Simple Dynamic Spanners with Near-Optimal Recourse Against an Adaptive Adversary

Sayan Bhattacharya
University of Warwick, Coventry, UK

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

Pattara Sukprasert
Northwestern University, Evanston, IL, USA

Abstract

Designing dynamic algorithms against an adaptive adversary whose performance match the ones assuming an oblivious adversary is a major research program in the field of dynamic graph algorithms. One of the prominent examples whose oblivious-vs-adaptive gap remains maximally large is the *fully dynamic spanner* problem; there exist algorithms assuming an oblivious adversary with near-optimal size-stretch trade-off using only polylog(n) update time [Baswana, Khurana, and Sarkar TALG'12; Forster and Goranci STOC'19; Bernstein, Forster, and Henzinger SODA'20], while against an adaptive adversary, even when we allow infinite time and only count recourse (i.e. the number of edge changes per update in the maintained spanner), all previous algorithms with stretch at most $\log^5(n)$ require at least $\Omega(n)$ amortized recourse [Ausiello, Franciosa, and Italiano ESA'05].

In this paper, we completely close this gap with respect to recourse by showing algorithms against an adaptive adversary with near-optimal size-stretch trade-off and recourse. More precisely, for any $k \geq 1$, our algorithm maintains a $(2k-1)$-spanner of size $O(n^{1+1/k}\log n)$ with $O(\log n)$ amortized recourse, which is optimal in all parameters up to a $O(\log n)$ factor. As a step toward algorithms with small update time (not just recourse), we show another algorithm that maintains a 3-spanner of size $\tilde{O}(n^{1.5})$ with polylog(n) amortized recourse *and* simultaneously $\tilde{O}(\sqrt{n})$ worst-case update time.

2012 ACM Subject Classification Theory of computation → Sparsification and spanners

Keywords and phrases Algorithms, Dynamic Algorithms, Spanners, Recourse

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.17

Related Version *Full Version*: arXiv

Funding Sayan Bhattacharya is supported by Engineering and Physical Sciences Research Council, UK (EPSRC) Grant EP/S03353X/1.

1 Introduction

Increasingly, algorithms are used interactively for data analysis, decision making, and classically as data structures. Often it is not realistic to assume that a user or an adversary is *oblivious* to the outputs of the algorithms; they can be *adaptive* in the sense that their updates and queries to the algorithm may depend on the previous outputs they saw. Unfortunately, many classical algorithms give strong guarantees only when assuming an oblivious adversary. This calls for the design of algorithms that work against an adaptive adversary whose performance match the ones assuming an oblivious adversary. Driven by this question, there have been exciting lines of work across different communities in theoretical computer science, including streaming algorithms against an adaptive adversary [10, 52, 66, 3, 56, 28], statistical algorithms against an adaptive data analyst [51, 37, 7, 63], and very recent algorithms for machine unlearning [47].

In the area of this paper, namely dynamic graph algorithms, a continuous effort has also been put on designing algorithms against an adaptive adversary. This is witnessed by dynamic algorithms for maintaining spanning forests [53, 60, 67, 61, 34], shortest paths [13, 11, 14, 35, 36, 49, 50, 48, 33], matching [20, 21, 22, 24, 64, 23], and more. This development led to new powerful tools, such as the expander decomposition and hierarchy [62, 42, 59] applicable beyond dynamic algorithms [57, 58, 1, 68], and other exciting applications such as the first almost-linear time algorithms for many flow and cut problems [27, 26, 33, 17]. Nevertheless, for many fundamental dynamic graph problems, including graph sparsifiers [2], reachability [18], directed shortest paths [49], the performance gap between algorithms against an oblivious and adaptive adversary remains large, waiting to be explored and, hopefully, closed.

One of the most prominent dynamic problems whose oblivious-vs-adaptive gap is maximally large is the *fully dynamic spanner* problem [5, 38, 8, 25, 40, 16, 12]. Given an unweighted undirected graph $G = (V, E)$ with n vertices, an *α-spanner* H is a subgraph of G whose pairwise distances between vertices are preserved up to the *stretch* factor of α, i.e., for all $u, v \in V$, we have $\mathrm{dist}_G(u, v) \leq \mathrm{dist}_H(u, v) \leq \alpha \cdot \mathrm{dist}_G(u, v)$.[1] In this problem, we want to maintain an α-spanner of a graph G while G undergoes both edge insertions and deletions, and for each edge update, spend as small update time as possible.

Assuming an oblivious adversary, near-optimal algorithms have been shown: for every $k \geq 1$, there are algorithms maintaining a $(2k-1)$-spanner containing $\tilde{O}(n^{1+1/k})$ edges[2], which is nearly tight with the $\Omega(n^{1+1/k})$ bound from Erdős' girth conjecture (proven for the cases where $k = 1, 2, 3, 5$ [65]). Their update times are either $O(k \log^2 n)$ amortized [8, 40] or $O(1)^k \log^3 n$ worst-case [16], both of which are polylogarithmic when k is a constant.

In contrast, the only known dynamic spanner algorithm against an adaptive adversary by [5] requires $O(n)$ amortized update time and it can maintain a $(2k-1)$-spanner of size $O(n^{1+1/k})$ only for $k \leq 3$. Whether the $O(n)$ bound can be improved remained open until very recently. Bernstein et al. [12] show that a $\log^6(n)$-spanner can be maintained against an adaptive adversary using $\mathrm{polylog}(n)$ amortized update time. The drawback, however, is that their expander-based technique is too crude to give any stretch smaller than $\mathrm{polylog}(n)$. Hence, for $k \leq \log^6(n)$, it is still unclear if the $\Theta(n)$ bound is inherent. Surprisingly, this holds even if we allow infinite time, and only count *recourse*, i.e., the number of edge changes per update in the maintained spanner. The stark difference in performance between the two adversarial settings motivates the main question of this paper:

Is the $\Omega(n)$ recourse bound inherent for fully dynamic spanners against an adaptive adversary?

Recourse is an important performance measure of dynamic algorithms. There are dynamic settings where changes in solutions are costly while computation itself is considered cheap, and so the main goal is to directly minimize recourse [45, 44, 6, 46]. Even when the final goal is to minimize update time, there are many dynamic algorithms that crucially require the design of subroutines with recourse bounds *stronger than* update time bounds to obtain small final update time [31, 42, 32]. Historically, there are dynamic problems, such as planar embedding [55, 54] and maximal independent set [29, 9, 30], where low recourse algorithms were first discovered and later led to fast update-time algorithms. Similar to dynamic spanners, there are other fundamental problems, including topological sorting [15] and edge coloring [19], for which low recourse algorithms remain the crucial bottleneck to faster update time.

[1] Here, $\mathrm{dist}_G(u, v)$ denotes the distance between u and v in graph G.
[2] $\tilde{O}(\cdot)$ hides a $\mathrm{polylog}(n)$ factor.

In this paper, we successfully break the $O(n)$ recourse barrier and completely close the oblivious-vs-adaptive gap with respect to recourse for fully dynamic spanners against an adaptive adversary.[3]

▶ **Theorem 1.** *There exists a deterministic algorithm that, given an unweighted graph G with n vertices undergoing edge insertions and deletions and a parameter $k \geq 1$, maintains a $(2k-1)$-spanner of G containing $O(n^{1+1/k} \log n)$ edges using $O(\log n)$ amortized recourse.*

As the above algorithm is deterministic, it automatically works against an adaptive adversary. Each update can be processed in polynomial time. Both the recourse and stretch-size trade-off of Theorem 1 are optimal up to a $O(\log n)$ factor. When ignoring the update time, it even dominates the current best algorithm assuming an oblivious adversary [8, 40] that maintains a $(2k-1)$-spanner of size $O(n^{1+1/k} \log n)$ using $O(k \log^2 n)$ recourse. Therefore, the oblivious-vs-adaptive gap for amortized recourse is closed.

