Fully Dynamic Spanners with Worst-Case Update Time*

Greg Bodwin† Sebastian Krinninger‡

Abstract

An α-*spanner* of a graph G is a subgraph H such that H preserves all distances of G within a factor of α. In this paper, we give fully dynamic algorithms for maintaining a spanner H of a graph G undergoing edge insertions and deletions with worst-case guarantees on the running time after each update. In particular, our algorithms maintain:

- a 3-spanner with $\tilde{O}(n^{1+1/2})$ edges with worst-case update time $\tilde{O}(n^{3/4})$, or
- a 5-spanner with $\tilde{O}(n^{1+1/3})$ edges with worst-case update time $\tilde{O}(n^{5/9})$.

These size/stretch tradeoffs are best possible (up to logarithmic factors). They can be extended to the weighted setting at very minor cost. Our algorithms are randomized and correct with high probability against an oblivious adversary. We also further extend our techniques to construct a 5-spanner with suboptimal size/stretch tradeoff, but improved worst-case update time.

To the best of our knowledge, these are the *first* dynamic spanner algorithms with sublinear worst-case update time guarantees. Since it is known how to maintain a spanner using small *amortized* but large *worst-case* update time [Baswana et al. SODA'08], obtaining algorithms with strong worst-case bounds, as presented in this paper, seems to be the next natural step for this problem.

*To be presented at the European Symposium on Algorithms (ESA) 2016. This work was partially done while the authors were visiting the Simons Institute for the Theory of Computing.

†Stanford University

‡Max Planck Institute for Informatics

Contents

1 Introduction

An *α-spanner* of a graph G is a sparse subgraph that preserves all original distances within a multiplicative factor of α. Spanners are an extremely important and well-studied primitive in graph algorithms. They were formally introduced by Peleg and Schäfer [PS89] in the late eighties after appearing naturally in several network problems [PU89]. Today, they have been successfully applied in diverse fields such as routing schemes [Cow01, CW04, PU89, RTZ08, TZ01], approximate shortest paths algorithms [DHZ00, Elk05, BK10], distance oracles [BK10, Che14, Che15, PR14, TZ05], broadcasting [FPZ+04], etc. A landmark upper bound result due to Awerbuch [Awe85] states that for any integer k, every graph has a $(2k-1)$-spanner on $O(n^{1+1/k})$ edges. Moreover, the extremely popular *girth conjecture* of Erdős [Erd64] implies the existence of graphs for which $\Omega(n^{1+1/k})$ edges are necessary in any $(2k-1)$-spanner. Thus, the primary question of the optimal sparsity of a graph spanner is essentially resolved.

The next natural question in the field of spanners is to obtain efficient algorithms for computing a sparse spanner of an input graph G. This problem is well understood in the static setting; notable results include [Awe85, BS07, RTZ05, TZ01]. However, in many of the above applications of spanners, the underlying graph can experience minor changes and the application requires the algorithm designer to have a spanner available at all times. Here, it is very wasteful to recompute a spanner from scratch after every modification. The challenge is instead to *dynamically maintain* a spanner under edge insertions and deletions with only a small amount of time required per update. This is precisely the problem we address in this paper.

The pioneering work on dynamic spanners was by Ausiello et al. [AFI06], who showed how to maintain a 3- or 5-spanner with amortized update time proportional to the maximum degree Δ of the graph, i.e. for any sequence of u updates the algorithm takes time $O(u \cdot \Delta)$ in total. In sufficiently dense graphs, Δ might be $\Omega(n)$. Elkin [Elk11] showed how to maintain a $(2k-1)$ spanner of optimal size using $\tilde{O}(mn^{-1/k})$ expected update time; i.e. *super-linear* time for dense enough graphs. Finally, Baswana et al. [BKS12] gave fully dynamic algorithms that maintain $(2k-1)$-spanners with essentially optimal size/stretch tradeoff using *amortized* $O(k^2 \log^2 n)$ or $O(1)^k$ time per update. Their *worst-case* guarantees are much weaker: any individual update in their algorithm can require $\Omega(n)$ time. It is very notable that *every* previously known fully dynamic spanner algorithm carries the drawback of $\Omega(n)$ worst-case update time. It is thus an important open question whether this update time is an intrinsic part of the dynamic spanner problem, or whether this linear time threshold can be broken with new algorithmic ideas.

There are concrete reasons to prefer worst-case update time bounds to their amortized counterparts. In real-time systems, hard guarantees on update times are often needed to serve each request before the next one arrives. Amortized guarantees, meanwhile, can cause undesirable behavior in which the system periodically stalls on certain inputs. Despite this motivation, good worst-case update times often pose a veritable challenge to dynamic algorithm designers, and are thus significantly rarer in the literature. Historically, the fastest dynamic algorithms usually first come with amortized time bounds, and comparable worst-case bounds are achieved only after considerable research effort. For example, this was the case for the dynamic connectivity problem on undirected graphs [KKM13] and the dynamic transitive closure problem on directed graphs [San04]. In other problems, a substantial gap between amortized and worst-case algorithms remains, despite decades of research. This

holds in the cases of fully dynamically maintaining minimum spanning trees [HLT01, Fre85, EGI+97], all-pairs shortest paths [DI04, Tho05], and more. Thus, strong amortized update time bounds for a problem do not at all imply the existence of strong worst-case update time bounds, and once strong amortized algorithms are found it becomes an important open problem to discover whether or not there are interesting worst-case bounds to follow.

The main result of this paper is that highly nontrivial worst-case time bounds are indeed available for fully dynamic spanners. We present the first ever algorithms that maintain spanners with essentially optimal size/stretch tradeoff and *polynomially sublinear* (in the number of nodes in the graph) worst-case update time. Our main technique is a very general new framework for boosting the performance of an orientation-based algorithm, which we hope can have applications in related dynamic problems.

1.1 Our results

We obtain fully dynamic algorithms for maintaining spanners of graphs undergoing edge insertions and deletions. In particular, in the unweighted setting we can maintain:

- a 3-spanner of size $O(n^{1+1/2}\log^{1/2} n \log\log n)$ with worst-case update time $O(n^{3/4}\log^4 n)$, or
- a 5-spanner of size $O(n^{1+1/3}\log^{2/3} n \log\log n)$ with worst-case update time $O(n^{5/9}\log^4 n)$, or
- a 5-spanner of size $O(n^{1+1/2}\log^{1/2} n \log\log n)$ with worst-case update time $O(n^{1/2}\log^4 n)$.

Naturally, these results assume that the initial graph is empty; otherwise, a lengthy initialization step is unavoidable.

Using standard techniques, these results can be extended into the setting of arbitrary positive edge weights, at the cost of an increase in the stretch by a factor of $1+\epsilon$ and an increase in the size by a factor of $\log_{1+\epsilon} W$ (for any $\epsilon > 0$, where W is the ratio between the largest and smallest edge weights).

Our algorithms are randomized and correct with high probability against an *oblivious adversary* [BDBK+94] who chooses its sequence of updates independently from the random choices made by the algorithm.[1] This adversarial model is the same one used in the previous randomized algorithms with amortized update time [BKS12]. Since the girth conjecture has been proven unconditionally for $k = 2$ and $k = 3$ [Wen91], the first two spanners have optimal size/stretch tradeoff (up to the log factor). The third result sacrifices a non-optimal size/stretch tradeoff in exchange for improved update time.

1.2 Technical Contributions

Our main new idea is a general technique for boosting the performance of orientation-based algorithms.

