Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners[†]

Michael Elkin [*]

September 18, 2008

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 [25]. Specifically, the *processing time-per-edge* of our algorithm is $O(1)$, and it is drastically smaller than that of the algorithm of [25], 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 incremental update time of $O(1)$, 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. [26] were published. More recently, research in the streaming model was extended to traditional graph problems [7, 24, 23, 25]. 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* [18, 23, 17], *synchronization* [3, 29, 5], *routing* [30, 6, 34], and online load balancing [4]. The problem of constructing spanners with various parameters is a subject of intensive recent research [22, 18, 33, 13, 11, 35, 31, 36].

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. [25]. 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 $O(t^2 \cdot \log n \cdot n^{1/(t-1)})$. Also, their result gives rise immediately to a streaming algorithm for $(2t-1)$-approximate all-pairs-distance-computation (henceforth, $(2t-1)$-*APDC*) algorithm with the same parameters.

[†]Preliminary versions of this paper were published in ICALP 2007 [20] and in electronic archive [21].

[*]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel, `elkinm@cs.bgu.ac.il`. This research has been supported by the Israeli Academy of Science, grant 483/06.

	# passes	Processing time-per-edge	Stretch	# Edges (whp)
[25]	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	worst-case $O(1)$	$2t-1$	$O((\log n)^{1-1/t} \cdot n^{1+1/t})$

Table 1: A comparison between the algorithm of Feigenbaum et al. [25] and our new streaming algorithm. The word "whp" stands for "with high probability". The result of [25] provides non-trivial estimates for $t \geq 3$, while our algorithm does for for $t \geq 2$.

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 $O(t \cdot \log^{2-1/t} n \cdot n^{1+1/t})$ 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 just $O(1)$, which is *drastically smaller* than that of Feigenbaum et al. [25].

To summarize, our algorithm constructs a spanner with a smaller number of edges and number of bits of space used (by a factor of $n^{1/(t(t-1))}(\log n)^{1/t}$), and it does so using a *drastically* reduced (and 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. [25] can be found in Table 1.

Independently[1] us Baswana [9] came up with a streaming algorithm for computing sparse spanners. Most efficiency parameters of the algorithm of [10] are equal to those of our algorithm, but its bound on the processing time-per-edge is inferior to that of our algorithm. Specifically, it provides an *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, while in our algorithm it is $O(1)$. Also, the algorithm of [10] appears to be significantly more complex than ours.

A variant of our algorithm can be seen as a fully dynamic algorithm for maintaining a $(2t-1)$-spanner with $O(t \cdot (\log n)^{1-1/t} \cdot n^{1+1/t})$ edges. The incremental update time of this algorithm is exactly the processing time-per-edge of our streaming algorithm (that is, $O(1)$), 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 $O(\frac{m}{n^{1/t}} \cdot (\log n)^{1/t})$, and moreover, with probability at least $1 - \left(\frac{\log n}{n}\right)^{1/t}$, the decremental update time is $O(1)$. The size of the data structures maintained by an incremental variant of our algorithm is $O(t \cdot (\log n)^{2-1/t} \cdot n^{1+1/t})$ 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 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. [2]. 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.

[1]The preliminary version of this paper was published in the electronic archive on November 1, 2006 [21]. The preliminary version of the paper [9] was published in the electronic archive on November 6, 2006 [10].

Baswana [8] improved the result of Ausillo et al. [2] and devised a fully dynamic algorithm for maintaining spanners of optimal size with stretch at most 6 with expected constant update time. Baswana [8] 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 [8].

In addition, we remark that by combining our fully dynamic algorithm for maintaining sparse spanners with the dynamic All-Pairs-Almost-Shortest-Paths (henceforth, APASP) algorithm of Roditty and Zwick [32] we obtain tradeoffs for the dynamic APASP problem that are advantageous for a certain range of parameters. See Appendix A for more details.

Finally, another variant of our algorithm constructs $(2t-1)$-spanners with expected size $O(t \cdot n^{1+1/t})$ for *weighted* graphs in the *standard centralized* (that is, not streaming) model of computation. Its running time is $O(SORT(|E|))$ time with high probability, where $SORT(|E|)$ is the time required to sort $|E|$ numbers. Using the fastest known algorithm for sorting integers [27] we get the running time of $O(|E|\sqrt{\log\log n})$ for the case when all weights are integer. The current state-of-the-art algorithm for constructing sparse $(2t-1)$-spanners for weighted graphs requires $O(|E| \cdot t)$ expected time [13]. Hence for integer-weighted graphs our algorithm is more efficient than that of Baswana and Sen [13] whenever $t = \omega(\sqrt{\log\log n})$. (Observe that the range of t is $\Omega(1) = t = O(\log n)$.)

We also devise some improved algorithms for graphs with general (not necessarily integer) weights. First, if there are at most $\frac{|E|}{\log n}$ distinct edge weights in the graph, then a variant of our algorithm requires $O(|E|)$ time, with high probability. Within this time the algorithm constructs a $(2t-1)$-spanner of the input graph with expected $O(t \cdot n^{1+1/t})$ edges. Second, if the aspect ratio $\hat{\omega}$ of the graph (defined as the ratio between the maximum and minimum edge weight) satisfies $\hat{\omega} \leq 2^{n^{O(1)}}$, then another variant of our algorithm computes $(2t-1+n^{-O(1)})$-spanner with expected $O(t \cdot n^{1+1/t})$ edges, in expected $O(|E|)$ time.

Related Work and Our Techniques: A fully dynamic *distributed* algorithm for maintaining sparse spanners is presented in the companion paper [21, 19]. Our algorithm in this paper combines the techniques of Feigenbaum et al. [25], of Baswana and Sen [13], with those developed in [21].