The algorithm of Theorem 1 is as simple as possible. As it turns out, a variant of the classical greedy spanner algorithm [4] simply does the job! Although the argument is short and "obvious" in hindsight, for us, it was very surprising. This is because the greedy algorithm *sequentially* inspects edges in some fixed order, and its output solely depends on this order. Generally, long chains of dependencies in algorithms are notoriously hard to analyze in the dynamic setting. More recently, a similar greedy approach was dynamized in the context of dynamic maximal independent set [9] by choosing a *random order* for the greedy algorithm. Unfortunately, the random order must be kept secret from the adversary and so this fails completely in our adaptive setting. Despite these intuitive difficulties, our key insight is that we can *adaptively choose the order* for the greedy algorithm after each update. This simple idea is enough, see Section 2 for details.

Theorem 1 leaves open the oblivious-vs-adaptive gap for the update time. Below, we show a partial progress on this direction by showing an algorithm with near-optimal recourse and simultaneously non-trivial update time.

▶ **Theorem 2.** *There exists a randomized algorithm that, given an unweighted graph G with n vertices undergoing edge insertions and deletions, with high probability maintains against an adaptive adversary a 3-spanner of G containing $\tilde{O}(n^{1.5})$ edges using $\tilde{O}(1)$ amortized recourse* and *$\tilde{O}(\sqrt{n})$ worst-case update time.*

We note again that, prior to the above result, there was no algorithm against an adaptive adversary with $o(n)$ *amortized update time* that can maintain a spanner of stretch less than $\log^6(n)$. Theorem 2 shows that for 3-spanners, the update time can be $\tilde{O}(\sqrt{n})$ worst-case, while guaranteeing near-optimal recourse.

We prove Theorem 2 by employing a technique called *proactive resampling*, which was recently introduced in [12] for handling an adaptive adversary. We apply this technique on a modification of a spanner construction by Grossman and Parter [43] from distributed computation community. The modification is small, but seems inherently necessary for bounding the recourse.

To successfully apply proactive resampling, we refine the technique in two ways. First, we present a simple abstraction in terms of a certain load balancing problem that captures the power of proactive resampling. Previously, the technique was presented and applied specifically for the dynamic cut sparsifier problem [12]. But actually, this technique is

[3] We wish to mention here that our algorithm works against an adaptive adversary that, in addition to just seeing the algorithm's output, also has access to the internal randomness of the algorithm.

Table 1 The state of the art of fully dynamic spanner algorithms.

Ref.	Stretch	Size	Recourse	Update Time	Deterministic?
Against an oblivious adversary					
[8]	$2k-1$	$O(k^8 n^{1+1/k}\log^2 n)$	$O(7^{k/2})$ amortized		rand. oblivious
[8, 40]	$2k-1$	$O(n^{1+1/k}\log n)$	$O(k\log^2 n)$ amortized		rand. oblivious
[16]	$2k-1$	$\tilde{O}(n^{1+1/k})$	$O(1)^k\log^3 n$ worst-case		rand. oblivious
Against an adaptive adversary					
[5]	3	$O(n^{1+1/2})$	$O(\Delta)$ amortized		deterministic
	5	$O(n^{1+1/3})$	$O(\Delta)$ amortized		deterministic
[12]	$\tilde{O}(1)$	$\tilde{O}(n)$	$\tilde{O}(1)$ amortized		rand. adaptive
	$n^{o(1)}$	$\tilde{O}(n)$	$n^{o(1)}$ worst-case		deterministic
Ours	$2k-1$	$O(n^{1+1/k}\log n)$	$O(\log n)$ amortized	$\mathrm{poly}(n)$ worst-case	deterministic
	3	$\tilde{O}(n^{1+1/2})$	$\tilde{O}(1)$ amortized	$\tilde{O}(\sqrt{n})$ worst-case	rand. adaptive
	3	$O(n^{1+1/2})$	$O(\min\{\Delta,\sqrt{n}\}\log n)$ worst-case		deterministic

conceptually simple and quite generic, so our new abstraction will likely facilitate future applications. Our second technical contribution is to generalize and make the proactive resampling technique more flexible. At a very high level, in [12], there is a single parameter about sampling probability that is *fixed* throughout the whole process, and their analysis requires this fact. In our load-balancing abstraction, we need to work with multiple sampling probabilities and, moreover, they change through time. We manage to analyze this generalized process using probabilistic tools about *stochastic domination*, which in turn simplifies the whole analysis.

If a strong recourse bound is not needed, then proactive resampling can be bypassed and the algorithm becomes very simple, deterministic, and has slightly improved bounds as follows.

▶ **Theorem 3.** *There exists a deterministic algorithm that, given an unweighted graph G with n vertices undergoing edge insertions and deletions, maintains a 3-spanner of G containing $O(n^{1.5})$ edges using $O(\min\{\Delta,\sqrt{n}\}\log n)$ worst-case update time, where Δ is the maximum degree of G.*

Despite its simplicity, the above result improves the update time of the fastest deterministic dynamic 3-spanner algorithm [5] from $O(\Delta)$ amortized to $O(\min\{\Delta,\sqrt{n}\}\log n)$ worst-case. In fact, all previous dynamic spanner algorithms with worst-case update time either assume an oblivious adversary [38, 25, 16] or have a very large stretch of $n^{o(1)}$ [12]. See Table 1 for detailed comparison.

Organization. In Section 2, we give a very short proof of Theorem 1. In Section 3, we prove Theorem 2 assuming a crucial lemma (Lemma 8) needed for bounding the recourse. To prove this lemma, we show a new abstraction for the proactive resampling technique in Section 4 and complete the analysis in Section 5. Our side result, Theorem 3, is based on the the static construction presented in Section 3.1 and its simple proof is given in Section 3.2.

2 Deterministic Spanner with Near-optimal Recourse

Below, we show a decremental algorithm that *handles edge deletions only* with near-optimal recourse. This will imply Theorem 1 by a known reduction formally stated in Lemma 6. To describe our decremental algorithm, let us first recall the classic greedy algorithm.

The Greedy Algorithm. Althöfer et al. [4] showed the following algorithm for computing $(2k-1)$-spanners. Given a graph $G = (V, E)$ with n vertices, fix some *order* of edges in E. Then, we inspect each edge one by one according to the order. Initially $E_H = \emptyset$. When we inspect $e = (u, v)$, if $\text{dist}_H(u, v) \geq 2k$, then add e into E_H. Otherwise, ignore it. We have the following:

▶ **Theorem 4** ([4]). *The greedy algorithm above outputs a $(2k-1)$-spanner $H = (V, E_H)$ of G containing $O(n^{1+1/k})$ edges.*

It is widely believed that the greedy algorithm is extremely bad in dynamic setting: an edge update can drastically change the greedy spanner. In contrary, when we allow the order in which greedy scans the edges to be changed adaptively, we can avoid removing spanner edges until it is deleted by the adversary. This key insight leads to optimal recourse. When recourse is the only concern, prior to our work this result was known only for spanners with polylog stretch, which is a much easier problem.