Our algorithm contains three new high-level ideas. First, let $\vec{G}$ be an arbitrary orientation of the input graph G; i.e. replace every undirected edge $\{u, v\}$ by a directed edge, either (u, v) or (v, u). We give an algorithm ALG for maintaining either a 3-spanner or a 5-spanner

[1] In particular, this means that the adversary is *not* allowed to see the current edges of the spanner.

of G with update time proportional to the maximum *out-degree* of the oriented graph $\vec{G}$. This algorithm is based on the clustering approach used in [BS07]. For maintaining 3- and 5-spanners we only need to consider clusters of diameter at most 2 consisting of the set of neighbors of certain cluster centers.

This alone is of course not enough, as generally the maximum out-degree of $\vec{G}$ can be as large as $n-1$. To solve this problem, we combine ALG with the following simple out-degree reduction technique. Partition outgoing edges of every node into at most $t \leq \lceil n/s \rceil$ groups of size at most s each. For any $1 \leq i \leq t$, we combine the edges of the i-th groups and on the corresponding subgraph G_i we run an instance of ALG to maintain a 3-spanner with update time $O(s)$, the maximum out-degree in $\vec{G}_i$. By the decomposability of spanners, the union of all these sub-spanners $H_1 \cup \ldots H_t$ is a 3-spanner of G. In this way we can obtain an algorithm for maintaining a 3-spanner of size $|H_1| + \ldots |H_t| = O(n^{5/2}/s)$ with worst-case update time $O(s)$ for any $1 \leq s \leq n$. We remark that the general technique of partitioning a graph into subgraphs of low out-degree has been used before, e.g. [BE13]; however, our recursive conversion of these subgraphs into spanners is original and an important technical contribution of this paper.

The partitioning is still not enough, as the optimal size of a 3-spanner is $O(n^{3/2})$, which would then require $s = \Omega(n)$ worst-case update time. However, we can improve upon this tradeoff once more with a more fine-grained application of ALG. In particular, on each subgraph $\vec{G}_i$, ALG maintains two subgraphs A_i^1 and $\vec{B}_i^1$, such that:

- A_i^1 is a 'partial' 3-spanner of G_i of size $\tilde{O}(n^{1+1/2} \cdot s/n)$, and
- The maximum out-degree in $\vec{B}_i^1$ is considerably smaller than the maximum out-degree in $\vec{G}_i$.

We then recursively apply ALG on $\vec{B}_1^1 \cup \cdots \cup \vec{B}_t^1$ to some depth ℓ at which the out-degree can no longer be reduced by a meaningful amount. Our final spanner is then the union of all the sets A_i^j, for $1 \leq i \leq t$ and $1 \leq j \leq \ell$, as well as the "remainder" graphs $\vec{B}_1^\ell \cup \cdots \cup \vec{B}_t^\ell$, which have low out-degree and are thus sparse.

In principle, the recursive application of ALG could be problematic, as one update in G could lead to several changes to the edges in the B_i^1 subgraphs, which then propagate as an increasing number of updates in the recursive calls of the algorithm. This places another constraint on ALG. We carefully design ALG in such a way that it performs only a constant number of changes to each B_i^1 with any update in G, and we only recurse to depth $\ell = o(\log n)$ so that the total number of changes at each level is subpolynomial.

Overall, we remark that our framework for performing out-degree reduction is fairly generic, and seems likely applicable to other algorithms that admit the design of an ALG with suitable properties. The main technical challenges are designing ALG with these properties, and performing some fairly involved parameter balancing to optimize the running time used by the recursive calls. However, we do not know how to extend our approach to sparser spanners with larger stretches since corresponding constructions usually need clusters of larger diameter and maintaining such clusters with update time proportional to the maximum (out)-degree of the graph seems challenging.

1.3 Other Related Work

There has been some related work attacking the spanner problem in other models of computation. Some of the work on streaming spanner algorithms, in particular [Bas08, FKM+08], was converted into purely *incremental* dynamic algorithms, which maintain spanners under edge insertions but cannot handle deletions. This line of research culminated in an incremental algorithm with worst-case update time $O(1)$ per edge insertion [Elk11]. Elkin [Elk07] also gave a near-optimal algorithm for maintaining spanners in the distributed setting.

A concept closely related to spanners are *emulators* [DHZ00], in which the graph H for approximately preserving the distances may contain arbitrary weighted edges and is not necessarily a subgraph of G. Dynamic algorithms for maintaining emulators have been commonly used as subroutines to obtain faster dynamic algorithms for maintaining (approximate) shortest paths or distances. Some of the work on this problem includes [RZ12, BR11, HKN14b, HKN14a, ACT14, AC13].

As outlined above, one of the main technical contributions of this paper is a framework for exploiting orientations of undirected graphs. The idea of orienting undirected graphs has been key to many recent advances in dynamic graph algorithms. Examples include [NS13, KKP+14, PS16, AKL+15, ADK+15].

2 Preliminaries

We consider unweighted, undirected graphs $G = (V, E)$ undergoing edge insertions and edge deletions. For all pairs of nodes u and v we denote by $d_G(u, v)$ the distance between u and v in G. An *α-spanner* of a graph $G = (V, E)$ is a subgraph $H = (V, E') \subseteq G$ such that $d_H(u, v) \leq \alpha \cdot d_G(u, v)$ for all $u, v \in V$.[2] The parameter α is called the *stretch* of the spanner. We will use the well-known fact that it suffices to only span distances over the edges of G.

Lemma 2.1 (Spanner Adjacency Lemma (Folklore))**.** *If $H = (V, E')$ is a subgraph of $G = (V, E)$ that satisfies $d_H(u, v) \leq \alpha \cdot d_G(u, v)$ for all $(u, v) \in E$, then H is an α-spanner of G.*

We will work with *orientations* of undirected graphs. We denote an undirected edge with endpoints u and v by $\{u, v\}$ and a directed edge from u to v by (u, v). An *orientation* $\vec{G} = (V, \vec{E})$ of an undirected graph $G = (V, E)$ is a directed graph on the same set of nodes such that for every edge $\{u, v\}$ of G, $\vec{G}$ either contains the edge (u, v) or the edge (v, u). Conversely, G is the *undirected projection* of $\vec{G}$. In an undirected graph G, we denote by $N(v) := \{w \mid \{v, w\} \in G\}$ the set of neighbors of v. In an oriented graph $\vec{G}$, we denote by $Out(v) := \{w \mid (v, w) \in \vec{G}\}$ the set of outgoing neighbors of v. Similarly, by $In(v) := \{u \mid (u, v) \in \vec{G}\}$ we denote the set of incoming neighbors of v. We denote by $\Delta^+(\vec{G})$ the maximum out-degree of $\vec{G}$.

Our algorithms can easily be extended to graphs with edge weights, via the standard technique of weight binning:

[2] If u and v are disconnected in G, then $d_G(u, v) = \infty$ and so they may be disconnected in the spanner as well.

Lemma 2.2 (Weight Binning, e.g. [BKS12])**.** *Suppose there is an algorithm that dynamically maintains a spanner of an arbitrary unweighted graph with some particular size, stretch, and update time. Then for any $\epsilon > 0$, there is an algorithm that dynamically maintains a spanner of an arbitrary graph with positive edge weights, at the cost of an increase in the stretch by a factor of $1+\epsilon$ and an increase in the update time by a factor of $O(\log_{1+\epsilon} W)$ (and no change in update time). Here, W is the ratio between the largest and smallest edge weight in the graph.*

Since this extension is already well known, we will not discuss it further. Instead, we will simplify the rest of the paper by focusing only on the unweighted setting; that is, all further graphs in this paper are unweighted and undirected.

