Dynamic Algorithms for Graph Spanners

Surender Baswana

Max-Planck Institute for Computer Science,
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.
Email : sbaswana@mpi-sb.mpg.de

Abstract. Let $G = (V, E)$ be an undirected weighted graph on $|V| = n$ vertices and $|E| = m$ edges. For the graph G, A spanner with stretch $t \in \mathbb{N}$ is a subgraph (V, E_S), $E_S \subseteq E$, such that the distance between any pair of vertices in this subgraph is at most t times the distance between them in the graph G. We present simple and efficient dynamic algorithms for maintaining spanners with essentially optimal (expected) size versus stretch trade-off for any given unweighted graph. The main result is a decremental algorithm that takes expected $O(\text{polylog}\, n)$ time per edge deletion for maintaining a spanner with arbitrary stretch. This algorithm easily leads to a fully dynamic algorithm with sublinear (in n) time per edge insertion or deletion. Quite interestingly, this paper also reports that for stretch at most 6, it is possible to maintain a spanner fully dynamically with expected constant time per update. All these algorithms use simple randomization techniques on the top of an existing static algorithm [6] for computing spanners, and achieve drastic improvement over the previous best deterministic dynamic algorithms for spanners.

1 Introduction

A spanner is a (sparse) subgraph of a given graph that preserves approximate distance between each pair of vertices. More precisely, a t-spanner of a graph $G = (V, E)$, for any $t \geq 1$ is a subgraph $(V, E_S), E_S \subseteq E$ such that, for any pair of vertices, their distance in the subgraph is at most t times their distance in the original graph. The parameter t is called the *stretch factor* associated with the t-spanner. The concept of spanners was defined formally by Peleg and Schäffer [15] though the associated notion was used implicitly by Awerbuch [3] in the context of network synchronizers. Since then, spanner has found numerous applications in the area of distributed systems, communication networks and all pairs approximate shortest paths [3, 7, 16, 17].

Each application of spanners requires, for a specified $t \in \mathbb{N}$, a t-spanner of smallest possible size (the number of edges). Based on the famous girth conjecture by Erdős [11], Bollobás [8], and Bondy and Simonovits [9], it follows that for any $k \in \mathbb{N}$, there are graphs on n vertices whose $(2k-1)$-spanner or a $2k$-spanner will require $\Omega(n^{1+1/k})$ edges. (The conjecture has been proved for $k = 1, 2, 3$ and 5). Note that the conjectured lower bound is the same for stretch $2k$ and $(2k-1)$, and by definition, a $(2k-1)$-spanner is also a $2k$-spanner, Therefore, from perspective of an algorithmist, the aim would be to design a static (or dynamic) algorithm to compute (or maintain) a $(2k-1)$-spanner of $(n^{1+1/k})$ size for a given graph. For unweighted graphs, Halperin and Zwick [13] designed a deterministic $O(m)$ time algorithm to compute a $(2k-1)$-spanner of $O(n^{1+1/k})$ size. However, for weighted graphs, it took a series of improvements [1, 4, 10, 20, 6, 5] till an expected $O(m)$ time algorithm for computing a $(2k-1)$-spanner could be designed. This linear time randomized algorithm [6, 5] computes a $(2k-1)$-spanner of size $O(kn^{1+1/k})$ for a given weighted graph. Recently Roditty *et al.* [18] derandomized this algorithm.

In this paper, we consider the problem of efficiently maintaining a $(2k-1)$-spanner in a dynamic environment : Given a graph $G = (V, E)$, we receive an online sequence of updates which could be insertions or deletions of edges, the aim is to maintain a data structure which stores a $(2k-1)$-spanner for the graph at each moment and is very efficient to handle these updates. It is also desirable that the algorithm ensures $O(n^{1+1/k})$ size of the $(2k-1)$-spanner after each update.

Previous work :

Ausiello *et al.* [2] are the first to design dynamic algorithms for spanners. They present dynamic algorithms for maintaining spanners with stretch at most 6 only. They first design an $O(n)$ time decremental algorithm, and then employ the idea of handling insertions in a lazy fashion to design a fully dynamic algorithm with $O(n)$ time per update. The spanners maintained are of optimal size. However, the worst case space requirement of the associated data structure is $\theta(n^2)$. They extend their algorithm to weighted graphs with at most d different weights by maintaining separate spanner for the set of edges with the same weight. This leads to an increase in the size of the spanner and the update time by a factor of d.