More specifically, both the algorithm of Feigenbaum et al. [25] and our algorithm build upon the techniques of Baswana and Sen [13]. 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. [25] 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. [25].

In another related work, a streaming algorithm for constructing $(1+\epsilon, \beta)$-spanners was devised in [23]. Finally, after the preliminary version of our paper was published in ICALP 2007 [20], a significantly improved dynamic algorithm for maintaining sparse spanners was devised in [12].

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. Section 2.5 is devoted to two extensions of our algorithm to weighted graphs. In Section 3 we extend it to the dynamic setting. In Section 4 we summarize and list some open problems. In Appendix A we describe an application of our result to dynamic APASP problem.

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 $O((\log n)^{1-1/t} \cdot n^{1+1/t})$ edges using only $O(|H| \cdot \log n) = O(t \cdot (\log n)^{2-1/t} \cdot n^{1+1/t})$ bits of storage space, and processing each edge in $O(1)$ 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.

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 $p = \left(\frac{\log n}{n}\right)^{1/t}$. 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. See Figure 1. 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 $L(P) \leq t-2$.

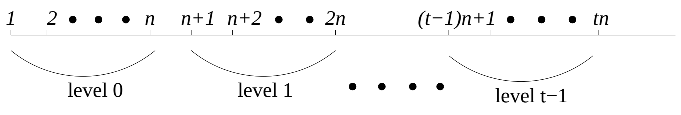

Figure 1: Labels of different levels. A label $in + j$ has base value j and level i.

An alternative way to see a label P is as a pair $(B(P), L(P))$ such that $P = n \cdot L(P) + B(P)$. Intuitively, the base value $B(P)$ determines the identity of a *tree* τ to which the vertex v labeled by P belongs currently, and the value $L(P)$ is equal to the distance between v and the root w_P of τ. The radius $r(w_P)$ of w_P reflects the maximum distance to which the label of w_P is allowed to propagate in the network during an execution of the algorithm. When a label P is adopted by a vertex from one of its neighbors, the level of P is incremented, and its base value is preserved. However, the level of P is not allowed to grow beyond $r(w_P)$.

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

Proof: Let $i \in [n]$, and let $u \in V$ be the vertex with $I(u) = i$. The probability that the label $n \cdot (t-1) + i$ occurs is equal to the probability that $r(u) = t-1$, that is, p^{t-1}. Hence the expected number of labels of level $t-1$ is $n \cdot p^{t-1} = O(n^{1/t} \cdot (\log n)^{1-1/t})$. The lemma follows by Chernoff bound. ▮

We remark that the way that we define and manipulate labels is closely related to the way it is done in Feigenbaum et al. [25].

