bernstein
Rutgers University, Piscataway, NJ, USA

Jan van den Brand
Simons Institute, Berkeley, CA, USA
University of California Berkeley, CA, USA

Maximilian Probst Gutenberg
ETH Zürich, Switzerland

Danupon Nanongkai
University of Copenhagen, Denmark
KTH Royal Institute of Technology, Stockholm, Sweden

Thatchaphol Saranurak
University of Michigan, Ann Arbor, MI, USA

Aaron Sidford
Stanford University, CA, USA

He Sun
University of Edinburgh, UK

Abstract

Designing efficient dynamic graph algorithms against an *adaptive* adversary is a major goal in the field of dynamic graph algorithms and has witnessed many exciting recent developments in, e.g., dynamic matching (Wajc STOC'20) and decremental shortest paths (Chuzhoy and Khanna STOC'19). Compared to other graph primitives (e.g. spanning trees and matchings), designing such algorithms for *graph spanners* and (more broadly) *graph sparsifiers* poses a unique challenge since there is no fast deterministic algorithm known for static computation and the lack of a way to adjust the output slowly (known as "small recourse/replacements").

This paper presents the first non-trivial efficient adaptive algorithms for maintaining many sparsifiers against an adaptive adversary. Specifically, we present algorithms that maintain

1. a polylog(n)-spanner of size $\tilde{O}(n)$ in polylog(n) amortized update time,
2. an $O(k)$-approximate cut sparsifier of size $\tilde{O}(n)$ in $\tilde{O}(n^{1/k})$ amortized update time, and
3. a polylog(n)-approximate spectral sparsifier in polylog(n) amortized update time.

Our bounds are the first non-trivial ones even when only the recourse is concerned. Our results hold even against a stronger adversary, who can access the random bits previously used by the algorithms and the amortized update time of all algorithms can be made worst-case by paying sub-polynomial factors. Our spanner result resolves an open question by Ahmed et al. (2019) and our results and techniques imply additional improvements over existing results, including (i) answering open questions about decremental single-source shortest paths by Chuzhoy and Khanna (STOC'19) and Gutenberg and Wulff-Nilsen (SODA'20), implying a nearly-quadratic time algorithm for approximating minimum-cost unit-capacity flow and (ii) de-amortizing a result of Abraham et al. (FOCS'16) for dynamic spectral sparsifiers.

Our results are based on two novel techniques. The first technique is a generic black-box reduction that allows us to assume that the graph is initially an expander with almost uniform-degree and, more importantly, stays as an almost uniform-degree expander while undergoing only edge deletions. The second technique is called *proactive resampling*: here we constantly *re-sample* parts of the input graph so that, independent of an adversary's computational power, a desired structure of the underlying graph can be always maintained. Despite its simplicity, the analysis of this sampling scheme is far from trivial, because the adversary can potentially create dependencies between the random choices used by the algorithm. We believe these two techniques could be useful for developing other adaptive algorithms.

© Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun; licensed under Creative Commons License CC-BY 4.0

49th International Colloquium on Automata, Languages, and Programming (ICALP 2022).
Editors: Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff;
Article No. 20; pp. 20:1–20:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic graph algorithm, adaptive adversary, spanner, sparsifier

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.20

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2004.08432 [24]

Funding *Aaron Bernstein*: Funded by NSF Career grant 1942010.
Jan van den Brand: Funded by ONR BRC grant N00014-18-1-2562 and by the Simons Institute for the Theory of Computing through a Simons-Berkeley Postdoctoral Fellowship. Research partially done at KTH.
Maximilian Probst Gutenberg: Research partially done while at University of Copenhagen.
Danupon Nanongkai: This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement No 715672. Also partially supported by the Swedish Research Council (Reg. No. 2015-04659 and 2019-05622).
Thatchaphol Saranurak: Research partially done at KTH and supported by the Swedish Research Council (Reg. No. 2015-04659).
Aaron Sidford: Supported in part by a Microsoft Research Faculty Fellowship, NSF CAREER Award CCF-1844855, NSF Grant CCF-1955039, a PayPal research award, and a Sloan Research Fellowship.
He Sun: Funded by an EPSRC Early Career Fellowship (EP/T00729X/1).

Acknowledgements We thank Julia Chuzhoy and Gramoz Goranci for discussions.

1 Introduction

Dynamic graph algorithms maintain information in an input graph undergoing edge *updates*, which typically take the form of edge insertions and deletions. Many efficient algorithms have been developed in this setting, such as those for maintaining a minimum spanning tree, maximum matching, shortest distances, and sparsifiers. However, many of these algorithms are randomized and, more importantly, make the so-called *oblivious adversary assumption*, which assumes that each update given to the algorithm cannot depend on answers of the algorithm to earlier queries. In other words, the entire update sequence is fixed by some adversary in advance, and then each update is given to the algorithm one by one. This assumption is crucial for many recent advances in the design of efficient randomized algorithms for dynamic problems (e.g. [68, 63, 35, 22, 10, 1, 82, 9, 25]).

The oblivious-adversary assumption significantly limits the use of dynamic algorithms in certain interactive environments and, in particular, the setting where these dynamic algorithms are employed as *subroutines* for other algorithms. For example, the recent partially-dynamic single-source shortest paths algorithm without this assumption [21] has been used to obtain an almost-linear time approximate min-cost flow and balanced separator algorithm in the static setting (see also, e.g., [17, 16, 61, 62, 42, 43], for prior attempts in this direction). In addition, [40] pointed out that their goal of computing a (static) short cycle decomposition could have been achieved easily using existing dynamic spanner algorithms, if such algorithms worked without the oblivious-adversary assumption. Because of this, designing dynamic algorithms *without* the oblivious adversary assumption has become a major goal in the field of dynamic graph algorithms in recent years. We call such algorithms *adaptive* and say that they work against an *adaptive adversary*.

Thanks to the recent efforts in developing adaptive algorithms, such algorithms now exist for maintaining a number of graph primitives, such as minimum spanning trees with bounded worst-case update time [41, 77, 76, 88], partially-dynamic single-source shortest

paths [50, 42, 17, 18, 16, 62, 61, 60], and fully-dynamic matching [26, 27, 28, 29, 87]. This line of research on dynamic graph algorithms has also brought new insights on algorithm design in the static setting (e.g. flow, vertex connectivity, matching, and traveling salesman problem [73, 42, 86, 37, 38]). One very recent exciting application is the use of an adaptive dynamic algorithm called *expander decomposition* to compute maximum-weight matching and related problems in nearly-linear time on moderately dense graphs [86].

Graph sparsifiers

Despite the fast recent progress, very little was known for certain important primitives, like maintaining *graph sparsifiers* against an adaptive adversary. To formalize our discussion, we say that a sparsifier of a graph $G = (V, E)$ is a sparse graph $H = (V, E')$ that approximately preserves properties of G, such as all cuts (*cut sparsifiers*), all-pairs distances (*spanners*), and spectral properties (*spectral sparsifiers*). For any integer $\alpha \geq 1$, an *α-spanner* of graph $G = (V, E)$ is a subgraph H such that for any pair of nodes (u, v), the distance between u and v in H is at most α times their distance in G. An *α-cut sparsifier* of G is a sparse graph H that preserves all cut sizes up to an α factor: that is, $\delta_H(S) \in [\delta_G(S), \alpha\delta_G(S)]$ for every $S \subseteq V$, where $\delta_G(S)$ (respectively $\delta_H(S)$) is the total weight of edges between S and $V \setminus S$ in G (respectively H) for any $S \subset V$. An α-spectral sparsifier is a sparse graph that provides an even stronger guarantee than an α-cut sparsifier (see full version for the definition).

A dynamic algorithm for maintaining a spanner or a cut sparsifier is given a weighted undirected n-node graph G to preprocess, and returns a spanner or cut-sparsifier H of G. After this, it must process a sequence of *updates*, each of which is an edge insertion or deletion of G. After each update, the algorithm outputs edges to be inserted and deleted to H so that the updated H remains an α-spanner or cut-sparsifier of the updated G. The algorithm's performance is measured by the *preprocessing time* (the time to preprocess G initially); the *update time* (the time to process each update); the *stretch* (the value of α); and the *size* of the spanner (the number of edges). The update time is typically categorized into two types: *amortized case* update time and *worst case* update time. The more desirable one is the *worst-case* update time which holds for every single update. This is in contrast to an *amortized* update time which holds "on average"; i.e., for any t, an algorithm is said to have an amortized update time of t if, for any k, the total time it spends to process the first k updates is at most kt.