New results :

1. *Decremental algorithm*
   We present a partial dynamic algorithm for maintaining a $(2k-1)$-spanner under deletion of edges for any $k \in \mathbb{N}$. Our algorithm ensures an expected $O(kn^{1+1/k})$ size for the $(2k-1)$-spanner and the expected update time required is $O(\text{polylog}\, n)$ per edge deletion. We employ the static algorithm [6], and overcome a few subtle problems in dynamizing it by introducing a new clustering of vertices. Our algorithm also leads to an efficient decremental algorithm for all-pairs approximate shortest paths (see Corollary 1).
2. *Fully dynamic algorithms*
   We make our decremental algorithm fully dynamic by handling the edge insertions in a lazy fashion and rebuilding the entire data structure after a period of $kn^{1+1/k}$ edge insertions. This leads to a fully dynamic algorithm for a $(2k-1)$-spanner with amortized $\tilde{O}(\frac{m}{n^{1+1/k}})$ time per edge insertion/deletion. The expected size of the $(2k-1)$-spanner maintained by the algorithm is $O(kn^{1+1/k})$.
   We also show that a fully dynamic algorithm for stretch at most 6, can be maintained with expected constant update time and expected optimal size. This algorithm follows by adding additional randomization to the static

algorithm of [6], followed by dynamizing it. However, there are some potential difficulties in extending the algorithm for arbitrary stretch. Nevertheless, it is worth exploring whether the result can be extended for arbitrary stretch.

Our algorithms can be extended to weighted graphs with d different weights in the same way as done by Ausiello *et al.* [2]. For these graphs, the bounds on the spanner size and the update time of our algorithm will increase by a factor of d. All our algorithms require $\theta(m)$ space, which is much better than the $\theta(n^2)$ space requirement of [2].

2 Preliminaries

Throughout the paper, we deal with graphs which are undirected and unweighted. We assume that the vertices are numbered from 1 to n. We shall maintain the set of edges of the graph using a dynamic hash table (see [14]). Using this hash table, it requires $O(1)$ worst-case time for lookup and $O(1)$ expected time for any insertion and deletion. The space occupied by the hash table at any moment will be of the order of the number of edges present in the graph at that moment. Each edge of the graph will have a field to denote whether or not it is a spanner edge at that moment of time. We also assume without loss of generality that $m = \Omega(kn^{1+1/k})$, since otherwise for maintaining a $(2k-1)$-spanner we just keep all the edges in the $(2k-1)$-spanner and just update the hash table for edge insertion or deletion. The distance between any two vertices in not merely a function of the edges in their local neighborhood. However, the task of maintaining a spanner - a sparse set of edges that approximates all pairs distances - can be achieved by ensuring the following somewhat local proposition for each non-spanner edge (x, y).

$\boxed{\mathcal{P}_t(x, y) : \text{the vertices } x \text{ and } y \text{ are connected in the subgraph } (V, E_S) \text{ by a path consisting of at most } t \text{ edges}}$

In order to maintain a t-spanner in dynamic scenario, it suffices to maintain $\mathcal{P}_t$ for each non-spanner edge. This will be achieved by a careful partitioning of vertices, called clustering [6].

Definition 1. *A* **cluster** *is a subset of vertices. A* **clustering** $\mathcal{C}$*, is a union of disjoint clusters. Each cluster will have a unique vertex which will be called its* **center***. A clustering can be represented by an array* $C[]$ *such that* $C[v]$ *for any* $v \in V$ *is the center of cluster to which* v *belongs, and* $C[v] = 0$ *if* v *is unclustered (does not belong to any cluster).*

The following notations will be used throughout the paper in the context of a given graph $G = (V, E)$, and $S, Y \subseteq V$.

- S_u : the set of vertices from S neighboring to u.
- $\delta(u, v)$: distance between u and v in the graph G.
- $\delta(u, Y)$: $\min\{\delta(u, v) \mid v \in Y\}$.

3 A decremental $O(\text{polylog}\, n)$ time algorithm

3.1 New clustering

Definition 2. *Given a permutation σ of some set $S \subseteq V$, and $i \in \mathbb{N}$, clustering $\mathcal{C}(\sigma, i)$ can be defined as follows.*
A vertex $u \in V$ with distance $\delta(u, S) \leq i$ is assigned to the cluster centered at the vertex in S nearest to u. In case of a tie, i.e., if there are multiple vertices at distance $\delta(u, S)$ from u, it is the nearest vertex that appears first in the permutation σ.