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

```
1: Let $u$ be the vertex s.t. $P(u) \succ P(v)$
2: if $P(u)$ is a selected label then
3:     $P(v) \leftarrow P(u) + n$
4:     $Sp(v) \leftarrow Sp(v) \cup \{e\}$
5: else if $B(P(u)) \notin M(v)$ then
6:     $M(v) \leftarrow M(v) \cup \{B(P(u))\}$
7:     $Sp(v) \leftarrow Sp(v) \cup \{e\}$
8: 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 2.2 *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.,

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

Next, we argue that the set $\bigcup_{v \in V} X(v)$ is sparse as well. First, by Lemma 2.1, the number of distinct labels of level $t-1$ that occur during the algorithm is, with high probability, $O(n^{1/t} \cdot (\log n)^{1-1/t})$. 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 $O(n^{1/t} \cdot (\log n)^{1-1/t})$.

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 2.3 *$X^{(t-1)}(v) = \emptyset$.*

Proof: Suppose for contradiction that there exists an edge $e = (v, u) \in X^{(t-1)}(v)$. By definition of $X^{(t-1)}(v)$, the label of v at the time when the algorithm read the edge e satisfied $L(P(v)) = t-1$. Since e was inserted into $X(v)$, it follows that $P(u) \succ P(v)$, where $P(u)$ is the label of u at that time. Consequently, $L(P(u)) = t-1$ as well. This is a contradiction to the definition if $X^{(t-1)}(v)$. ▮

In the next lemma we argue that the cardinalities of the sets $X^{(i)}(v)$, $0 \le i \le t-2$, are small as well. (Though these sets are not necessarily empty.)

Lemma 2.4 *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, $|X^{(i)}(v)| = O(n^{1/t} \cdot \log^{1-1/t} n)$.*

Proof: The value $L(P(v))$ grows each time that the algorithm encounters an edge (u, v) with $P(u) \succ P(v)$ and such that $P(u)$ is a selected label. On the other hand, for a fixed index $i \in [(t-2)]$, during the time period when the condition $L(P(v)) = i$ holds, the size of the set $X^{(i)}(v)$ is incremented each time the algorithm encounters an edge (u, v) with $P(u) \succ P(v)$, such that $P(u)$ is not a selected label, and such that $B(P(u))$ does not belong to $M(v)$.

Fix an execution of the algorithm, and an index i, $i \in [(t-2)]$. Consider the sequence η of all edges $\eta = (e_1 = (u_1, v), e_2 = (u_2, v), \ldots, e_k = (u_k, v))$, for some integer $k \ge 0$, that arrived during the time period when the condition $L(P(v)) = i$ holds, and such that $P(v)$ did not grow as a result of processing these edges. Let $\sigma = (P_1, P_2, \ldots, P_k)$, $P_j = P(u_j)$, $j \in [k]$, be the sequence of labels of vertices u_j such that the edge (u_j, v) appears in η. Note that the edge $e_j = (u_j, v)$ is inserted into $X(v)$ only if the base value B_j of the label P_j appears in σ for the first time. Moreover, the edge contributes to $X^{(i)}(v)$ only if $L(P_j) < t-1$.

Let $\sigma' = (P_{j_1}, P_{j_2}, \ldots, P_{j_\ell})$, for some integer $0 \le \ell \le k$, be the subsequence of σ that contains only labels $P_{j_q} = P(u_{j_q})$, $q \in [\ell]$, of level at most $t-2$ such that no other label with the same base value appears in σ with an index smaller than j_q. It follows that $|X^{(i)}(v)| = \ell$. Moreover, all labels that appear in σ' are not selected, as the algorithm increases $L(P(v))$ whenever it encounters an edge (u_j, v) as above with a selected label $P(u_j)$.

Since all labels of σ' have distinct base values, and they are of level at most $t-2$, each of these labels has a probability exactly p to be selected independently of other labels in the sequence. Hence the probability that ℓ or more unselected labels of level $t-2$ or less with distinct base values will appear *in a row* (with no selected label in-between them) is at most

$$(1-p)^\ell = \left(1 - \left(\frac{\log n}{n}\right)^{1/t}\right)^\ell .$$

In other words,

$$\mathbb{P}(|X^{(i)}(v)| \ge \ell) \le \left(1 - \left(\frac{\log n}{n}\right)^{1/t}\right)^\ell . \qquad (2)$$

For $\ell = c \log n \cdot \left(\frac{n}{\log n}\right)^{1/t}$ for a sufficiently large value of c, this probability is at most $\frac{1}{n^c}$.

Hence, with high probability, $|X^{(i)}(v)| = O(n^{1/t} \cdot \log^{1-1/t} n)$. ▮

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

Corollary 2.5 *Under the assumption of Lemma 2.4, for every vertex $v \in V$, with high probability, the overall number of edges inserted into $X(v)$ is $O(t \cdot n^{1/t} \cdot (\log n)^{1-1/t})$.*

Proof: By Lemma 2.4, with high probability, the number of edges (u,v) with $L(P(u)) < t-1$ inserted into $X(v)$ is at most

$$\sum_{i=0}^{t-2} |X^{(i)}(v)| = O(t \cdot n^{1/t} \cdot (\log n)^{1-1/t}) \ .$$

As was argued in the discussion preceding Lemma 2.3, the number of edges (v,u) with $L(P(u)) = t-1$ inserted into $X(v)$ is, with high probability, $O(t \cdot n^{1/t} \cdot (\log n)^{1-1/t})$ as well. ▮

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

Corollary 2.6 *Under the assumptions of Lemma 2.4, with high probability, the spanner H constructed by the algorithm contains $O(t \cdot n^{1+1/t} \cdot (\log n)^{1-1/t})$ edges. Moreover, each table $M(v)$, $v \in V$, stores, with high probability, at most $O(t \cdot n^{1/t} \cdot (\log n)^{1-1/t})$ values, and consequently, overall the algorithm uses $O(|H| \cdot \log n) = O(t \cdot n^{1+1/t} \cdot (\log n)^{2-1/t})$ 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), $|\bigcup_{v\in V} T(v)| \leq n \cdot (t-1)$. By Corollary 2.5, with high probability, $|\bigcup_{v\in V} X(v)| = O(t \cdot n^{1+1/t} \cdot (\log n)^{1-1/t})$, 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 2.5, with high probability, $|X(v)| = O(t \cdot n^{1/t} \cdot (\log n)^{1-1/t})$, 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 $|\bigcup_{v\in V} M(v)| \leq |\bigcup_{v\in V} X(v)| \leq |\bigcup_{v\in V} X(v)| + |\bigcup_{v\in V} T(v)| = O(|H|)$. 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

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