The Decremental Greedy Algorithm. Now we describe our deletion-only algorithm. Let G be an initial graph with m edges and $H = (V, E_H)$ be a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges. Suppose an edge $e = (u, v)$ is deleted from the graph G. If $(u, v) \notin E_H$, then we do nothing. Otherwise, we do the following. We first remove e from E_H. Now we look at the only remaining non-spanner edges $E \setminus E_H$, one by one in an arbitrary order. (Note that the order is *adaptively* defined and not fixed through time because it is defined only on $E \setminus E_H$.) When we inspect $(x, y) \in E \setminus E_H$, as in the greedy algorithm, we ask if $\text{dist}_H(x, y) \geq 2k$ and add (x, y) to E_H if and only if it is the case. This completes the description of the algorithm.

Analysis. We start with the most crucial point. We claim that the new output after removing e is as if we run the greedy algorithm that first inspects edges in E_H (the order within E_H is preserved) and later inspects edges in $E \setminus E_H$.

To see the claim, we argue that if the greedy algorithm inspects E_H first, then the whole set E_H must be included, just like E_H remains in the new output. To see this, note that, for each $(x, y) \in E_H$, $\text{dist}_H(x, y) \geq 2k$ when (x, y) was inspected according to some order. But, if we move the whole set E_H to be the prefix of the order (while the order within E_H is preserved), it must still be the case that $\text{dist}_H(x, y) \geq 2k$ when (x, y) is inspected and so e must be added into the spanner by the greedy algorithm.

So our algorithm indeed "simulates" inspecting E_H first, and then it explicitly implements the greedy algorithm on the remaining part $E \setminus E_H$. So we conclude that it simulates the greedy algorithm. Therefore, by Theorem 4, the new output is a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges.

The next important point is that, whenever an edge e added into the spanner H, the algorithm never tries to remove e from H. So e remains in H until it is deleted by the adversary. Therefore, the total recourse is $O(m)$. With this, we conclude the following key lemma:

▶ **Lemma 5.** *Given a graph G with n vertices and m initial edges undergoing only edge deletions, the algorithm above maintains a $(2k-1)$-spanner H of G of size $O(n^{1+1/k})$ with $O(m)$ total recourse.*

By plugging Lemma 5 to the fully-dynamic-to-decremental reduction by [8] below, we conclude Theorem 1.

▶ **Lemma 6** ([8]). *Suppose that for a graph G with n vertices and m initial edges undergoing only edge deletions, there is an algorithm that maintains a $(2k-1)$-spanner H of size $O(S(n))$ with $O(F(m))$ total recourse where $F(m) = \Omega(m)$, then there exists an algorithm that maintains a $(2k-1)$-spanner H' of size $O(S(n)\log n)$ in a fully dynamic graph with n vertices using $O(F((U)\log n))$ total recourse. Here U is the total number of updates, starting from an empty graph.*

3 3-Spanner with Near-optimal Recourse and Fast Update Time

In this section, we prove Theorem 2 by showing an algorithm for maintaining a 3-spanner with small update time *in addition* to having small recourse. We start by explaining a basic static construction and needed data structures in Section 3.1 and show the dynamic algorithm in Section 3.2. Assuming our key lemma (Lemma 8) about proactive resampling, most details here are quite straight forward. Hence, due to space constraint, some proofs are omitted. They will appear in our full version.

Throughout this section, we let $N_G(u) = \{v \in V : (u,v) \in E\}$ denote the set of neighbors of a node $u \in V$ in a graph $G = (V,E)$, and we let $\deg_G(u) = |N_G(u)|$ denote the degree of the node u in the graph G.

3.1 A Static Construction and Basic Data Structures

A Static Construction. We now describe our static algorithm. Though our presentation is different, our algorithm is almost identical to [43]. The only difference is that we do not distinguish small-degree vertices from large-degree vertices.

We first arbitrarily partition V into $\sqrt{n}$ equal-sized *buckets* $V_1, \ldots, V_{\sqrt{n}}$. We then construct three sets of edges E_1, E_2, E_3. For every bucket $V_i, i \in [1, \sqrt{n}]$, we do the following. First, for all $v \in V \setminus V_i$, if $V_i \cap N_G(v)$ is not empty, we choose a neighbor $c_i(v) \in V_i \cap N_G(v)$ and add $(v, c_i(v))$ to E_1. We call $c_i(v)$ an *i-partner* of v. Next, for every edge $e = (u,v)$, where both $u, v \in V_i$, we add e to E_2. Lastly, for $u, u' \in V_i$ with overlapping neighborhoods, we pick an arbitrary common neighbor $w_{uu'} \in N_G(u) \cap N_G(u')$ and add $(u, w_{uu'}), (w_{uu'}, u')$ to E_3. We refer to the node $w_{uu'}$ as the *witness* for the pair u, u'.

▷ Claim 7. The subgraph $H = (V, E_1 \cup E_2 \cup E_3)$ is a 3-spanner of G consisting of at most $O(n\sqrt{n})$ edges.

Proof. We need to show that (1) H is a 3-spanner and (2) $E_H = E_1 \cup E_2 \cup E_3$ has size at most $O(n\sqrt{n})$.

Stretch. Consider an edge $e = (u,v)$ where $u \in V_i, v \in V_j$. We show that H has a path of length at most 3 between u and v. The easy case is when $(u,v) \in E_H$. This gives us a path of one edge. It happens when $u = c_i(v)$ or $v = c_j(u)$, or $i = j$. Suppose $(u,v) \notin E_H$. Consider $v' = c_j(u)$. Since u is a common neighbor between v and v', $P(v,v')$ is not empty. As $e \notin E_H$, $u \neq w_{vv'}$. As, the path $u, v', w_{vv'}, v$ has length exactly 3, the stretch part is concluded.

Size. Each vertex u has upto $\sqrt{n}$ partners. Since we have n vertices, $|E_1| = O(n\sqrt{n})$. For E_2, the graph induced on V_i has at most $O(n)$ edges. Since we have $\sqrt{n}$ buckets, $|E_2| = O(n\sqrt{n})$. For E_3, $|E_3|$ is bounded by the number of witnesses we need. Since we have $O\sqrt{n}$ buckets, and we have $O(n)$ pairs of vertices within the same bucket, for all buckets, $|E_3|$ must be bounded by $O(n\sqrt{n})$. We conclude the proof by saying that $|E_H| = O(|E_1| + |E_2| + |E_3|) = O(n\sqrt{n})$ ◁

Dynamizing the Construction. Notice that it suffices to separately maintain E_1, E_2, E_3, in order to maintain the above dynamic 3-spanner. Maintaining E_1 and E_2 is straightforward and can be done in a fully-dynamic setting in $O(1)$ worst-case update time. Indeed, if $e = (u, c_i(u)) \in E_1$ is deleted, then we pick a new i-partner $c_i(u) \in V_i \cap N_G(u)$ for u. Maintaining $V_i \cap N_G(u)$ for all u allows us to update $c_i(u)$ efficiently. If $e = (u, u') \in E_2$, where $u, u' \in V_i$, is deleted, then we do nothing.

The remaining task, maintaining E_3, is the most challenging part of our dynamic algorithm. Before we proceed, let us define a subroutine and a data structure needed to implement our algorithm.

Resampling Subroutine. We define $\textsc{Resample}(u, u')$ as a subroutine that *uniformly samples* a witness $w_{uu'}$ (i.e. a common neighbor of u and u'), if exists. Notice that, we can obtain E_3 by calling $\textsc{Resample}(u, u')$ for all $u, u' \in V_i$ and for all $i \in [1, \sqrt{n}]$.

