Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners

Michael Elkin

[1] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel, elkinm@cs.bgu.ac.il
[2] This research has been supported by the Israeli Academy of Science, grant 483/06.

Abstract. We present a streaming algorithm for constructing sparse spanners and show that our algorithm out-performs significantly the state-of-the-art algorithm for this task [20]. Specifically, the *processing time-per-edge* of our algorithm is drastically smaller than that of the algorithm of [20], and all other efficiency parameters of our algorithm are no greater (and some of them are strictly smaller) than the respective parameters for the state-of-the-art algorithm.
We also devise a fully dynamic centralized algorithm maintaining sparse spanners. This algorithm has a very small incremental update time, and a non-trivial decremental update time. To our knowledge, this is the first fully dynamic centralized algorithm for maintaining sparse spanners that provides non-trivial bounds on both incremental and decremental update time for a wide range of stretch parameter t.

1 Introduction

The study of the streaming model became an important research area after the seminal papers of Alon, Matias and Szegedy [1], and Feigenbaum et al. [21] were published. More recently, research in the streaming model was extended to traditional graph problems [8, 19, 18, 20]. The input to a graph algorithm in the streaming model is a sequence (or *stream*) of edges representing the edge set E of the graph. This sequence can be an arbitrary permutation of the edge set E.

In this paper we devise a streaming algorithm for constructing sparse spanners for unweighted undirected graphs. Informally, graph *spanners* can be thought of as sparse skeletons of communication networks that approximate to a significant extent the metric properties of the respective networks. Spanners serve as an underlying graph-theoretic construct for a great variety of distributed algorithms. Their most prominent applications include *approximate distance computation* [15, 18, 14], *synchronization* [4, 22, 6], *routing* [23, 7, 26], and online load balancing [5]. The problem of constructing spanners with various parameters is a subject of intensive recent research [17, 15, 25, 12, 11, 27, 24, 28].

The state-of-the-art streaming algorithm for computing a sparse spanner for an input (unweighted undirected) n-vertex graph $G = (V, E)$ was presented in a recent breakthrough paper of Feigenbaum et al. [20]. For an integer parameter

$t \geq 2$, their algorithm, with high probability, constructs a $(2t-1)$-spanner with $O(t \cdot \log n \cdot n^{1+1/(t-1)})$ edges *in one pass* over the input using $O(t \cdot \log^2 n \cdot n^{1+1/(t-1)})$ bits of space. It processes each edge in the stream in time $O(t^2 \cdot \log n \cdot n^{1/(t-1)})$. Their result also immediately gives rise to a streaming algorithm for $(2t-1)$-approximate all-pairs-distance-computation (henceforth, $(2t-1)$-*APDC*) algorithm with the same parameters.

Our algorithm constructs a $(2t-1)$-spanner with $O((t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$ edges in one pass over the input using $O(t^{1-1/t} \cdot \log^{2-1/t} n \cdot n^{1+1/t})$ bits of space, for an integer parameter $t \geq 1$. (The size of the spanner and the number of bits are with high probability.) Most importantly, the *processing time-per-edge* of our algorithm is *drastically smaller* than that of Feigenbaum et al. [20]. Specifically, the expectation of the processing time-per-edge in our algorithm is $O(1)$, it is $O\left(\sqrt{\frac{\log \log n}{\log^{(3)} n}}\right)$ with high probability, and in the worst-case processing an edge $e = (v, u)$ requires $O\left(\sqrt{\frac{\log deg(e)}{\log \log deg(e)}}\right)$, where $deg(e) = \max\{deg(v), deg(u)\}$.

To summarize, our algorithm constructs a spanner with a smaller number of edges and number of bits of space used (by a factor of $n^{1/(t(t-1))}(t \cdot \log n)^{1/t}$), and it does so using a *drastically* reduced (and very close to optimal) *processing time-per-edge*, *at no price whatsoever*. Our result also gives rise to an improved streaming $(2t-1)$-approximate APDC algorithm with the same parameters. Observe also that for the stretch guarantee equal to 3, our algorithm is the first one to provide a non-trivial bound. A concise comparison of our result with the state-of-the-art result of Feigenbaum et al. [20] can be found in Table 1.

Independently of us Baswana [9] came up with a streaming algorithm for computing sparse spanners. The efficiency parameters of the algorithm of [9] are equal to those of our algorithm, except that it provides *amortized* bound of $O(1)$ on the processing time-per-edge. However, processing an individual edge by this algorithm may require $\Omega(n)$ time in the worst-case, and [9] provides no guarantee on the expectation of the processing time-per-edge. Also, the algorithm of [9] appears to be significantly more complex than ours.