For an integer $k \geq 1$, 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 2.7 *Let $v, v' \in V$ be a pair of vertices such that there exist positive integers $k, k' \geq 1$ 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 $\bigcup_{v\in V} T(v)$.*

Proof: The proof is by induction of $L(P_k(v)) + L(P_{k'}(v'))$. The induction base is when $L(P_k(v)) = L(P_{k'}(v')) = 0$. Hence $B(P_k(v)) = B(P_{k'}(v'))$. It follows that $P_k(v) = B(P_k(v)) = B(P_{k'}(v')) = P_{k'}(v')$, and so $I(v) = I(v')$. Hence $v = v'$, and the statement follows. (The path that starts and ends in v, and contains no other vertex is said to have length 0.)

For the induction step we first observe that the only way for a vertex v to be labeled by a label of level greater than 0 is by adopting the label from a neighboring vertex (and incrementing its level).

Since $L(P_k(v)) + L(P_{k'}(v')) > 0$, and $L(P_k(v)), L(P_{k'}(v')) \geq 0$, it follows that either $L(P_k(v)) > 0$ or $L(P_{k'}(v')) > 0$. Suppose without loss of generality that $L(P_k(v)) > 0$. Let u be the vertex such that when the edge (u,v) was read, the vertex v adapted the label $P(u)+n = P_k(v)$. Let k'' be the time step on which this happened. It follows that on time step k'' the edge (u,v) was inserted into $T(v)$ (on line 4 of Algorithm 1), and on this time step the label of u, $P_{k''}(u)$, was equal to $P_k(v) - n$, and so $L(P_{k''}(u)) = L(P_k(v)) - 1$, and $B(P_{k''}(u)) = B(P_k(v)) = B(P_{k'}(v'))$. Hence $L = L(P_{k''}(u)) + L(P_{k'}(v')) = L(P_k(v)) + L(P_{k'}(v')) - 1$, and so the induction hypothesis is applicable to the pair of vertices $\{u, v'\}$. Hence there exists a path of length L between u and v' in $\bigcup_{v \in V} T(v)$, and so there exists a path of length $L+1 = L(P_k(v)) + L(P_{k'}(v'))$ connecting v and v' in the edge 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 2.8 *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.*

Proof: If there exist indices $k, k' \geq 1$ such that $B(P_k(v)) = B(P_{k'}(v'))$ then by Lemma 2.7 there exists a path of length at most $L(P_k(v)) + L(P_{k'}(v')) \leq 2t-2$ between v and v' in the spanner.

Otherwise, consider the time step k, $k \geq 1$, of the execution of the algorithm on which the edge $e = (v, v')$ was read. Let $P = P_k(v)$, $P' = P_k(v')$ be the labels of v and v', respectively, at time k. Suppose without loss of generality that $P' \succ P$. If P' is a selected label, then the edge e was added to $T(v)$, and so there exists a path of length 1 connecting v and v' in the spanner.

Otherwise, consider the case that P' is not a selected label. Let $M_k(v)$ (respectively, $X_k(v)$) denote the set $M(v)$ (resp., $X(v)$) at time k. If $B(P') \notin M_k(v)$ then the edge $e = (v, v')$ was added to $X(v)$ (line 7 of Algorithm 1), and again the distance in the spanner between v and v' is equal to 1.

We are left with the case that the label of v', P', is not a selected label, and $B(P') \in M_k(v)$. In this case, by construction, there exists an edge $(u', v) \in X(v)$ such that u' was labeled by P'', $B(P'') = B(P')$, at some earlier time k', $k' \leq k$. (This can be seen by a straightforward induction on the time moment k.) Hence, by Lemma 2.7, since $B(P_k(v')) = B(P_{k'}(u')) = B(P')$, there exists a path of length at most $2t-2$ connecting the vertices v' and u' in the set $\bigcup_{v \in V} T(v)$. Since $(u', v) \in X(v)$, it follows that there exists a path of length at most $2t-1$ between v and v' in the spanner. ▮

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 2.6, 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. (Since $|M(v)| \leq |X(v)|$, by inequality (2) and union-bound, 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.)

To maintain the sets $M(v)$ we will use the *dynamic hashing dictionary* (henceforth, *dictionary*) due to Dietzfelbinger and Meyer auf der Heide [16]. The following result is implicit in [16].

Theorem 2.9 *[16, 15] Let N' be a fixed positive integer known to the algorithm in the beginning of the computation, and c' be a universal arbitrarily large constant. There exists a fixed function $f(\cdot)$ that satisfies the following statement.*

Suppose that a dictionary of size N' is allocated in the beginning of computation, and that there is a sequence of poly(N') insertion, deletion, and lookup operations applied to the dictionary, such that at all times the number of items stored does not exceed N'. Then each lookup operation requires worst-case time $f(c') = O(1)$, and each update (insertion or deletion) operation requires time $f(c') = O(1)$ with probability at least $1 - O((N')^{-c'})$. Finally, this dictionary uses space $O(N')$.

We set $N' = N \cdot n = c \cdot t \cdot (\log n)^{1-1/t} \cdot n^{1+1/t}$, set c' to be a sufficiently large universal constant ($c' = 2$ will suffice), and use one single dictionary $\mathcal{D}$ (given by Theorem 2.9) for *all sets* $M(v)$. In other words, every element stored in $\mathcal{D}$ is a pair (B, v), indicating that the base value B belongs to the set $M(v)$. As we have shown above, with probability at least $\frac{1}{n^{c-2}}$, the number of elements stored by the algorithm in the dictionary $\mathcal{D}$ is at most N'.