Note that the clustering $\mathcal{C}(\sigma, i)$ partitions only those vertices of the graph that are within distance i from S.
Efficient construction and maintenance : Given a permutation σ of some set $S \subseteq V$, and $i \in \mathbb{N}$, clustering $\mathcal{C}(\sigma, i)$ can be constructed in $O(m)$ time by algorithm described in Figure 1. A simple proof by induction on the distance from S shows that C stores the clustering $\mathcal{C}(\sigma, i)$. Moreover, the forest $\mathcal{F}$ spans each cluster by a tree rooted at its center such that for each vertex $v \in \mathcal{C}(\sigma, i)$, there is a path in $\mathcal{F}$ of length $\delta(v, S)$ connecting v to $C[v]$.

```
Let Q be a queue initialized to contain
elements of S in the order as defined
by σ.
Initially visited(v)=false  ∀v ∈ V,
C[s] = s ∀s ∈ S, and F ← ∅.
While not_empty(Q) do
{    x ← Dequeue(Q);
     For all (x, y) ∈ E do
          If visited(y)=false
          {    visited(y) ← true;
               C[y] ← C[x];
               F ← F ∪ {(x, y)};
               ℓ(y) ← ℓ(x) + 1;
               If ℓ(y) < i  Enqueue(y)}}
```

Fig. 1. Computing $\mathcal{C}(\sigma, i)$

The clustering $\mathcal{C}(\sigma, i)$ and the forest $\mathcal{F}$ can be associated with a breadth first search (BFS) tree in an augmented graph in the following way. If G' is a graph formed by adding a dummy vertex g and the edges $\{(g, s) | s \in S\}$ in G, then $\mathcal{F} \cup \{(g, s) | s \in S\}$ is a BFS tree rooted at g in G'. This BFS tree (not necessarily a unique one) satisfies the following condition. *Every vertex $v \in \mathcal{C}(\sigma, i)$ lies in the subtree rooted at $C[v]$.* Maintaining the clustering amounts to maintaining a BFS tree which also satisfies this condition at all times. An arbitrary BFS tree of depth i can be maintained in total $O(mi)$ cost over any sequence of edge deletions [12]. The algorithm involves finding new depth of each vertex $v \in V$ whose depth has increased, and then hooking it to any arbitrary neighbor from the level just above it. To maintain the clustering $\mathcal{C}(\sigma, i)$, we need to maintain a BFS tree wherein we hook a vertex v to its appropriate neighbor to satisfy the condition stated above. For this we need to maintain a search data structure for every vertex storing the centers of the clusters to which its neighbors belong. This will lead to $O(mi \log n)$ total update time over any sequence of edge deletions.

Lemma 1. *Given a graph $G = (V, E)$, an integer i, a permutation σ of a set $S \subseteq V$, we can maintain the clustering $\mathcal{C}(\sigma, i)$ and its spanning forest $\mathcal{F}$ with amortized $O(i \log n)$ time per edge deletion.*

The decremental algorithm employs a k-level hierarchy of clusterings : $\{\mathcal{C}(\sigma_i, i) | i \leq k\}$ whose defining sets S_i's and the permutations σ_i's are computed as follows.

1. Let $S_0 \leftarrow V$, $S_k = \emptyset$. For $0 < i < k$, let S_i contain each element of set S_{i-1} independently with probability $n^{-1/k}$.
2. For $0 \leq i < k$ let σ_i be a uniformly random permutation of set S_i.

3.2 Decremental Algorithm

Our decremental algorithm for $(2k-1)$-spanner will maintain the following structures and functions which can be initialized in $\tilde{O}(m)$ time easily.

1. **Clustering $\mathcal{C}(\sigma_i, i), i < k$:** Let C_i stores the clustering at level i, and $\mathcal{F}_i$ be its spanning forest. We maintain arrays C_i's and the set $\mathcal{F} = \cup_i \mathcal{F}_i$.
2. **Highest levels of vertices:** Let $H[v]$ be the highest level l such that v is present in $\mathcal{C}(\sigma_l, l)$). We maintain $H[v], \forall v \in V$.
3. **Assignment of edges to appropriate levels and endpoints:** Each edge $(u, v) \in E$ is kept at level $i = \max(H[u], H[v])$, and belongs to u if $H[u] \leq H[v]$ and to v otherwise. Let $E_i(u)$ denote the edges thus assigned to u at level i.
4. **Vertex-cluster Connectivity:** Let $E_i(u, o)$ be the set of edges from $E_i(u)$ which are incident from cluster centered at $o \in S_i$. Keep a set $\mathcal{E}$ which contains, for every $v \in V,\ i < k,\ o \in S_i$, one edge from $E_i(v, o)$.