Spanners and cut sparsifers are fundamental objects that have been studied extensively in various settings (e.g., [5, 79, 11, 78, 44, 58, 57, 15, 53, 3, 4]). In the dynamic setting, they have been actively studied since 2005 (e.g. [7, 51, 8, 48, 23, 10, 31, 20]). A fairly tight algorithm with amortized update time for maintaining dynamic spanners was known in 2008 due to Baswana et al. [10]. For any $k \geq 1$, their algorithm maintains, with high probability, a $(2k-1)$-spanner of size $\tilde{O}(kn^{1+1/k})$ in $O(k^2 \log^2 n)$ amortized update time[1]. The stretch and size tradeoff is almost tight assuming Erdős' girth conjecture, which implies that a $(2k-1)$-spanner must contain $\Omega(n^{1+1/k})$ edges. Recently, Bernstein et al. [20] showed how to "de-amortize" the result of Baswana et al. [10], giving an algorithm that in $O(1)^k \log^3(n)$ *worst-case* update time maintains, w.h.p., a $(2k-1)$-spanner of size $\tilde{O}(n^{1+1/k})$.

We refer the reader to the full version for other related results and clear comparison. For dynamic cut sparsifiers, the only result we are aware of is [1], which maintains a $(1+\epsilon)$-cut sparsifier in polylogarithmic worst-case update time. [1] can also maintain a $(1+\epsilon)$-spectral sparsifier within the same update time, but this holds only for the amortized update time.

[1] Throughout, $\tilde{O}$ hides $O(\text{polylog}(n))$ factors. With high probability (w.h.p.) means with probability at least $1 - 1/n^c$ for any constant $c > 1$.

Similar to other dynamic algorithms, most existing dynamic spanner and cut sparsifier algorithms are *not* adaptive. The exceptions are the algorithms of [7], which can maintain a 3-spanner (respectively a 5-spanner) of size $O(n^{1+1/2})$ (respectively $O(n^{1+1/3})$) in $O(\Delta)$ time, where Δ is the maximum degree. These algorithms are deterministic, and thus work against an adaptive adversary. Since Δ can be as large as $\Omega(n)$, their update time is rather inefficient as typically polylog(n) or $n^{o(1)}$ update times are desired. Designing dynamic algorithms for this low update time is one of the major objectives of our paper.

Challenges

Developing efficient adaptive algorithms for maintaining graph sparsifiers poses great challenges in the general research program towards adaptive dynamic algorithms. First, computing many sparsifiers inherently relies on the use of randomness. Even in the static setting, existing fast algorithms for constructing cut and spectral sparsifiers are all randomized, and known deterministic algorithms require $\Omega(n^4)$ time [12, 89]. In fact, a nearly-linear time deterministic algorithm for a certain cut sparsifier would resolve a major open problem about computing the minimum cut deterministically [70, 56, 75]. Thus, in contrast to other primitives, such as the minimum spanning tree or approximate maximum matching, for which efficient deterministic algorithms exist in the static setting, there is little chance to dynamically maintain sparsifiers *deterministically*. (Deterministic dynamic algorithms always work against adaptive adversaries.)

Secondly, even if we allowed infinite update time and focused on the strictly simpler objective of minimizing the changes in the maintained sparsifier (the so-called *recourse* or *replacements* in online algorithms), it is unclear from existing techniques whether it is possible to maintain such a sparsifier against an adaptive adversary while only making (amortized) polylog(n) changes to the sparsifier per update to the input graph. For example, an $O(\log n)$-spanner of $\tilde{O}(n)$ edges can be easily maintained with $\tilde{O}(n)$ recourse per update by replacing the entire spanner by a new one after every update. Is it possible that an adversary who can see the output spanner can make a few changes to the graph so that a new $O(\log n)$-spanner has to change completely? An answer to this question is unclear. This is in contrast to many dynamic graph primitives where bounding the changes is obvious even against adaptive adversaries. For example, it can be easily shown that maintaining the minimum spanning tree requires at most one edge insertion and one edge deletion after each update to the input graph.

Designing algorithms with low recourse is a prerequisite for fast dynamic algorithms, and there are several graph problems where low-recourse algorithms were the crucial bottleneck, e.g. maximal independent set [33, 6, 36, 13, 74], planar embeddings [65, 66], and topological sorting [19].The lack of recourse-efficient algorithms makes it very challenging to maintain sparsifiers against an adaptive adversary.

1.1 Our Results

We show how to dynamically maintain both spanners, cut sparsifiers, and spectral sparsifiers against an adaptive adversary in *poly-logarithmic* update time and recourse. We summarize these results as follows:

▶ **Theorem 1** (Adaptive Spanner). *There is a randomized* adaptive *algorithm that, given an n-vertex graph undergoing edge insertions and deletions, with high probability, explicitly maintains a* polylog(n)*-spanner of size $\tilde{O}(n)$ using* polylog(n) *amortized update time.*

▶ **Theorem 2** (Adaptive Cut Sparsifier). *There is a randomized* adaptive *algorithm that, given an n-vertex graph undergoing edge insertions and deletions and a parameter $k \geq 1$, with high probability, maintains an $O(k)$-cut sparsifier of size $\tilde{O}(n)$ using $\tilde{O}(n^{1/k})$ amortized update time, which is* $\text{polylog}(n)$ *time when $k = \log n$.*

▶ **Theorem 3** (Adaptive Spectral Sparsifier). *There is a randomized* adaptive *algorithm that, given an n-vertex graph undergoing edge insertions and deletions, with high probability, maintains a* $\text{polylog}(n)$*-spectral sparsifier of size $\tilde{O}(n)$ using* $\text{polylog}(n)$ *amortized update time.*

All results above hold even against a stronger adversary, called *randomness-adaptive* in [76]. This adversary can access the random bits previously used by our algorithms (but not the future random bits). Theorem 1 is the first algorithm with $o(n)$ update time against an adaptive adversary, and answers the open problem in [2]. The only previous adaptive algorithm is by [7] which can take $O(n)$ update time. No non-trivial dynamic adaptive algorithm for cut sparsifiers and spectral sparsifiers is known before Theorems 2 and 3.

Compared to results assuming the oblivious-adversary assumption (e.g. [10, 20, 1]), our bounds are not as tight. For example, Theorem 1 does not achieve the standard $(2k-1)$-spanner with $O(n^{1+1/k})$ edges. One reason for this limitation is that it is not clear if such trade-off is possible even when we focus on the *recourse*, as discussed above. Maintaining spanners or other sparsifiers against adaptive adversaries with tight trade-offs and polylogarithmic recourse is a challenging barrier that is beyond the scope of this paper. Additionally, the sparse-spanner regime studied in this paper is generally the most useful for applications to other problems (see discussion in Section 1.2); getting a sharper trade-off would not lead to significant improvements for most of these applications.

All the above results can be *deamortized*.[2] For example, a $2^{O(\sqrt{\log n}\log\log(n))}$-spanner of size $\tilde{O}(n)$ can be maintained in $2^{O(\sqrt{\log n}\log\log(n))}$ *worst-case* update time. Also, for any k, a $2^{O(k\,\text{polylog}(k))}$-cut sparsifier of size $\tilde{O}(n)$ can be maintained in $\tilde{O}(n^{1/k})$ worst-case time. In particular, we can maintain an $O(\log^* n)$-cut sparsifier and an $O(1)$-cut sparsifier in $n^{o(1)}$ and n^{ϵ} time for any constant ϵ, respectively.

Our deamortization technique also implies, as a side result, the first non-trivial algorithm with *worst-case* update time against an oblivious adversary for maintaining spectral sparsifiers.

▶ **Theorem 4** (Oblivious Spectral Sparsifier). *There is a randomized algorithm against an oblivious adversary that, given an n-vertex graph undergoing edge insertions and deletions and $\epsilon \geq 1/\text{polylog}(n)$, with high probability, maintains a $(1+\epsilon)$-spectral sparsifier of size $n \cdot 2^{O(\sqrt{\log n})}$ using $2^{O(\log^{3/4} n)}$ worst-case update time.*

The previous algorithm by Abraham et al. [1] maintains a $(1+\epsilon)$-spectral sparsifier of size $\tilde{O}(n)$ using $\text{polylog}(n)$ amortize update time. Further [1] asked if the update time can be made worst-case. Theorem 4 answers this open question modulo $n^{o(1)}$ factors.