By Theorem 2.9, using this dictionary we get $f(c') = O(1)$ processing time-per-edge *with high probability.* (There is a probability of $O((N')^{-c'}) = O(n^{-c'(1+1/t)})$ that one of the insertions into $\mathcal{D}$ will require more than $f(c')$ time.) To achieve *worst-case* processing time $O(1)$, we modify the insertion operation as follows. Consider an insertion of a pair (B, v) into $\mathcal{D}$. If it did not complete within $f(c')$ operations, we stop the insertion and return the dictionary $\mathcal{D}$ to its state before the execution of this insertion started. (The corresponding edge is, however, inserted into the spanner.) Each operation of the insertion algorithm of [16] can be undone, and undoing it requires by at most a constant factor more time than doing it. Hence this may require additional at most $O(f(c'))$ operations. (For this end we need to maintain a list with at most $f(c') = O(1)$ operations that modified the dictionary during the current insertion.) In addition, we maintain a counter with the number of elements currently stored in the dictionary $\mathcal{D}$. If the algorithm attempts to insert an element when $\mathcal{D}$ contains N' elements, we just ignore this insertion and leave the dictionary unchanged. (Here too, the corresponding edge is inserted into the spanner.) Other operations of the dictionary are left unchanged.

Observe that with this modification, the worst-case processing time-per-edge of the algorithm becomes $O(1)$. However, it may cause some false negatives during lookup operations. In other words, if the algorithm will lookup for a value B that should have belonged to $M(v)$, the dictionary will answer that $B \notin M(v)$, even though ideally, the answer should be positive.

On the other hand, for a single false negative to occur, either at least one of the at most N' insertions should have required more than $f(c')$ time or the spanner should contain more than N' edges. By Corollary 2.6, Theorem 2.9, and union-bound, the probability of this event is, however, $O((N')^{-c'} + n^{-(c-2)}) = O(n^{-c'(1+1/t)} + n^{-(c-2)})$. Hence, with high probability, no false negative will occur. In addition, the effect that false negatives may have on the execution of the algorithm is not that bad either. Specifically, each negative lookup (a true or a false one) leads to an insertion of a single edge into the spanner on line 7 of the algorithm. Hence each false negative leads to an insertion of an edge e which is not necessary for the spanner. In other words, the spanner H' that is produced by our modified algorithm will contain the spanner H that would have been produced by our original algorithm, and might also contain a few more edges, one for each false negative that occurred during the algorithm. Since H is a $(2t-1)$-spanner of the original graph, so is its supergraph H'. By Corollary 2.6, with probability at least $1 - \frac{1}{n^{c-2}}$, it holds that $|H| \leq c \cdot t \cdot (\log n)^{1-1/t} \cdot n^{1+1/t}$. Since with probability at least $1 - \frac{1}{n^{c'}}$, the number of false negatives is zero, it follows that with probability at least $1 - \frac{1}{n^{c-2}} - \frac{1}{n^{c'}}$, the subgraphs H' and H are equal (i.e., $H = H'$), and thus $|H'| \leq c \cdot t \cdot (\log n)^{1-1/t} \cdot n^{1+1/t}$. This completes the proof of correctness of the modified algorithm.

The following corollary summarizes this argument.

Corollary 2.10 *The processing time-per-edge of our algorithm is* $O(1)$.

The properties of our streaming algorithm are summarized in the following theorem. This theorem follows directly from Corollary 2.6, Lemma 2.8, and Corollary 2.10.

Theorem 2.11 *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 $O(1)$ processing time-per-edge. The space used by the algorithm is $O(|H| \cdot \log n) = O(t \cdot \log^{2-1/t} n \cdot n^{1/t})$ bits with high probability. The preprocessing of the algorithm requires $O(n)$ time.

Remark: It is easy to verify that setting $p = n^{-1/t}$ in our algorithm guarantees the expected size of $O(t \cdot n^{1+1/t})$ for the spanner, but slightly increases the upper bound on the number of edges of the spanner that holds with high probability. The latter becomes $O(t \cdot n^{1+1/t} \cdot \log n)$.

2.5 Weighted graphs

In this section we describe a number of extensions of our algorithm to weighted graphs.

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 graph. (The *aspect ratio* $\hat{\omega}$ is defined as $\frac{w_{max}}{w_{min}}$, where w_{max} (respectively, w_{min}) is the maximum (resp., minimum) weight of an edge in the graph.) 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., [25]), 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 $\ell = \lceil \log_{(1+\epsilon)} \hat{\omega} \rceil$ categories, $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_\ell$, 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.

Finally, one can also use this algorithm in the standard (not streaming) centralized model of computation for constructing $(2t-1)$-spanner of expected size $O(t \cdot n^{1+1/t})$ in expected time $O(SORT(|E|))$ for *weighted* graphs, where $SORT(|E|)$ is the time required to sort $|E|$ numbers. For this end we start with sorting the edge set by their weights, and then provide them to the streaming algorithm in the increasing order of weights. It follows from Theorem 2.11 (see also the argument that appears later in this section) that the resulting subgraph is a $(2t-1)$-spanner of the original weighted graph. Using the fastest known algorithm for sorting integers [27] we have $SORT(|E|) = O(|E|\sqrt{\log\log n})$ (expected time). Our streaming algorithm spends $O(1)$ time per edge, and so the overall expected running time is $O(|E|\sqrt{\log\log n})$.

We summarize these properties in the next theorem.

Theorem 2.12 *For any parameter $t \geq 1$ and n-vertex* integer-weighted *graph $G = (V,E)$, the algorithm described above constructs a $(2t-1)$-spanner with expected $O(t \cdot n^{1+1/t})$ edges. The expected running time of the algorithm is $O(|E|\sqrt{\log\log n})$.*

The current state-of-the-art algorithm for constructing sparse $(2t-1)$-spanners for weighted graphs requires $O(|E| \cdot t)$ expected time [13]. Hence our algorithm is more efficient than that of [13] whenever $t = \omega(\sqrt{\log\log n})$ and the weights are integer.

If edge weights are not necessarily integer, still under certain conditions this technique provides an improved running time. In particular, if the set of edge weights $\{w(e) \mid e \in E\}$ contains at most $\frac{|E|}{\log n}$