Algorithm 1 Maintaining a 3-spanner after one edge deletion.

```
Data: $G = (V, E)$, $H = (V, E_1 \cup E_2 \cup E_3)$
Input: $e = (u, v)$ is an edge being deleted
Result: $G' = (V, E')$ , $H' = (V, E'_1 \cup E'_2, E'_3)$
begin
    $E' \longleftarrow E \setminus e$
    Let $i_u$ be such that $u \in V_{i_u}$
    Let $i_v$ be such that $v \in V_{i_v}$

    if $u = c_{i_u}(v)$ then
        Reselect $c_{i_u}(v)$ from $V_{i_u} \cap N_{G'}(v)$
    if $v = c_{i_v}(u)$ then
        Reselect $c_{i_v}(u)$ from $V_{i_v} \cap N_{G'}(u)$
    $E'_1 \longleftarrow E_1 \setminus e \cup \{(u, c_{i_u}(v)), (v, c_{i_v}(u))\}$

    if $i_u = i_v$ then
        $E'_2 \longleftarrow E_2 \setminus e$

    $E'_3 \longleftarrow E_3 \setminus e$
    for $u' \in V_{i_u} \setminus u$ do
        if $v = w_{uu'}$ then
            $w_{uu'} \longleftarrow \textsc{Resample}(u, u')$ /* $\textsc{Resample}(\cdot,\cdot)$ is drawn from $G'$ */

            $E'_3 \longleftarrow E'_3 \cup \{(u, w_{uu'}), (w_{uu'}, u')\}$
    for $v' \in V_{i_v} \setminus v$ do
        if $u = w_{vv'}$ then
            $w_{vv'} \longleftarrow \textsc{Resample}(v, v')$
            $E'_3 \longleftarrow E'_3 \cup \{(v, w_{vv'}), (w_{vv'}, v')\}$
```

Partnership Data Structures. The subroutine above hints that we need a data structure for maintaining the common neighborhoods for all pairs of vertices that are in the same bucket. For vertices u and v within the same bucket, we let $P(u, v) = N_G(u) \cap N_G(v)$ be the *partnership* between u and v. To maintain these structures dynamically, when an edge (u, v) is inserted, if $u \in V_i$ and $v \in V_j$, we add v to $P(u, u')$ for all $u' \in V_i \cap N_G(v) \setminus \{u\}$, and symmetrically add u to $P(v, v')$ for all $v' \in V_j \cap N_G(u) \setminus \{v\}$. This clearly takes $O(\sqrt{n} \log n)$ worst-case time for edge insertion. and this is symmetric for edge deletion.

As we want to prove that our final update time is $\tilde{O}(\sqrt{n})$, we can assume from now that E_1, E_2, and all partnerships are maintained in the background. The pseudocode for maintaining the 3-spanner is provided in Algorithm 1.

3.2 Maintaining Witnesses via Proactive Resampling

Remark. For clarity of exposition, we will present an amortized update time analysis. Using standard approach, we can make the update time worst case. We will discuss this issue at the end of this section.

Our dynamic algorithm runs in *phases*, where each phase lasts for $n\sqrt{n}$ consecutive updates (edge insertions/deletions). As a spanner is decomposable[4], it suffices to maintain a 3-spanner H of the graph undergoing only edge deletions within this phase and then include all edges inserted within this phase into H, which increases the size of H by at most $n\sqrt{n}$ edges. Henceforth, *we only need to present how to initialize a phase and how to handle edge deletions within each phase.* The reason behind this reduction is because our proactive resampling technique naturally works for decremental graphs.

Initialization. At the start of the phase, since our partnerships structures only processed edge deletions from the previous phase, we first update partnerships with all the $O(n\sqrt{n})$ inserted edges from the previous phase. Then, we call $\text{Resample}(u, u')$ for all $u, u' \in V_i$ for all $i \in [1, \sqrt{n}]$ to replace all witnesses and initialize E_3 of this phase.

Difficulty of Bounding Recourse. Maintaining E_3 (equivalently, the witnesses) in $\tilde{O}(\sqrt{n})$ worst-case time is straightforward because the partnership data structure has $O(\sqrt{n} \log n)$ update time. However, our goal is to show $\tilde{O}(1)$ amortized recourse, which is the most challenging part of our analysis. To see the difficulty, if (u, v) is deleted and $u \in V_i$, a vertex v may serve as a witness $\{w_{uu'}\}$ for all $u' \in V_i$. In this case, deleting (u, v) causes the algorithm to find a new witness $w_{uu'}$ for all $u' \in V_i$. This implies a recourse of $|V_i| = \Omega(\sqrt{n})$. To circumvent this issue, we apply the technique of *proactive resampling*, as described below.

Proactive Resampling. We keep track of a *time-variable* T; the number of updates to G that have occurred in this phase until now. T is initially 0. We increment T each time an edge gets deleted from G.

In addition, for all $i \in [1, \sqrt{n}]$ and $u, u' \in V_i$ with $u \neq u'$, we maintain: (1) $w_{uu'}$, the *witness* for the pair u and u' and (2) a set $\text{Schedule}[u, u']$ of positive integers, which is the set of timesteps where our algorithm intends to *proactively resample* a new witness for u, u'. This set grows adaptively each time the adversary deletes $(u, w_{uu'})$ or $(w_{uu'}, u')$.

Finally, to ensure that the update time of our algorithm remains small, for each $\lambda \in [1, n\sqrt{n}]$ we maintain a $\text{List}[\lambda]$, which consists of all those pairs of nodes (x, x') such that $\lambda \in \text{Schedule}[x, x']$.

[4] Let $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$. If H_1 and H_2 are α-spanners G_1 and G_2 respectively, then $H_1 \cup H_2$ is a α-spanner of $G_1 \cup G_2$.

When an edge (u, v), where $u \in V_i$ and $v \in V_j$ is deleted, we do the following operations. First, for all $u' \in V_i$ that had $v = w_{uu'}$ as a common neighbor with u before deleting (u, v), we add the timesteps $\{T + 2^k \mid T + 2^k \leq n\sqrt{n}, k \in \mathbb{N}\}$ to $\textsc{Schedule}[u, u']$. Second, analogous to the previous one, for all $v' \in V_j$ that had $u = w_{vv'}$ as a common neighbor with v before deleting (u, v), we add the timesteps $\{T + 2^k \mid T + 2^k \leq n\sqrt{n}, k \in \mathbb{N}\}$ to $\textsc{Schedule}[v, v']$. Third, we set $T \leftarrow T + 1$. Lastly, for each $(x, x') \in \textsc{List}[T]$, we call the subroutine $\textsc{Resample}(x, x')$.

The key lemma below summarizes a crucial property of this dynamic algorithm. Its proof appears in Section 4.

▶ **Lemma 8.** *During a phase consisting of $n\sqrt{n}$ edge deletions, our dynamic algorithm makes at most $\tilde{O}(\sqrt{n})$ calls to the* Resample *subroutine after each edge deletion. Moreover, the* total *number of calls to the* Resample *subroutine during an* entire phase *is at most $\tilde{O}(n\sqrt{n})$ w.h.p. Both these guarantees hold against an adaptive adversary.*

Analysis of Recourse and Update Time. Our analysis are given in the lemmas below.