In our algorithms, we will use the well-known fact that good hitting sets can be obtained by random sampling. This technique was first used in the context of shortest paths by Ullman and Yannakakis [UY91]. A general lemma on the size of the hitting set can be formulated as follows.

Lemma 2.3 (Hitting Sets)**.** *Let $a \geq 1$, let V be a set of size n and let $U_1, U_2, \ldots, U_r$, be subsets of V of size at least q. Let S be a subset of V obtained by choosing each element of V independently at random with probability $p = \min(x/q, 1)$ where $x = a \ln(rn) + 1$. Then, with high probability (whp), i.e. probability at least $1 - 1/n^a$, both the following two properties hold:*

1. *For every $1 \leq i \leq r$, the set S contains a node in U_i, i.e. $U_i \cap S \neq \emptyset$.*
2. *$|S| \leq 3xn/q = O(an \ln(rn)/q)$.*

A well-known property of spanners is *decomposability*. We will exploit this property to run our dynamic algorithm on carefully chosen subgraphs.

Lemma 2.4 (Spanner Decomposability, [BKS12])**.** *Let $G = (V, E)$ be an undirected (possibly weighted) graph, let $E_1, \ldots, E_t$ be a partition of the set of edges E, and let, for every $1 \leq i \leq t$, H_i be an α-spanner of $G_i = (V, E_i)$ for some $\alpha \geq 1$. Then $H = \bigcup_{i=1}^{t} H_i$ is an α-spanner of G.*

In our algorithms we use a reduction for getting a fully dynamic spanner algorithm for an arbitrarily long sequence of updates from a fully dynamic spanner algorithm that only works for a polynomially bounded number of updates. This is particularly useful for randomized algorithms whose high-probability guarantees are obtained by taking a union bound over a polynomially bounded number of events.

Lemma 2.5 (Update Extension, Implicit in [ADK+15])**.** *Assume there is a fully dynamic algorithm for maintaining an α-spanner (for some $\alpha \geq 1$) of size at most $S(m, n, W)$ with worst-case update time $T(m, n, W)$ for up to $4n^2$ updates in G. Then there also is a fully dynamic algorithm for maintaining an α-spanner of size at most $O(S(m, n, W))$ with worst-case update time $O(T(m, n, W))$ for an arbitrary number of updates.*

For completeness, we give the proof of this lemma in an appendix. We remark that is is entirely identical to the one given in [ADK+15].

3 Algorithms for Partial Spanner Computation

Our goal in this section is to describe fully dynamic algorithm for *partial* spanner computation. We prove lemmas that can informally be summarized as follows: given a graph G with an orientation $\vec{G}$, one can build a very sparse spanner that only covers the edges leaving nodes with large out-degree in $\vec{G}$. There is a smooth tradeoff between the sparsity of the spanner and the out-degree threshold beyond which edges are spanned.

As a crucial subroutine, our algorithms employ a fully dynamic algorithm for maintaining certain structural information related to a *clustering* of G. We will describe this subroutine first.

3.1 Maintaining a clustering structure

In the spanner literature, a *clustering* of a graph $G = (V, E)$ is a partition of the nodes V into *clusters* $C_1, \ldots, C_k$, as well as a "leftover" set of *free* nodes F, with the following properties:

- For each cluster C_i, there exists a "center" node $x_i \in V$ such that all nodes in C_i are adjacent to x_i.
- The free nodes F are precisely the nodes that are not adjacent to any cluster center.

In this paper, we will represent clusterings with a vector c indexed by V, such that for any clustered $v \in V$ we have $c[v]$ equal to its cluster center, and for any free $v \in V$ we use the convention $c[v] = \infty$.

We will use the following subroutine in our main algorithms:

Lemma 3.1. *Given an oriented graph $\vec{G} = (V, \vec{E})$ and a set of cluster centers $S = \{s_1, \ldots, s_k\}$, there is a fully dynamic algorithm that simultaneously maintains:*

1. *A clustering c of $G = (V, E)$ with centers S*
2. *For each node v and each cluster index $i \in \{1, \ldots, k\}$, the set*
$$In(v, i) := \{u \in In(v) \mid c[u] = i\}$$
(i.e. the incoming neighbors to v from cluster i)
3. *For every pair of cluster indices $i, j \in \{1, \ldots, k\}$, the set*
$$In(i, j) := \{(u, v) \in \vec{E} \mid c[u] = j, c[v] = i\}$$
(i.e. the incoming neighbors to cluster i from cluster j).

This algorithm has worst-case update time $O(\Delta^+(\vec{G}) \log n)$, where $\Delta^+(\vec{G})$ is the maximum out-degree of $\vec{G}$.

The second $In(v, i)$ sets will be useful for the 3-spanner, while the third $In(i, j)$ sets will be useful for the 5-spanner.

The implementation of this lemma is extremely straightforward; it is not hard to show that the necessary data structures can be maintained in the naive way by simply passing a message along the outgoing edges from u and v whenever an edge (u, v) is inserted or deleted. Due to space constraints, we defer full implementation details and pseudocode to Appendix A.

3.2 Maintaining a partial 3-spanner

We next show how to convert Lemma 3.1 into a fully dynamic algorithm for maintaining a partial 3-spanner of a graph, as described in the introduction. Specifically:

Lemma 3.2. *For every integer $1 \leq d \leq n$, there is a fully dynamic algorithm that takes an oriented graph $\vec{G} = (V, \vec{E})$ on input and maintains subgraphs $A = (V, E_A), \vec{B} = (V, \vec{E}_B)$ (i.e. $\vec{B}$ is oriented but A is not) over a sequence of $4n^2$ updates with the following properties:*

- *$d_A(u, v) \leq 3$ for every edge $\{u, v\}$ in $E \setminus E_B$*
- *A has size $|A| = O(n^2(\log n)/d + n)$*
- *The maximum out-degree of $\vec{B}$ is $\Delta^+(\vec{B}) \leq d$.*
- *With every update in G, at most 4 edges are changed in $\vec{B}$.*

Further, this algorithm has worst-case update time $O(\Delta^+(\vec{G}) \log n)$. The algorithm is randomized, and all of the above properties hold with high probability against an oblivious adversary.

Informally, this lemma states the following. Edges leaving nodes with high out-degree are easy for us to span; we maintain A as a sparse spanner of these edges. Edges leaving nodes with low out-degree are harder for us to span, and we maintain $\vec{B}$ as a collection of these edges.

Note that this lemma is considerably *stronger* than the existence of a 3-spanner. In particular, by setting $d = \sqrt{n \log n}$ and then using $A \cup \vec{B}$ as a spanner of G, we obtain a fully dynamic algorithm for maintaining a 3-spanner:

Corollary 3.3. *There is a fully dynamic algorithm for maintaining a 3-spanner of size $O(n^{1+1/2}\sqrt{\log n})$ for an oriented graph $\vec{G}$ with worst-case update time $O(\Delta^+(\vec{G}) \log n)$. The stretch and the size guarantee both hold with high probability against an oblivious adversary.*

The proof is essentially immediate from Lemma 3.2; we omit it because it is non-essential. The detail of handling only $4n^2$ updates is not necessary in this corollary, due to Lemma 2.5.

Looking forward, we will wait until Lemma 4 to show precisely how the extra generality in Lemma 3.2 is useful towards strong worst-case update time. The rest of this subsection is devoted to the proof of Lemma 3.2.

3.2.1 Algorithm

It will be useful in this algorithm to fix an arbitrary ordering of the nodes in the graph. This allows us to discuss the "smallest" or "largest" node in a list, etc.