1.2 Applications

Our results imply several interesting applications. Our first set of applications are for the decremental $(1+\epsilon)$-approximate single-source shortest paths (SSSP) problem. There has been a line of work [17, 16, 18, 62] on fast adaptive algorithms for solving this problem.

[2] To do this, we use, e.g., the sparsification technique [49] and a sophisticated dynamic expander decomposition; see Section 2.3 for an overview.

Although all these algorithms are adaptive, they share a drawback that they cannot return the shortest path itself; they can only maintain distance estimates. Very recently, Chuzhoy and Khanna [42] showed a partial fix to this issue for some algorithms [17, 16] and consequently obtained impressive applications to static flow algorithms. Unfortunately, this fix only applies in a more restricted setting, and moreover it is not clear how the technique from [42] can be used to fix the same issue in other algorithms (e.g. [18, 62]).

We show that our main result from Theorem 1 can be employed in a consistent and simple way, such that the path-reporting issue in all previous algorithms in [17, 16, 18, 62] can be fixed. This resolves an open question posed in multiple papers [16, 62, 42]. We summarize these applications below:

▶ **Corollary 5** (Fixing the path-reporting issue of [18, 62]). *For any decremental unweighted graph $G = (V, E)$, fixed source s, and constant $\epsilon > 0$, there is an adaptive algorithm $\mathcal{B}$ that maintains the $(1+\epsilon)$-approximate distances from vertex s to every vertex $v \in V$* and supports corresponding shortest path queries. *The algorithm $\mathcal{B}$ has expected total update time $mn^{0.5+o(1)}$, distance estimate query time $O(\log\log n)$ and shortest path query time $\tilde{O}(n)$.*

Corollary 5 gives the first adaptive algorithm without the path-reporting issue that can take $o(n^2)$ total update time. The next algorithm works on weighted graphs and is near-optimal on dense graphs:

▶ **Corollary 6** (Fixing the path-reporting issue of [17, 16]). *For any decremental weighted graph $G = (V, E, w)$ with W being the ratio between maximum and minimum edge weight, fixed source s, and $\epsilon > 0$, there is an adaptive algorithm $\mathcal{A}$ that maintains the $(1+\epsilon)$-approximate distances from vertex s to every vertex $v \in V$* and supports corresponding shortest path queries. *The algorithm $\mathcal{A}$ has expected total update time $\tilde{O}(n^2 \log W)$, distance estimate query time $O(\log\log(nW))$ and shortest path query time $\tilde{O}(n \log W)$.*

Corollary 6 can be compared to two previous results [42, 43]. In [42], their algorithm requires slower $n^{2+o(1)} \log W$ total update time, and needs to assume that the input graph undergoes only *vertex* deletions which is more restrictive. So Corollary 6 strictly improves the algorithm by [42]. In [43], they use very different techniques than ours and show an algorithm with $n^{2+o(1)} \log W$ total update time, distance query time $O(\log\log(nW))$, shortest path query time $O(|P| n^{o(1)})$ when a path P is returned, and is deterministic. This algorithm is incomparable to Corollary 6. Our result has slightly faster total update time, but their algorithm is deterministic and guarantees faster shortest path query time.

By plugging Corollary 6 into the standard multiplicative weight update framework (e.g. [55, 52, 42, 73]), we get the following:

▶ **Corollary 7.** *There exist $(1+\epsilon)$-approximate algorithms with expected running time $\tilde{O}(n^2)$ for the following problems:*

1. *minimum-cost maximum s-t flow in undirected vertex-capacitated graphs, and*
2. *minimum-cost maximum s-t flow in undirected unit-edge-capacity graphs.*

Corollary 7 slightly reduces the $n^{2+o(1)}$ run time from [42, 43] to $\tilde{O}(n^2)$.[3]

The second set of applications are faster algorithms for variants of multi-commodity flow problems using Theorem 1. For example, we achieve a static $\tilde{O}((n+k)n)$-time polylog(n)-approximation algorithm for the *congestion minimization* problem with k demand pairs on unweighted vertex-capacitated graphs. This improves the $\tilde{O}((m+k)n)$-time $O(\log(n)/\log\log(n))$-approximation algorithms implied by Karakostas [69] in terms of the running time at the cost of a worse approximation ratio. See the full version for more detail.

[3] We note that the result of [43] is deterministic.

Finally, we apply Theorem 4 to the problem of maintaining *effective resistance*. Durfee et al. [47, 46] presented a dynamic algorithm with $\tilde{O}(n^{6/7})$ amortized update time for $(1+\epsilon)$-approximately maintaining the effective resistance between a fixed pair of nodes. Plugging our result in the algorithm of Durfee et al. leads to an $n^{6/7+o(1)}$ worst-case update time. (Both of these results assume an oblivious adversary.)

1.3 Techniques

To prove the above results, the first key tool is the black-box reduction in Theorem 8 that allows us to focus on *almost-uniform-degree expanders*[4]. Theorem 8 works for a large class of problems satisfying natural properties (defined in Section 2) which includes spanners, cut sparsifiers, and spectral sparsifiers. Hence the theorem uses the term "α-approximate sparsifier" without defining the exact type of sparsifier.

▶ **Theorem 8** (Informal Blackbox Reduction, see full version for the formal statements)**.** *Assume that there is an algorithm $\mathcal{A}$ that can maintain an α-approximate sparsifier on an n-vertex graph G with the following promises:*

- *G undergoes batches of edge deletions*[5] *(isolated nodes are automatically removed),*
- *G is unweighted,*
- *after each batch of deletions, G is an expander graph, and*
- *after each batch of deletions, G has almost uniform degree, i.e. the maximum degree $\Delta_{\max}$ and the minimum degree $\Delta_{\min}$ are within a* polylog(n) *factor.*

Then, there is another algorithm $\mathcal{B}$ with essentially the same amortized update time for maintaining an α-approximate sparsifier of essentially the same size on a general weighted *graph undergoing both edge insertions and deletions. If $\mathcal{A}$ is adaptive or deterministic, then so is $\mathcal{B}$.*

As it is well-known that many problems become much easier on expanders (e.g., [84, 83, 81, 67, 34, 72]), we believe that this reduction will be useful for future developments of dynamic algorithms. For example, if one can come up with an adaptive algorithm for maintaining $(1+\epsilon)$-cut sparsifiers on expanders, then one can immediately obtain the same result on general graphs.

Our second technique is a new sampling scheme called *proactive resampling*: here we constantly re-sample parts of the input graph so that, independent of an adversary's computational power, a desired structure of the underlying graph can be always maintained; see Section 2 for a high-level discussion of this technique. Since there are still few known tools for designing algorithms that work against an adaptive adversary, we expect that our technique will prove useful for the design of other adaptive algorithms in the future.

We further extend the black-box reduction from Theorem 8 to algorithms with worst-case update time, which allows us to deamortize both Theorem 1 and Theorem 2 with slightly worse guarantees. It also easily implies Theorem 4.

1.4 Subsequent Development

Since this paper has appeared in April 2020, there have been exciting subsequent developments based on techniques of this paper.

[4] Expanders are graphs with high conductance (see Section 2). Intuitively, they are "robustly connected" graphs.

[5] That means, in each iteration the algorithm is given a set $D \subset E$ of edges that are to be deleted.

Our dynamic spectral sparsifiers with $(1+\epsilon)$-approximation but large query time (see full version) has been employed as a blackbox subroutine in a line of work on faster algorithms for exact max-flow [86, 85, 54], and also in [39] for making a dynamic polylog(n)-approximate all-pairs max flow algorithm work against an adaptive adversary. In [14], the authors opened the blackbox of our spectral sparsifier and combined our dynamic expander decomposition with our sparsification techniques to obtain the first dynamic algorithms with $o(n)$ update time against an adaptive adversary for $(1+\epsilon)$-approximate global min cuts and all-pairs effective resistances. Our almost-uniform-degree expander decomposition is also used in the context of online algorithms [59].