▶ **Lemma 9** (Recourse). *The amortized recourse of our algorithm is $\tilde{O}(1)$ w.h.p., against an adaptive adversary.*

Proof. To maintain the edge-sets E_1 and E_2, we pay a worst-case recourse of $O(1)$ per update. For maintaining the edge-set E_3, our total recourse during the entire phase is at most $O(1)$ times the number of calls made to the $\textsc{Resample}(., .)$ subroutine, which in turn is at most $\tilde{O}(n\sqrt{n})$ w.h.p. (see Lemma 8). Finally, while computing E_3 in the beginning of a phase, we pay $O(n\sqrt{n})$ recourse. Therefore, the overall total recourse during an entire phase is $\tilde{O}(n\sqrt{n})$ w.h.p.. Since a phase lasts for $n\sqrt{n}$ time steps, we conclude the lemma. ◀

▶ **Lemma 10** (Worst-case Update Time within a Phase). *Within a phase, our algorithm handles a given update in $\tilde{O}(\sqrt{n})$ worst case time w.h.p..*

Proof. Recall that the sets E_1, E_2 can be maintained in $O(1)$ worst case update time. Henceforth, we focus on the time required to maintain the edge-set E_3 after a given update in G.

Excluding the time spent on maintaining the partnership data structure (which is $\tilde{O}(\sqrt{n})$ in the worst-case anyway), this is proportional to $\tilde{O}(1)$ times the number of calls made to the $\textsc{Resample}(., .)$ subroutine, *plus* $\tilde{O}(1)$ times the number of pairs $u, u'(v, v')$ where we need to adjust SCHEDULE$[u, u']$. The former is w.h.p. at most $\tilde{O}(\sqrt{n})$ according to Lemma 8, while the latter is also at most $\tilde{O}(\sqrt{n})$ since $|V_i|, |V_j| \leq \sqrt{n}$. Thus, within a phase we can also maintain the set E_3 w.h.p. in $\tilde{O}(\sqrt{n})$ worst case update time. ◀

Although the above lemma says that we can handle each edge deletion in $\tilde{O}(\sqrt{n})$ worst-case update time, our current algorithm does not guarantee worst-case update time yet because the intialization time exceed the $\tilde{O}(\sqrt{n})$ bound. In more details, observe that the total initialization time is $O(n\sqrt{n}) \times O(\sqrt{n} \log n)$ because we need to insert $O(n\sqrt{n})$ edges into partnership data structures, which has $O(\sqrt{n} \log n)$ update time. Over a phase of $n\sqrt{n}$ steps, this implies only $\tilde{O}(\sqrt{n})$ amortized update time.

However, since the algorithm takes long time only at the initialization of the phase, but takes $\tilde{O}(\sqrt{n})$ worst-case step for each update during the phase, we can apply the standard building-in-the-background technique to de-amortized the update time.[5] We conclude the following:

[5] The proof for this standard technique will be included in our full version.

▶ **Lemma 11** (Worst-case Update Time for the Whole Update Sequence)**.** *W.h.p., the worst-case update time of our dynamic algorithm is* $\tilde{O}(\sqrt{n})$*.*

4 Proactive Resampling: Abstraction

The goal of this section is to prove Lemma 8 for bounding the recourse of our 3-spanner algorithm. This is the most technical part of this paper. To ease the analysis, we will abstract the problem situation in Lemma 8 as a particular dynamic problem of assigning jobs to machines while an adversary keeps deleting machines and the goal is to minimize the total number of reassignments. Below, we formalize this problem and show how to use it to bound the recourse of our 3-spanner algorithm.

Our abstraction has two technical contributions: (1) it allows us to easily work with multiple sampling probabilities, while in [12], they fixed a single parameter on sampling probability, (2) the simplicity of this abstraction can expose the generality of the proactive resampling technique itself; it is not specific to the cut sparsifier problem as used in [12].

Jobs, Machines, Routines, Assignments, and Loads. Let J denote a set of *jobs* and M denote a set of *machines*. We think of them as two sets of vertices of the (hyper)-graph $G = (J, M, R)$.[6] A *routine* $r \in R$ is a hyperedge of G such that r contains exactly one job-vertex from J, denoted by $\text{job}(r) \in J$, and may contain several machine-vertices from M, denoted by $M(r) \subseteq M$. Each routine r in G means there is a routine for handling $\text{job}(r)$ by *simultaneously* calling machines in $M(r)$. Note that $r = \{\text{job}(r)\} \cup M(r)$. We say that r is a *routine for* $\text{job}(r)$. For each machine $x \in M(r)$, we say that routine r *involves* machine x, or that r *contains* x. The set $R(x)$ is then defined as the set of routines involving machine x. Observe that there are $\deg_G(u)$ different routines for handling job u. An *assignment* $A = (J, M, F \subseteq R)$ is simply a subgraph of G. We say assignment A is *feasible* iff $\deg_A(u) = 1$ for every job $u \in J$ where $\deg_G(u) > 0$. That is, every job is handled by some routine, if exists. When $r \in A$, we say that $\text{job}(r)$ is *handled by* routine r. Finally, given an assignment A, the *load* of a machine x is the number of routines in A involving x, or in other words, is the degree of x in A, $\deg_A(x)$. We note explicitly that our end-goal is not to optimize loads of machines. Rather, we want to minimize the number of reassignments needed to maintain feasible assignments throughout the process.

In this section, we usually use u, v, w to denote jobs, use x, y, z to denote machines, and use e or r to denote routines or (hyper)edges.

The Dynamic Problem. Our problem is to maintain a feasible assignment A while the graph G undergoes a sequence of machine deletions (which might stop before all machines are deleted). More specifically, when a machine x is deleted, all routines r containing x are deleted as well. But when routines in A are deleted, A might not be feasible anymore and we need to add new edges to A to make A feasible. Our goal is to minimize the total number of routines ever added to A.

To be more precise, write the graph G and the assignment A after t machine-deletions as $G^t = (J, M, R^t)$ and $A^t = (J, M, F^t)$, respectively. Here, we define *recourse* at timestep t to be $|F^t \setminus F^{t-1}|$, which is the number of routined added into A at timestep t. When the adversary deletes T machines, the goal is then to minimize the total recourse $\sum_{t=1}^{T} |F^t \setminus F^{t-1}|$.

[6] This graph is different from the graph that we maintain a spanner in previous sections.

The Algorithm: Proactive Resampling. To describe our algorithm, first let $\textsc{Resample}(u)$ denote the process of reassigning job u to a uniformly random routine for u. In the graph language, $\textsc{Resample}(u)$ removes the edge r such that $\text{job}(r) = u$ from A, sample an edge r' from $\{r \in R \mid \text{job}(r) = u\}$, and then add r' into A. At timestep 0, we initialize a feasible assignment A^0 by calling $\textsc{Resample}(u)$ for every job $u \in J$, i.e., assign each job u to a random routine for u. Below, we describe how to handling deletions.

Let T be the total number of machine-deletions. For each job u, we maintain $\textsc{Schedule}(u) \subseteq [T]$ containing all time steps that we will invoke $\textsc{Resample}(u)$. That is, at any timestep t, before an adversary takes any action, we call $\textsc{Resample}(u)$ if $t \in \textsc{Schedule}(u)$.

We say that an adversary *touches* u at timestep t if the routine $r \in A^t$ handling u at time t is deleted. When u is touched, we call $\textsc{Resample}(u)$ and, very importantly, we put $t+1, t+2, t+4, \ldots$ where $t + 2^i \leq T$ into $\textsc{Schedule}(u)$. This is the action that we call *proactive resampling* because we do not just resample a routine for u only when u is touched, but do so proactively in the future as well. This completes the description of the algorithm.

Clearly, A remains a feasible assignment throughout because whenever a job u is touched, we immediately call $\textsc{Resample}(u)$. The key lemma below states that the algorithm has low recourse, even if the adversary is adaptive in the sense that each deletion at time t depends on previous assignment before time t.