We initialize the algorithm by determining a set of cluster centers S via random sampling. Specifically, every node of G is added to S independently with probability $p = \min(x/d, 1)$ where $x = a \ln(4n^5) + 1$ for some error parameter $a \geq 1$. We then use the algorithm of Lemma 3.1 above to maintain a clustering with $S = \{s_1, \ldots, s_k\}$ as the set of cluster centers. The subgraphs A and $\vec{B}$ are defined according to the following three rules:

1. For every clustered node v (i.e. $c[v] \neq \infty$), A contains the edge $\{v, c[v]\}$ from v to its cluster center in S.

2. For every clustered node v (i.e. $c[v] \neq \infty$) and every cluster index $1 \leq i \leq k$, A contains the edge $\{u, v\}$ to the first node $u \in In(v, i)$ (unless $In(v, i) = \emptyset$).

3. For every node u and every node v among the *first* d neighbors of u in $N(u)$ (with respect to an arbitrary fixed ordering of the nodes), $\vec{B}$ contains the edge (u, v). Alternately, if $|N(u)| \leq d$, then $\vec{B}$ contains all such edges (u, v).

We maintain the subgraph $\vec{B}$ in the following straightforward way. For every node u we store $N(u)$, the set of neighbors of u, in two self-balancing binary search trees: $N_{\leq d}(u)$ for the first d neighbors and $N_{>d}(u)$ for the remaining neighbors. Every time an edge (u, v) or an edge (v, u) is inserted into $\vec{G}$, we add v to $N_{\leq d}(u)$ and we add (u, v) to $\vec{B}$. If $N_{\leq d}(u)$ now contains more than d nodes, we remove the largest element v', add it to $N_{>d}(u)$, and remove (u, v') from $\vec{B}$.[3] Similarly, every time an edge (u, v) or an edge (v, u) is deleted from $\vec{G}$, we first check if v is contained in $N_{>d}(u)$ and if so remove it from $N_{>d}(u)$. Otherwise, we first remove v from $N_{\leq d}(u)$ and (u, v) from $\vec{B}$. Then we find the smallest node v' in $N_{>d}(u)$, remove v' from $N_{>d}(u)$, add v' to $N_{\leq d}(u)$, and add (u, v) to $\vec{B}$.

We now explain how to maintain the subgraph A. As an underlying subroutine, we use the algorithm of Lemma 3.1 to maintain a clustering w.r.t. centers S. On each edge insertion/deletion, we first update the clustering, and then perform the following steps:

1. For every node v for which $c[v]$ has just changed from some center s_i to some other center s_j, we remove the edge $\{v, s_i\}$ from A (if $i \neq \infty$) and add the edge $\{v, s_j\}$ to A (if $j \neq \infty$).

2. For every node u that has been added to $In(v, i)$ for some node v and some $1 \leq i \leq k$, we check if u is now the first node in $In(v, i)$. If so, we add the edge $\{u, v\}$ to A and remove the edge $\{u', v\}$ for the previous first node u' of $In(v, i)$ (if $In(v, i)$ was previously non-empty).

3. For every node u that is removed from $In(v, i)$ for some node v and some $1 \leq i \leq k$, we check if u was the first node in $In(v, i)$ and if so remove the edge $\{u, v\}$ from A and add the edge $\{u', v\}$ for the new first node u' of $In(v, i)$ (if $In(v, i)$ is still non-empty).

3.2.2 Analysis

To bound the update time required by this algorithm, we will argue that we spend $O(\Delta^+(\vec{G}) \log n)$ time per update maintaining A, and $O(\log n)$ time per update maintaining $\vec{B}$ (which, in our applications, is always dominated by $O(\Delta^+(\vec{G}) \log n)$). By Lemma 3.1, the clustering structure can be updated in time $O(\Delta^+(\vec{G}) \log n)$. Each operation in steps 1, 2, and 3 above can be charged to the corresponding changes in s_i and $In(v, i)$ and thus can also be carried out within the same $O(\Delta^+(\vec{G}) \log n)$ time bound. Updating the subgraph $\vec{B}$ takes time $O(\log n)$, since we must perform a constant number of queries and updates in the corresponding self-balancing binary search trees.

We now show that the subgraphs A and $\vec{B}$ have all of the properties claimed in Lemma 3.2. First, we will discuss the sparsity bounds on A and $\vec{B}$. Observe that rule 1 contributes at most n edges to A, since every node is contained in at most one cluster. Next, recall

[3] Note that the node v' that is removed from $N_{\leq d}(u)$ might be the node v we have added in the first place.

that the number of cluster centers S is $|S| = k = O(n(\log n)/d)$ (by Lemma 2.3, with high probability). Thus, A contains only $O(nk) = O(n^2(\log n)/d)$ edges due to rule 2. As the only edges of $\vec{B}$ come from rule 3, the maximum out-degree in $\vec{B}$ is d. The claimed sparsity bounds therefore hold. Furthermore, with every insertion or deletion of an edge $\{u, v\}$ in G, at most one edge is added to or removed from the first d neighbors of u and v, respectively. This implies that there are at most 4 changes to $\vec{B}$ with every update in G. It now only remains to show that A is a 3-spanner of $G \setminus B$.

Lemma 3.4. *For up to $4n^3$ updates, $d_A(u, v) \leq 3$ for every edge $\{u, v\}$ in $E \setminus E_B$ with high probability.*

Proof. Let $\{u, v\}$ be an edge of $E \setminus E_B$. Assume without loss of generality that the edge is oriented from u to v in $\vec{G}$. As $\{u, v\}$ is not contained in B, by rule 3 above we have $|N(u)| > d$. Thus, by Lemma 2.3, since the cluster centers S were chosen by random sampling, with high probability there exists a cluster center in the first d outgoing neighbors of each node in all of up to $4n^3$ different versions of G (i.e. one version for each of the $4n^3$ updates considered). Therefore $c[u] = i$ for some $1 \leq i \leq k$ and, by rule 1, A contains the edge $\{u, s_i\}$. Since $c[u] = i$, and u is an incoming neighbor of v in $\vec{G}$, we have $In(v, i) \neq \emptyset$, and thus, for the first element u' of $In(v, i)$, A contains the edge $\{u', v\}$ (by rule 2). As $c[u'] = i$, A contains the edge $\{s_i, u'\}$ by rule 1. This means that A contains the edges $\{u, s_i\}$, $\{s_i, u'\}$, and $\{u', v\}$, and thus there is a path from u to v of length 3 in A as desired. □

This now also completes the proof of Lemma 3.2.

3.3 5-spanner

The 5-spanner algorithm is very similar to the 3-spanner algorithm above, but we define the edges of the spanner in a slightly different way. Instead of including an edge from each node to each cluster, we have an edge between each *pair* of clusters. Thus, the subgraphs A and $\vec{B}$ are defined according to the following three rules:

1. For every clustered node v (i.e. $c[v] \neq \infty$), A contains the edge $\{v, c[v]\}$ from v to its cluster center in S.

2. For every pair of distinct cluster indices $1 \leq i, j \leq k$, A contains the edge $\{u, v\}$, where $\{u, v\}$ is the first element in $In(i, j)$ (unless $In(i, j) = \emptyset$).

3. For every node u and every node v among the *first* d neighbors of u in $N(u)$ (with respect to an arbitrary fixed ordering of the nodes), $\vec{B}$ contains the edge (u, v). Alternately, if $|N(u)| \leq d$, then $\vec{B}$ contains all such edges (u, v)..

Beyond this slightly altered definition, we use the same approach for maintaining A and $\vec{B}$ as in the 3-spanner. The guarantee on the stretch can be proved as follows.