A variant of our algorithm can be seen as a fully dynamic centralized algorithm for maintaining a $(2t-1)$-spanner with $O((t \cdot \log n)^{1-1/t} n^{1+1/t})$ edges. The incremental update time of this algorithm is exactly the processing time-per-edge of our streaming algorithm, and, in fact, the way that the dynamic algorithm processes incremental updates is identical to the way that our streaming algorithm processes each edge of the stream. The expected decremental update time (the time required to update the data structures of the algorithm when an edge e is deleted) is $O(\frac{m}{n^{1/t}} \cdot (t \log n)^{1/t})$, and moreover, with probability at least $1 - \left(\frac{t \cdot \log n}{n}\right)^{1/t}$, the decremental update time is $O\left(\sqrt{\frac{h}{\log h}}\right)$, where $h = \max\{\log deg(e), \log \log n\}$. The size of the data structures maintained by an incremental variant of our algorithm is $O(t^{1-1/t} \cdot (\log n)^{2-1/t} \cdot n^{1+1/t})$ bits. To cope with decremental updates as well, our algorithm needs to maintain $O(|E| \cdot \log n)$ bits. (Note that the incremental algorithm maintains data structures of overall size *sublinear* in the size of the input $|E|$. This is not really

	# passes	Processing time-per-edge	Stretch	# Edges (whp)
[20]	1	whp $O(t^2 \cdot \log n \cdot n^{1/(t-1)})$	$2t-1$	$O(t \cdot \log n \cdot n^{1+1/(t-1)})$
New	1	Expected $O(1)$, whp $O\left(\sqrt{\frac{\log\log n}{\log^{(3)} n}}\right)$, worst-case $O\left(\sqrt{\frac{\log deg(e)}{\log\log deg(e)}}\right)$	$2t-1$	$O((t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$

Table 1. A comparison between the algorithm of Feigenbaum et al. [20] and our new streaming algorithm. The degree of an edge $e = (u, v)$ is $\max\{deg(v), deg(u)\}$. The word "whp" stands for "with high probability". The result of [20] applies for $t \geq 2$, while our algorithm applies for $t \geq 1$.

surprising, since this is essentially a streaming algorithm.)

To our knowledge, this is the first fully dynamic centralized algorithm for maintaining sparse spanners that provides non-trivial bounds for a wide range of stretch parameter t. The first algorithm of this type was devised recently by Ausillo et al. [3]. This algorithm maintains 3- and 5-spanners of optimal size with $O(n)$ amortized time per operation for an intermixed sequence of $\Omega(n)$ edge insertions and deletions.

Very recently Baswana [10] improved the result of Ausillo et al. [3] and devised a fully dynamic algorithm for maintaining spanners of optimal size with stretch at most 6 with expected constant update time. Baswana [10] presented also a *decremental* algorithm for maintaining $(2t-1)$-spanner of optimal size with expected update time of $O(t^2 \cdot \log^2 n)$. However, the latter algorithm provides no non-trivial bound for *incremental* update time. Also, note that with high probability the decremental update time of our algorithm is significantly smaller than that of [10].

Related Work and Our Techniques: A fully dynamic distributed algorithm for maintaining sparse spanners is presented in the companion paper [16]. Our algorithm in this paper combines the techniques of Feigenbaum et al. [20], of Baswana and Sen [12], with those developed in [16].

More specifically, both the algorithm of Feigenbaum et al. [20] and our algorithm build upon the techniques of Baswana and Sen [12]. Both algorithms label vertices by numbers, and use the labels to decide whether a newcoming edge needs to be inserted into the spanner or not. The main conceptual difference between the two algorithms is that in the algorithm of Feigenbaum et al. [20] for every vertex v, the entire list of labels L such that v was ever labeled by L is stored. These lists are then manipulated in a rather sophisticated manner to ensure that only the "right" edges end up in the spanner. On the other hand, in our algorithm only one (current) label is stored for every vertex, and decisions are made on the basis of this far more restricted information. As a result, our algorithm avoids manipulating lists of labels, and is, consequently, much simpler, and far more efficient.

One could expect that using a smaller amount of information to make decisions may result in a denser spanner, or/and in a spanner with a relaxed stretch guarantee. Surprisingly, however, we show that this is not the case, and that the parameters of spanners produced by our algorithm are better than the parameters of spanners produced by the algorithm of Feigenbaum et al. [20].

