Streaming Algorithm for Graph Spanners - Single Pass and Constant Processing Time per Edge

Surender Baswana *
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur - 208016, INDIA.
Email : sbaswana@cse.iitk.ac.in

Keywords : Analysis of algorithms, On-line algorithms, Streaming, Spanner, Shortest path.

1 Introduction

A spanner is a sparse subgraph of a given graph that preserves approximate distance between each pair of vertices. In precise words, a t-spanner of a graph $G = (V, E)$, for any $t \in \mathbb{N}$, is a subgraph $(V, E_S), E_S \subseteq E$ such that, for any $u, v \in V$, their distance in the subgraph is at most t times their distance in the original graph. The parameter t is called the *stretch* associated with the t-spanner. The concept of spanner was defined formally by Peleg and Schäffer [13] though the associated notion was used implicitly by Awerbuch [3] in the context of network synchronizers. Computing t-spanner of smallest size for a given graph is a well motivated combinatorial problem with numerous applications in the area of distributed systems, communication networks and all pairs approximate shortest paths (see [4, 13] and references therein). However, computing t-spanner of smallest size for a graph is NP-hard. In fact, for $t > 2$, it is NP-hard [6] even to approximate the smallest size of t-spanner of a graph with ratio $O(2^{(1-\mu)\ln n})$ for any $\mu > 0$. Having realized this fact, researchers have pursued another direction : to design an efficient algorithm which, for a given graph on n vertices, outputs a t-spanner whose size is of the order of the maximum size of the sparsest t-spanner of a graph on n vertices. A 43 years old girth lower bound conjecture by Erdős [7] implies that there are graphs on n vertices whose $2k$ as well as $(2k-1)$-spanner will require $\Omega(n^{1+1/k})$ edges. This conjecture has been proved for k = 1,2, 3 and 5. Given the hardness result for the original spanner problem, therefore, a valid algorithmic goal would be to design an efficient algorithm that, for any weighted graph on n vertices, computes a $(2k-1)$-spanner of size $O(n^{1+1/k})$. In the classical RAM model, there exist efficient algorithms [1, 10] which achieve this goal. In this article we address the problem of computing spanners of unweighted graphs in streaming model. This problem has recently gained lot of importance due to its application in computing all-pairs approximate distances in streaming environment [9].

The streaming model [11] has the following two characteristics: firstly the input data can be accessed only sequentially in the form of a stream; secondly the working memory is considerably smaller than the size of the entire input stream. An algorithm in this model is allowed to make one or more passes over the input stream to solve a given computational problem. The sequentiality in accessing the data and the small working memory size enforce the following restriction : during a pass, a data item once evicted from the memory can't be brought back into the memory. The number of passes, the size of working memory, and the processing time per data item are the parameters which one aims to optimize in a streaming algorithm. For the existing streaming algorithms for various graph problems please refer to [9, 12].

1.1 Computing spanner in streaming environment and new results

Feigenbaum et al. [9] designed the first streaming algorithm for spanners of unweighted graphs. For any $k \in \mathbb{N}$, their algorithm computes a $(2k+1)$-spanner of expected size $O(kn^{1+1/k})$ in one pass and requires ex-

*This research work was supported by a Young Faculty Chair award from Research I Foundation, CSE, IIT Kanpur.

pected $\Theta(k^2 n^{1/k})$ processing time per edge. Note that the size of the spanner thus computed is away from the conjectured optimal bound by a factor of $\Theta(kn^{1/k^2})$.

We present a one pass streaming algorithm that spends amortized constant time per edge and computes a $(2k-1)$-spanner of expected size $O(kn^{1+1/k})$. Note that the size of the spanner (and the working memory) is away from the conjectured optimal bound just by a factor of k, and so is essentially optimal for any constant k. Moreover, one pass and $O(m)$ time to process the stream is the best one can hope for in the streaming model.

Recently and independently Elkin [5] came up with a streaming algorithm for graph spanners. The efficiency parameters of his algorithm are the same as that of our algorithm except a slight difference in the processing time per edge. Our algorithm takes $O(m)$ time to process the entire stream of edges, and thus amortized $O(1)$ time per edge. The processing time per edge achieved by his algorithm has $O(1)$ expected bound and $O(\sqrt{\frac{\log n}{\log \log n}})$ worst case bound. Another difference is that our algorithm has an advantage of using only elementary data structures for implementation at the expense of slightly involved analysis.