distinct weights then it can be sorted in $O(|E|)$ time, and the overall running time of the algorithm becomes $O(|E|)$.

Moreover, if one is willing to settle for a relaxed stretch guarantee of $(2t-1+\delta)$ for an arbitrarily small $\delta > 0$, then one can always guarantee the running time of $O(|E|\sqrt{\log\log n})$. In addition, if the aspect ratio $\hat{\omega}$ of the graph is at most exponential in $n^{O(1)}$ then $(2t-1+n^{-O(1)})$-spanners can be constructed in time $O(|E|)$.

The algorithm that guarantees these bounds works as follows. It starts with computing w_{min} and w_{max}, and fixing an arbitrarily small $\epsilon > 0$. Basically, it then partitions the edge set E of the input graph G into $\ell = \lceil \log_{(1+\epsilon)} \hat{\omega} \rceil$ categories $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_\ell$, according to edge weights, defined exactly as in the beginning of this section. Then it invokes the streaming algorithm, and feeds it first with the edges from the category $\mathcal{E}_1$, then with edges from $\mathcal{E}_2$, etc. (The last edges that are given to the streaming algorithm are those from $\mathcal{E}_\ell$.) The order of the edges within a given category is arbitrary.

Consider the constructed subgraph H. It is a $(2t-1)$-spanner of the unweighted version of G, and it contains expected $O(t \cdot n^{1+1/t})$ edges. Moreover, for an edge $e = (u, v) \in \mathcal{E}_i$, for some $i \in [\ell]$, either it belongs to H, or H contains a path P with $2t-1$ or less edges connecting u and v so that all edges of P belong to $\bigcup_{j=1}^{i} \mathcal{E}_j$. Consequently, each edge e' in P satisfies $w(e') \leq (1+\epsilon) \cdot w(e)$, and thus the weight of P is at most $(2t-1)(1+\epsilon) \cdot w(e)$. Hence H is a $(2t-1)(1+\epsilon)$-spanner of G.

However, if implemented naively, this algorithm requires $O(\epsilon^{-1} \log \hat{\omega} + |E|)$ time and space. As long as $\hat{\omega} = 2^{O(\epsilon|E|)}$, this is $O(|E|)$. For the case of larger aspect ratio we next provide a more careful implementation of this algorithm. For this end we use a number of additional data structures. First, we employ a dynamic dictionary $\mathcal{D}$ of [16] of size $O(|E|)$ that stores pointers to linked lists with edges from different categories. Specifically, for each index $i \in [\ell]$ such that there exists at least one edge $e \in \mathcal{C}_i$ (i.e., $(1+\epsilon)^{i-1} \leq w(e) < (1+\epsilon)^i$), there is a linked list $\mathcal{D}_i$ associated with the key i that contains all edges from the category $\mathcal{C}_i$. A category $\mathcal{C}_i$ as above (such that $\mathcal{C}_i \neq \emptyset$) will be called a *non-empty* category. Note that the number of non-empty categories is at most $|E|$, even if $\ell = \lceil \log_{1+\epsilon} \hat{\omega} \rceil$ is larger than $|E|$. The second data structure that we use is an array I of size $|E|$ that stores indices of non-empty categories.

The algorithm goes through the edge set E. For each edge $e \in E$, it computes the index $i = i(e)$ such that $e \in \mathcal{C}_i$. Then the algorithm tests whether the index i is in the dictionary $\mathcal{D}$. If i is not in $\mathcal{D}$ then it is inserted into $\mathcal{D}$ as a key, and the linked list $\mathcal{D}_i$ associated with the key i is initialized to contain the single edge e. Moreover, in this case the index i is stored in the array I. (For this end one can keep a counter ctr initialized as 0 that counts the number of elements inserted into I. Once a new index i needs to be inserted, we insert it into the entry $I[ctr+1]$, and increment ctr.)

With high probability, each insertion and search query for the dictionary $\mathcal{D}$ requires $O(1)$ time, and thus, this step requires $O(|E|)$ time.

The second step is to sort the array I of indices. Since the indices are integer, this can be accomplished in expected $O(|E|\sqrt{\log\log n})$ time [27]. Moreover, if ℓ is at most polynomial in n, then this can be done in $O(|E|)$ time using Radix Sort. (See, e.g., [14], ch. 8, or [28] for a yet more efficient algorithm.)

Next, the algorithm goes over the array I in the increasing order of indices. It starts with the smallest index i stored in I, traverses the linked list $\mathcal{D}_i$ associated with the key i in the dictionary $\mathcal{D}$, and feeds the streaming algorithm with the edges stored in $\mathcal{D}_i$. Once it exhausts $\mathcal{D}_i$, it continues to the next smallest index stored in I. Since the streaming algorithm spends $O(1)$ time per edge, this step requires $O(|E|)$ time.

Clearly, this is an implementation of the algorithm described earlier. Thus, it constructs a $(2t-1)(1+\epsilon)$-spanner with expected $O(t \cdot n^{1+1/t})$ edges. It requires $O(|E|)$ space. The expected running time of the algorithm is $O(|E|\sqrt{\log\log n})$. Also, by setting $\epsilon = \frac{\delta}{2t-1}$, for an arbitrarily small $\delta > 0$, we get the stretch bound of $(2t-1+\delta)$. Moreover, if ℓ is not too large (at most polynomial in n) then the running time is $O(|E|)$, with high probability. This condition on ℓ means that $\ell = \lceil \log_{1+\epsilon} \hat{\omega} \rceil = O(t \cdot \delta^{-1} \cdot \log \hat{\omega})$ should be at most n^c, for some constant c. Recall that $t = O(\log n)$. Hence, for $\delta = n^{-O(1)}$, the condition is equivalent to $\hat{\omega} \leq 2^{n^{O(1)}}$. We summarize this section with the following theorem.

Theorem 2.13 *For an integer parameter $t \geq 1$, a real $\delta > 0$, and an n-vertex* weighted *graph $G = (V, E)$ different variants of our algorithm construct subgraphs H of expected size $O(t \cdot n^{1+1/t})$ so that one of the following holds.*
(i) If $|\{w(e) : e \in E\}| = O\left(\frac{|E|}{\log n}\right)$ then H is a $(2t-1)$-spanner, and the expected running time is $O(|E|)$.