Lemma 2. *The set $E_S = \mathcal{E} \cup \mathcal{F}$ is a $(2k-1)$-spanner of expected size $O(n^{1+1/k})$ for the graph at any time.*

Proof. We need to ensure that $\mathcal{P}_{2k-1}$ holds for every $(u, v) \notin E_S$. Let $(u, v) \in E_i(u, C_i[v])$ for some $i < k$. Vertex-cluster connectivity ensures that there must be an edge (u, w) satisfying $C_i[w] = C_i[v]$ which is present in $\mathcal{E}$. It follows from the clustering that v and w are connected in $\mathcal{F}_i$ by a path of length at most $2i$, so there is a path in E_S of length at most $2i + 1 < 2k - 1$ joining u and v. Hence $\mathcal{P}_{(2k-1)}(u, v)$ holds. For bounding the size, we bound the expected number of spanner edges contributed to $\mathcal{E}$ by any vertex v at any level $i < k$. Consider any clustering $\mathcal{C}(\sigma_i, i)$. With respect to any arbitrary set $E' \subseteq E$, let $c_1, ..., c_\ell$ be the clusters in this clustering which are neighboring to v. The vertex v will contribute at most one edge per neighboring cluster to $\mathcal{E}$, and a necessary (though not sufficient) condition for this contribution is that v is not present in any clustering of level $i + 1$ or higher. Given any clustering $\mathcal{C}(\sigma_i, i)$, the vertex v would appear in the clustering at next level only if the center of at least one of $c_1, ..., c_\ell$ is sampled. Therefore, it follows by elementary probability that the expected number of edges contributed by v at level i is at most $\ell \cdot (1 - n^{-1/k})^\ell + 1$, which is at most $n^{1/k}$ for any value of ℓ. It can be observed that this upper bound is derived for any set $E' \subseteq E$.

Data structures : In addition to the hash table storing all the edges of the graph and the usual adjacency lists for each vertex, we keep the following data structures. For each $v \in V$ and $i, 0 \le i < k$, let $o_1, ..., o_\ell$ be the centers of the cluster which are adjacent to v through edges $E_i(v)$ allocated to v. Keep a search tree for the set $\{o_1, ..., o_\ell\}$ and the node associated with $o_j, 1 \le j \le \ell$ in this tree would store a doubly linked list storing the edges $E_i(v, o_j) \subseteq E_i(v)$ incident on v from cluster centered at o_j. Furthermore, each edge in the set $E_i(v, o)$ will keep a pointer to and from the entry in the hash table storing all the edges of the graph. These data structures will help in efficient maintenance of the structures and function mentioned above.

Deletion of an edge may cause two kinds of changes in $\mathcal{C}_i, i < k$: some vertices change their clusters within $\mathcal{C}_i$ and/or some vertices cease to belong to the clustering $\mathcal{C}_i$ forever. The former change alters vertex-cluster connectivity as follows. Let a vertex v move from cluster c to join another cluster c'. Let $w \in V$ be a neighbor of v. As a result of this movement, it might be that c is no longer adjacent to w, and the cluster c', which earlier might be non adjacent to w, has become adjacent to w. Hence vertex-cluster connectivity needs to be updated. We describe below a subroutine for handling this case. When a vertex v ceases to belong to a clustering we will reassign all the edges present at level i which have u as one endpoint to their new levels and endpoints, and update the vertex-cluster connectivity accordingly.

Change-cluster(v, o, o')
(*when* v *moves from cluster centered at* o *to cluster centered at* o' *in* $\mathcal{C}(\sigma_i, i)$)
For each neighbor w of v in the graph with $(v, w) \in E_i(w, o)$ do

1. Delete (v, w) from $E_i(w, o)$. If (v, w) was in $\mathcal{E}$, choose some other edge from $E_i(w, o)$ in $\mathcal{E}$ (unless $E_i(w, o) = \emptyset$ now).
2. Insert edge (v, w) to $E_i(w, o')$, and choose it in $\mathcal{E}$ if $E_i(w, o')$ was empty earlier.

