Approximating Steiner Trees in the Semi-Streaming Model

Devon R. Graham (23142094), Raunak Kumar (47037130) *

July 22, 2017

Abstract

We present an algorithm for constructing approximate Steiner trees in the graph streaming context. Massive graphs arise in many modern settings. But traditional methods for analyzing such graphs are infeasible, due to resource limitations. However, recent work has dealt with many well-known problems on such graphs. Here, we consider the *semi-streaming* setting, where an algorithm is limited to $O(n \operatorname{polylog} n)$ space while processing its stream. We emulate a well known 2-approximate algorithm for calculating Steiner trees in an offline setting, making use of existing semi-streaming algorithms for constructing spanners and minimum spanning trees. The result is a $2(1+\epsilon)\left(2\frac{\log n}{\log\log n}+1\right)$-approximation for the weighted metric Steiner tree problem, where ϵ is a user defined accuracy parameter.

1 Introduction

In recent years there has been a lot of work on analyzing properties of massive graphs. Such graphs, for example, representing friends in a social network, require prohibitively large amounts of memory to store. One approach is to process the edges in a *data stream model*. That is, edges (and their weights) are revealed in a sequential fashion, and an algorithm must process the data in the order it is seen, while using only a limited amount of memory. In particular, there has been focus on the *semi-streaming* model, where an algorithm is permitted $O(n \operatorname{polylog} n)$ space, where n is the number of vertices of the graph. See [5] for a survey of recent work in this model.

The Steiner tree problem in undirected graphs is an important problem in combinatorial optimization. Formally, the problem is defined as: Given an undirected graph $G = (V, E)$, with edge weights $w_e \in \mathbb{R}^+$ and a subset $T \subset V$ of "terminal" nodes, we wish to compute the weight of the minimum spanning tree of G that contains all vertices in T. That is, what is the minimum cost of connecting all the terminal nodes together? The metric Steiner tree problem is the same as above with the additional restriction that the edge weights of G follow the triangle inequality. In our project, we consider the latter and refer to it as the Steiner tree problem. The decision version of the Steiner tree problem can be shown to be NP-complete via a reduction from vertex cover [4]. However, a simple 2-approximation algorithm is known. We present an adaptation of this algorithm to the semi-streaming model. We make use of existing developments in the semi-streaming literature, in particular graph sketches and graph spanners [5].

2 Background

In the past few years, researchers have developed models and algorithms that can efficiently process large data streams. It has been demonstrated that $O(n \operatorname{polylog} n)$ space is sufficient to compute both graph spanners [3] and minimum spanning trees [2]. We briefly describe the main techniques and existing work that our algorithm relies on, and the reader is encouraged to read the references for a detailed exposition.

*authors listed in alphabetical order

2.1 Count-Sketch

A common task in big data processing is to maintain a space efficient representation of the data, called a "sketch". This allows us to estimate properties of the data without having to store it explicitly.

Count-Sketch [6] is one such example. The setting is as follows. There is a stream of updates $\{a_1, a_2, \ldots, a_m\}$ to a vector $x \in \mathbb{R}^n$. Each update a_i is of the form $(i_t, \delta_t) \in [n] \times \mathbb{R}$. This means that at time t, the vector x is updated as $x_{i_t} \leftarrow x_{i_t} + \delta_t$. We want to be able to estimate properties of x such as it's p^{th} frequency moment, it's k largest coordinates, etc. without storing x explicitly.

Count-Sketch maintains a 2D array $C \in \mathbb{R}^{d \times w}$. Each coordinate i of x is hashed to some column in each row of C. So, C can be thought of as having d compact representations of x and every cell of C maintains a signed count of the elements of x that land in that "bucket". We use $2d$ mutually independent hash functions:

- $\forall j \in [d]$. $h_j : [n] \to [w]$ hashes indices of x to columns of C in row j.
- $\forall j \in [d]$. $r_j : [n] \to \{-1, +1\}$ outputs a random sign for indices of x in row j.

At time t, an update $a_i = (i_t, \delta_t)$ is processed in the following way: For $j = 1, 2, \ldots, d$, Increment $C_{j,h_j(i_t)}$ by $r_j(i_t) \cdot \delta_t$. We compute $\tilde{x}$, an estimate of x, as $\tilde{x} = \text{median}(z_j)$ where $z_j = r_j(i) \cdot C_{j,h_j(i)}$ $\forall j \in [d]$. The following claim is proved in [6].