(ii) H is a $(2t - 1 + \delta)$-spanner, and the expected running time is $O(|E|\sqrt{\log\log n})$.
(iii) If $\hat{\omega} \leq 2^{n^{O(1)}}$ then H is a $(2t - 1 + n^{-O(1)})$-spanner, and the expected running time is $O(|E|)$.

We remark that these results hold under the assumption (called also the *RAM model assumption*) that each arithmetic operation involving $O(1)$ edge weights requires $O(1)$ time. The algorithm of [13] relies on a slightly weaker assumption. Specifically, the assumption in [13] is that comparing two edge weights requires $O(1)$ time.

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 $O(|E|)$ time in the worst case. Each edge that is added to the graph is processed using our streaming algorithm. The spanner and the satellite data structures are updated in $O(1)$ time-per-edge in the worst case.

Next, we 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 $p = \left(\frac{\log n}{n}\right)^{1/t}$ 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.

Now we argue 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 $O(1)$. To adapt the algorithm to deal with crashes of this kind, we change the rule that specifies the behavior of the algorithm when it scans an edge (v, u) with $P(v) \prec P(u)$ and $B(P(u)) \in M(v)$. In this scenario Algorithm 1 realizes that another edge $e' = (v, u')$ with $B(P(u')) = B(P(u))$ was already read, and "drops" this edge (that is, does not insert it into the spanner). The set of edges of this type will be called the set of edges *dropped* by v, and denoted $D(v)$. The new version of the algorithm will maintain a set $M(v)[B]$ for every vertex v and value B recording all edges of this type. (A pointer to the linked list with these edges will be stored with the pair (v, B) in the dictionary $\mathcal{D}$.)

Another modification of the algorithm is that when an edge $e = (v, u)$ is read, the algorithm associates the endpoint v with a smaller label $P(v) \prec P(u)$ with this edge, and also records whether e was inserted into $T(v)$, $X(v)$ or $D(v)$. If e was inserted into $X(v)$ or $D(v)$, the value $B(P(u))$ is also recorded and associated with the edge e.

Consider a crash of an edge $e = (v, u)$, and suppose without loss of generality that e is associated with v. Suppose also that e does not belong to $T(v)$. If $e \in X(v)$ then the algorithm pops an arbitrary edge $e' = (v, u')$ from $M(v)[B]$ (if the latter set is not empty), and inserts it into $X(v)$. (Consequently, in this case the edge e' joins the spanner.) If $e \in D(v)$ then it is removed from $M(v)[B]$.

For the analysis we introduce the following notation. The *appearance* (respectively, *crash*) of an edge e is denoted $a(e)$ (resp., $c(e)$). Our algorithm processes a sequence of events $f_1(e_1), f_2(e_2), \ldots$, where each

f_i is either an appearance or a crash (that is, either a or c), and $e_1, e_2, \ldots$ is a sequence of not necessarily disjoint edges. These edges are assumed not to belong to $\bigcup_{v \in V} T(v)$, i.e., to be *non-tree* edges.

By a straight-forward induction on the length of this sequence it is easy to verify that the algorithm maintains the following invariant.

Lemma 3.1 *Consider the state of the data structures of the algorithm after processing a sequence* $f_1(e_1), f_2(e_2), \ldots, f_h(e_h)$, $h = 1, 2, \ldots$ *is a positive integer, and* $e_1, e_2, \ldots, e_h$ *are all non-tree edges. Then for every edge* $e = (v, u)$ *present in the graph at this point hold the two properties listed below.*

1. *The spanner maintained by the algorithm spans* e.
2. *If* $e \in D(v)$ *and* B *is the value associated with it, then there exists an edge* $e' = (v, u') \in X(v)$ *with the same associated value* B.

Each crash of a non-tree edge invokes one deletion operation from the dictionary $\mathcal{D}$. Hence by Theorem 2.9, it requires $O(1)$ time with high probability, and expected time $O(1)$. Since the entire spanner can be recomputed in time $O(|E|)$ by our algorithm, it follows that the expected decremental update time of our algorithm is $O(\frac{|E|}{n^{1/t}} \cdot (\log n)^{1/t})$. The size of the data structure maintained by the incremental variant of the algorithm is $O(t \cdot (\log n)^{2-1/t} \cdot n^{1+1/t})$ bits, and the fully dynamic algorithm maintains a data structure of size $O(|E| \cdot \log n)$.

We summarize this section with a following corollary.

Corollary 3.2 *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* $O(t \cdot (\log n)^{1-1/t} \cdot n^{1+1/t})$ *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* $O(|E|)$ *time in the worst-case. The incremental update time of the algorithm is* $O(1)$. *The expected decremental update time is* $O(\frac{|E|}{n^{1/t}} \cdot (\log n)^{1/t})$, *and with probability at least* $1 - \left(\frac{\log n}{n}\right)^{1/t}$ *the decremental update time is* $O(1)$.

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. An application of this algorithm to the dynamic All-Pairs-Almost-Shortest-Paths problem can be found in Appendix A.

4 Conclusion and Open Problems

Our streaming algorithm produces spanners with optimal stretch $2t-1$, does so in one pass over the stream of the edges, has optimal processing time-per-edge $O(1)$. The constructed spanners have expected size $O(t \cdot n^{1+1/t})$, and with high probability, their size is $O(t \cdot (\log n)^{1-1/t} \cdot n^{1+1/t})$. One open problem is to devise spanners of size $O(n^{1+1/t})$ by a streaming algorithm with similar, or even somewhat inferior, other parameters. Another challenging question is to devise a deterministic streaming algorithm with similar properties. Finally, devising a single-pass streaming algorithm for constructing sparse $(1+\epsilon, \beta)$-spanners is an interesting open problem as well.

Acknowledgements

The author is grateful to Martin Dietzfelbinger for very inspiring and helpful correspondence.