Decremental algorithm for $(2k-1)$-spanner

Deletion of an edge (u, v) is processed as follows. Let (u, v) be present at level i, and belong to $E_i(u, o)$ where $o = \mathcal{C}_i[v]$. If $(u, v) \in \mathcal{E}$ delete it from $\mathcal{E}$, and select some other edge from $E_i(u, o)$ in $\mathcal{E}$ (unless $E_i(u, o) = \emptyset$ now). The edge (u, v) could be in $\mathcal{F}$ (simultaneously as well). In this case, deletion might cause change of the clusterings at various levels, and we handle it as follows.
For $i = 1$ to $k - 1$ do

1. Update the clustering $\mathcal{C}(\sigma_i, i)$. Let $\Delta \subseteq V$ be the set of vertices that changed their clusters within $\mathcal{C}_i$, and $U \subseteq V$ be the set of vertices that ceased to be member of $\mathcal{C}_i$.
2. For each vertex $x \in \Delta$ do
   Let x moved from cluster centered at o to cluster centered at o' in $\mathcal{C}_i$.
   Change-cluster(x, o, o')
3. For each vertex $x \in U$, reassign all the edges present at level i which have x as one endpoint to their new levels and endpoints, and update the vertex-cluster connectivity accordingly.

Lemma 3. *At any level i, a vertex changes its cluster expected $O(i \log n)$ times.*

Proof. Consider a vertex $v \in V$. On leaving a cluster in $\mathcal{C}(\sigma_i, i)$ whenever v joins the same cluster again, its distance from S_i must have increased. We shall now estimate the number of times a vertex changes its cluster within $\mathcal{C}(\sigma_i, i)$ while keeping $\delta(v, S_i) = d$, for some fixed $d \leq i$. Consider the first time when $\delta(v, S_i)$ becomes d. Fix any order in which the edges are being deleted. Corresponding to this order, let $o_1, ...o_\ell$ be the sequence of all the vertices of S_i at distance d from v at present, and arranged in the chronological order of their cessation from being at distance d from v. The number of times v changes its cluster while keeping $\delta(v, S_i) = d$ is the same as the number of clusters in this sequence which v joins during the period for which $\delta(v, S_i) = d$. The vertex v will join cluster centered at o_j if and only if o_j appears first among $\{o_j, ..., o_\ell\}$ in the permutation σ_i. Since σ_i is a uniformly random permutation of S_i, the probability of this event is $1/(\ell - j + 1)$. Hence the expected number of cluster changes for the vertex v while remaining at a fixed distance d from S_i is $\sum_{j=1}^{\ell} \frac{1}{\ell-j+1} = O(\log n)$. Since the vertex v may change $\delta(v, S_i)$ at most i times before losing membership from the clustering $\mathcal{C}_i$, the lemma follows.

Analyzing the running time: When an edge is deleted from $\mathcal{E}$, it requires $O(\log n)$ cost to look for a replacement edge using the data structure amounting to a total of $O(m \log n)$ cost over any sequence of edge deletions. It follows from Lemma 1 that the total cost of maintaining clustering at any level over any sequence of edge deletions is $O(km \log n)$. The remaining cost incurred is for processing of the edges due to the changes in the clusterings and will be charged to the respective edges. An edge will be processed at most $2k$ times due to cessation of one of its endpoints from being member of clusterings. It follows from Lemma 3 that an edge will be processed expected $O(ki \log n)$ times due to change of clusters of its endpoints within any clustering since there are total k levels of clusterings. Using the data structures each processing of an edge costs $O(\log n)$ time. So the total expected cost charged to an edge throughout the algorithm is of the order of $\sum_{i<k} ik \log^2 n \leq k^2 \log^2 n = O(\text{polylog}\, n)$ since $k \leq \log n$. So we can conclude the following theorem.

Theorem 1. *Given a graph on n vertices undergoing edge deletions and $k \in \mathbb{N}$, we can maintain its $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size with expected $O(\text{polylog}\, n)$ time per edge deletion.*

Choosing $k = \log n$, we get the following corollary.

Corollary 1. *Given a graph on n vertices undergoing edge deletions, we can maintain all-pairs $O(\log n)$-approximate shortest paths with $O(\text{polylog}\, n)$ update time, $\tilde{O}(n)$ query time, and $O(m)$ space requirement.*

Roditty and Zwick [19] gave a decremental algorithm that maintains all-pairs $O(\log n)$-approximate shortest paths with $O(1)$-query time, $\tilde{O}(n)$ update time, and $O(m)$ space. For the scenario, where we want to minimize the update time at the expense of increased query time, our algorithm offers a better choice.