Our proactive resampling technique has been extended in a follow-up work [30] for maintaining fully dynamic 3-spanners against an adaptive adversary, in contrast with polylog(n)-spanners.

2 Overview

All our algorithms use a common framework based on expanders, which results in a reduction from fully dynamic algorithms on general graphs to the special case of decremental algorithms on expander graphs. The reduction holds for a general class of graph problems that satisfy some criteria. These criteria are satisfied for spectral-sparsifiers, cut-sparsifiers and spanners. In this section, we define the abstract criteria needed for our reduction (See Conditions 1-5 below), so that we only need to prove our algorithm once and apply it to all these types of sparsifiers.

The overview is split into three parts. In Section 2.1 we show the reduction for amortized update time. In Section 2.2 we show how to take advantage of the reduction by designing efficient algorithms on expanders. Finally, in Section 2.3 we finish the overview with a sketch of how to extend the reduction to worst-case update-time algorithms.

2.1 Reduction to Expanders: Amortized Update Time

We now outline our black-box reduction, which can preserve several nice properties of the algorithms. That is, given an algorithm with property x running on expander, we obtain another algorithm with property x with essentially the same running time and approximation guarantee, where the property x can be "deterministic", "randomized against an adaptive adversary", or "worst-case update time". In this subsection, we focus on amortized update time: see Section 2.3 for an overview of how to extend the reduction to apply to worst-case algorithms.

The reduction holds for any graph problem that satisfies a small number of conditions. We formalize a graph problem as a function $\mathcal{H}$ that maps (G, ϵ) for a graph G and parameter $\epsilon > 0$ to a set of graphs. We say a dynamic algorithm $\mathcal{A}$ solves $\mathcal{H}(\epsilon)$ if for every input graph G, algorithm $\mathcal{A}$ maintains/computes a graph $H \in \mathcal{H}(G, \epsilon)$. For example we could define $\mathcal{H}(G, \epsilon)$ to be the set of all $(1+\epsilon)$-cut sparsifiers. So then saying "data structure $\mathcal{A}$ solves $\mathcal{H}(\epsilon)$" means that $\mathcal{A}$ maintains for any input graph an $(1+\epsilon)$-cut sparsifier.

Pertubation Property

The first property required by our reduction allows us to slightly perturb the edges, i.e. scale each edge $\{u, v\}$ by some small factor $f_{u,v}$ bounded by $1 \leq f_{u,v} \leq e^{\epsilon}$. Define $\zeta \cdot G$ to be the graph G with all edge-weights multiplied by ζ.

$$\text{Let } G' \text{ be } G \text{ scaled by up to } e^{\epsilon}, \text{ then } G' \in \mathcal{H}(G, \epsilon) \text{ and } e^{\epsilon} \cdot G \in \mathcal{H}(G', \epsilon). \tag{1}$$

Property (1) implies that $G \in \mathcal{H}(G, \epsilon)$ for all $\epsilon > 0$. For example any graph is a (potentially dense) spectral approximation or spanner of itself. The property is also useful when we want to discretize the edge weights. A common technique is to round edge weights to the nearest power of e^{ϵ} in order to discretize the set of possible edge weights without changing graph properties such as the spectrum or distances too much. Combined with the following *union* property, this also allows us to generalize algorithm for unweighted graphs to support weighted graphs.

Union Property

Say that $G = \bigcup_{i=1}^{k} G_i$ for some k and that $s_1, ..., s_k \in \mathbb{R}$. Then the union property is defined as follows:

$$\text{If } H_i \in \mathcal{H}(G_i, \epsilon) \text{ and } 0 \leq s_i, \text{ then } \bigcup_i s_i \cdot H_i \in \mathcal{H}\left(\bigcup_i s_i \cdot G_i, \epsilon\right). \tag{2}$$

Combining this property with the previous *pertubation* property (1) gives us the following reduction. Given a graph G with real edge weights from $[1, W]$, one can decompose G into graphs $G_1, ..., G_k$, such that each G_i contains edges with weights in $[e^{(i-1)\epsilon}, e^{i\epsilon})$. One can then use any algorithm that solves $\mathcal{H}$ on *unweighted* graphs to obtain $H_i \in \mathcal{H}(G'_i, \epsilon)$ for all $i = 1, ..., k$, where G'_i is the graph G_i when ignoring the edge weights. Then $\bigcup_i e^{i\epsilon} \cdot H_i \in \mathcal{H}(\bigcup_i e^{i\epsilon} \cdot G_i, \epsilon) \subset \mathcal{H}(G, \epsilon)$ by combining property (2) and (1). Thus one obtains an algorithm that solves $\mathcal{H}$ on *weighted* graphs.

Reduction for Amortized Update Time

Loosely speaking, our black-box states the following. Say that we have a data structure $\mathcal{A}_X$ on a graph X that at all times maintains a sparsifier in $\mathcal{H}(X, \epsilon)$ with amortized update time $T(\mathcal{A}_X)$, but assumes the following restricted setting: **1)** Every update to X is an edge *deletion* (no insertion), and **2)** X is always an expander. We claim that $\mathcal{A}_X$ can be converted into a *fully dynamic* algorithm $\mathcal{A}$ that works on any graph G, and has amortized update time $T(\mathcal{A}) = \tilde{O}(T(\mathcal{A}_X))$.

We first outline this black-box under the assumption that we have a dynamic algorithm that maintains a decomposition of $G = \bigcup_i G_i$ into edge disjoint expander graphs $G_1, G_2, ...$ This dynamic algorithm will have the property that whenever the main graph G is updated by an adversarial edge insertion/deletion, each expander G_i receives only edge deletions, though occasionally a new expander G_j is added to the decomposition. Thus one can simply initialize $\mathcal{A}_X$ on the expander G_j to obtain some $H_j \in \mathcal{H}(G, \epsilon)$, when G_j is added to the decomposition. Then whenever an edge deletion is performed to G_j, we simply update the algorithm $\mathcal{A}_X$ to update the graph H_j. By the union property (2) we then have that

$$H := \bigcup_i H_i \in \mathcal{H}\left(\bigcup_i G_i, \epsilon\right) = \mathcal{H}(G, \epsilon).$$

So we obtain an algorithm $\mathcal{A}$ that can maintain a sparsifier H of G. We are left with proving how to obtain this dynamic algorithm for maintaining the expander decomposition of G.

Dynamic Expander Decomposition

The idea is based on the expander decomposition and expander pruning of [80]. Their expander decomposition splits V into disjoint node sets $V_1, V_2, ...$, such that the induced subgraphs $G[V_i]$ on each V_i are expanders, and there are only $o(m)$ edges between these

expanders. In the full version we show that by recursively applying this decomposition on the subgraph induced by the inter-expander edges, we obtain a partition of the *edges* of G into a union of expanders. This means, we can decompose G into subgraphs $G_1, G_2, ...$, where each G_i is an expander and $\bigcup_i G_i = G$. We show in the full version that the time complexity of this decomposition algorithm is $\tilde{O}(m)$.

We now outline how we make this decomposition dynamic in the full version. Assume for now, that we have a decomposition of G into $G = \bigcup_i G^{(i)}$, where for all i, the graph $G^{(i)}$ has at most 2^i edges, but each $G^{(i)}$ is not necessarily an expander. Further, each $G^{(i)}$ is decomposed into expanders $G^{(i)} = \bigcup_j G^{(i)}_j$. To make this decomposition dynamic, we will first consider edge insertions where we use a technique from [64]. Every edge insertion performed by the adversary is fed into the graph $G^{(1)}$. Now, when inserting some edges into some $G^{(i)}$, there are two cases: (i) the number of edges in $G^{(i)}$ remains at most 2^i. In that case recompute the expander decomposition $G^{(i)} = \bigcup_j G^{(i)}_j$ of $G^{(i)}$. Alternatively we have case (ii) where $G^{(i)}$ has more than 2^i edges. In that case we set $G^{(i)}$ to be an empty graph and insert all the edges that previously belonged to $G^{(i)}$ into $G^{(i+1)}$. Note that on average it takes 2^{i-1} adversarial insertions until $G^{(i)}$ is updated, and we might have to pay $\tilde{O}(2^i)$ to recompute its decomposition, so the amortized update time for insertions is simply $\tilde{O}(1)$.