Remark. Our algorithm at each stage maintains a $(2k-1)$-spanner of the graph seen so far. Therefore, it can also be viewed as a partial dynamic (incremental) algorithm for computing a $(2k-1)$-spanner of an unweighted graph with amortized $O(1)$ time per edge insertion.

Our algorithm is based on new simple ideas and a combination of the techniques used by the algorithms of Baswana and Sen [4], and Feigenbaum *et al.* [9].

1.2 Preliminaries

We assume, like the previous streaming algorithms [8, 9], that n, the number of vertices is known in advance and the vertices are numbered from 1 to n. The central idea of the algorithm is a suitable grouping of vertices called clustering.

Definition 1.1 *A cluster is a subset of vertices, and a clustering $\mathcal{C}$, is a union of disjoint clusters. Each cluster will have a unique vertex which will be called its center.*

The uniqueness of the center of a cluster can be used to represent a clustering $\mathcal{C}$ as an array (of the same label $\mathcal{C}$) of size n in the following way : $\mathcal{C}[v]$ will denote the center of the cluster containing v unless v does not belong to any cluster, in which case $\mathcal{C}[v] = 0$. A cluster c is said to be *adjacent* to a vertex u if there is some edge (u, v) in the graph for some $v \in c$. With respect to a given clustering $\mathcal{C}$, a vertex $u \in V$ is said to be a *clustered* vertex if it belongs to some cluster in $\mathcal{C}$, and an *unclustered* vertex otherwise.

2 Streaming algorithm for $(2k-1)$-spanners

Prior to processing the stream of edges, we have the graph on n vertices without edges. As a preprocessing step the algorithm employs random sampling to construct initial k clusterings $\{\mathcal{C}_i | 0 \leq i \leq k-1\}$ as shown in Figure 1.

$S_0 \leftarrow V$; $S_k \leftarrow \emptyset$;
For $(0 < i < k)$
  S_i is formed by selecting each $v \in S_{i-1}$ independently with probability $n^{-1/k}$;
For (each $v \in V$ and $0 \leq i < k$)
  if $(v \in S_i)$ $\mathcal{C}_i[v] \leftarrow v$ **else** $\mathcal{C}_i[v] \leftarrow 0$.

Figure 1: Forming the initial k clusterings

Let $\ell_c(v)$ be the highest level $i < k$ such that v appears as center of some cluster in $\mathcal{C}_i$. It can be observed that initially each cluster in $\mathcal{C}_i, i < k$ is a singleton set storing only its center, and a small fraction of cluster centers at each level $i < k-1$ are selected randomly to form cluster centers at level $i+1$. We will say that a cluster $c \in \mathcal{C}_i$ is a *sampled* cluster at level i if its center was selected to form a cluster center at $(i+1)$th level. As will become evident from the algorithm below, the only change in a cluster during processing the stream will be that other vertices from lower levels might join it, and the following assertion will hold at each stage.

$\mathcal{A}$: For each cluster $c' \in \mathcal{C}_{i+1}$, there exists a unique sampled cluster c at level i such that $c \subseteq c'$, and vice versa.

An overview of the algorithm: Let $\ell(u)$ denote the highest level $i < k$ such that u appears as a member

of some cluster in $\mathcal{C}_i$. In the beginning $\ell(u) = \ell_c(u)$ for each $u \in V$. When an edge, say (u, v), appears in the stream, the algorithm will process u if $\ell(u) \leq \ell(v)$ and will process v otherwise. This processing and in fact the whole algorithm from view point of a vertex can be described informally as follows.

Each vertex $u \in V$ waits at its present level $\ell(u)$ for an opportunity to move to a level higher than $\ell(u)$. Vertex u gets such an opportunity only when an edge, say (u, v), appears in the stream such that $\ell(v) > \ell(u)$ and the cluster c containing v is a sampled cluster at level $\ell(u)$. In this case, it follows from assertion $\mathcal{A}$ that there must be some cluster $c' \in \mathcal{C}_{\ell(u)+1}$ such that $c \subseteq c'$. So u adds the edge to the spanner, moves to the next level and joins the cluster c'. If c' too is a sampled cluster at $\ell(u) + 1$, the vertex u further moves to the next level and joins appropriate cluster c'' satisfying $c' \subseteq c''$, and so on. However, u will get an opportunity to move to a higher level on very few occasions since there is only a small fraction of sampled clusters at each level. Most of the times it will receive an edge incident from an unsampled cluster in $\mathcal{C}_{\ell(u)}$. In this case, u just adds that edge to the spanner provided no edge appeared in the stream earlier which was incident on u from the same unsampled cluster. In order to check the latter condition, it is wasteful to explore the entire list of neighboring clusters of u at level $\ell(u)$. So we adopt a buffering approach such that the vertex u will initially add the edge to its temporary buffer $Temp(u)$, and prune this set once its size grows *sufficiently*.