4 Fully dynamic algorithms

For a given graph (V, E), let $\mathcal{D}(V, E)$ be the data structure associated with our decremental algorithm for maintaining a $(2k-1)$-spanner. Our fully dynamic algorithm maintains $\mathcal{D}(V, E)$ and handles edge insertions in a lazy fashion by inserting the edge directly into the spanner and rebuilding the data structure $\mathcal{D}$ for the new graph periodically once there are $kn^{1+1/k}$ insertions. In this manner, cost of each insertion is $O(1)$. Using Theorem 1 the total update cost (including the initialization cost) for maintaining $\mathcal{D}$ is $O(m \operatorname{polylog} n)$ for each interval of rebuilding. So the fully dynamic algorithm achieves an amortized $\tilde{O}(m/n^{1+1/k})$ cost per edge insertion/deletion. We can thus conclude with the following theorem.

Theorem 2. *Given a graph on n vertices and $k \in \mathbb{N}$, we can maintain its $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size in fully dynamic environment with expected $\tilde{O}(\frac{m}{n^{1+1/k}})$ update time per edge insertion or deletion.*

5 A fully dynamic algorithm for small stretch spanners

We present a fully dynamic algorithm for 3-spanner. Our fully dynamic algorithm for 5-spanner is along similar lines, and we present a sketch of it. First we state a simple lemma whose proof follows from elementary probability.

Lemma 4. *Given a set A of ℓ elements, let S be a sample formed by selecting each element of set A independently with some probability. The following assertion holds :*

$$\forall a \in A, \qquad \mathbf{Pr}[a \in S \mid |S| = i] = \frac{i}{\ell}$$

The algorithm begins with the following preprocessing. A sample $S \subseteq V$ is formed by selecting each vertex independently with some probability p. During the whole algorithm, the set S serves as the set of the centers of clusters, and the clustering is essentially grouping each vertex $v \in V$ satisfying $S_v \neq \emptyset$ to some vertex from S_v. We now relabel the vertices of the graph so that the vertices of set S get labels which are a permutation of $1..S$. (This relabeling takes $O(m)$ time and is required as a minor technicality for sake of clarity of exposition of the algorithm).
The algorithm will maintain the following three invariants at each moment.

- $\mathcal{I}_1$: Each vertex $v \in V \backslash S$, with $S_v \neq \emptyset$, belongs to the the cluster centered at any of the vertex from S_v with equal probability.
- $\mathcal{I}_2$: Each vertex $v \in V \backslash S$ contributes all its edge to the spanner.
- $\mathcal{I}_3$: Each clustered vertex v has the edge $(v, C[v])$ and one edge to each of its neighboring clusters in the spanner.

The invariants $\mathcal{I}_2$ and $\mathcal{I}_3$ will ensure that the spanner has stretch 3 and expected size $O(n^{3/2})$, whereas the invariant $\mathcal{I}_1$ will play a key role in achieving expected constant time for handling any edge insertion/deletion.

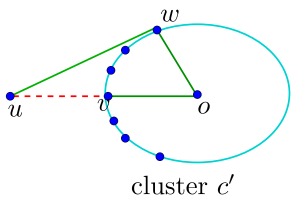

Fig. 2. For a non spanner edge, a stretch of 3.

Note that all the edges incident on an unclustered vertex are included in the spanner. So let us consider an edge (u, v) which is not a spanner edge. In this case both u and v must be clustered. It follows from invariant $\mathcal{I}_2$ that a clustered vertex is connected to the center of its cluster by a spanner edge. If both u and v belong to the same cluster, there is a path between them consisting of two edges, both from the spanner. So consider the case when u and v belong to different clusters, say c and c' respectively (see Figure 5). The existence of the edge (u, v) in E shows that the cluster c' is neighboring to u, so $\mathcal{I}_3$ ensures that there must be some edge $(u, w), w \in c'$ in the spanner. This implies a path of three spanner edges between u and v (see Figure 5). Hence the spanner is surely a 3-spanner.