Preliminaries: For a parameter α, $\alpha \geq 1$, a subgraph G' of the graph $G = (V, E)$ is called an *α-spanner* of G if for every pair of vertices $x, y \in V$, $dist_{G'}(x, y) \leq \alpha \cdot dist_G(x, y)$, where $dist_G(u, w)$ denotes the distance between u and w in G. The parameter α is called the *stretch* or *distortion* parameter of the spanner. Also, for a fixed value of $t = 1, 2, \ldots$, we say that a subgraph G' *spans* an edge $e = (v, u) \in E$, if $dist_{G'}(v, u) \leq 2t - 1$.

Structure of this paper: In Section 2 we present and analyze our streaming algorithm. In Section 3 we extend it to the dynamic centralized setting. Most proofs are omitted from this extended abstract.

2 The Streaming Model

In this section we present and analyze the version of our algorithm that constructs spanners in the streaming model of computation.

2.1 The Algorithm

The algorithm accepts as input a *stream* of edges of the input graph $G = (V, E)$, and an integer positive parameter t, and constructs a $(2t-1)$-spanner $G' = (V, H)$, $H \subseteq E$, of G with $O((t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$ edges using only $O(|H| \cdot \log n) = O(t^{1-1/t} (\log n)^{2-1/t} \cdot n^{1+1/t})$ bits of storage space, and processing each edge in $O(1)$ expected time, in one pass over the stream. Note that the space used by the algorithm is linear in the size of the representation of the spanner. Regarding the processing time-per-edge, processing the edge e requires $O\left(\sqrt{\frac{\log deg(e)}{\log\log deg(e)}}\right)$ time in the worst-case, and moreover, with high probability, the processing time-per-edge is $O\left(\sqrt{\frac{\log\log n}{\log^{(3)} n}}\right)$.

At the beginning of the execution (before the first edge of the stream arrives), the vertices of V are assigned unique identifiers from the set $\{1, 2, \ldots, n\} = [n]$, $n = |V|$. (Henceforth, for any positive integer k, the set $\{1, 2, \ldots, k\}$ is denoted $[k]$, and the set $\{0, 1, \ldots, k\}$ is denoted $[(k)]$.) Let $I(v)$ denote the identifier of the vertex v. Also, as a part of preprocessing, the algorithm picks a non-negative integer *radius* $r(v)$ for every vertex v of the graph from the *truncated geometric probability distribution* given by $\mathbb{P}(r = k) = p^k \cdot (1-p)$, for every $k \in [(t-2)]$, and

$\mathbb{P}(r = t-1) = p^{t-1}$, with $p = \left(\frac{t \log n}{n}\right)^{1/t}$. Note that this distribution satisfies $\mathbb{P}(r \geq k+1 \mid r \geq k) = p$ for every $k \in [(t-2)]$.

We next introduce a few definitions that will be useful for the description of our algorithm. During the execution, the algorithm maintains for every vertex v the variable $P(v)$, called the *label* of v, initialized as $I(v)$. The labels of vertices may grow as the execution proceeds, and they accept values from the set $\{1, 2, \ldots, n \cdot t\}$. A label P in the range $i \cdot n + 1 \leq P < (i+1)n$, for $i \in [(t-1)]$ is said to be a label of *level* i; in this case we write $L(P) = i$. The value $B(P)$ is given by $B(P) = n$ if n divides $P(v)$, and by $B(P) = P(v) \pmod{n}$, otherwise. This value is called the *base value* of the label P. The vertex $w = w_P$ such that $I(w) = B(P)$ is called the *base vertex* of the label P. A label P is said to exist if the level $L(P)$ of P is no greater than the radius of the base vertex w_P, i.e., $L(P) \leq r(w_P)$. The label P is called *selected* if $L(P) < r(w_P)$. Note that for a label P to be selected, it must satisfy $L(P) \leq t-2$.

One of the basic primitives of the algorithm is comparing the labels. We say that the labels $P(v)$ and $P(v')$ of the vertices v and v', respectively, satisfy the relation $P(v) \succ P(v')$ if and only if either $P(v) > P(v')$ or ($P(v) = P(v')$ and $I(v) > I(v')$). Note that for every two vertices v and v', either $P(v) \succ P(v')$ or $P(v') \succ P(v)$.

For a label P of level $t-2$ or smaller,

$$\mathbb{P}(P \text{ is selected}) = \mathbb{P}(r(w_P)) \geq L(P) + 1 \mid r(w_P) \geq L(P)) = p\ .$$

Lemma 1. *With high probability, the number of distinct labels of level* $t-1$ *that occur in the algorithm is* $O(n^{1/t} \cdot (t \cdot \log n)^{1-1/t})$.

We remark that the way that we define and manipulate labels is closely related to the way it is done in Feigenbaum et al. [20].