The algorithm : First we describe the data structures used by the algorithm. We shall use k arrays $\mathcal{C}_i, i < k$ to store clustering at each level. Each vertex $u \in V$ keeps lists $Temp(u)$ and $\mathcal{E}(u)$. The list $\mathcal{E}(u)$ will store one edge per unsampled cluster at level $\ell(u)$ which is adjacent to u, and $Temp(u)$ will act as a temporary buffer for these edges which we prune once the number of edges in $Temp(u)$ equals the number of edges in $\mathcal{E}(u)$. Whenever vertex u moves to a higher level, it moves all edges of the lists $\mathcal{E}(u)$ and $Temp(u)$ to a bigger list $\mathcal{E}_S$ which will store spanner partially. Initially all the lists are empty. The algorithm for processing an edge (u, v) of the stream and the procedure $Prune(u, i)$ are described in Figures 2 and 3 respectively. For the procedure $Prune()$ we shall use an array A. Before the first call of $Prune()$ every entry of A is initialized to 0 and at the end of every call all non-zero entries of A are reset to zero.

If $(\ell(u) > \ell(v))$ swap (u, v) **Endif**
$i \leftarrow \ell(u)$; $x \leftarrow \mathcal{C}_i[v]$; $h \leftarrow \ell_c(x)$;
If $(h > i)$ *// opportunity for u to move up*
  For $j = i + 1$ to h **do** $\mathcal{C}_j[u] \leftarrow x$;
  $\ell(u) \leftarrow h$;
  $\mathcal{E}_S \leftarrow \mathcal{E}_S \cup Temp(u) \cup \mathcal{E}(u)$;
  $Temp(u) \leftarrow \emptyset$; $\mathcal{E}(u) \leftarrow \{(u, v)\}$;
Else
  $Temp(u) \leftarrow Temp(u) \cup \{(u, v)\}$;
  If $(|Temp(u)| = |\mathcal{E}(u)|)$ *Prune*(u, i) **Endif**
Endif

Figure 2: Processing an edge (u, v) of the stream.

1. **For** each $(u, w) \in \mathcal{E}(u)$ **do**
   $A[\mathcal{C}_i[w]] \leftarrow 1$.
2. **For** each $(u, v) \in Temp(u)$ **do**
   If $(A[\mathcal{C}_i[v]] = 0)$
   $A[\mathcal{C}_i[v]] \leftarrow 1$;
   $\mathcal{E}(u) \leftarrow \mathcal{E}(u) \cup \{(u, v)\}$ **Endif**
   $Temp(u) \leftarrow Temp(u) \backslash (u, v)$.
3. **For** each $(u, w) \in \mathcal{E}(u)$ **do** $A[\mathcal{C}_i[w]] \leftarrow 0$.

Figure 3: The procedure *Prune*(u, i).

From the way a vertex joins cluster at next higher level during the algorithm it follows easily that each $u \in V$ is a clustered vertex at all the levels from 0 to $\ell(u)$. Furthermore, the processing of an edge always preserves the validity of assertion $\mathcal{A}$. We now state an important observation which will be used in the analysis of the algorithm.

Observation 2.1
For each vertex $u \in V$, $|Temp(u)| < |\mathcal{E}(u)|$ always, except just before the invocation of $Prune(u, i)$ when $|Temp(u)| = |\mathcal{E}(u)|$.