For edge deletions we use the expander pruning technique based on [80] (refined from [77, 76, 88]). An over-simplified description of this technique is that, for each update to the graph, we can repeatedly prune (i.e. remove) some $\tilde{O}(1)$ edges from the graph, such that the remaining part is an expander. So whenever an edge is deleted from G, we remove the edge from its corresponding $G^{(i)}_j$, and remove/prune some $\tilde{O}(1)$ extra edges from $G^{(i)}_j$, so that it stays an expander graph. These pruned edges are immediately re-inserted into $G^{(1)}$ to guarantee that we still have a valid decomposition $G = \bigcup_i G^{(i)}$. In summary, we are able to maintain a decomposition $G = \bigcup_{i,j} G^{(i)}_j$ of G into expander graphs. This decomposition changes only by creating new expanders and removing edges from existing expanders, so we can run the decremental expander algorithm $\mathcal{A}_X$ on each $G^{(i)}_j$.

Contraction Property and Reduction to Uniform Degree Expanders

Many problems are easier to solve on graphs with (near) uniform degree. Thus, we strengthen our reduction to work even if the decremental algorithm $\mathcal{A}_X$ assumes that graph X has near-uniform degree. On its own, the expander decomposition described above is only able to guarantee that for each expander the minimum degree is close to the average degree; the maximum degree could still be quite large. In order to create a (near) uniform degree expander, we split these high degree nodes into many smaller nodes of smaller degree. In order to perform this operation, we need the condition that whichever graph problem $\mathcal{H}$ we are trying to solve must be able to handle the reverse operation, i.e. when we contract many small degree nodes into a single large degree node.

$$\begin{aligned}&\text{When contracting } W \subset V \text{ in both } G \text{ and } H \in \mathcal{H}(G, \epsilon),\\ &\text{let } G' \text{ and } H' \text{ be the resulting graphs, then } H' \in \mathcal{H}(G', \epsilon).\end{aligned} \tag{3}$$

All in all, our black-box reduction shows that in order to solve a sparsification problem $\mathcal{H}$ in the fully dynamic model on general graphs, we need to **1)** show that $\mathcal{H}$ satisfies the perturbation, union, and contraction properties above (Properties 1-3) AND **2)** Design an algorithm $\mathcal{A}_X$ for $\mathcal{H}$ in the simpler setting where the dynamic updates are purely decremental (only edge deletions), and where the dynamic graph G is always guaranteed to be a near-uniform degree expander.

We now present the second main contribution of our paper, which is a new adaptive algorithm $\mathcal{A}_X$ on expanders. We conclude the overview with a discussion of the worst-case reduction (Section 2.3), for which we will need two additional properties of the problem $\mathcal{H}$.

2.2 Adaptive Algorithms on Expanders

We showed above that maintaining a sparsifier in general graphs can be reduced to the same problem in a near-uniform-degree expander. Thus, for the rest of this section we assume that $G = (V, E)$ is *at all times* a ϕ-expander with max degree $\Delta_{\max}$ and min-degree $\Delta_{\min}$, and that G is only subject to edge deletions. Let $n = |V|, m = |E|$. In this overview, we assume that $1/\phi$ and $\Delta_{\max}/\Delta_{\min}$ are $O(\text{polylog}\, n)$, and we assume $\Delta_{\min} \gg 1/\phi$. Define $\textsc{Inc}_G(v)$ to the edges incident to v in G.

We now show how to maintain a $O(\log(n))$-approximate cut-sparsifier H in G against an adaptive adversary; it is not hard to check that H is also a spanner of stretch $\tilde{O}(1/\phi)$, because a cut-sparsifier of a ϕ-expander is itself a $\tilde{\Omega}(\phi)$-expander, and hence has diameter $\tilde{O}(1/\phi)$. See full version for details.

Static Expander Construction

We first show a very simple *static* construction of $H \subseteq G$. Define $\rho = \tilde{\Theta}\left(\frac{\Delta_{\max}}{\Delta_{\min}^2 \phi^2}\right) = \tilde{\Theta}\left(\frac{1}{\Delta_{\min}}\right)$, with a sufficiently large polylog factor. Now, every edge is independently sampled into H with probability ρ, and if sampled, is given weight $1/\rho$. To see that H is a cut sparsifier, consider any cut $X, \bar{X}$, with $|X| \leq n/2$. We clearly have $\mathbb{E}[|E_H(X, \bar{X})|] = \rho |E_G(X, \bar{X})|$, so since every edge in H has weight $1/\rho$, we have the same weight *in expectation*. For a high probability bound, want to show that $\Pr[|E_H(X, \bar{X})| \sim \rho |E_G(X, \bar{X})|] \geq 1 - n^{-2|X|}$; we can then take a union bound over the $O(n^{|X|})$ cuts of size $|X|$.

Since the graph is an expander, we know that $|E_G(X, \bar{X})| \geq \text{vol}_G(X) \cdot \phi \geq |X| \cdot \Delta_{\min} \cdot \phi = \tilde{\Omega}(|X|\Delta_{\min})$. Thus, by our setting of $\rho = \tilde{\Theta}(1/\Delta_{\min})$, we have $\mathbb{E}[|E_H(X, \bar{X})|] \geq |X| \log^2(n)$. Since each edge is sampled independently, a chernoff bound yields the desired concentration bound for $|E_H(X, \bar{X})|$.

Naive Dynamic Algorithms

The most naive dynamic algorithm is: whenever the adversary deletes edge (u, v), resample all edges in $\textsc{Inc}_G(u)$ and $\textsc{Inc}_G(v)$: that is, include each such edge in H with probability ρ. Efficiency aside, the main issue with this protocol is that the adversary can cause some target vertex x to become *isolated* in H, which clearly renders H not a cut sparisifer. To see this, let $y_1, \ldots, y_k$ be the neighbors of x. The adversary then continually deletes arbitrary edges $(y_1, z) \neq (y_1, x)$, which has the effect of resampling edge (x, y_1) each time. With very high probability, the adversary can ensure within $\log(n)$ such deletions (y_1, z) that (x, y_1) is NOT included in H; the adversary then does the same for y_2, then y_3, and so on.

Slightly Less Naive Algorithm

To fix the above issue, we effectively allow vertices u and v to have separate copies of edge (u, v), where u's copy can only be deleted if u itself is resampled. Formally, every vertex v will have a corresponding set of edges S_v and we will always have $H = \bigcup_{v \in V} S_v$, where all edges in H have weight $1/\rho$. We define an operation $\textsc{SampleVertex}(v)$ that *independently* samples each edge in $\textsc{Inc}_G(v)$ into S_v with probability ρ. The naive implementation of $\textsc{SampleVertex}(v)$ takes time $O(\deg_G(v)) = O(\Delta_{\max})$ time, but an existing technique used

in [71, 45, 32] allows us to implement SampleVertex(v) in time $O(\rho\Delta_{\max}\log(n)) = \tilde{O}(1)$. (The basic idea is that the sampling can be done in time proportional to the number of edges successfully chosen, rather than the number examined.)