For every vertex the algorithm maintains an edge set $Sp(v)$, initialized as an empty set. During the execution the algorithm inserts some edges into $Sp(v)$, and never removes them. In other words, the sets $Sp(v)$ grow monotonely during the execution of the algorithm. It is useful to think of the sets $Sp(v)$ as divided into two disjoint subsets $T(v)$ and $X(v)$, $Sp(v) = T(v) \cup X(v)$. The set $T(v)$ is called the set of the *tree edges* of the vertex v, and the set $X(v)$ is called the set of the *cross edges* of the vertex v. During the execution the algorithm constructs implicitly a *tree cover* of the graph. The edges of this tree cover are (implicitly) maintained in the sets $T(v)$. In addition, the spanner will also contain some edges that connect different trees of the tree cover; these edges are (implicitly) maintained in the sets $X(v)$. Each edge e that is (implicitly) inserted into the set $T(v)$ will also be labeled by a label of v at the time of the insertion. An insertion of an edge $e = (v, u)$ into the set $T(v)$ will cause v to change its label to the label of u plus n, that is $P(u) + n$. The edge e will also be labeled by this label.

In addition, for every vertex v a table $M(v)$ is maintained. These tables are initially empty. Each table $M(v)$ is used to store all the base values of levels P such that there exists at least one neighbor z of v that was labeled by P at some point of the execution of the algorithm, and such that the edge (v, z) was inserted into the set $X(v)$ at that point of the execution.

The algorithm itself is very simple. It iteratively invokes the Procedure *Read_Edge* on every edge of the stream, until the stream is exhausted. At this point it outputs the set $\bigcup_v Sp(v) = \bigcup_v T(v) \cup \bigcup_v X(v)$ as the resulting spanner. The Procedure *Read_Edge* accepts as input an edge $e = (u, v)$ that it is supposed to "read". The procedure finds the endpoint x of the edge e that has a greater label $P(x)$ (with respect to the order relation $\succ$). Suppose without loss of generality that $x = u$, i.e., $P(u) \succ P(v)$. Then the procedure tests whether $P(u)$ is a selected label. If it is, the edge e is inserted into the set of tree edges $T(v)$ of v, and v adapts the label $P(u) + n$. If $P(u)$ is not a selected label, then the procedure tests whether the base value $B(P(u))$ of the label $P(u)$ is stored in the table $M(v)$. If it is not, then the edge e is inserted into the set $X(v)$ of the cross edges of v, and the label of v does not change. If $P(u)$ is already stored in $M(v)$ then nothing needs to be done.

The pseudo-code of the Procedure *Read_Edge* is provided below. Its main difference from the description above is that the sets $X(v)$ and $T(v)$ are not maintained explicitly, but rather instead there is just one set $Sp(v)$ maintained. The reason for this difference is that we aim to present the simplest version of the algorithm for which we can prove the desired bounds. However, it is more convenient to reason about the sets $X(v)$ and $T(v)$ explicitly, rather than about the set $Sp(v)$ as a whole, and thus in the analysis we will analyze the version of the algorithm that maintains the sets $T(v)$ and $X(v)$ explicitly. (It is obvious that the two versions are equivalent.)

Algorithm 1: The streaming algorithm for constructing a sparse $(2t-1)$-spanner, and Procedure *Read_Edge*$(e = (u, v))$.

1. For all the edges e of the input stream
   invoke *Read_Edge*(e)
2. Let u be the vertex s.t. $P(u) \succ P(v)$
3. If ($P(u)$ is a selected label) then
      $P(v) \leftarrow P(u) + n$
      $Sp(v) \leftarrow Sp(v) \cup \{e\}$
   else if ($B(P(u)) \notin M(v)$) then
         $M(v) \leftarrow M(v) \cup \{B(P(u))\}$
         $Sp(v) \leftarrow Sp(v) \cup \{e\}$
   end-if

The set $T(v)$ (resp., $X(v)$) is the set of edges inserted into the set $Sp(v)$ on line 4 (resp., 7) of the Procedure *Read_Edge*. We will say that an edge e *is inserted into* $T(v)$ *(resp.,* $X(v)$*)* if it is inserted into $Sp(v)$ on line 4 (resp., 7) of the algorithm.

Note that the Procedure *Read_Edge* is extremely simple, and the only operations that might require a super-constant time are lines 5 and 6, which require testing a membership of an element in a data structure, and an insertion of an element into the data structure if it is not there already. These operations can be implemented very efficiently in a general scenario via a balanced search tree, or a hash table. Moreover, we will also show later that with high probability, the size of each table is quite small, specifically $\tilde{O}(n^{1/t})$, and thus, in our setting these operations can be implemented even more efficiently.