Lemma 3.5. *For up to $4n^3$ updates, $d_A(u, v) \leq 5$ for every edge $\{u, v\}$ in $E \setminus E_B$ with high probability.*

Proof. Let $\{u, v\}$ be an edge of $E \setminus E_B$. Assume without loss of generality that the edge is oriented from u to v in $\vec{G}$. As $\{u, v\}$ is not contained in B, by rule 3 above we have

$|N(v)| > d$. We now apply Lemma 2.3 to argue that there is a cluster center in the first d outgoing neighbors of each node in up to $4n^3$ versions of the graph (one version for each update to be considered). Thus, $N(v)$ contains a cluster center from S with high probability. Therefore $c[v] = i$ for some $1 \leq i \leq k$ and, by rule 1, A contains the edge $\{v, s_i\}$. By the same argument, $N(u)$ contains a cluster center from S with high probability and thus A contains an edge $\{u, s_j\}$ where $c[u] = j$ for some $1 \leq j \leq k$. Since $c[v] = i$, $c[u] = j$, and u is an incoming neighbor of v in $\vec{G}$, we have $In(i, j) \neq \emptyset$, and thus, for the first element (u', v') of $In(i, j)$, A contains the edge $\{u', v'\}$ (by rule 2). As $c[v'] = i$, $c[u'] = j$, A contains the edges $\{v', s_i\}$ and $\{u', s_j\}$ by rule 1. This means that A contains the edges $\{u, s_i\}$, $\{s_i, u'\}$, $\{u', v'\}$, $\{v', s_i\}$ and $\{s_i, v\}$, and thus there is a path from u to v of length 5 in A as desired. □

Note that in this proof we exploit the fact that we have cluster centers for both u and v whenever the edge $\{u, v\}$ is missing. This motivates our design choice for considering the whole neighborhood of a node to determine its cluster. If we only considered cluster centers in the outgoing neighbors of a node, the resulting clustering would still be good enough for the 3-spanner, but the argument above for the 5-spanner would break down.

All other properties of the 5-spanner can be proved in an essentially identical manner to the 3-spanner. We can summarize the obtained guarantees as follows.

Lemma 3.6. *For every integer $1 \leq d \leq n$, there is a fully dynamic algorithm that takes an oriented graph $\vec{G} = (V, \vec{E})$ on input and maintains subgraphs $A = (V, E_A), \vec{B} = (V, \vec{E}_B)$ (i.e. $\vec{B}$ is oriented but A is not) over a sequence of $4n^2$ updates with the following properties:*

- *$d_A(u, v) \leq 5$ for every edge $\{u, v\}$ in $E \setminus E_B$*
- *A has size $|A| = O((n^2 \log^2 n)/d^2 + n)$*
- *The maximum out-degree of $\vec{B}$ is $\Delta^+(\vec{B}) \leq d$.*
- *With every update in G, at most 4 edges are changed in $\vec{B}$.*

Further, this algorithm has worst-case update time $O(\Delta^+(\vec{G}) \log n)$. The algorithm is randomized, and all of the above properties hold with high probability against an oblivious adversary.

Once again, this lemma generalizes the construction of a sparse 5-spanner. By setting $d = (n \log n)^{2/3}$ we can obtain:

Corollary 3.7. *There is a fully dynamic algorithm for maintaining a 5-spanner of size $O(n^{1+1/3} \log^{2/3} n)$ for an oriented graph $\vec{G}$ with worst-case update time $O(\Delta^+(\vec{G}) \log n)$. The stretch and the size guarantee both hold with high probability against an oblivious adversary.*

4 Out-degree Reduction for Improved Update Time

Our goal is now to use Lemmas 3.2 and 3.6 to obtain spanner algorithms with sublinear update time. Since we obtain our 3-spanner and 5-spanner in an essentially identical manner, we will explain only the 3-spanner in full detail, and then sketch the 5-spanner construction.

We next establish the following simple generalization of Lemma 3.2:

Lemma 4.1. *For every integer $1 \leq s \leq n$ and $1 \leq d \leq n$, there is a fully dynamic algorithm that takes an oriented graph $\vec{G} = (V, \vec{E})$ on input and maintains subgraphs $A = (V, E_A), \vec{B} = (V, \vec{E}_B)$ (i.e. $\vec{B}$ is oriented but A is not) over a sequence of $4n^2$ updates with the following properties:*

- *$d_A(u, v) \leq 3$ for every edge $\{u, v\}$ in $E \setminus E_B$*
- *A has size $|A| = O(\Delta^+(\vec{G})n^2(\log n)/(sd))$*
- *The maximum out-degree of $\vec{B}$ is $\Delta^+(\vec{B}) \leq \Delta^+(\vec{G}) \cdot d/s$.*
- *With every update in G, at most 4 edges are changed in $\vec{B}$.*

Further, this algorithm has worst-case update time $O(s \log n)$. The algorithm is randomized, and all of the above properties hold with high probability against an oblivious adversary.

In particular, Lemma 3.2 is the special case of this lemma in which $s = \Delta^+(\vec{G})$.

Proof. We orient each incoming edge of G in an arbitrary way. We then maintain a partitioning of the (oriented) edges of G into $t := \lceil \Delta^+(\vec{G})/s \rceil$ groups, such that in each group each node has at most s outgoing edges. Specifically, we perform this partitioning by maintaining the current out-degree of each node u in $\vec{G}$, and we assign a new edge (u, v) which is the x^{th} edge leaving u in $\vec{G}$ to the subgraph $\vec{G}_{\lceil x/s \rceil}$. In this way, we form t subgraphs $\vec{G}_1, \ldots \vec{G}_t$ of $\vec{G}$, each of which has $\Delta^+(\vec{G}_i) \leq s$.

We now run the algorithm of Lemma 3.2 on each $\vec{G}_i$ to maintain, for each $1 \leq i \leq t$, two subgraphs A_i and $\vec{B}_i$ as specified in the lemma. Let $A = \bigcup A_i$ and $\vec{B} = \bigcup \vec{B}_i$ denote the unions of these subgraphs.

Observe that every update in G only changes exactly one of the subgraphs $\vec{G}_i$ and thus only must be executed in one corresponding instance of the algorithm of Lemma 3.2. As we have "artificially" bounded the maximum out-degree of every subgraph $\vec{G}_i$ by s, the claimed bounds on the update time and the properties of A and $\vec{B}$ now follow simply from Lemma 3.2. □

We now recursively apply the "out-degree reduction" of the previous lemma to obtain subgraphs $\vec{B}$ of smaller and smaller out-degree. Finally, at bottom level, the maximum out-degree is small enough that we can apply a "regular" spanner algorithm to it.

Theorem 4.2. *There is a fully dynamic algorithm for maintaining a 3-spanner of size $O(n^{1+1/2} \log^{1/2} n \log \log n)$ with worst-case update time $O(n^{3/4} \log^4 n)$.*

Proof. Our spanner construction is as follows (we temporarily omit details related to parameter choices, which influence the resulting update time). Apply Lemma 4.1 to obtain subgraphs $A_1, \vec{B}_1$. Include all edges in A in the spanner, and then recursively apply Lemma 4.1 to $\vec{B}$ to obtain $A_2, \vec{B}_2$. Repeat to depth ℓ (for some parameter ℓ that will be chosen later). At bottom level, instead of recursing, we apply the algorithm from Corollary 3.3 to obtain a 3-spanner of $\vec{B}$.