The dynamic algorithm is as follows. At initialization, construct each S_v by calling SampleVertex(v), and then set $H = \bigcup_{v\in V} S_v$. Whenever the adversary deletes edge (u, v), replace S_u and S_v with new sets SampleVertex(u) and SampleVertex(v), and modify $H = \bigcup_{v\in V} S_v$ accordingly. By the above discussion, the update time is clearly $\tilde{O}(1)$. We now show that this algorithm effectively guarantees a good *lower bound* on the weight of each cut in H, but might still lead to an overly high weight. Consider any cut $(X, \bar{X})$. By the expansion of G, the *average* vertex $x \in X$ has $\text{Inc}_G(x) \cap E_G(X, \bar{X}) \geq \phi\Delta_{\min}$. For simplicity, let us assume that *every* vertex $x \in X$ has $\text{Inc}_G(x) \cap E_G(X, \bar{X}) = \tilde{\Omega}(\phi\Delta_{\min}) = \tilde{\Omega}(\Delta_{\min})$, as we can effectively ignore the small fraction of vertices for which this is false. Now, say that an operation SampleVertex(x) succeeds if it results in $|S_x \cap E_G(X, \bar{X})| \sim \rho|\text{Inc}_G(x) \cap E_G(X, \bar{X})|$. Because of our setting for ρ and our assumption that $\text{Inc}_G(x) \cap E_G(X, \bar{X}) = \tilde{\Omega}(\Delta_{\min})$, a Chernoff bound guarantees that each SampleVertex(x) succeeds with probability $1-n^{-10}$. Now, since the adversary makes at most m updates before the graph is empty, each SampleVertex(x) is called at most n^2 times, so there is a $1-n^{-8}$ probability that *every* call SampleVertex(x) is successful; we call such vertices always-successful. A simple probability calculation shows that $\Pr[\text{at least } |X|/2 \text{ vertices in } X \text{ are always-successful}] \geq 1 - n^{-2|X|}$, which allows us to union bound over all cuts of size X. Thus, at all times, half the vertices in X have $|S_x \cap E_G(X, \bar{X})| \sim \rho|\text{Inc}_G(x) \cap E_G(X, \bar{X})|$; assuming for simplicity that this is an "average" half of vertices, i.e. that these vertices have around half of the edges crossing the cut, we have $|E_H(X, \bar{X})| \geq |\bigcup_{x\in X} S_x \cap E_G(X, \bar{X})| \gtrsim \rho|E_G(X, \bar{X})|/2$.

The above idea already implies that we can maintain a sparse graph H where each cut is expanding, i.e. a sparse expander, against an adaptive adversary. As an expander has low diameter, H is a spanner. Therefore, we are done if our goal is a dynamic spanner algorithm.

Unfortunately, this algorithm is not strong enough for maintaining cut sparsifiers, as the algorithm may result in $|E_H(X, \bar{X})| \gg \rho|E_G(X, \bar{X})|$. Let $\Delta_{\max} \sim \sqrt{n}$, and consider the following graph G. There is a set X of size $\sqrt{n}$ such that $G[X]$ is a clique and $G[\bar{X}]$ is $\sqrt{n}$-degree-expander. There is also a $\sqrt{n}$-to-1 matching from X to $\bar{X}$: so every vertex in $y \in \bar{X}$ has *exactly one* edge e_y crossing the cut. It is easy to check that G is an expander. The adversary then does the following. For each $y \in \bar{X}$, it keeps deleting edges in $E(y, \bar{X})$ until e_y is sampled into S_y; with high probability, this occurs within $O(\log(n)/\rho)$ deletions for each vertex y. Thus, at the end, $H \supseteq \bigcup_{y\in\bar{X}} S_y$ contains all of $E_G(X, \bar{X})$.

Better Algorithm via Proactive Sampling

We now show how to modify the above algorithm to ensure that w.h.p., $|E_H(X, \bar{X})| = \tilde{O}(\rho|E_G(X, \bar{X})|)$; we later improve this to $|E_H(X, \bar{X})| = O(\rho\log(n)|E_G(X, \bar{X})|)$. We let time t refer to the tth adversarial update. As before, we always have $H = \bigcup_{v\in V} S_v$, and if the adversary deletes edge (u, v) at time t, the algorithm immediately calls SampleVertex(u) and SampleVertex(v). The change is that the algorithm also calls SampleVertex(u) and SampleVertex(v) at times $t+1, t+2, t+4, t+8, t+16, \ldots$; we call this *proactive sampling*. The proof that $|E_H(X, \bar{X})| \gtrsim \rho|E_G(X, \bar{X})|/2$ remains basically the same as before. We now upper bound $|E_H(X, \bar{X})|$.

The formal analysis is somewhat technical, but the crux if the following **key claim:** for any $(u, v) \in G$, we have that after t adversarial updates, $\Pr[(u, v) \in H \text{ at time t}] \leq 2\rho\log(t) \leq 2\rho\log(m)$. We then use the key claim as follows: consider any cut $(X, \bar{X})$. If every edge in $E_G(X, \bar{X})$ was *independently* sampled into H with probability at most $2\rho\log(m)$,

then a Chernoff bound would show that $|E_H(X, \bar{X})| \leq 4\rho \log(m)|E_G(X, \bar{X})|$ with probability at least $1 - n^{-2|X|}$, as desired. Unfortunately, even though every individual edge-sampling occurs with probability ρ, independent of everything that happened before, it is NOT the case that event $e \in H$ is independent from event $e' \in H$: the adversary is adaptive, so its sampling strategy for e' can depend on whether or not e was successfully sampled into H at an earlier time. Nonetheless, we show in the full proof that these dependencies can be disentangled.

Let us now sketch the proof for the key claim. The edge (u, v) can appear in H because it is in S_u or S_v at the time t. Let us bound the probability that $(u, v) \in S_u$ at time t. Let $T_{\text{schedule}}(u)$ be all times before t for which SampleVertex(u) has been scheduled by proactive sampling: so whenever the adversary updates an edge (u, v) at time t', times $t', t' + 1, t' + 2, t' + 4, t' + 8, \ldots$ are added to $T_{\text{schedule}}(u)$. Let $T^{t'}_{\text{schedule}}(u) \subset [t', t]$ be the state of $T_{\text{schedule}}(u)$ at time t'. Now, we say that a call to SampleVertex(u) at time t' is *relevant* if $T^{t'+1}_{\text{schedule}}(u) = \emptyset$. Observe that for (u, v) to be in S_u at time t, it must have been added during some relevant call SampleVertex(u), because every non-relevant call is followed by another call before time t which invokes again SampleVertex(u) and thereby deletes the previously sampled set S_u and replaces it by a new one. We complete the proof by claiming that there are at most $\log(t)$ relevant calls SampleVertex(u). This is because if a relevant call occurs at t', then proactive sampling adds some time t^* to $T_{\text{schedule}}(u)$ such that $(t' + t)/2 \leq t^* \leq t$; thus, there can be no relevant calls in time interval $[t', (t' + t)/2]$. So each relevant call halves the possible time interval for other relevant calls, so there are at most $\log(t)$ relevant calls.

We now briefly point out why this modified algorithm has $|E_H(X, \bar{X})| = \tilde{O}(\rho|E_G(X, \bar{X})|)$, rather than the desired $|E_H(X, \bar{X})| = O(\rho \log(n)|E_G(X, \bar{X})|)$. Consider again the graph G consisting of a vertex set X of size $\sqrt{n}$ such that $G[X]$ is a complete graph, and let $\overline{X}$ be a $\sqrt{n}$-degree expander in $G[\overline{X}]$. Additionally, we have a $\sqrt{n}$-to-one matching, i.e. every vertex in X is matched to $\sqrt{n}$ vertices in $\overline{X}$. The graph is still an expander as argued before.

Now observe that $\Delta_{\max} = 2\sqrt{n}$ and we obtain a first sparsifier H at time 0 of G where we have weight on the cut, i.e. $|E_H(X, \bar{X})|/\rho$, of size $\sim n$ (which is the number of edges crossing). In particular, the weight on edges in $|E_H(X, \bar{X}) \cap \bigcup_{x \in \overline{X}} S_x|/\rho \sim n$, i.e. the vertices in $\overline{X}$ carry half the weight of the cut in the sparsifier. But over the course of the algorithm, the adversary can delete edges in the cut $(X, \overline{X})$ that are in $G \setminus H$. Observe that the resampling events do not affect edges in $E_H(X, \bar{X}) \cap \bigcup_{x \in \overline{X}} S_x$ since none of the deleted edges is incident to any such edge (recall the $\sqrt{n}$-to-one matching). The adversary can continue until the cut only has weight in G of $|X|\Delta_{\min}\phi$ without violating the expander and min-degree guarantees. But then the weight in H on the cut is still $\sim n$ while the weight in G is only $\sim n(\Delta_{\max}/\Delta_{\min})\phi$. Thus, we only obtain a $\sim (\Delta_{\max}/\Delta_{\min})\phi$-approximation (plus a $\log n$-factor from proactive sampling might appear).