2.2 The Size of the Spanner

We start with showing that the resulting spanner is sparse. For this end we show that both sets $\bigcup_{v \in V} T(v)$ and $\bigcup_{v \in V} X(v)$ are sparse.

Lemma 2. *For every vertex $v \in V$, $|T(v)| \leq t-1$.*

Proof. Each time an edge $e = (v, u)$ is inserted into $T(v)$, the label of v grows from $P(v)$ to $P(u) + n$. Moreover, note that $P(u) \geq P(v)$ for such an edge. Consequently, the level of $P(v)$ grows at least by 1. Hence at any given time of an execution of the algorithm, $L(P(v))$ is an upper bound on the number of edges currently stored in $T(v)$. Since $L(P(v))$ never grows beyond $t-1$, it follows that $|T(v)| \leq t-1$. ∎

Consequently, the set $\bigcup_v T(v)$ contains at most $n \cdot (t-1)$ edges, i.e.,

$$|\bigcup_{v \in V} T(v)| \leq n \cdot (t-1) \ . \tag{1}$$

We next argue that the set $\bigcup_{v \in V} X(v)$ is sparse as well. First, by Lemma 1, the number of distinct labels of level $t-1$ that occur during the algorithm is, with high probability, $O(n^{1/t} \cdot (t \cdot \log n)^{1-1/t})$. Fix a vertex $v \in V$. Since, by line 5 of Algorithm 1 for each such a label P at most one edge (u, v) with $P(u) = P \succ P(v)$ is inserted into $X(v)$, it follows that the number of edges (u, v) with $P(u) \succ P(v)$, $L(P(u)) = t-1$, inserted into $X(v)$, is, with high probability, at most $O(n^{1/t} \cdot (t \cdot \log n)^{1-1/t})$.

For an index $i \in [(t-1)]$, let $X^{(i)}(v)$ denote the set of edges (u, v), with $L(P(u)) < t-1$, inserted into $X(v)$ during the period of time that $L(P(v))$ was equal to i.

Lemma 3. $X^{(t-1)}(v) = \emptyset$.

Cardinalities of the sets $X^{(i)}(v)$, $0 \leq i \leq t-2$, are small as well. (Though these sets may be not empty.)

Lemma 4. *For every input sequence of edges $(e_1, e_2, \ldots, e_m)$ determined obliviously of the coin tosses of the algorithm, for every vertex v, and index $i \in [(t-2)]$, with high probability, $|X^{(i)}(v)| = O\left(n^{1/t} \cdot \frac{\log^{1-1/t} n}{t^{1/t}}\right)$.*

We are now ready to state the desired upper bound on $|\bigcup_{v \in V} X(v)|$.

Corollary 5. *Under the assumption of Lemma 4, for every vertex $v \in V$, with high probability, the overall number of edges inserted into $X(v)$ is $O(n^{1/t} \cdot (t \cdot \log n)^{1-1/t})$.*

We summarize the size analysis of the spanner constructed by Algorithm 1 with the following corollary.

Corollary 6. *Under the assumptions of Lemma 4, with high probability, the spanner H constructed by the algorithm contains $O(n^{1+1/t} \cdot (t \cdot \log n)^{1-1/t})$ edges. Moreover, each table $M(v)$, $v \in V$, stores, with high probability, at most $O(n^{1/t} \cdot (t \cdot \log n)^{1-1/t})$ values, and consequently, overall the algorithm uses $O(|H| \cdot \log n) = O(n^{1+1/t} \cdot t^{1-1/t} \cdot (\log n)^{2-1/t})$ bits of space.*

Proof. The resulting spanner is $\left(\bigcup_{v\in V} T(v) \cup \bigcup_{v\in V} X(v)\right)$. By the inequality (1), $|\bigcup_{v\in V} T(v)| \le n\cdot(t-1)$. By Corollary 5, with high probability, $|\bigcup_{v\in V} X(v)| = O(n^{1+1/t} \cdot (t \cdot \log n)^{1-1/t})$, and so the first assertion of the corollary follows.

For the second assertion recall that a new value is added to $M(v)$ only when a new edge (u, v) is introduced into the set $X(v)$. By Corollary 5, with high probability, $|X(v)| = O(n^{1/t} \cdot (t \cdot \log n)^{1-1/t})$, and therefore the same bound applies for $|M(v)|$ as well.