▶ **Lemma 12.** *Let T be the total number of machine-deletions. The total recourse of the algorithm running against an adaptive adversary is $O\big(|J| \log(\Delta) \log^2 |M| + T \log^2 |M|\big)$ with high probability where Δ is the maximum degree of any job. Moreover, if the load of a machine never exceeds λ, then our algorithm has $O(\lambda \log T)$ worst-case recourse.*

We will prove Lemma 12 in Section 5. Before proceeding any further, however, we argue why Lemma 12 directly bounds the recourse of our 3-spanner algorithm.

Back to 3-spanners: Proof of Lemma 8. It is easy to see that maintaining E_3 in our 3-spanner algorithm can be framed exactly as the job-machine load-balancing problem. Suppose the given graph is $G = (V, E)$ where $n = |V|$ and $m = |E|$. We create a job $j_{uu'}$ for each pair of vertices $u, u' \in V_i$ with $u \neq u'$. For each edge $e \in E$, we create a machine x_e. Hence, $|J| = O(n^{1.5})$ and $|M| = |E| = m$. For each job, as we want to have a witness $w_{uu'}$, this witness is corresponding to two edges $e = (u, w_{uu'})$ and $e' = (u', w_{uu'})$. Hence, we create a routine $r = (j_{uu'}, e, e')$ for each $u, u' \in V_i$ and a common neighbor $w_{uu'}$. Since there are at most n common neighbors between each u and u', $\Delta = O(n)$. A feasible assignment is then corresponding to finding a witness for each job. Our algorithm that maintains the spanner is also exactly this load-balancing algorithm. Hence, the recourse of the 3-spanner construction follows from Lemma 12 where we delete exactly $T = O(n^{1.5})$ machines. As $|J| = O(n^{1.5})$, the total recourse bound then becomes $O(n^{1.5} \log^3 n)$. As $T = O(n^{1.5})$, averaging this bound over all timesteps yields $O(\log^3 n)$ amortized recourse.

5 Proactive Resampling: Analysis (Proof of Lemma 12)

The first step to prove Lemma 12 is to bound the loads of machines x. This is because whenever machine x is deleted, its load of $\deg_A(x)$ would contribute to the total recourse.

What would be the expected load of each machine? For intuition, suppose that the adversary was *oblivious*. Recall that $R(x)$ denote the set of all routines involving machine x. Then, the expected load of machine x would be $\sum_{r \in R(x)} 1/\deg_G(\text{job}(r))$ because each job samples its routine uniformly at random, and this is concentrated with high probability using Chernoff's bound. Although in reality our adversary is *adaptive*, our plan is to still prove that the loads of machines do not exceed its expectation in the oblivious setting too much. This motivates the following definitions.

▶ **Definition 13.** *The* target load *of machine x is* $\operatorname{target}(x) = \sum_{r \in R(x)} 1/\deg_G(\operatorname{job}(r))$. *The target load of x at time t is* $\operatorname{target}^t(x) = \sum_{r \in R^t(x)} 1/\deg_{G^t}(\operatorname{job}(r))$. *An assignment A has* overhead (α, β) *iff* $\deg_A(x) \le \alpha \cdot \operatorname{target}(x) + \beta$ *for every machine $x \in M$.*

Our key technical lemma is to show that, via proactive resampling, the loads of machines are indeed close to its expectation in the oblivious setting. That is, the maintained assignment has small overhead. Recall that T is the total number of machine-deletions.

▶ **Lemma 14.** *With high probability, the assignment A maintained by our algorithm always has overhead $(O(\log T), O(\log|M|))$ even when the adversary is adaptive.*

We defer the proof of Lemma 14 to Section 6. Assuming Lemma 14, we formally show how to bound of machine loads implies the total recourse, which proves Lemma 12.

Proof of Lemma 12. Let T be the total number of deletions. Observe that the total recourse up to time T is precisely the total number of Resample(.) calls up to time T, which in turn is at most the total number of Resample(.) calls put into Schedule(.) since time 1 until time T. Therefore, our strategy is to bound, for each time t, the number of Resample(.) calls *newly generated* at time t. Let x^t be the machine deleted at time t. Observe this is at most $O(\log T) \times \deg_{A^t}(x^t)$ where $\deg_{A^t}(x^t)$ is x^t's load at time t and the $O(\log T)$ factor is because of proactive sampling.

By Lemma 14, we have $\deg_{A^t}(x^t) \le O\left(\log(T)\operatorname{target}^t(x^t) + \log|M|\right)$. Also, we claim that $\sum_{t=1}^{T} \operatorname{target}^t(x^t) = O(|J| \log \Delta)$ where Δ is the maximum degree of jobs (to be proven below). Therefore, the total recourse up to time T is at most

$$\begin{aligned} O(\log T) \sum_{t=1}^{T} \deg_{A^t}(x^t) &\le O(\log T) \sum_{t=1}^{T} O\left(\log(T)\operatorname{target}^t(x^t) + \log|M|\right) \\ &\le O\left(|J| \log(\Delta) \log^2|M| + T \log^2|M|\right) \end{aligned}$$

as $T \le |M|$.

It remains to show that $\sum_{t=1}^{T} \operatorname{target}^t(x^t) = O(|J| \log \Delta)$. Recall that $\operatorname{target}^t(x) = \sum_{r \ni x} \frac{1}{\deg_{G^t}(\operatorname{job}(r))}$. Imagine when machine x^t is deleted at time t. We will show how to charge $\operatorname{target}^t(x^t)$ to jobs with edges connecting to x^t. For each job u with c (hyper)edges connecting to x^t, u's contribution of $\operatorname{target}^t(x^t)$ is $c/\deg_{G^t}(u)$. So we distribute the charge of $c/\deg_{G^t}(u) \le \frac{1}{\deg_{G^t}(u)} + \frac{1}{\deg_{G^t}(u)-1} + \ldots + \frac{1}{\deg_{G^t}(u)-c+1}$ to u. Since these edges are charged from machine x^t to job u only once, the total charge of each job u at most $\frac{1}{\deg_G(u)} + \frac{1}{\deg_G(u)-1} + \cdots + 1/2 + 1 = O(\log \Delta)$. Since there are $|J|$ jobs, the bound $\sum_{t=1}^{T} \operatorname{target}^t(x^t) = O(|J| \log \Delta)$ follows.

To see that we have worst-case recourse, one can look at any timestep t. There are $O(\log t)$ timesteps that can cause Resample to be invoked at timestep t, namely, $t-1, t-2, t-4, \ldots$. At each of these timesteps t', one machine is deleted, so the number of Resample calls added from timestep t' is also bounded by the load of the deleted machine $x_{t'}$, which does not exceed λ. Summing this up, the number of calls we make at timestep t is at most $O(\lambda \log t) = O(\lambda \log T)$. This concludes our proof. ◀

6 Proof of Lemma 14

Here, we show that the load $\deg_{A^t}(x)$ of every machine x at each time t is small. Some basic notions are needed in this analysis.

Experiments and Relevant Experiments. An *experiment* X is a binary random variable associated with an edge/routine e and time step t, where $X = 1$ iff $\textsc{Resample}(\text{job}(e))$ is called at time t and e is chosen to handle job(e), among all edges incident to job(e). Observe that $\mathbb{P}[X = 1] = 1/\deg_{G^t}(\text{job}(e))$. Note that each call to $\textsc{Resample}(u)$ at time t creates new $\deg_{G^t}(u)$ experiments. We order all experiments $X_1, X_2, X_3, \ldots$ by the time of their creation. For convenience, for each experiment X, we let $e(X)$, $t(X)$, and job(X) denote its edge, time of creation, and job respectively.

Next, we define the most important notion in the whole analysis.