Final Algorithm

To resolve the issue above, we would like to ensure that the edges in $E_H(X, \bar{X})$ are resampled when $|E_G(X, \bar{X})|$ changes by a large amount. We achieve this with one last modification to the algorithm: for every $v \in V$, whenever $\deg_G(v)$ decreases by $\zeta = \phi\Delta_{\min}$, we run SampleVertex$(w)$ for every edge $(v, w) \in G$. It is not hard to check that each vertex will only resampled a total of $O(\Delta^2_{\max}/\zeta) = \tilde{O}(\Delta_{\max})$ additional times as a result of this change which is subsumed by $\tilde{O}(m)$ when summing over the vertices. (A naive implementation of the above modification only leads to small *amortized* update time, but this can easily be worst-case by staggering the work over several updates using round-robin scheduling.) We leave the analysis of the approximation ratio for the full version.

By using the convenient lemma which says that any cut sparsifier on an expander is also a spectral sparsifier, we also obtain an adaptive algorithm for spectral sparsifier.

2.3 Reduction to Expanders: Worst-Case Update Time

We now outline how to extend our black-box reduction to work with worst-case update time. We again assume there exists some algorithm $\mathcal{A}_X$ that maintains for any graph G a sparsifier $H \in \mathcal{H}(G, \epsilon)$, provided that G stays a uniform degree expander throughout all updates, all of which are only edge deletions.

The condition that G always remains an expander is too strong, but we can use expander pruning to maintain the property from the perspective of $\mathcal{A}_X$. Consider some deletion in G: although G may not be an expander, we can use pruning to find a subset of edges $P \subset E(G)$, such that $G \setminus P$ is an expander. We then input all edges in P as deletions to $\mathcal{A}_X$, so the graph $G \setminus P$ in question is still an expander: $\mathcal{A}_X$ thus returns $H \in \mathcal{H}(G \setminus P, \epsilon)$. Then based on property (1) and (2) it can be shown that $H \cup P \in \mathcal{H}(G, \epsilon)$. So by taking all the pruned edges together with the sparsifier H of $G \setminus P$ we obtain a sparsifier of G, even when G itself is no longer an expander.

Unfortunately this dynamic sparsifier algorithm has two downsides: (i) The maintained sparsifier is only sparse for a short sequence of updates, as otherwise the set of pruned edges becomes too large and thus the output $H \cup P$ becomes too dense. (ii) The algorithm only works on graphs that are initially an expander.

Extending the algorithm to general graphs

To extend the previous algorithm to work on general graphs, we run the static expander decomposition algorithm. As outlined before, we can decompose G into subgraphs $G_1, G_2, ...$, where each G_i is an expander and $\bigcup_i G_i = G$. We can then run the algorithm, outlined in the previous paragraph, on each of these expanders and the union of all the obtained sparsifiers will be a sparsifier of the original input G.

Similar as before, one downside of this technique is that the size of the sparsifier will increase with each update, because more and more edges will be pruned. Thus, the resulting dynamic algorithm can only maintain a sparsifier for some limited number of updates.

Extending the number of updates

A common technique for dynamic algorithms, which only work for some limited number of updates (say k updates), is to reset the algorithm after k updates. If the algorithm has preprocessing time p and update time u, then one can obtain an algorithm with amortized update time $O(p/k + u)$. However, the worst-case complexity would be quite bad, because once the reset is performed, the old sparsifier (from before the reset) must be replaced by the new one. Listing all edges of the new sparsifier within a single update would be too slow. There is a standard technique for converting such an amortized bound to an equivalent worst-case bound. The idea is to slowly translate from the old sparsifier to the new one, by only listing few edges in each update. For this we require another property for $\mathcal{H}$ that guarantees that the sparsifier stays valid, even when removing a few of its edges.

Transition Property

Consider some $H_1, H_2 \in \mathcal{H}(G, \epsilon)$, and we now want to have a slow transition from H_1 to H_2, by slowly removing edges from H_1 from the output (and slowly inserting edges of H_2). The exact property we require is as follows:

$$\text{Let } H_1, H_2 \in \mathcal{H}(G, \epsilon) \text{ and } H \subset H_1, \text{ then } (e^{\delta} - 1)H \cup H_2 \in \mathcal{H}(G, \epsilon + \delta). \tag{4}$$

Here $H \subset H_1$ represents the remaining to be removed edges (or alternatively $H_1 \setminus H$ are the remaining to be inserted edges). Exploiting this property we are able to obtain a $O(p/k + u)$ worst-case update time.

As the output grows with each update, we must perform the reset after $k = O(n)$ updates, otherwise the output becomes too dense. This is unfortunate as the preprocessing time is $p = \Omega(m)$, because one must read the entire input, which is too slow to obtain a subpolynomial update time. This issue can be fixed via a sparsification technique based on [49], presented in the full version. By using this technique, we can make sure that $m = O(n^{1+o(1)})$ and thus the preprocessing time will be fast enough to allow for $O(n^{o(1)})$ update time.

For this sparsification technique we require the following transitivity property.

Transitivity Property

$$\text{If } H \in \mathcal{H}(G, \epsilon), \text{ then } \mathcal{H}(H, \delta) \subset \mathcal{H}(G, \delta + \epsilon). \tag{5}$$

Intuitively this means that an approximation H of G and an approximation H' of H, then H' is also a (slightly worse) approximation of G.

Properties 1-5 above are precisely the properties required of a graph problem by our black-box reduction for worst-case update time. We show in the full version that the sparsifiers discussed in this paper (spectral sparsifier, cut sparsifier, spanner) satisfy all these properties.

Sparsification Technique

We now outline the sparsification technique, whose formal proof is presented in the full version. Let G be an arbitrary graph. We partition the edges of G into equally sized subgraphs $G_1, G_2, ..., G_d$ for some $d > 1$. Note that, if we have ε-approximate sparsifiers $H_1, ..., H_d$ of $G_1, ..., G_d$, then $\bigcup_i H_i$ is a ε-approximate sparsifier of G by property (2). In addition, if we have a ε-approximate sparsifier H of $\bigcup_i H_i$, then H is a (2ε)-approximate sparsifier of G by property (5). This allows us to obtain a faster algorithm as follows: If G has m edges, then each G_i has only m/d edges, so the dynamic algorithm runs faster on these sparse G_i. Further, since the H_i are sparse (let's say $O(n)$ edges), the graph $\bigcup_i H_i$ has only $O(dn)$ edges and maintaining H is also faster than maintaining a sparsifier of G directly, if $dn = o(m)$. The next idea is to repeat this trick recursively: We repeatedly split each G_i into $d = n^{o(1)}$ graphs, until the graphs have only $O(n^{1+o(1)})$ edges. This means we obtain some tree-like structure rooted at G, where each tree-node G' represents a subgraph of G and its tree-children are the d subgraphs $G'_1, ..., G'_d$ of G'. For the graphs that form leaves of this tree, we run our dynamic sparsifier algorithm. We also obtain a sparsifier H' of any non-leaf tree-node G', by running our dynamic algorithm on $\bigcup_{i=1}^{d} H'_i$, where the H'_i are sparsifiers of the child-tree-nodes G'_i of the tree-node G'. Thus all instances of our dynamic algorithm always run on sparse input graphs. However, there is one downside: When some sparsifier H'_i changes, the sparsifier H' must also change. Let's say some edge is deleted from G, then the edge is deleted from one leaf-node of the tree-structure, and this update will propagate

from the leaf-node all the way to the root of the tree. This can be problematic because when the dynamic algorithm changes some $c > 1$ many edges of the sparsifier for each edge update, then the number of updates grows exponentially with the depth of the tree-like structure. In [49] Eppstein et al. circumvented this issue by assuming an extra property which they call *stability property*, which essentially says that this exponential growth does not occur. Our modified sparsification technique no longer requires this assumption, instead we balance the parameter d carefully to make sure the blow-up of the propagation is only some sub-polynomial factor.

References

1 Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *FOCS*, pages 335–344, 2016. doi:10.1109/FOCS.2016.44.

2 Abu Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen G. Kobourov, and Richard Spence. Graph spanners: A tutorial review. *CoRR*, abs/1909.03152, 2019.

3 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14, 2012.

4 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *APPROX-RANDOM*, pages 1–10, 2013.

5 Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993. doi:10.1007/BF02189308.

6 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *STOC*, pages 815–826. ACM, 2018.

7 Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. Small stretch spanners on dynamic graphs. *J. Graph Algorithms Appl.*, 10(2):365–385, 2006. Announced at ESA'05. URL: http://jgaa.info/accepted/2006/AusielloFranciosaItaliano2006.10.2.pdf, doi:10.7155/jgaa.00133.