To calculate the overall size of the data structures used by the algorithms we note that $|\bigcup_{v\in V} M(v)| \le |\bigcup_{v\in V} X(v)| \le |\bigcup_{v\in V} X(v)| + |\bigcup_{v\in V} T(v)| = O(|H|)$. Since each label and edge requires $O(\log n)$ bits to represent, the desired upper bound on the size of the data structures follows. ∎

2.3 The Stretch Guarantee of the Spanner

We next show that the subgraph constructed by the algorithm is a $(2t-1)$-spanner of the original graph G.

For an integer $k \ge 1$, and a vertex $v \in V$, let $P_k(v)$ denote the label of v, $P(v)$, before reading the kth edge of the input stream.

Lemma 7. *Let $v, v' \in V$ be a pair of vertices such that there exist positive integers $k, k' \ge 1$ such that $B(P_k(v)) = B(P_{k'}(v'))$. Then there exists a path of length at most $L(P_k(v)) + L(P_{k'}(v'))$ between v and v' in the (final) set $\bigcup_{v\in V} T(v)$.*

The next lemma shows that the edge set $H = \bigcup_{v\in V} T(v) \cup \bigcup_{v\in V} X(v)$ is a $(2t-1)$-spanner.

Lemma 8. *Let $e = (v, v') \in E$ be an edge. Then there exists a path of length at most $2t-1$ between v and v' in the edge set H.*

2.4 The Processing Time-per-edge

To conclude the analysis of our streaming algorithm for constructing sparse spanners, we show that it has a very small processing time-per-edge. For this purpose we now fill in a few implementation details that have so far been unspecified. Specifically, on lines 5 and 6 of the Procedure *Read_Edge* the algorithm tests whether an element B belongs to a set $M(v)$, and if it does not, the algorithm inserts it there. The set $M(v)$ is a subset of the universe $[n]$, and by Corollary 6, its size is, with high probability, $O((t \cdot \log n)^{1-1/t} \cdot n^{1/t})$. Moreover, since $|M(v)| \leq |X(v)|$, it follows that $|M(v)| \leq deg(v)$.

Let $N = c \cdot (t \cdot \log n)^{1-1/t} \cdot n^{1/t}$, for a sufficiently large constant c. (The probability that $|M(v)| \leq c \cdot (t \cdot \log n)^{1-1/t} \cdot n^{1/t}$ for every vertex $v \in V$ is at least $1 - \frac{1}{n^{c-2}}$. Hence choosing $c = 4$ is sufficient.) As a part of preprocessing the algorithm computes a random hash function $h : [n] \rightarrow [N]$. Specifically, for each number $i \in [n]$, the algorithm picks a value $j \in [N]$ uniformly at random, and sets $h(i) = j$. The table representation of this hash function is written down, and is used throughout the execution of the algorithm for the tables $M(v)$ for all the vertices $v \in V$. This representation requires $O(n \cdot \log n)$ space, and can be computed in $O(n)$ time during the preprocessing.

For every vertex v the algorithm maintains a hash table $M(v)$ of size N. Every base value B for which the algorithm needs to test its membership in $M(v)$ on line 5 of the Procedure *Read_Edge* is hashed to $h(B)$ using the hash function h. To resolve collisions, for each entry of the hash table $M(v)$ we use a dynamic dictionary data structure of Beame and Fich [13] (with the dynamization result of Andersson and Thorup [2]). This data structure maintains a dynamic set of q keys from an arbitrary universe using $O\left(\sqrt{\frac{\log q}{\log\log q}}\right)$ time per update (insertion or deletion) and membership queries. This completes the description of the implementation details of the algorithm.

Note that the preprocessing of the algorithm requires $O(n)$ time. In the full version of this paper we show that implemented this way, the algorithm enjoys an extremely low processing time-per-edge.

The properties of our streaming algorithm are summarized in the following theorem.

Let $\lambda(n)$ (respectively, $\sigma(n)$) denote the function $\sqrt{\frac{\log\log n}{\log^{(3)} n}}$ (resp., $\sqrt{\frac{\log n}{\log\log n}}$).

Theorem 9. *Let n, t, $n \geq t \geq 1$, be positive integers. Consider an execution of our algorithm in the streaming model on an input (unweighted undirected) n-vertex graph $G = (V, E)$ such that both the graph and the ordering ρ of its edges are chosen by a non-adaptive adversary obliviously of the coin tosses of the algorithm. The algorithm constructs a $(2t-1)$-spanner H of the input graph. The expected size of the spanner is* $O(t \cdot n^{1+1/t})$ or *the size of the spanner is* $O((t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$ *with high probability (depending on the choice of p; in the first case the guarantee on the size that holds with high probability is $O(t \cdot \log n \cdot n^{1+1/t})$). The algorithm does so in one pass over the input stream, and requires $O(1)$ expected processing time-per-edge, $O(\lambda(n))$ processing time-per-edge with high probability, and $O(\sigma(deg(e)))$ processing time-per-edge in the worst-case, for an edge $e = (v, u)$. The space used by the algorithm is $O(|H| \cdot \log n) = O(t^{1-1/t} \cdot \log^{2-1/t} n \cdot n^{1/t})$ bits with high probability. The preprocessing of the algorithm requires $O(n)$ time.*