Now let us analyze the expected size of this 3-spanner. An unclustered vertex will contribute all its edges, whereas a clustered vertex will contribute one edge per incident cluster. Hence the expected number of edges contributed by a vertex will be at most $\deg(v) \cdot (1-p)^{\deg(v)} + np$, which is at most $1/p + np$. Hence the expected size of the 3-spanner is $O(n/p + n^2 p)$, which for $p = 1/\sqrt{n}$, is $O(n^{3/2})$. Hence we can conclude that

Lemma 5. *Maintaining invariants $\mathcal{I}_2$ and $\mathcal{I}_3$ for a dynamic graph ensures that the spanner is a 3-spanner of expected size $O(n^{3/2})$ at each stage.*

5.1 Data structure

In order to efficiently maintain the invariants, we shall use the following data structures, which will require $O(m + n^{3/2})$ space, which is $O(m)$ since we have assumed that $m = \Omega(n^{3/2})$ (see Preliminaries).

- Let C be the array representing the clustering.
- Each vertex $v \in V$ keeps an array N_v of size $|S|$ such that $N_v[i]$ is (or points to) the head of a doubly linked list storing all those edges incident on v from a cluster centered at i. A node storing the edge (v, w) in this doubly linked list will also keep a pointer to (and from) the entry for the same edge in the hash table storing all the edges.
- Each vertex v maintains the set S_v of all the sampled vertices that are adjacent to it using a doubly linked list. A node storing w in this list will have a pointer to (and from) the node storing edge (v, w) in the list (pointed by) $N_v[C[w]]$. This will facilitate insertion/deletion of a vertex w from set S_v in $O(1)$ time whenever the corresponding edge (v, w) is inserted/deleted from the graph.

We shall employ the subroutine *Change-cluster* that we designed for our decremental algorithm. In addition, we shall use the following two subroutines.

Join-clustering(v, i) : (*an unclustered vertex v joins a cluster centered at i*)
$C[v] \leftarrow i$.
Process each clustered neighbor w of v as follows.
$j \leftarrow C[w]$.
  Insert edge (v, w) to the lists $N_w[i]$ and $N_v[j]$.
  Make (v, w) a non-spanner edge unless it is the only spanner edge present in either of $N_w[i]$ and $N_v[j]$.

Leave-clustering(v, i) : (*a vertex v clustered at i becomes unclustered*)
Process each clustered neighbor w of v as follows.
  $j \leftarrow C[w]$.
  Delete (v, w) from the lists $N_w[i]$ and $N_v[j]$, and make (v, w) a spanner edge.

Fully dynamic algorithm for 3-spanner

– **Deletion of an edge** (u, v) :
If the edge (u, v) does not belong to the current spanner, it suffices to delete the edge from the data structures of u as well as v. So let us consider the situation when (u, v) is a spanner edge. If either u or v is an unclustered vertex, it also suffices to just delete the edge. Otherwise let u and v belong to clusters centered at i and j respectively. We process the vertex u as follows (the vertex v is processed in a similar manner).
If $C[u] = v$
{ $S_u \leftarrow S_u \backslash \{v\}$;
  If $S_u = \emptyset$ { $C[v] \leftarrow 0$; *Leave-clustering*(u, v)}
  Else
  { let s be uniformly selected vertex from S_u;
    Make (u, s) a spanner edge;
    $C[u] \leftarrow s$; *Change-cluster*(u, i, s)}}
Else delete the edge (u, v) from $N_u[j]$, choose another edge from $N_u[j]$ (unless $N_u[j] = \emptyset$ now), and make that a spanner edge.

– **Insertion of an edge** (u, v) :
We process the vertex u as follows (the vertex v is processed similarly).
If u is unclustered
{ Make (u, v) a spanner edge;
  If $v \in S$ { $C[u] \leftarrow v$; *Join-clustering*(u, v)}}
Else
{ **If** v is clustered
  { $i \leftarrow C[u]$; $j \leftarrow C[v]$; Insert the edge (u, v) to $N_u[j]$;
    If ($i \neq j$ and $|N_u[j]| = 1$) make (u, v) a spanner edge;
    If $v \in S$
    { $S_u \leftarrow S_u \cup \{v\}$;
      With probability $1/|S_u|$ do
      { $C[u] \leftarrow v$; *Change-cluster*(u, i, j)}}}}.

It is easy to verify that the fully dynamic algorithm described above maintains the invariants $\mathcal{I}_1, \mathcal{I}_2$ and $\mathcal{I}_3$. The invariant $\mathcal{I}_1$ combined with Lemma 4 would imply the following crucial lemma whose proof follows by elementary probability.