▶ **Definition 15.** *For any time t and edge $e \in R^t$ at time t, an experiment X is (t, e)-relevant if*
- *$e(X) = e$, and*
- *there is no $t(X) < t' < t$ such that $t' \in \textsc{Schedule}^{t(X)}(\text{job}(e))$.*

Moreover, X is (t, x)-relevant if it is (t, e)-relevant and edge $e \in R^t(x)$ is incident to x.

Intuitively, X is a (t, e)-relevant experiment if X could cause e to appear in the assignment A^t at time t. To see why, clearly if $e(X) \neq e$, then X cannot cause e to appear. Otherwise, if $e(X) = e$ but there is $t' \in (t(X), t)$ where $t' \in \textsc{Schedule}^{t(X)}(\text{job}(e))$, then X cannot cause e to appear at time t either. This is because even if X is successful and so e appears at time $t(X)$, then later at time $t' > t(X)$, e will be resampled again, and so X has nothing to do whether e appears at time $t > t'$. With the same intuition, X is (t, x)-relevant if X could contribute to the load $\deg_{A^t}(x)$ of machine x at time t.

Notice that we decide whether X is a (t, e)-relevant based on $\textsc{Schedule}^{t(X)}(\text{job}(e))$ at time $t(X)$. If it was based on $\textsc{Schedule}^{t}(\text{job}(e))$ at time t, then there would be only a single experiment X that is (t, e)-relevant (which is the one with $e(X) = e$ and maximum $t(X) < t$).

According to Definition 15, there could be more than one experiments that are (t, e)-relevant. For example, suppose X is (t, e)-relevant. At time $t(X) + 1$, the adversary could touch job(e), hence, adding $t(X) + 2, t(X) + 4, \ldots$ into $\textsc{Schedule}(\text{job}(e))$. Because of this action, there is another experiment X' that is (t, e)-relevant and $t(X') > t(X)$. This motivates the following definition.

▶ **Definition 16.** *Let $Rel(t, e)$ be the random variable denoting the number of (t, e)-relevant experiments, and let $Rel(t, x) = \sum_{e \in R^t(x)} Rel(t, e)$ denote the total number of (t, x)-relevant experiments.*

To simplify the notations in the proof of Lemma 14 below, we assume the following.

▶ **Assumption 17** (The Machine-disjoint Assumption). *For any routines e, e' with* job$(e) =$ job(e'), *then $M(e) \cap M(e') = \emptyset$. That is, the edges adjacent to the same job are machine-disjoint.*

Note that this assumption indeed holds for our 3-spanner application. This is because any two paths of length 2 between a pair of centers u and u' must be edge disjoint in any simple graph. We show how remove this assumption in our full version. The notations there will slightly be more complicated.

Roadmap for Bounding Loads. We are now ready to describe the key steps for bounding the load $\deg_{A^t}(x)$, for any time t and machine x.

First, we write $\mathcal{X}^{(t,x)} = X_1^{(t,x)}, X_2^{(t,x)}, \ldots, X_{Rel(t,x)}^{(t,x)}$ as the sequence of all (t,x)-relevant experiments (ordered by time step the experiments are taken). The order in $\mathcal{X}^{(t,x)}$ will be important only later. For now, we write

$$\overline{\deg}_{A^t}(x) = \sum_{X \in \mathcal{X}^{(t,x)}} X,$$

as the total number of success (t,x)-relevant experiments. As any edge e adjacent to x in A^t may appear only because of some successful (t,x)-relevant experiment $X \in \mathcal{X}^{(t,x)}$, we conclude the following:

▶ **Lemma 18** (Key Step 1). $\deg_{A^t}(x) \le \overline{\deg}_{A^t}(x)$.

Lemma 18 reduces the problem to bounding $\overline{\deg}_{A^t}(x)$. If all (t,x)-relevant experiments $\mathcal{X}^{(t,x)} = \{X_i^{(t,x)}\}_i$ were independent, then we could have easily applied standard concentration bounds to $\overline{\deg}_{A^t}(x) = \sum_{X \in \mathcal{X}^{(t,x)}} X$. Unfortunately, they are not independent as the outcome of earlier experiments can affect the adversary's actions, which in turn affect later experiments.

Our strategy is to relate the sequence $\mathcal{X}^{(t,x)}$ of (t,x)-relevant experiments to another sequence $\hat{\mathcal{X}}^{(t,x)} = \hat{X}_1^{(t,x)}, \hat{X}_2^{(t,x)} \ldots, \hat{X}_{Rel(t,x)}^{(t,x)}$ of *independent* random variables defined as follows. For each (t,x)-relevant experiment $\hat{X}_i^{(t,x)}$ where $e = e(\hat{X}_i^{(t,x)})$ and $u = \text{job}(e)$, we carefully define $\hat{X}_i^{(t,x)}$ as an *independent* binary random variable such that

$$\mathbb{P}[\hat{X}_i^{(t,x)} = 1] = 1/\deg_{G^t}(u),$$

which is the probability that RESAMPLE(u) chooses e at time t. We similarly define

$$\widehat{\deg}_{A^t}(x) = \sum_{\hat{X} \in \hat{\mathcal{X}}^{(t,x)}} \hat{X},$$

that sums independent random variables, where each term in the sum is closely related to the corresponding (t,x)-relevant experiments. By our careful choice of $\mathbb{P}[\hat{X}_i^{(t,x)} = 1]$, we can relates $\widehat{\deg}_{A^t}(x)$ to $\overline{\deg}_{A^t}(x)$ via the notion of *stochastic dominance* defined below.

▶ **Definition 19.** *Let Y and Z be two random variables not necessarily defined on the same probability space. We say that Z* stochastically dominates *Y, written as $Y \preceq Z$, if for all $\lambda \in \mathbb{R}$, we have $\mathbb{P}[Y \ge \lambda] \le \mathbb{P}[Z \ge \lambda]$.*

Our second important step is the following lemma. The proof of Lemma 20 is omitted and will appear in our full version.

▶ **Lemma 20** (Key Step 2). $\overline{\deg}_{A^t}(x) \preceq \widehat{\deg}_{A^t}(x)$.

Lemma 20 reduces the problem to bounding $\widehat{\deg}_{A^t}(x)$, which is indeed relatively easy to bound because it is a sum of independent random variables. The last key step of our proof does exactly this:

▶ **Lemma 21** (Key Step 3). $\widehat{\deg}_{A^t}(x) \le 2\log(t) \cdot \text{target}^t(x) + O(\log|M|)$ *with probability* $1 - 1/|M|^{10}$.

We show the proof Lemma 21 in our full version. Here, we only mention one important point about the proof. The $\log(t)$ factor above follows from the factor the number of (t,e)-relevant experiment is always at most $Rel(t,e) \le \log(t)$ for any time t and edge e. This property is so crucial and, actually, is what the proactive resampling technique is designed for.

Given three key steps above (Lemmas 18, 20, and 21), we can conclude the proof of Lemma 14.