Claim 1. *Let $d \geq \log \frac{1}{\delta}$ and $w > \frac{3}{\epsilon^2}$. Then $\forall i \in [n]$:*

1. $E[z_j] = x_i$
2. $\Pr[|\tilde{x_i} - x_i| \geq \epsilon \|x\|_2] \leq \delta$

By picking $d = O(\log 2n^2)$, we only need $O\left(\frac{\log 2n^2}{\epsilon^2}\right)$ space to store C versus $O(n)$ space if we were to store x explicitly.

2.2 Sparse Recovery

Definition 1. A vector x is called k-sparse if it has $\leq k$ non-zero coordinates.

Algorithm 1 Sparse Recovery

```
procedure SPARSE-RECOVERY
    Set d = 100 log_{3/2} n = O(log n)
    Set w ≥ 3k = O(k)
    Construct Count-Sketch table C for x
    if Any row of C has > k non-zero columns then
        output DENSE
    else
        output x̃    ▷ x̃ is constructed in the usual way
```

Often, we want to be able to approximate a vector x by a sparse vector. Sparse vectors have obvious space benefits and operations on sparse vectors can be significantly faster. Claim 2 [8] turns out to be a key component of the ℓ_0-sampling algorithm presented in the next subsection. We describe the required modification to Count-Sketch in Algorithm 1.

Claim 2. *With $O(k \log n)$ random bits, Count-Sketch can:*

1. *If x is k-sparse, output x with probability $\geq 1 - 1/n$.*
2. *If x is not k-sparse, output `DENSE` with probability $\geq 1 - 1/n$.*

2.3 ℓ_0-sampling

Definition 2. The *support* of a vector x, $\text{supp}(x)$, is the set of non- zero coordinates of x.

The goal of ℓ_0-sampling is to sample a coordinate of x uniformly at random from $\text{supp}(x)$, i.e. sample coordinate i with probability $\frac{\mathcal{I}[x_i \neq 0]}{\|x\|_0}$. Graph sketching relies heavily on ℓ_0-sampling. Various other applications are mentioned in [7]. The following claim and algorithm are presented in [7] and [9].

Claim 3. *There is an ℓ_0-sampler using $O\left(\log^2 n \log \frac{1}{\delta}\right)$ bits of space that*

1. *with probability $\geq 1 - \delta$, gives a uniform sample from supp(x).*

2. with probability $\leq \delta$, fails.

Algorithm: Let $I_0 = [n]$ and let $I_1, I_2, \ldots, I_{\log n}$ be random subsets of $[n]$, where each I_h chooses each element of x independently with probability $\frac{2^h}{n}$. We run $\log n + 1$ copies of Sparse-Recovery (SR) where SR_h only considers coordinates of x in I_h and set $k = 12 \log \frac{1}{\delta}$. SR_h could fail either by outputting `DENSE` or by returning the 0 vector. Let h be the smallest index such that SR_h succeeded, ie. it output a non-zero k-sparse vector. We output a random non-zero coordinate from this vector, which is a uniform random sample from $\text{supp}(x)$.

2.4 Graph Sketching

An important tool in our algorithm is the strategy of *graph sketching*, where the graph is first mapped to a smaller-dimensional "sketch" space [1]. Since the graphs we are dealing with are assumed to be very large, performing such computations on them directly is infeasible. Instead, we hope that we can infer useful properties of our original graph from computations performed in the sketch space, with high probability. Ahn et al. have shown that many useful and interesting properties of graphs can be computed in this manner including (but not limited to) connectivity, bipartiteness, and the weight of a minimum spanning tree [1].

We consider the *dynamic graph streaming* model, where the algorithm is presented with a stream of updates to the edges of the graph. Specifically, the updates are in the form of edge insertions and deletions. The algorithm sees the stream only once (or perhaps a small number of times), and must process the edges dynamically, as it sees them. In particular, the algorithms we consider fit into the *semi-streaming* model, where we are allowed $O(n \operatorname{polylog} n)$ storage space [1]. When n is very large, this is considerably less space than would be required to store an arbitrary n-vertex graph.

One factor distinguishing streaming algorithms is the number of passes over the data they require. In the case of connectivity and bipartiteness, a single pass is sufficient to check these properties exactly, with high probability. Also, a single pass is sufficient to compute a $(1+\epsilon)$ approximate weight to a minimum spanning tree using $O(\epsilon^{-1} n \log^3 n)$ space. However, in order to compute the weight of a minimum spanning tree exactly in the semi-streaming model, $O(\log n / \log \log n)$ passes over the data are required [1].