More formally, we set $\vec{B}_0 = \vec{G}_0$, and for every $1 \leq j \leq \ell$ we let A_j and $\vec{B}_j$ be the graphs maintained by the algorithm of Lemma 4.1 on input $\vec{B}_{j-1}$ using parameters s and d_j to be

chosen later.[4] Further, we let H' be the spanner maintained by the algorithm of Corollary 3.3 on input $\vec{B}_\ell$. The resulting graph maintained by our algorithm is $H = \bigcup_{1 \le j \le \ell} A_j \cup H'$. Then, by Lemma 4.1, we have the following properties for every $1 \le j \le \ell$:

- $d_{A_j}(u, v) \le 3$ for every edge $\{u, v\}$ in $B_{j-1} \setminus B_j$
- A_j has size $|A_j| = O(\Delta^+(\vec{B}_{j-1})n^2(\log n)/(sd_j))$
- The maximum out-degree of $\vec{B}_j$ is $\Delta^+(\vec{B}_j) \le \Delta^+(\vec{B}_{j-1}) \cdot d_j/s$.
- With every update in $\vec{B}_{j-1}$, at most 4 edges are changed in $\vec{B}_j$.

It is straightforward to see that the resulting graph H is a 3-spanner of G: At each level j of the recursion, A_j spans all edges of B_{j-1} *except* those that appear in the current subgraph $\vec{B}_j$. Thus, at bottom level, the only non-spanned edges of G are those in the final subgraph $\vec{B}_\ell$. For these edges we explicitly add a 3-spanner H' of $\vec{B}_\ell$ to H. By Lemma 2.1, this suffices to produce a 3-spanner of all of G.

Now that we have correctness of the construction, it remains to bound the number of edges in the output spanner. First, observe that, by induction,

$$\Delta^+(\vec{B}_j) \le n \cdot \prod_{1 \le j' \le j} d_{j'}/s^j$$

for all $1 \le j \le \ell$. Since additionally H' has size $O(n^{1+1/2}\log^{1/2} n)$ by Corollary 3.3, the total number of edges in H is

$$\begin{aligned} |H| = \sum_{1 \le j \le \ell} |A_j| + |H'| &\le \sum_{1 \le j \le \ell} O\left(\frac{\Delta^+(B_{j-1})n^2 \log n}{sd_j}\right) + O(n^{1+1/2}\log^{1/2} n) \\ &\le \sum_{1 \le j \le \ell} O\left(\frac{\left(\prod_{1 \le j' \le j-1} d_{j'}\right) n^3 \log n}{s^j d_j}\right) + O(n^{1+1/2}\log^{1/2} n)\,. \end{aligned}$$

Thus, our spanner satisfies the claimed sparsity bound so long as the union of all ℓ of the A_j subgraphs fit within the claimed sparsity bound; this will be the case if we balance all summands.

We next bound the update time of our algorithm. Each change to some $\vec{B}_j$ causes at most 4 changes in the next level $\vec{B}_{j+1}$, and thus the number of changes to $\vec{B}_j$ can propagate exponentially. Thus, for every $0 \le j \le \ell - 1$, a single update in $\vec{G}$ could cause at most 4^j changes to $\vec{B}_j$. Each of the ℓ instances of the algorithm of Lemma 4.1 has a worst-case update time of $O(s \log n)$ and the algorithm of Corollary 3.3 has a worst-case update time of $\Delta^+(\vec{B}_\ell \log n)$. Since

$$\Delta^+(\vec{B}_\ell) \le n \cdot \prod_{1 \le j \le \ell} d_j/s^\ell$$

[4]Note that the parameter s is the same for all levels of the recursion, whereas the parameter d_j is not.

the worst-case update time of our overall algorithm is

$$O\left(\left(\sum_{j=0}^{\ell-1} 4^j s + 4^\ell \Delta^+(\vec{B}_\ell)\right) \cdot \log n\right) \leq O\left(\left(s + \frac{n \cdot \prod_{1 \leq j \leq \ell} d_j}{s^\ell}\right) \cdot 4^\ell \log n\right).$$

Our goal is now to choose parameters s_j, d, ℓ to minimize this expression subject to the constraint on spanner size given above. To achieve this, we set parameters as follows:

$$\begin{aligned} \ell &= \log\log n\,, \\ s &= n^{(3\cdot 2^\ell - 1)/(2^{\ell+2}-2)} \log n\,, \text{ and} \\ d_j &= n^{(3\cdot 2^\ell - 2^{j-1} - 1)/(2^{\ell+2}-2)} \log n\,. \end{aligned}$$

These values were obtained with the help of a computer algebra solver, so we do not have explicit computations to show for them. □

We now turn our attention to the 5-spanner. Similar to Lemma 4.1 above, we can use Lemma 3.6 to perform a similar out-degree reduction step for our dynamic 5-spanner algorithm.

Lemma 4.3. *For every integer $1 \leq s \leq n$ and $1 \leq d \leq n$, there is a fully dynamic algorithm that takes an oriented graph $\vec{G} = (V, \vec{E})$ on input and maintains subgraphs $A = (V, E_A), \vec{B} = (V, \vec{E}_B)$ (i.e. $\vec{B}$ is oriented but A is not) over a sequence of $4n^2$ updates with the following properties:*

- *$d_A(u, v) \leq 5$ for every edge $\{u, v\}$ in $E \setminus E_B$*
- *A has size $|A| = O(\Delta^+(\vec{G})n^2(\log^2 n)/(sd^2))$*
- *The maximum out-degree of $\vec{B}$ is $\Delta^+(\vec{B}) \leq \Delta^+(\vec{G}) \cdot d/s$.*
- *With every update in G, at most 4 edges are changed in $\vec{B}$.*

Further, this algorithm has worst-case update time $O(s \log n)$. The algorithm is randomized, and all of the above properties hold with high probability against an oblivious adversary.

The proof of this lemma is essentially identical to the proof of Lemma 4.1 and has thus been omitted.

Just as in the case of the 3-spanner, we use this lemma to show:

Theorem 4.4. *There is a fully dynamic algorithm for maintaining a 5-spanner of size $O(n^{1+1/3} \log^{2/3} n \log\log n)$ with worst-case update time $O(n^{5/9} \log^4 n)$.*

Proof. The proof is identical to the proof of Theorem 4.2, except that the proper parameter balance is now:

$$\begin{aligned} \ell &= \log\log n\,, \\ s &= n^{(5\cdot 3^\ell - 2^{\ell+1})/(3^{\ell+2} - 3\cdot 2^{\ell+1})} \log n\,, \text{ and} \\ d_j &= n^{(5\cdot 3^\ell - 3^{j-1}2^{\ell-j+2} - 2^{\ell+1})/(3^{\ell+2} - 3\cdot 2^{\ell+1})} \log n\,. \end{aligned}$$

□

Finally, we can also show:

Theorem 4.5. *There is a fully dynamic algorithm for maintaining a 5-spanner of size* $O(n^{1+1/2}\log^{1/2} n \log\log n)$ *with worst-case update time* $O(n^{1/2}\log^4 n)$.

Proof. The proof is identical to the proof of Theorems 4.2 and 4.4, except that we now use the parameter balance

$$\begin{aligned}\ell &= \log\log n\,,\\ s &= n^{(3^{\ell+1}-2^{\ell})/(2\cdot 3^{\ell+1}-2^{\ell+2})}\log n\,,\ \text{and}\\ d_j &= n^{(3^{\ell+1}-3^j\cdot 2^{\ell-j}-2^{\ell})/(2\cdot 3^{\ell+1}-2^{\ell+2})}\log n\,.\end{aligned}$$

and we maintain the dynamic 3-spanner H' of size $O(n^{1+1/2}\log^{1/2} n)$ from Corollary 3.7 at bottom level. □

This spanner has non-optimal size/stretch tradeoff, but enjoys the best worst-case update time that we are currently able to construct.