3 Analysis of the algorithm

Analyzing the running time : The time complexity of the algorithm shown in Figure 2 is determined

by the iterations of the **For** loop and various calls of $Prune()$. Each iteration of the loop increases the level of some vertex, and since the level of a vertex never exceeds $k-1$, the total time spent on these loops is $O(nk)$. The total time required by $Prune(u,i)$ is of the order of $|\mathcal{E}(u)| + |Temp(u)|$, which by Observation 2.1 is $O(|Temp(u)|)$. So it suffices to charge $O(1)$ cost to each edge of $Temp(u)$ to account for the time spent in a call of $Prune(u,i)$. Note that an edge $(u,v) \in E$ is processed only once by $Prune(u,i)$ while being a member of $Temp(u)$. This is because, after $Prune(u,i)$ procedure, either the edge gets discarded forever or it becomes a member of $\mathcal{E}(u)$. Hence each edge will be charged only $O(1)$ cost during all calls of $Prune()$ during the algorithm. So total time spent in processing the stream of edges is $O(m+nk)$ which is certainly $O(m)$ when $m = \Omega(nk)$. In order to ensure that time taken is $O(m)$ always, we may start our algorithm only after we have seen nk edges in the stream; and in case there are only nk edges in the stream, we just output all of them as spanner edges. We now define two sets of edges $\mathcal{E}^+$ and $Temp$ as follows.

- $\mathcal{E}^+ = \cup_{u \in V} \mathcal{E}(u) \cup \mathcal{E}_S$.
- $Temp = \cup_{u \in V} Temp(u)$.

In the following subsections, we shall now prove that the set $\mathcal{E}^+ \cup Temp$ at any moment is a $(2k-1)$-spanner for the set of edges appeared in the stream till that moment, and its expected size is $O(kn^{1+1/k})$.

3.1 Analyzing the stretch of the spanner

Lemma 3.1 *Let c' be any cluster in $\mathcal{C}_i$. Each vertex $u \in c'$ is connected to its center through a path of at most i edges from $\mathcal{E}^+$.*

Proof: The proof is based on induction on i and the number of edges of the stream seen so far. Let x be the center of the cluster c'. If c' is a singleton cluster, there is nothing to prove, so assuming otherwise, let $u \neq x$ be a vertex which belongs to c'. The vertex u would have become member of c' only in the following situation in the past. Vertex u was at some level $j < i$ and some edge (u,v) appeared in the stream with vertex v being a member of some sampled cluster c in $\mathcal{C}_j$. The assertion $\mathcal{A}$ implies that c is a subset of c', and x is its center too. Now applying induction hypothesis, there is a path $\subseteq \mathcal{E}^+$ between v and x with length at most j. Also note that vertex u adds the edge (u,v) to $\mathcal{E}(u)$ while joining c'. Hence there is a path $\subseteq \mathcal{E}^+$ of length at most $j+1 \leq i$ between u and the center of the cluster c' □

At any stage of the algorithm, let (u,v) be an edge that has appeared in the stream. It is certain that either (u,v) belongs to $\mathcal{E}^+ \cup Temp$ or it has been discarded by $Prune()$ in the past. Let (u,v) got discarded during $Prune(u,i)$. Now it follows from $Prune(u,i)$ that the edge (u,v) could be discarded only if we had already selected some other edge (u,w) in $\mathcal{E}(u)$ incident from the same cluster in $\mathcal{C}_i$ to which v belongs. Lemma 3.1 implies that vertices v and w, being the members of the same cluster, are connected by a path $\subseteq \mathcal{E}^+$ which passes through the center of the cluster and has length at most $2i$. This path concatenated with the edge $(u,w) \in \mathcal{E}(u)$, is a path in $\mathcal{E}^+$ between u and v with length at most $2i+1$, which is at most $2k-1$ since $i < k$ always. Thus for each edge discarded by $Prune()$, there is a path of length at most $2k-1$ in $\mathcal{E}^+$ between its endpoints. This implies that any shortest path in the original graph is stretched by a factor of at most $(2k-1)$ in the subgraph with edges $\mathcal{E}^+ \cup Temp$. Hence we can conclude that, at any moment, the set $\mathcal{E}^+ \cup Temp$ is a $(2k-1)$-spanner for the stream of edges seen so far.

3.2 Analyzing the spanner size

A vertex contributes edges to the partial spanner $\mathcal{E}_S$ only when it moves to next higher level, and the contribution is $|\mathcal{E}(u)| + |Temp(u)|$. It follows from Observation 2.1 that $|Temp(u)| \leq |\mathcal{E}(u)|$ holds always. So it suffices to bound the number of edges accumulated in $\mathcal{E}(u)$ for the period when u stays at a particular level $i < k$.