The algorithms described in [1], and the algorithm we construct, require that we can sample uniformly at random from the neighbours of a given vertex v (at least approximately). In order to do this, we can construct sketches for ℓ_0-sampling from the characteristic vector of the neighbourhood of v.

Let $\gamma^v \in \{0,1\}^n$ be the characteristic vector of the neighbourhood of vertex v. That is, $\gamma^v[u] = 1$ if and only if u is a neighbour of v. Then we apply a sketch $\mathcal{S}$ for ℓ_0-sampling to γ^v, such that querying $\mathcal{S}(\gamma^v)$ returns a uniformly random sample from the neighbours of v, with high probability [1]. This key observation will be used extensively in our algorithm for finding the weight of a minimum spanning tree.

However, we must be aware of the following caveat. We may naturally wish to query our sketch $\mathcal{S}(\gamma^v)$ more than once, and would expect to be returned uniformly random *independent* neighbours of v. However, this in general is not possible with ℓ_0-samplers [1].

2.5 Minimum Spanning Tree

In order to emulate the simple algorithm for computing an approximate Steiner tree in the semi-streaming model, our algorithm must compute a minimum spanning tree in a streaming fashion. We follow the method of [1] for doing this using $O(\log n / \log \log n)$ passes over the data stream. The approach is a modification of Boruvka's algorithm, presented in Algorithm 2.

A naive adaptation of this to the streaming setting is to emulate each iteration of the loop in Algorithm 2 using $O(\log n)$ passes over the data. In the first pass we construct ℓ_0-samplers for each vertex v, sample an incident edge at random and remember its weight w_v. In the next pass we ignore any edge that arrives with weight greater than w_v when constructing the ℓ_0-samplers. Thus, in $O(\log n)$ passes we will have found the minimum weight edge incident on v, and

Algorithm 2 Boruvka's offline MST algorithm

```
1: Input: G = (V, E)
2: Output: T, an MST of G
3: procedure BORUVKA(G)
4:     for i = 1, ..., log_2 n do
5:         for v in V do
6:             e_v ← min weight edge incident on v
7:         for v in V do
8:             Add e_v to T
9:             Contract e_v in G
10:    return T
```

we repeat this process $O(\log n)$ times to find a minimum weight spanning tree. Ahn et al. make two further optimizations which improve the analysis of this algorithm to $O(\log n/\log\log n)$ passes over the data [1].

2.6 Graph Spanners

An important subroutine in numerous graph algorithms, and a fundamental problem in its own right, is that of computing shortest path distances between vertices in a graph. One method for doing this is through the use *graph spanners*, sparse graphs in which the notion of distance (i.e., shortest path length) is maintained.

Definition 3. Given a graph $G = (V, E)$, an α-spanner of G is a subgraph $S = (V, E')$ such that, for any $u, v \in V$, $d_G(u, v) \leq d_S(u, v) \leq \alpha d_G(u, v)$, where $d_G(\cdot)$ is the length of the shortest path from u to v in G.

Our algorithm uses the method of [3] for spanner construction in the streaming setting. In the semi-streaming setting, this gives a $(2\log n/\log\log n + 1)$-spanner for unweighted graphs, and a $(1+\epsilon)(2\log n/\log\log n + 1)$-spanner for weighted graphs. Intuitively, the algorithm constructs a collection of small, dense subgraphs, which are then connected together to form a spanner using the sparse part of the original graph. The algorithm operates on unweighted graphs, but can be extended to the weighted case by a technique of geometric grouping [3].

The first step is a randomized vertex-labeling phase, before any of the data stream is seen. Then, as the stream is processed, the algorithm grows BSTs around the dense clusters of G. The final spanner output is the union of these BFS trees along with additional edges from the "sparse part" of the graph. Feigenbaum et al. show that with appropriate choice of parameters, this algorithm constructs a $(1+\epsilon)(2\log n/\log\log n + 1)$-spanner for weighted graphs using $O(\epsilon^{-1} n \operatorname{polylog} n \log W)$ space [3], where W is the ratio between the maximum and minimum edge weights in G.