Our algorithm can be easily adapted to construct a $(2t-1)(1+\epsilon)$-spanners for weighted graphs, for an arbitrary $\epsilon > 0$. The size of the obtained spanner becomes $O(\log_{(1+\epsilon)} \hat{\omega} \cdot (t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$, where $\hat{\omega}$ is the aspect ratio of the network. The algorithm still works in one pass, and has the same processing time-per-edge. This adaptation is achieved in a standard way (see, e.g., [20]), by constructing $\log_{(1+\epsilon)} \hat{\omega}$ different spanners in parallel. For completeness, we next overview this adaptation.

The edge weights can be scaled so that they are all greater or equal to 1 and smaller or equal to $\hat{\omega}$. All edges are partitioned logically into $\lceil \log_{(1+\epsilon)} \hat{\omega} \rceil$ categories, indexed $i = 1, 2, \ldots, \lceil \log_{(1+\epsilon)} \hat{\omega} \rceil$, according to their weights, with the category i containing the edges with weights greater of equal to $(1+\epsilon)^{i-1}$ and smaller than $(1+\epsilon)^i$. When an edge $e = (u, v)$ is read, it is processed according to its category, and it is either inserted into the spanner for the edges of category i, or discarded.

Obviously, after reading all the edges, we end up with $\lceil \log_{(1+\epsilon)} \hat{\omega} \rceil$ subgraphs, with the ith subgraph being a $(2t-1)(1+\epsilon)$-spanner for the edges of the category i. Consequently, the union of all these edges is a $(2t-1)(1+\epsilon)$-spanner for the entire graph. The cardinality of this union is at most $\lceil \log_{(1+\epsilon)} \hat{\omega} \rceil$ times the maximum cardinality of one of these subgraphs, which is, in turn, at most $O((t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$ with high probability.

3 A Centralized Dynamic Algorithm

Our streaming algorithm can be seen as an incremental dynamic algorithm for maintaining a $(2t-1)$-spanner of size $O((t \cdot \log n)^{1-1/t} \cdot n^{1+1/t})$ for unweighted graphs, where n is an upper bound on the number of vertices that are allowed to appear in the graph.

The initialization of the algorithm is as follows. Given a graph $G = (V, E)$, we run our streaming algorithm with the edge set E, where the order in which the edge set is read is arbitrary. As a result, the spanner, and the satellite data structures $\{M(v), Sp(v) \mid v \in V\}$ are constructed. This requires $O(|E|)$ expected time, $O(|E| \cdot \lambda(n))$ time with high probability, and $O(|E| \cdot \sigma(\Delta))$ in the worst-case, where Δ is the maximum degree of a vertex in G.

Each edge that is added to the graph is processed using our streaming algorithm. The spanner and the satellite data structures are updated in expected $O(1)$ time-per-edge, $O(\lambda(n))$ time-per-edge with high probability, and $O(\sigma(\Delta))$ time in the worst-case.

We next make the algorithm robust to decremental updates (henceforth, *crashes*) as well. Note that for an edge $e = (v, u)$ to become a T-edge (an edge of $\bigcup_{x \in V} T(x)$), it must hold that at the time that the algorithm reads the edge, the greater of the two labels $P(u)$ and $P(v)$ (with respect to the order relation $\succ$) is selected. The probability of a label to be selected is at most $p = \left(\frac{t \cdot \log n}{n}\right)^{1/t}$ if its level is smaller than $t-1$, and is 0 otherwise. Hence the probability of e to become a T-edge is at most p.

In the companion paper [16] it is shown that a crash of an edge $e = (v, u)$ that does not belong to $\bigcup_{x \in V} T(x)$ can be processed in expected time $O(1)$. Moreover, with high probability the processing of such a crash requires $\sigma(h)$ time, where $h = \max\{deg(e), \log n\}$. Since the entire spanner can be recomputed in expected time $O(|E|)$ by our algorithm, it follows that the expected decremental update time of our algorithm is $O(\frac{|E|}{n^{1/t}} \cdot (t \cdot \log n)^{1/t})$. The size of the data structure maintained by the incremental variant of the algorithm is $O(t^{1-1/t} \cdot (\log n)^{2-1/t} \cdot n^{1+1/t})$ bits, and the fully dynamic algorithm maintains a data structure of size $O(|E| \cdot \log n)$.