Lemma 3.2 *For any vertex u and any level $i < k$, during the period $\ell(u) = i$, the expected number of edges in $\mathcal{E}(u)$ is at most $n^{1/k}$*

Proof: Let us first consider any level $i < k-1$. For any arbitrary but fixed stream of edges, let $\langle c_1, c_2, \ldots \rangle$ be the clusters at level i arranged in the chronological order of their getting adjacent to u. When a cluster from $\mathcal{C}_i$ gets adjacent to u and the cluster is a sampled cluster, the vertex u will hook onto that cluster and move to the next level. So an edge incident from c_j

will be added to $\mathcal{E}(u)$ if none of $c_1, ..., c_j$ were sampled clusters in $\mathcal{C}_i$. Now each cluster at level i is a sampled cluster independently with probability $p = n^{-1/k}$. So an edge incident on u from c_j will be added to $\mathcal{E}(u)$ with probability $(1 - n^{-1/k})^j$. Hence the expected number of edges in $\mathcal{E}(u)$ accumulated during the period $\ell(u) = i$ is at most $\sum_{1 \leq j}(1 - n^{-1/k})^j \leq n^{1/k}$. Now let us consider level $k-1$. Note that the expected number of clusters at level $k-1$ is $n^{1/k}$, and none of them is a sampled cluster since $S_k = \emptyset$. So a vertex u on reaching this level would continue to stay there and would contribute at most one edge per neighboring cluster to $\mathcal{E}(u)$. □

It thus follows from Lemma 3.2 and the preceding discussion that the expected number of edges contributed by each vertex to the spanner $\mathcal{E}^+ \cup Temp$ is $O(kn^{1/k})$. Hence the expected size of the spanner will be $O(\min(m, kn^{1+1/k}))$.

We can thus conclude the following theorem.

Theorem 3.1 *Given any $k \in \mathbb{N}$, a $(2k-1)$-spanner of expected size $O(\min(m, kn^{1+1/k}))$ for an unweighted graph can be computed in streaming model in one pass with $O(m)$ time to process the entire stream. The working memory required is $O(kn^{1+1/k})$.*

4 Conclusion and open problems

We presented a single pass and linear time streaming algorithm for computing a $(2k-1)$-spanner of size $O(\min(m, kn^{1+1/k}))$ for any unweighted graph. The size bound is away from the conjectured optimal bound by a multiplicative factor of k; an important open problem is to explore whether it is possible to get rid of this factor while maintaining single pass and linear time. Another problem is to devise deterministic streaming algorithm for graph spanners. Recently Ausiello et al. [2] designed such an algorithm for spanners of stretch at most 6. Another interesting open problem is to design streaming algorithm for spanners of weighted graphs. Note that, for weighted graphs, an $O(k)$ pass algorithm for computing a $(2k-1)$-spanner follows easily from the static algorithm of Baswana and Sen [4]. The expected size of the spanner and the working memory will be $O(\min(m, kn^{1+1/k})$, and it will take amortized $O(1)$ processing time per edge per pass. So the real challenge is to design a single pass streaming algorithm for weighted graphs without affecting the optimal bound on the spanner size and constant processing time per edge.

Acknowledgment. We are very thankful to anonymous referee for his comments which improved the readability of this article.

References

[1] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.

[2] G. Ausiello, C. Demetrescu, P. G. Franciosa, G. F. Italiano, and A. Ribichini. Small stretch spanners in the streaming model: New algorithms and experiments. In *Proceedings of 15th Annual European Symposium on Algorithms*, volume 4698 of *LNCS*, pages 605–617. Springer, 2007.

[3] B. Awerbuch. Complexity of network synchronization. *Journal of ACM*, pages 804–823, 1985.

[4] S. Baswana and S. Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures and Algorithms*, 30:532–563, 2007.

[5] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. In *Proceedings of 13th Annual European Symposium on Algorithms*, volume 4596 of *LNCS*, pages 716–727. Springer, 2007.

[6] M. Elkin and D. Peleg. Strong inapproximability of the basic k-spanner problem. In *Proceedings of 27th International Colloquim on Automata, Languages and Programming*, volume 1853 of *LNCS*, pages 636–648. Springer, 2000.

[7] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.

[8] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proceedings of 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *LNCS*, pages 531–543. Springer, 2004.

[9] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, 2005.

[10] S. Halperin and U. Zwick. Linear time deterministic algorithm for computing spanners for unweighted graphs. *unpublished manuscript*, 1996.

[11] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 107–118, 1999.

[12] A. McGregor. Finding graph matchings in streaming model. In *Proceedings of 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, volume 3624 of *LNCS*, pages 170–181. Springer, 2005.

[13] D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.