Proof of Lemma 14. Recall that we ultimately want to show that, for every timestep t, the maintained assignment A^t has overhead $O(\log(T), \log|M|)$. In other words, for every $t \in T$ and every $x \in M$, we want to show that

$$\deg_{A^t}(x) \leq \text{target}^t(x) \cdot O(\log(T)) + O(\log|M|).$$

By Lemma 18, it suffices to show that

$$\overline{\deg}_{A^t}(x) \leq \text{target}^t(x) \cdot O(\log(T)) + O(\log|M|).$$

By Lemmas 20 and 21,

$$\begin{aligned} &\mathbb{P}[\overline{\deg}_{A^t}(x) \geq 2\log(t) \cdot \text{target}^t(x) + O(\log|M|)] \\ &\leq \mathbb{P}[\widehat{\deg}_{A^t}(x) \geq 2\log(t) \cdot \text{target}^t(x) + O(\log|M|)] && \text{(Lemma 20)} \\ &\leq 1/|M|^{10}. && \text{(Lemma 21)} \end{aligned}$$

Now we apply union bound to the probability above. There are $T \leq |M|$ timesteps and $|M|$ machines, hence the probability that a bad event happens is bounded by $\frac{T|M|}{|M|^{10}} = \frac{1}{|M|^8}$. Here, we conclude the proof of Lemma 14. ◀

7 Conclusion

In this paper, we study fully dynamic spanner algorithms against an adaptive adversary. Our algorithm in Theorem 1 maintains a spanner with near-optimal stretch-size trade-off using only $O(\log n)$ amortized recourse. This closes the current oblivious-vs-adaptive gap with respect to amortized recourse. Whether the gap can be closed for *worst-case recourse* is an interesting open problem.

The ultimate goal is to show algorithms against an adaptive adversary with polylogarithmic amortized update time or even worst-case. Via the multiplicative weight update framework [39, 41], such algorithms would imply $O(k)$-approximate multi-commodity flow algorithm with $\tilde{O}(n^{2+1/k})$ time which would in turn improve the state-of-the-art. We made partial progress toward this goal by showing the first dynamic 3-spanner algorithms against an adaptive adversary with $\tilde{O}(\sqrt{n})$ update time in Theorem 3 and *simultaneously* with $\tilde{O}(1)$ amortized recourse in Theorem 2, improving upon the $O(n)$ amortized update time since the 15-year-old work by [5].

Generalizing our Theorem 3 to dynamic $(2k-1)$-spanners of size $\tilde{O}(n^{1+1/k})$, for any $k \geq 2$, is also a very interesting and challenging open question.

References

1 Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Apmf< apsp? gomory-hu tree for unweighted graphs in almost-quadratic time. *arXiv preprint*, 2021. `arXiv:2106.02981`.

2 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`.

3 Noga Alon, Omri Ben-Eliezer, Yuval Dagan, Shay Moran, Moni Naor, and Eylon Yogev. Adversarial laws of large numbers and optimal regret in online classification. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 447–455, 2021.

4 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.

5 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. `doi:10.7155/jgaa.00133`.

6 Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. *SIAM Journal on Discrete Mathematics*, 34(3):1791–1812, 2020.

7 Raef Bassily, Kobbi Nissim, Adam D. Smith, Thomas Steinke, Uri Stemmer, and Jonathan R. Ullman. Algorithmic stability for adaptive data analysis. *SIAM J. Comput.*, 50(3), 2021. `doi:10.1137/16M1103646`.

8 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`.

9 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 382–405, 2019. `doi:10.1109/FOCS.2019.00032`.

10 Omri Ben-Eliezer, Rajesh Jayaram, David P Woodruff, and Eylon Yogev. A framework for adversarially robust streaming algorithms. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pages 63–80, 2020.

11 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`.

12 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. *arXiv preprint*, 2020. `arXiv:2004.08432`.

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

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

15 Aaron Bernstein and Shiri Chechik. Incremental topological sort and cycle detection in expected total time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 21–34, 2018. `doi:10.1137/1.9781611975031.2`.

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

17 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`.

18 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.

19 Sayan Bhattacharya, Fabrizio Grandoni, and David Wajc. Online edge coloring algorithms via the nibble method. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2830–2842. SIAM, 2021.

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

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

22 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.

23 Sayan Bhattacharya and Peter Kiss. Deterministic rounding of dynamic fractional matchings. *arXiv preprint*, 2021. `arXiv:2105.01615`.

24 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.

25 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`.

26 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 *STOC*, pages 859–869. ACM, 2021.

27 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.

28 Vladimir Braverman, Avinatan Hassidim, Yossi Matias, Mariano Schain, Sandeep Silwal, and Samson Zhou. Adversarial robustness of streaming algorithms through importance sampling. *arXiv preprint*, 2021. `arXiv:2106.14952`.

29 Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 217–226, 2016. `doi:10.1145/2933057.2933083`.

30 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 370–381, 2019. `doi:10.1109/FOCS.2019.00031`.

31 Shiri Chechik and Tianyi Zhang. Dynamic low-stretch spanning trees in subpolynomial time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 463–475. SIAM, 2020.

32 Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146. IEEE, 2020.

33 Julia Chuzhoy. Decremental all-pairs shortest paths in deterministic near-linear time. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 626–639, 2021. `doi:10.1145/3406325.3451025`.

34 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.

35 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.

36 Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2478–2496. SIAM, 2021. `doi:10.1137/1.9781611976465.147`.

37 Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Leon Roth. Preserving statistical validity in adaptive data analysis. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 117–126, 2015.

38 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`.

39 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`.

40 Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *STOC*, pages 377–388. ACM, 2019.

41 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`.

42 Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2212–2228. SIAM, 2021. doi:10.1137/1.9781611976465.132.

43 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.

44 Anupam Gupta and Amit Kumar. Online steiner tree with deletions. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 455–467. SIAM, 2014.

45 Anupam Gupta, Amit Kumar, and Cliff Stein. Maintaining assignments online: Matching, scheduling, and flows. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 468–479. SIAM, 2014.

46 Anupam Gupta and Roie Levin. Fully-dynamic submodular cover with bounded recourse. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1147–1157. IEEE, 2020.

47 Varun Gupta, Christopher Jung, Seth Neel, Aaron Roth, Saeed Sharifi-Malvajerdi, and Chris Waites. Adaptive machine unlearning. *arXiv preprint*, 2021. arXiv:2106.04378.

48 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.

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

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

51 Moritz Hardt and Jonathan Ullman. Preventing false discovery in interactive data analysis is hard. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 454–463. IEEE, 2014.

52 Avinatan Hasidim, Haim Kaplan, Yishay Mansour, Yossi Matias, and Uri Stemmer. Adversarially robust streaming algorithms via differential privacy. *Advances in Neural Information Processing Systems*, 33, 2020.

53 Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

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

55 Jacob Holm and Eva Rotenberg. Worst-case polylog incremental spqr-trees: Embeddings, planarity, and triconnectivity. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2378–2397, 2020. doi:10.1137/1.9781611975994.146.

56 Haim Kaplan, Yishay Mansour, Kobbi Nissim, and Uri Stemmer. Separating adaptive streaming from oblivious streaming. *arXiv preprint*, 2021. arXiv:2101.10836.

57 Jason Li. Deterministic mincut in almost-linear time. In *STOC*, pages 384–395. ACM, 2021.

58 Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. A nearly optimal all-pairs min-cuts algorithm in simple graphs. *arXiv preprint*, 2021. arXiv:2106.02233.

59 Jason Li and Thatchaphol Saranurak. Deterministic weighted expander decomposition in almost-linear time. *arXiv preprint*, 2021. arXiv:2106.01567.

60 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.

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

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

63 Thomas Steinke and Jonathan Ullman. Tight lower bounds for differentially private selection. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 552–563. IEEE, 2017.

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

65 Rephael Wenger. Extremal graphs with no c4's, c6's, or c10's. *Journal of Combinatorial Theory, Series B*, 52(1):113–116, 1991.

66 David P Woodruff and Samson Zhou. Tight bounds for adversarially robust streams and sliding windows via difference estimators. *arXiv preprint*, 2020. `arXiv:2011.07471`.

67 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`.

68 Tianyi Zhang. Faster cut-equivalent trees in simple graphs. *arXiv preprint*, 2021. `arXiv:2106.03305`.