Lemma 6. *For the graph G undergoing deletion and insertion of edges in any arbitrary order, our algorithm ensures the following equality at all times.*

$$\forall (u,v) \in E \qquad \mathbf{Pr}[C[u] = v \mid u \text{ is clustered}] \;=\; \frac{1}{\deg(u)}$$

Analyzing the complexity of the algorithm :
Consider deletion of an edge (u,v). It follows from the description of the algorithm that the processing of vertex u will take $O(1)$ time for all the cases except for the case when $C[u] = v$, in which case the update time is $O(\deg(u))$. Now applying Lemma 6 prior to deletion of edge (u,v), it follows that the probability of the latter case is $1/\deg(u)$. Hence the expected processing time for vertex u is $O(1)$ when an edge (u,v) is deleted. Similarly analyzing the vertex v, we conclude that an edge deletion can be processed in expected $O(1)$ time.

Now consider insertion of edge (u,v). It follows from the description of the algorithm that the update time is $O(1)$ for all the cases except when u gets assigned to cluster centered at v, in which case the update time is $O(\deg(u))$. Applying Lemma 6 just after the insertion (u,v), it follows that the probability of the latter case is $1/\deg(u)$. Hence the expected update time for maintaining a 3-spanner on inserting an edge (u,v) is constant. Combined with Lemma 5,

Theorem 3. *Given a graph on n vertices, we can maintain its 3-spanner of expected size $O(n^{3/2})$ with expected $O(1)$ time per edge insertion/deletion.*

5.2 Fully dynamic algorithm for stretch 5 (or 6)

The algorithm is the same as the algorithm for 3-spanner except with the following modifications.

1. The sampling probability is $p = 1/n^{1/3}$.
2. **The data structure :** The data structure is identical to that of 3-spanner except that each cluster (instead of each vertex) c keeps an array N_c such that $N_c[c']$ is a doubly linked list storing the edges incident on c from c'.
3. **The invariants :** The fully dynamic algorithm will maintain the three invariants : The first two are just identical to $\mathcal{I}_1$ and $\mathcal{I}_2$ for the 3-spanner, while the invariant $\mathcal{I}_3$ is defined as : Each clustered vertex v has the edge $(v, C[v])$ in the spanner and each cluster has one spanner edge to each of its neighboring clusters.

With the slight difference in the data structure and the invariant $\mathcal{I}_3$ as described above, our fully dynamic algorithm for 5-spanner is identical to our fully dynamic algorithm for 3-spanner which we have described and analyzed in complete details earlier. Hence, we can state the following theorem.

Theorem 4. *Given a graph on n vertices, we can maintain its 5-spanner of expected size $O(n^{4/3})$ with expected $O(1)$ time per edge insertion/deletion.*

Acknowledgement. The author is grateful to Sandeep Sen and anonymous referees for their useful comments.

References

1. I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.
2. G. Ausiello, P. G. Franciosa, and G. F. Italiano. Small stretch spanners on dynamic graphs. In *Proceedings of 13th Annual European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 532–543. Springer, 2005.
3. B. Awerbuch. Complexity of network synchronization. *Journal of Ass. Compt. Mach.*, pages 804–823, 1985.
4. B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhod covers. *SIAM Journal on Computing*, 28:263–277, 1998.
5. S. Baswana and S. Sen. A simple linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures and Algorithms (to appear)*.
6. S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 384–396, 2003.
7. S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $\tilde{O}(n^2)$ time. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280, 2004.
8. B. Bollobás. *Extremal Graph Theory*. Academic Press, 1978.
9. J. A. Bondy and M. Simonovits. Cycles of even length in graphs. *Journal of Combinatorial Theory, Series B*, 16:97–105, 1974.
10. E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.
11. P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.
12. S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of association for computing machinery*, 28:1–4, 1981.
13. S. Halperin and U. Zwick. Linear time deterministic algorithm for computing spanners for unweighted graphs. *unpublished manuscript*, 1996.
14. R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.
15. D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.
16. D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.
17. D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of Assoc. Comp. Mach.*, 36(3):510–530, 1989.
18. L. Roditty, M. Thorup, and U. Zwick. Deterministic construction of approximate distance oracles and spanners. In *Proceedings of 32nd International Colloquim on Automata, Languagaes and Programming (ICALP)*, volume 3580 of *LNCS*, pages 261–272. Springer, 2005.
19. L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proceedings of 12th Annual European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 580–591. Springer, 2004.
20. M. Thorup and U. Zwick. Approximate distance oracles. *Journal of Association of Computing Machinery*, 52:1–24, 2005.