We summarize this discussion with a following corollary.

Corollary 10. *For positive integer n, t, $n \geq t \geq 1$, the algorithm is a fully dynamic algorithm for maintaining $(2t-1)$-spanners with expected $O(t \cdot n^{1+1/t})$ number of edges (or $O((t \log n)^{1-1/t} \cdot n^{1+1/t})$ edges with high probability, depending on the choice of p) for graphs with at most n vertices. If $G = (V, E)$ is the initial graph, then the initialization of the algorithm requires $O(|E|)$ expected time, $O(|E| \cdot \lambda(n))$ time with high probability, and $O(|E| \cdot \sigma(\Delta))$ in the worst-case. The expected incremental update time of the algorithm is $O(1)$, with high probability it is $O(\lambda(n))$, and in the worst-case it is $O(\sigma(deg(e)))$ (for an edge e that joins the graph). The expected decremental update time is $O(\frac{|E|}{n^{1/t}} \cdot (t \cdot \log n)^{1/t})$, and with probability at least $1 - \left(\frac{t \cdot \log n}{n}\right)^{1/t}$ the decremental update time is $O(\sigma(h))$, where $h = \max\{deg(e), \log n\}$.*

To our knowledge, this is the first fully dynamic algorithm for maintaining sparse spanners for a wide range of values of the stretch parameter t with non-trivial guarantees on both the incremental and decremental update times.

References

1. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.
2. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. of the 32nd Annual ACM Symp. on Theory of Computing*, pages 335–342, 2000.
3. G. Ausillo, P. G. Franciosa, and G. F. Italiano. Small stretch spanners on dynamic graphs. In *Proc. of the 13th European Symp. on Algorithms (ESA)*, pages 532–543, 2005.
4. B. Awerbuch. Complexity of network synchronization. *J. ACM*, 4:804–823, 1985.
5. B. Awerbuch, S. Kutten, and D. Peleg. Online load balancing in a distributed network. In *Proc. 24th ACM Symp. on Theory of Comput.*, pages 571–580, 1992.
6. B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.
7. B. Awerbuch and D. Peleg. Routing with polynomial communication-space tradeoff. *SIAM J. Discrete Mathematics*, 5:151–162, 1992.
8. Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an applications to counting triangles in graphs. In *Proc. 13th ACM-SIAM Symp. on Discr. Algor.*, pages 623–632, 2002.
9. S. Baswana. personal communication, 2006.

10. S. Baswana. Dynamic algorithms for graph spanners. In *Proc. of the European Symp. of Algorithms, ESA'06 (to appear)*, 2006.
11. S. Baswana, T. Kavitha, K. Mehlhorn, and S. Pettie. New constructions of (a,b)-spanners and additive spanners. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 672–681, 2005.
12. 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*, volume 2719 of *LNCS*, pages 384–396. Springer, 2003.
13. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proc. of the 31st Annual ACM Symp. on Theory of Computing*, pages 295–304, 1999.
14. D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29:1740–1759, 2000.
15. M. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 53–62, 2001.
16. M. Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. *Manuscript*, 2006.
17. M. Elkin and D. Peleg. Spanner constructions for general graphs. In *Proc. of the 33th ACM Symp. on Theory of Computing*, pages 173–182, 2001.
18. M. Elkin and J. Zhang. Efficient algorithms for constructing $(1+\epsilon,\beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18:375–385, 2006.
19. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proc. of the 31st International Colloq. on Automata, Languages and Progr.*, pages 531–543, 2004.
20. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: The value of space. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, pages 745–754, 2005.
21. J. Feigenbaum, S. K. M. Strauss, and M. Viswanathan. An approximate L^1 difference algorithm for massive data streams. *Journal on Computing*, 32:131–151, 2002.
22. D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Comput.*, 18:740–747, 1989.
23. D. Peleg and E. Upfal. A tradeoff between size and efficiency for routing tables. *J. of the ACM*, 36:510–530, 1989.
24. S. Pettie. Ultrasparse spanners with sublinear distortion. *Manuscript*, 2005.
25. M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of the 33rd ACM Symp. on Theory of Computing*, pages 183–192, 2001.
26. M. Thorup and U. Zwick. Compact routing schemes. In *Proc. of the 13th Symp. on Parallelism in Algorithms and Architectures*, pages 1–10, 2001.
27. M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *Proc. of Symp. on Discr. Algorithms*, pages 802–809, 2006.
28. D. Woodruff. Lower bounds for additive spanners, emulators, and more. *Manuscript*, 2006.