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] 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.

[3] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 4:804–823, 1985.

[4] 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.

[5] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.

[6] B. Awerbuch and D. Peleg. Routing with polynomial communication-space tradeoff. *SIAM J. Discrete Mathematics*, 5:151–162, 1992.

[7] 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.

[8] S. Baswana. Dynamic algorithms for graph spanners. In *Proc. of the European Symp. of Algorithms, ESA'06*, pages 76–87, 2006.

[9] S. Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Information Processing Letters*, 106(3):110–114, 2008.

[10] S. Baswana. Faster streaming algorithms for graph spanners. ArXiv:cs/0611023, November 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. Sarkar. Fully dynamic algorithms for graph spanners with poly-logarithmic update time. In *Proc. of the 19th Annual Symp. on Discr. Algorithms*, pages 672–681, 2008.

[13] 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.

[14] T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill Book Company, 2001.

[15] M. Dietzfelbinger. personal communication, 2008.

[16] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. of the 17th International Colloq. on Automata, Languages and Programming, ICALP*, pages 6–19, 1990.

[17] D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29:1740–1759, 2000.

[18] M. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 53–62, 2001.

[19] M. Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *In Proc. of the 26th Annual Symp. on Principles of Distrib. Computing*, pages 185–194, 2007.

[20] M. Elkin. A streaming and fully dynamic centralized algorithm for maintaining sparse spanners. In *Proc. of the Internat. Colloq. Autonata, Languages, and Progr.*, pages 716–727, 2007.

[21] M. Elkin. A near-optimal fully dynamic distributed algorithm for maintaining sparse spanners. ArXiv:cs/0611001, November 2006.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Y. Han and M. Thorup. Integer sorting in $o(n\sqrt{\log\log n})$ time and linear space. pages 135–144, 2002.

[28] D. Kirpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[29] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Comput.*, 18:740–747, 1989.

[30] D. Peleg and E. Upfal. A tradeoff between size and efficiency for routing tables. *J. of the ACM*, 36:510–530, 1989.

[31] S. Pettie. Ultrasparse spanners with sublinear distortion. *Manuscript*, 2005.

[32] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proc. 45th Annual IEEE Symp. on Foundations of Comp. Science*, pages 499–508, 2004.

[33] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of the 33rd ACM Symp. on Theory of Computing*, pages 183–192, 2001.

[34] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. of the 13th Symp. on Parallelism in Algorithms and Architectures*, pages 1–10, 2001.

[35] M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *Proc. of Symp. on Discr. Algorithms*, pages 802–809, 2006.

[36] D. Woodruff. Lower bounds for additive spanners, emulators, and more. In *In Proc. of the Annual Symp. on Foundations of Computer Science*, pages 389–398, 2006.

Appendix

A Dynamic All-Pairs Almost Shortest Paths Problem

Using Corollary 3.2 in conjunction with the dynamic All-Pairs-Almost-Shortest-Paths (henceforth, APASP) algorithm of Roditty and Zwick [32] we obtain tradeoffs for the incremental APASP problem that are advantageous for a certain range of parameters. Specifically, we use the following result.

Theorem A.1 *[32] For every $\epsilon, \delta > 0$, and $z \leq m^{1/2-\delta}$, there exists a fully dynamic $(1+\epsilon)$-approximate APASP algorithm with an amortized update time $\tilde{O}(|E|n/z)$, and query time $O(z)$.*

Consequently, by maintaining an incremental $(2t-1)$-spanner with $\tilde{O}(n^{1+1/t})$ edges, and maintaining the dynamic data structure of [32] computed for the maintained spanner, we obtain the following result.

Theorem A.2 *For every $\epsilon, \delta > 0$, positive integer $t \geq 1$, and $z \leq n^{\frac{1}{2}(1+1/k)-\delta}$, there exists a fully dynamic $(2t-1)(1+\epsilon)$-approximate APASP algorithm for unweighted undirected n-vertex graphs with amortized expected update time of $\tilde{O}\left(\frac{n^{2+1/t}}{z} + \frac{|E|}{n^{1/t}} \cdot (\log n)^{1/t}\right)$, and query time at most $O(z)$.*

Note that when choosing $z = \omega(n^{1/t})$, this algorithm has a *sublinear* amortized update time in terms of the number of edges even when the number of edges $m = \Theta(n^2)$. To our knowledge, this is the first dynamic APASP algorithm that achieves an amortized update of $o(n^2)$ for *all graphs* with a non-trivial query time is non-trivial ($z = n^{1/t+\eta}$, $\eta > 0$ is a constant).

However, note that our algorithm supports only *incremental updates*, while the algorithm of [32] is fully dynamic. In addition, our algorithm suffers from a significantly higher approximation ratio than that of [32]; it is $(2t-1)(1+\epsilon)$ instead of $1+\epsilon$.

To overcome one of these drawbacks, we can make this algorithm fully dynamic by recomputing the spanner every time there is an edge deletion query. This way decremental update time becomes $\tilde{O}\left(\frac{n^{2+1/t}}{z} + m\right)$, which is still significantly faster than $\tilde{O}\left(\frac{m \cdot n}{z}\right)$ of [32] as long as $m = \omega(n^{1+1/t})$. We summarize this extension in the following corollary.

Corollary A.3 *For every $\epsilon, \delta > 0$, positive integer t, and $z \leq n^{\frac{1}{2}(1+1/t)-\delta}$, there is a fully dynamic $(2t-1)(1+\epsilon)$-approximate APASP algorithm for unweighted undirected n-vertex graphs with amortized update time of $\tilde{O}\left(\frac{n^{2+1/t}}{z} + m\right)$ (incremental update time of $\tilde{O}\left(\frac{n^{2+1/t}}{z}\right)$), and query time $O(z)$.*