3 Steiner Tree in the Streaming Model

3.1 Algorithm

In the offline setting, a simple 2-approximation algorithm [12] first finds the minimum spanning tree of a new graph H where the vertices $V(H)$ are the terminal nodes T and the edges $E(H)$ are the shortest paths between pairs of terminal nodes in the original graph G. We use 2 streams to adapt the above offline algorithm to the semi-streaming setting. Our algorithm proceeds in the same 2 phases: computing shortest paths between terminal nodes and finding the minimum spanning tree of H.

It is natural to try and implement a shortest path algorithm like Dijkstra's or Floyd-Warshall to perform phase 1. However, Feigenbaum et al. prove a lower bound on the space requirements to construct even a BFS tree [3]. In particular, they state that any algorithm that computes the first k layers of a BFS tree from any given vertex with probability $\geq \frac{2}{3}$ requires either greater than $\frac{k}{2}$ passes or $\Omega(n^{1+\frac{1}{k}} \operatorname{polylog} n)$ space, for any even constant k. So, we use graph spanners in our algorithm to approximate shortest path distances.

The input to our algorithm is the graph $G = (V, E)$, presented as a stream $I = \langle a_1, a_2, ..., a_m \rangle$, where each a_i represents an edge insertion and a weight. We are also given a set T of "terminal" nodes, which it can store explicitly, since there are at most n of them. The algorithm is allowed to

make $O(\log n/\log\log n)$ passes over I in a sequential manner. The output is a $2(1+\epsilon)\left(2\frac{\log n}{\log\log n}+1\right)$-approximate Steiner Tree of G. We proceed in two phases.

3.1.1 Phase 1

In the first phase, we execute the algorithm of [3] on I. The first step of their algorithm is a randomized vertex-labeling phase, before any of the data stream is seen. Set $L^0 = [n]$, the "level 0" labels. Then with probability $n^{-1/t}$ each label $l \in L^0$ will be chosen as *selected*, and placed in the set S^0 (here, t is an accuracy parameter). From each label in S^0, a new label $l' = l + n$ is created, and added to L^1. This continues until level $\lfloor t/2 \rfloor$, and the final set of labels is $L = \bigcup_{i=1}^{\lfloor t/2 \rfloor} L^i$.

As the stream is processed, the algorithm assigns labels from L to the vertices. For $l \in L$, let $C(l)$ be the set of vertices with label l. For each $C(l)$, we store a BFS tree, $Tree(l)$, which we build up as edges are processed. We also maintain a set D of edges which connect the clusters C together. For $l, l' \in L^{\lfloor t/2 \rfloor}$ if u has label l and v has label l', then we add (u, v) to D. The algorithm also remembers edges incident on each vertex that are not part of any $Tree(l)$, or of D. The spanner S we output is the union of the BFS trees $Tree(l)$, the set D, and the sets of other edges incident on each v.

Feigenbaum et al. show that by taking $t = \Omega(\log n/\log\log n)$, this algorithm constructs a $(2\log n/\log\log n+1)$-spanner for unweighted graphs. However, this can be extended to weighted graphs using a technique of "geometric grouping" to produce a $(1+\epsilon)(2\log n/\log\log n+1)$-spanner using $O(\epsilon^{-1} n \operatorname{polylog} n \log W)$ space [3], where W is the ratio between the maximum and minimum edge weights in G.

3.1.2 Phase 2

Now, we construct a stream to represent the graph H, as described earlier where $V(H) = T$ (the set of terminal nodes) and $E(H)$ are the shortest paths between pairs of terminal nodes in G. To do so, we first iterate over pairs of terminal nodes $(t_i, t_j) \in T^2$. For each pair, we query the spanner S constructed in phase 1 to obtain an approximate shortest path distance between the two in G. Note that we do not store all such pairs and pass them to the stream at once. Instead, we pass these pairs to the stream as they are created. This prevents us from exceeding the $O(n \operatorname{polylog} n)$ space restriction of the semi-streaming model.

Let J be the stream of edges representing H. We now take J to be the input to the minimum spanning tree algorithm presented in [1]. We proceed in $\log n$ rounds. Naively, in each round, we process the stream $O(\log n)$ times. On each pass we construct n ℓ_0-samplers, one for each vertex. We sample the neighbours of each vertex and "remember" the weight of the edge selected. While processing the stream in the following round we ignore any edges whose weight is greater than that of the "remembered" weight for its endpoints. In this fashion we ensure that we find the minimum weight edge incident on each vertex, in each round [1]. Having found this edge for each vertex, we add it to M, the minimum spanning tree of H. We then "contract" the edges we add to M in the following manner. If we add an edge (u, v) to M in round i, then in round $i+1$, we process updates to edges incident on u and v using the same ℓ_0-sampler. The output of the algorithm is T, an exact minimum spanning tree of the graph H, and thus an approximate Steiner tree for the graph G.