Acknowledgements

We want to thank Seeun William Umboh for many fruitful discussions at Simons.

References

[AC13] Ittai Abraham and Shiri Chechik. "Dynamic Decremental Approximate Distance Oracles with $(1+\epsilon, 2)$ stretch". In: *CoRR* abs/1307.1516 (2013) (cit. on p. 6).

[ACT14] Ittai Abraham, Shiri Chechik, and Kunal Talwar. "Fully Dynamic All-Pairs Shortest Paths: Breaking the $O(n)$ Barrier". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. 2014, pp. 1–16 (cit. on p. 6).

[ADK+15] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. "On Fully Dynamic Graph Sparsifiers". In: *CoRR* abs/1604.02094 (2015) (cit. on pp. 6, 7).

[AFI06] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. "Small Stretch Spanners on Dynamic Graphs". In: *Journal of Graph Algorithms and Applications* 10.2 (2006). Announced at ESA'05, pp. 365–385 (cit. on p. 3).

[AKL+15] Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. "Online Dictionary Matching with One Gap". In: *CoRR* abs/1503.07563 (2015) (cit. on p. 6).

[Awe85] Baruch Awerbuch. "Complexity of Network Synchronization". In: *Journal of the ACM* 32.4 (1985), pp. 804–823 (cit. on p. 3).

[Bas08] Surender Baswana. "Streaming algorithm for graph spanners—single pass and constant processing time per edge". In: *Information Processing Letters* 106.3 (2008), pp. 110–114 (cit. on p. 6).

[BDBK+94] Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. "On the Power of Randomization in On-Line Algorithms". In: *Algorithmica* 11.1 (1994). Announced at STOC'90, pp. 2–14 (cit. on p. 4).

[BE13] Leonid Barenboim and Michael Elkin. "Distributed Graph Coloring: Fundamentals and Recent Developments". In: Morgan and Claypool, 2013. Chap. Arb-defective Coloring (cit. on p. 5).

[BK10] Surender Baswana and Telikepalli Kavitha. "Faster Algorithms for All-pairs Approximate Shortest Paths in Undirected Graphs". In: *SIAM Journal on Computing* 39.7 (2010). Announced at FOCS'10, pp. 2865–2896 (cit. on p. 3).

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. "Fully Dynamic Randomized Algorithms for Graph Spanners". In: *ACM Transactions on Algorithms* 8.4 (2012). Announced at ESA'06, and SODA'08, 35:1–35:51 (cit. on pp. 1, 3, 4, 7).

[BR11] Aaron Bernstein and Liam Roditty. "Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions". In: *Symposium on Discrete Algorithms (SODA)*. 2011, pp. 1355–1365 (cit. on p. 6).

[BS07] Surender Baswana and Sandeep Sen. "A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs". In: *Random Structures & Algorithms* 30.4 (2007). Announced at ICALP'03, pp. 532–563 (cit. on pp. 3, 5).

[Che14] Shiri Chechik. "Approximate distance oracles with constant query time". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 654–663 (cit. on p. 3).

[Che15] Shiri Chechik. "Approximate Distance Oracles with Improved Bounds". In: *Symposium on Theory of Computing (STOC)*. 2015, pp. 1–10 (cit. on p. 3).

[Cow01] Lenore Cowen. "Compact Routing with Minimum Stretch". In: *Journal of Algorithms* 38.1 (2001). Announced at SODA'99, pp. 170–183 (cit. on p. 3).

[CW04] Lenore Cowen and Christopher G. Wagner. "Compact roundtrip routing in directed networks". In: *Journal of Algorithms* 50.1 (2004). Announced at PODC'00, pp. 79–95 (cit. on p. 3).

[DHZ00] Dorit Dor, Shay Halperin, and Uri Zwick. "All-Pairs Almost Shortest Paths". In: *SIAM Journal on Computing* 29.5 (2000). Announced at FOCS'96, pp. 1740–1759 (cit. on pp. 3, 6).

[DI04] Camil Demetrescu and Giuseppe F. Italiano. "A New Approach to Dynamic All Pairs Shortest Paths". In: *Journal of the ACM* 51.6 (2004). Announced at STOC'03, pp. 968–992 (cit. on p. 4).

[EGI+97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. "Sparsification—A Technique for Speeding Up Dynamic Graph Algorithms". In: *Journal of the ACM* 44.5 (1997). Announced at FOCS'92, pp. 669–696 (cit. on p. 4).

[Elk05] Michael Elkin. "Computing almost shortest paths". In: *ACM Transactions on Algorithms* 1.2 (2005). Announced at PODC'01, pp. 283–323 (cit. on p. 3).

[Elk07] Michael Elkin. "A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners". In: *Symposium on Principles of Distributed Computing (PODC)*. 2007, pp. 185–194 (cit. on p. 6).

[Elk11] Michael Elkin. "Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners". In: *ACM Transactions on Algorithms* 7.2 (2011). Announced at ICALP'07, 20:1–20:17 (cit. on pp. 3, 6).

[Erd64] Paul Erdős. "Extremal problems in graph theory". In: *Theory of graphs and its applications* (1964), pp. 29–36 (cit. on p. 3).

[FKM+08] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. "Graph Distances in the Data-Stream Model". In: *SIAM Journal on Computing* 38.5 (2008), pp. 1709–1727 (cit. on p. 6).

[FPZ+04] Arthur M. Farley, Andrzej Proskurowski, Daniel Zappala, and Kurt Windisch. "Spanners and message distribution in networks". In: *Discrete Applied Mathematics* 137.2 (2004), pp. 159–171 (cit. on p. 3).

[Fre85] Greg N. Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications". In: *SIAM Journal on Computing* 14.4 (1985). Announced at STOC'83, pp. 781–798 (cit. on p. 4).

[HKN14a] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "A Subquadratic-Time Algorithm for Dynamic Single-Source Shortest Paths". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1053–1072 (cit. on p. 6).

[HKN14b] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 146–155 (cit. on p. 6).

[HLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *Journal of the ACM* 48.4 (2001). Announced at STOC'98, pp. 723–760 (cit. on p. 4).

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. "Dynamic graph connectivity in polylogarithmic worst case time". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1131–1142 (cit. on p. 3).

[KKP+14] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. "Orienting Fully Dynamic Graphs with Worst-Case Time Bounds". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2014, pp. 532–543 (cit. on p. 6).

[NS13] Ofer Neiman and Shay Solomon. "Simple Deterministic Algorithms for Fully Dynamic Maximal Matching". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 745–754 (cit. on p. 6).

[PR14] Mihai Pătraşcu and Liam Roditty. "Distance Oracles beyond the Thorup-Zwick Bound". In: *SIAM Journal on Computing* 43.1 (2014). Announced at FOCS'10, pp. 300–311 (cit. on p. 3).

[PS16] David Peleg and Shay Solomon. "Dynamic $(1+\epsilon)$-Approximate Matchings: A Density-Sensitive Approach". In: *Symposium on Discrete Algorithms (SODA)*. 2016, pp. 712–729 (cit. on p. 6).

[PS89] David Peleg and Alejandro A. Schäffer. "Graph spanners". In: *Journal of Graph Theory* 13.1 (1989), pp. 99–116 (cit. on p. 3).

[PU89] David Peleg and Jeffrey D. Ullman. "An Optimal Synchronizer for the Hypercube". In: *SIAM Journal of Computing* 18.4 (1989). Announced at PODC'87, pp. 740–747 (cit. on p. 3).