8 Surender Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Inf. Process. Lett.*, 106(3):110–114, 2008. doi:10.1016/j.ipl.2007.11.001.

9 Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $o(\log n)$ update time (corrected version). *SIAM J. Comput.*, 47(3):617–650, 2018. doi:10.1137/16M1106158.

10 Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012. Announced at SODA'08. doi:10.1145/2344422.2344425.

11 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007. doi:10.1002/rsa.20130.

12 Joshua D. Batson, Daniel A. Spielman, and Nikhil Srivastava. Twice-Ramanujan sparsifiers. *SIAM Rev.*, 56(2):315–334, 2014.

13 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *FOCS*, pages 382–405. IEEE Computer Society, 2019.

14 Amos Beimel, Haim Kaplan, Yishay Mansour, Kobbi Nissim, Thatchaphol Saranurak, and Uri Stemmer. Dynamic algorithms against an adaptive adversary: Generic constructions and lower bounds. *arXiv preprint*, 2021. To appear at STOC'22. arXiv:2111.03980.

15 András A Benczúr and David R Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM Journal on Computing*, 44(2):290–319, 2015.

16 Aaron Bernstein. Deterministic partially dynamic single source shortest paths in weighted graphs. In *ICALP*, volume 80, pages 44:1–44:14, 2017. doi:10.4230/LIPIcs.ICALP.2017.44.

17 Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the $o(mn)$ bound. In *STOC*, pages 389–397, 2016.

18 Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *SODA*, pages 453–469, 2017.

19 Aaron Bernstein and Shiri Chechik. Incremental topological sort and cycle detection in expected total time. In *SODA*, pages 21–34. SIAM, 2018.

20 Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918, 2019.

21 Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. *arXiv preprint*, 2021. arXiv:2101.07149.

22 Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *STOC*, pages 365–376, 2019.

23 Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *SODA*, pages 1355–1365, 2011.

24 Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. *CoRR*, abs/2004.08432, 2020.

25 Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *SODA*, pages 1–20, 2018.

26 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA*, 2015.

27 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC*, pages 398–411, 2016.

28 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *SODA*, pages 470–489, 2017.

29 Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *SODA*, 2019.

30 Sayan Bhattacharya, Thatchaphol Saranurak, and Pattara Sukprasert. Simple dynamic spanners with near-optimal recourse against an adaptive adversary. In submission.

31 Greg Bodwin and Sebastian Krinninger. Fully dynamic spanners with worst-case update time. In *ESA*, pages 17:1–17:18, 2016. doi:10.4230/LIPIcs.ESA.2016.17.

32 Karl Bringmann and Konstantinos Panagiotou. Efficient sampling methods for discrete distributions. In *ICALP*, pages 133–144. Springer, 2012.

33 Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *PODC*, pages 217–226. ACM, 2016.

34 Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In *PODC*, pages 66–73, 2019.

35 Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *FOCS*, 2018.

36 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *FOCS*, pages 370–381. IEEE Computer Society, 2019.

37 Chandra Chekuri and Kent Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 801–820. SIAM, 2017.

38 Chandra Chekuri and Kent Quanrud. Fast approximations for metric-tsp via linear programming. *arXiv preprint*, 2018. arXiv:1802.01242.

39 Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *FOCS*, pages 1135–1146. IEEE, 2020.

40 Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. Graph sparsification, spectral sketches, and faster resistance computation, via short cycle decompositions. In *FOCS*, pages 361–372, 2018. doi:10.1109/FOCS.2018.00042.

41 Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. *CoRR*, abs/1910.08025, 2019.

42 Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *STOC*, pages 389–400, 2019.

43 Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In *SODA*, pages 2478–2496. SIAM, 2021.

44 Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Sublinear fully distributed partition with applications. *Theory Comput. Syst.*, 47(2):368–404, 2010. doi:10.1007/s00224-009-9190-x.

45 Luc Devroye. Nonuniform random variate generation. *Handbooks in operations research and management science*, 13:83–121, 2006.

46 David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. Fully dynamic effective resistances. *CoRR*, abs/1804.04038, 2018.

47 David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. Fully dynamic spectral vertex sparsifiers and applications. In *STOC*, pages 914–925, 2019.

48 Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2):20:1–20:17, 2011. Announced at ICALP'07. doi:10.1145/1921659.1921666.

49 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

50 Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

51 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: the value of space. In *SODA*, pages 745–754, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070537.

52 Lisa Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000. announced at FOCS'99. doi:10.1137/S0895480199355754.

53 Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. In *STOC*, pages 71–80, 2011.

54 Yu Gao, Yang P Liu, and Richard Peng. Fully dynamic electrical flows: sparse maxflow faster than goldberg-rao. *arXiv preprint*, 2021. arXiv:2101.07233.

55 Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37(2):630–652, 2007. doi:10.1137/S0097539704446232.

56 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in o(m $\log^2$ n) time. In *ICALP*, volume 168 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

57 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In *DISC*, pages 29:1–29:17, 2018. doi:10.4230/LIPIcs.DISC.2018.29.

58 Ofer Grossman and Merav Parter. Improved deterministic distributed construction of spanners. In *DISC*, pages 24:1–24:16, 2017. doi:10.4230/LIPIcs.DISC.2017.24.

59 Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Sahil Singla. Online carpooling using expander decompositions. In *FSTTCS*, volume 182 of *LIPIcs*, pages 23:1–23:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

60 Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In *Symposium on Theory of Computing*, 2020. arXiv:2001.10751.

61 Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Decremental SSSP in weighted digraphs: Faster and against an adaptive adversary. In *SODA*, pages 2542–2561, 2020.

62 Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *SODA*, pages 2522–2541, 2020.

63 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. *J. ACM*, 65(6):36:1–36:40, 2018. announced at FOCS'14.

64 Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 594–604. Springer, 1997.

65 Jacob Holm and Eva Rotenberg. Dynamic planar embeddings of dynamic graphs. *Theory Comput. Syst.*, 61(4):1054–1083, 2017.

66 Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *STOC*, pages 167–180. ACM, 2020.

67 Arun Jambulapati and Aaron Sidford. Efficient $\tilde{O}(n/\epsilon)$ spectral sketches for the laplacian and its pseudoinverse. In *SODA*, pages 2487–2503. SIAM, 2018.

68 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1142, 2013. doi:10.1137/1.9781611973105.81.

69 George Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Trans. Algorithms*, 4(1):13:1–13:17, 2008. doi:10.1145/1328911.1328924.

70 Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *STOC*, pages 665–674. ACM, 2015.

71 Donald Ervin Knuth. Seminumerical algorithms. *The art of computer programming*, 2, 1997.

72 Huan Li, He Sun, and Luca Zanetti. Hermitian Laplacians and a Cheeger inequality for the Max-2-Lin problem. In *ESA*, pages 71:1–71:14, 2019.

73 Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *STOC*, pages 121–130, 2010. doi:10.1145/1806689.1806708.

74 Morteza Monemizadeh. Dynamic maximal independent set. *CoRR*, abs/1906.09595, 2019.

75 Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: Sequential, cut-query and streaming algorithms. In *STOC*, 2020.

76 Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las vegas, and $O(n^{1/2-\epsilon})$-time. In *STOC*, pages 1122–1129, 2017. doi:10.1145/3055399.3055447.

77 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961, 2017.

78 Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In *ICALP*, pages 261–272, 2005. doi:10.1007/11523468_22.

79 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. doi:10.1007/s00453-010-9401-5.

80 Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *SODA*, pages 2616–2635, 2019.

81 Jonah Sherman. Nearly maximum flows in nearly linear time. In *FOCS*, pages 263–269, 2013.

82 Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS*, pages 325–334, 2016. doi:10.1109/FOCS.2016.43.

83 Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *STOC*, pages 81–90, 2004.

84 Luca Trevisan. Approximation algorithms for unique games. In *FOCS*, pages 197–205. IEEE Computer Society, 2005.

85 Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and l1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021.

86 Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *FOCS*, pages 919–930. IEEE, 2020.

87 David Wajc. Rounding dynamic matchings against an adaptive adversary. *Symposium on Theory of Computing*, 2020. `arXiv:1911.05545`.

88 Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *STOC*, pages 1130–1143, 2017. `doi:10.1145/3055399.3055415`.

89 Anastasios Zouzias. A matrix hyperbolic cosine algorithm and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 846–858. Springer, 2012.