We can reduce the number of passes of the algorithm with the following observations made by Ahn et al. First, at each round, the number of vertices of the graph is at least halved. So emulating the ith phase can be done in $\log_{2^i} n$ passes. So only a total of $\sum_{i=1}^{\log n} \log_{2^i} n = O(\log n \log\log n)$ passes are required. With this fact, and the idea that in each round we could find not just the lightest weight edge but the k lightest, Ahn et al. are able to show that only $O(\log n/\log\log n)$ passes are required, and that this is sufficient to fit the semi-streaming criteria [1].

3.2 Analysis

Let W be the ratio between the maximum and minimum edge weights in G. In the first phase, we con-

struct a $(1+\epsilon)\left(2\frac{\log n}{\log\log n}+1\right)$-spanner S that uses $O(\epsilon^{-1}n\,\text{polylog}\,n\log W)$ space and takes 1 pass over the data [3]. In the second phase, we use an MST algorithm that uses $O(n\,\text{polylog}\,n)$ space and takes $O\left(\frac{\log n}{\log\log n}\right)$ passes over the data [1]. The offline algorithm is a 2-approximation for the weighted metric Steiner tree problem. From these observations, it follows that our algorithm is a $2(1+\epsilon)\left(2\frac{\log n}{\log\log n}+1\right)$ -approximation for the weighted metric Steiner tree problem that uses $O(\epsilon^{-1}n\,\text{polylog}\,n\log W)$ space and takes $O\left(\frac{\log n}{\log\log n}\right)$ passes over the data.

4 Conclusion

We have provided an algorithm for approximating weighted metric Steiner trees in the semi-streaming model. We make use of existing developments in the streaming literature, namely graph spanners and MST.

Biological networks are network models of biological systems. Examples include protein-protein interaction networks, and DNA- protein interaction networks [13]. In such networks, a variant of the Steiner tree problem is used to detect biological relationships between selected proteins or genes [13]. Our technique can be used to adapt the Steiner method for biological networks to a semi-streaming setting which can operate on much larger protein and gene datasets. Our technique can also be used to solve well known Steiner tree problems in networks on a large scale.

In the future, we can explore constructing spanners that take more passes over the data but give us a better approximation of the shortest path distances between vertices. The MST algorithm of Ahn et al. works in an insert-delete setting. However, the spanner construction of Feigenbaum et al. works in an insert only setting. So another extension of our project would be to construct spanners in the insert-delete setting and adapt our algorithm accordingly.

References

[1] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms. 459-467. 2012.

[2] Graham Cormode, and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms 55.1: 58-75. 2005.

[3] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model." Theoretical Computer Science 348, no. 2-3: 207-216. 2005.

[4] Richard M Karp. Reducibility among combinatorial problems. Complexity of computer computations. springer US. 85-103. 1972.

[5] Andrew McGregor. Graph Stream Algorithms: A Survey. ACM SIGMOD Record 43.1: 9-20, 2014.

[6] Chandra Chekuri. CS 598CSC: Algorithms for Big Data. Lecture 6 Notes. 2014.

[7] Hossein Jowhari, Mert Sağlam, Gabór Tardos. Tight Bounds for Lp Samplers, Finding Duplicate in Streams and Related Problems. Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. 49-58. 2011

[8] Nicholas J.A. Harvey. CPSC 536N: Algorithms That Matter. Assignment 2. 2017.

[9] Nicholas J.A. Harvey. CPSC 536N: Algorithms That Matter. Lectures 11 - 13. 2017.

[10] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, Jian Zhang. Graph Distances in the Data Stream Model. SIAM J. Comput., 38(5). 1709-1727. 2008.

[11] Kook Jin Ahn, Sudipto Guha, Andrew McGregor. Graph Sketches: Sparsification, Spanners, and Subgraphs. PODS 2012.

[12] Luca Trevisan. CS 261. Lecture 2. 2011.

[13] Nadja Betzler. Steiner Tree Problems in the Analysis of Biological Networks. PhD Thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen. 2006.