[RTZ05] Liam Roditty, Mikkel Thorup, and Uri Zwick. "Deterministic Constructions of Approximate Distance Oracles and Spanners". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2005, pp. 261–272 (cit. on p. 3).

[RTZ08] Liam Roditty, Mikkel Thorup, and Uri Zwick. "Roundtrip spanners and roundtrip routing in directed graphs". In: *ACM Transactions on Algorithms* 4.3 (2008). Announced at SODA'02 (cit. on p. 3).

[RZ12] Liam Roditty and Uri Zwick. "Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs". In: *SIAM Journal on Computing* 41.3 (2012). Announced at FOCS'04, pp. 670–683 (cit. on p. 6).

[San04] Piotr Sankowski. "Dynamic Transitive Closure via Dynamic Matrix Inverse". In: *Symposium on Foundations of Computer Science (FOCS)*. 2004, pp. 509–517 (cit. on p. 3).

[Tho05] Mikkel Thorup. "Worst-Case Update Times for Fully-Dynamic All-Pairs Shortest Paths". In: *Symposium on Theory of Computing (STOC)*. 2005, pp. 112–119 (cit. on p. 4).

[TZ01] Mikkel Thorup and Uri Zwick. "Compact routing schemes". In: *Symposium on Parallel Algorithms and Architectures (SPAA)*. 2001, pp. 1–10 (cit. on p. 3).

[TZ05] Mikkel Thorup and Uri Zwick. "Approximate Distance Oracles". In: *Journal of the ACM* 52.1 (2005). Announced at STOC'01, pp. 74–92 (cit. on p. 3).

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. "High-Probability Parallel Transitive-Closure Algorithms". In: *SIAM Journal on Computing* 20.1 (1991). Announced at SPAA'90, pp. 100–125 (cit. on p. 7).

[Wen91] Rephael Wenger. "Extremal graphs with no C^4's, C^6's, or C^{10}'s". In: *Journal of Combinatorial Theory B, Series B* 52.1 (1991), pp. 113–116 (cit. on p. 4).

A Updating the Clustering

In the following we give the straightforward algorithm for maintaining the clustering with worst-case update time proportional to the maximum out-degree of the original graph mentioned in Section 3.1. To make the presentation of this algorithm more succinct we assume that there is some (arbitrary, but fixed) ordering on the nodes. Furthermore, we assume that the nodes of S are given according to this order, i.e. $s_1 \leq s_2 \leq \cdots \leq s_k$. For every node v, we maintain $c[v]$ as the smallest i such that s_i is a neighbor of v (or ∞ if no such neighbor exists). Additionally, we naturally extend the sets $In(v, i)$ and $In(i, j)$ to the case $i, j \in \{1, \ldots, k, \infty\}$.

We begin with an empty graph $G = (V, \emptyset)$, a cluster vector c with $c[v] = \infty$ for all $v \in V$, and empty sets $In(i, j)$ and $In(i, v)$ for all cluster indices $1 \leq i, j \leq k$ and $v \in V$. We then modify these data structures under edge insertions and deletions as follows.

Correctness of the algorithms that follow is immediate, and is not shown formally.

A.1 Insertion of an edge (u, v)

- Add u to $In(v, i)$ for $i = c[u]$.
- Add (u, v) to $In(j, i)$ for $i = c[u]$ and $j = c[v]$
- If $u = s_i$ for some $1 \leq i \leq k$:
  - Set $j = c[v]$ (might be ∞)
  - Add u to $C[v]$.
  - If $i < j$:
    - Set $c[v] = i$
    - For every outgoing neighbor v' of v:
      Remove v from $In(v', j)$ and add v to $In(v', i)$.
      Remove (v, v') from $In(i', j)$ and add (v, v') to $In(i', i)$ where $i' = c[v']$.
- If $v = s_i$ for some $1 \leq i \leq k$:
  - Set $j = c[u]$ (might be ∞)
  - Add v to $C[u]$.
  - If $i < j$:
    - Set $c[u] = i$
    - For every outgoing neighbor v' of u:
      Remove u from $In(v', j)$ and add u to $In(v', i)$.
      Remove (u, v') from $In(i', j)$ and add (u, v') to $In(i', i)$ where $i' = c[v']$.

A.2 Deletion of an edge (u, v):

- Remove u from $In(v, i)$ for $i = c[u]$.
- Remove (u, v) from $In(j, i)$ for $i = c[u]$ and $j = c[v]$

- If $u = s_i$ for some $1 \leq i \leq k$:
  - Remove u from $C[v]$.
  - If $c[v] = s_i$:
    - Let j be minimal such that s_j is in $C[v]$ (might be ∞)
    - Set $c[v] = j$
    - For every outgoing neighbor v' of v:
      Remove v from $In(v', i)$ and add v to $In(v', j)$.
      Remove (v, v') from $In(i', j)$ and add (v, v') to $In(i', i)$ where $i' = c[v']$
- If $v = s_i$ for some $1 \leq i \leq k$:
  - Remove v from $C[u]$.
  - If $c[u] = s_i$:
    - Let j be minimal such that s_j is in $C[u]$ (might be ∞)
    - Set $c[u] = j$
    - For every outgoing neighbor v' of u:
      Remove u from $In(v', i)$ and add u to $In(v', j)$.
      Remove (u, v') from $In(i', j)$ and add (u, v') to $In(i', i)$ where $i' = c[v']$

B Proof of Lemma 2.5

We exploit the decomposability of spanners. We maintain a partition of G into two disjoint subgraphs G_1 and G_2 and run two instances A_1 and A_2 of the dynamic algorithm on G_1 and G_2, respectively. These two algorithms maintain a t-spanner of H_1 of G_1 and a t-spanner H_2 of G_2. By Lemma 2.4, the union $H = H_1 \cup H_2$ is a t-spanner of $G = G_1 \cup G_2$.

We divide the sequence of updates into phases of length n^2 each. In each phase of updates one of the two instances A_1, A_2 is in the state *growing* and the other one is in the state *shrinking*. A_1 and A_2 switch their states at the end of each phase. In the following we describe the algorithm's actions during one phase. Assume without loss of generality that, in the phase we are fixing, A_1 is growing and A_2 is shrinking.

At the beginning of the phase we restart the growing instance A_1. We will orchestrate the algorithm in such a way that at the beginning of the phase G_1 is the empty graph and $G_2 = G$. After every update in G we execute the following steps:

1. If the update was the insertion of some edge e, then e is added to the graph G_1 and this insertion is propagated to the *growing* instance A_1.

2. If the update was the deletion of some edge e, then e is removed from the graph G_i it is contained in and this deletion is propagated to the corresponding instance A_i.

3. In addition to processing the update in G, if G_2 is non-empty, then one arbitrary edge e is first removed from G_2 and deleted from instance A_2 and then added to G_1 and inserted into instance A_1.

Observe that these rules indeed guarantee that G_1 and G_2 are disjoint and together contain all edges of G. Furthermore, since the graph G_2 of the shrinking instance has at most n^2 edges at the beginning of the phase, the length of n^2 updates per phase guarantees that G_2 is empty at the end of the phase. Thus, the growing instance always starts with an empty graph G_1.

As both H_1 and H_2 have size at most $S(n, m, W)$, the size of $H = H_1 \cup H_2$ is $O(S(n, m, W))$. With every update in G we perform at most 2 updates in each of A_1 and A_2. It follows that the worst-case update time of our overall algorithm is $O(T(m, n, W))$. Furthermore since each of the instances A_1 and A_2 is restarted every other phase, each instance of the dynamic algorithm sees at most $4n^2$ updates before it is restarted.