A Survey of Graph Algorithms in Extensions to the Streaming Model of Computation [1]

Thomas C. O'Connell [2]

Abstract

There has been a great deal of recent interest in the streaming model of computation where algorithms are restricted to a single pass over the data and have significantly less internal memory available than would be required to store the entire stream of data. Because of the inherent difficulty of solving graph problems in the streaming model, a number of extensions to the streaming model have been considered, namely the Semi-Streaming model, the W-Stream model, and the Stream-Sort model. In this chapter, we survey the algorithms developed for graph problems in each of these models. The survey is intended to be tutorial in nature although familiarity with graph algorithms is assumed.

1 Introduction

With our ever increasing ability to generate enormous amounts of information, comes a need to process that information more efficiently. In particular, we need to be able to process data that cannot fit into internal memory (i.e. RAM) and to do so in such a way that our access to external storage is efficient. Recently, there has been interest in a model of computation called the *streaming model* which has its origins in [HRR00] and also in [MP91]. In the streaming model, the data is presented sequentially in a single pass while the internal memory available is sufficient only to store a small portion of the data. The motivation for the streaming model is that sequential access to disk can be implemented very efficiently yet making multiple passes over large data sets may be prohibitively expensive or, in some cases, impossible because of the transient nature of the data.

A number of papers consider computing various statistics in one pass over a stream ([AMS99], [M03]). However, determining the types of graph problems that can be solved efficiently when the

[1]This is a preprint of a paper that will appear in *Fundamental Problems in Computing: Essays in Honor of Professor Daniel J. Rosenkrantz* (S. S. Ravi and Sandeep K. Shukla eds.), Springer-Verlag, 2007.

[2]Department of Mathematics and Computer Science, Skidmore College, Saratoga Springs, NY 12866. email: oconnellT@acm.org

graph is presented as a stream of edges is also an important research question. For many graph properties, it is impossible to determine whether a given graph has the property in a single pass using $o(n)$ space where n is the number of vertices in the graph ([FKM+05b]). (One notable exception is the problem of computing the number of triangles in a graph for which a 1-pass streaming algorithm appears in [BKS02]). Because of the inherent difficulty in solving graph problems in the 1-pass streaming model, extensions to the streaming model have been proposed. The most obvious extension is to allow multiple passes over the stream with the hope that the number of passes will be quite small in relation to the size of the stream. In [HRR00], it is suggested that studying the tradeoff between the number of passes and the amount of space required by streaming algorithms is an important research topic.

Beyond allowing multiple passes there are three main extensions currently discussed in the literature:

1. The *Semi-Streaming model* ([FKM+05a]) in which the algorithm is given $\Theta(n \log^k n)$ space where n is the number of vertices in the graph and k is any constant. In this case, the algorithm has enough internal memory to store the vertices but not necessarily the edges in the graph.

2. The *W-Stream model* ([R03]) in which the algorithm is allowed to write an intermediate stream as it reads the input stream. This intermediate stream, which can be at most a constant factor larger than the original stream, is used as the input stream for the next pass.

3. The *Stream-Sort model* ([R03], [ADR+04]) in which the algorithm is not only allowed to create intermediate streams but also to sort these streams in a single pass.

In this chapter, we survey the algorithms that have been developed for these extended models. For a more general survey of streaming algorithms, see [M03] and [BBD+02]. While the emphasis of this chapter is on the actual algorithms developed, we begin with a discussion of lower bounds on the space required to solve graph problems in the streaming model to motivate the discussion of the other models.

2 Lower Bounds

When receiving an input graph $G(V, E)$ as a stream, we assume, unless otherwise stated, the graph is given as a stream of edges $(u, v) \in E$ in no particular order. If the graph is weighted, an additional

weight component is added to each edge, giving us data items of the form $(u, v, w(u, v))$. Some lower bounds on the space required for streaming algorithms can be proven using counting techniques. For example, in [BGW03], lower bounds are provided for deterministic and randomized algorithms for $O(1)$-pass streaming algorithms for finding common neighborhoods of vertices. Another approach to proving lower bounds is to use results from communication complexity ([KN96]). We provide a brief description of the main ideas behind the communication complexity approach below.

2.1 Communication Complexity

The **two-party communication complexity** model originally defined in [Y79] consists of two players A and B. The players' combined goal is to compute a function $f : X \times Y \rightarrow Z$. When computing $f(x, y)$, Player A is given x but does not know y while player B is given y but does not know x. To compute the result, the players must communicate their private information to each other. The **communication complexity of** f is the number of bits of information that must be exchanged for f to be computed. In other words, if we define the cost of a communication protocol for the two players to be the number of bits that must be exchanged to compute $f(x, y)$ in the worst case, the communication complexity of f is the cost of the communication protocol with the least cost. The **one-way communication complexity** of a function f is the communication complexity of f when Player A is allowed to transmit information to Player B but Player B cannot transmit information to Player A. For example, consider the following problem referred to as **Bit Vector Probing** in [HRR00]:

Definition 2.1 *In the* **Bit Vector Probing (BVP)** *problem, Player A is given a bit string x of length n, while Player B is given an index $i, 1 \leq i \leq n$. The objective of the two players is to output the i-th bit. In other words, $f(x, i) = x_i$.*

The (two-way) communication complexity of this problem is $\log n$ since B can simply send A the index i and A can output x_i. The one-way communication complexity where B is prohibited from sending any information to A is n. (See [KN96] for details on this and other results in communication complexity.)

In [HRR00], results from communication complexity are used to study several graph problems related to database queries as well as other problems relating to databases. In each of the graph problems, a directed multigraph is given as input. The vertices of the graph can be partitioned into k sets, $V_1, V_2, \ldots, V_k$, such that each edge is directed from a node in V_i to a node in V_{i+1} for

some $i, 1 \leq i < k$. Four different types of queries are considered:

- Max: Let u_1 be the node of largest degree in V_1. Let $u_i \in V_i$ be a node of largest degree among those incident to u_{i-1} for $2 \leq i \leq k$. Find u_k.
- MaxNeighbor: Let u_1^* be the node with the largest outdegree in V_1. Let u_i^* be the node of largest degree incident on u_{i-1}^* for $2 \leq i \leq k$. Find u_k^*.
- MaxTotal: Find a node $u_1 \in V_1$ such that u_1 is connected to the largest number of nodes of V_k.
- MaxPath: Find nodes $u_1 \in V_1, u_k \in V_k$ such that they are connected by the largest possible number of paths.

In [HRR00], Henzinger et al prove that a 1-pass streaming algorithm requires $\Omega(kn^2)$ space to solve any one of these problems where $n = \max_{1 \leq i \leq k} |V_i|$. They also prove that a p-pass algorithm for MAX requires $\Omega(kn^2/p)$ space and provide a p-pass, $O(kn^2 \log n/p)$-space algorithm for MAX. For the remaining three problems, 1-pass, $O(kn^2 \log n)$-space algorithms are provided.

Rather than restate the proofs from [HRR00], we provide a lower bound proof for a more familiar graph problem, namely determining whether a graph is bipartite, to provide the flavor of the communication complexity arguments for space lower bounds on streaming algorithms. We can reduce BVP to Bipartiteness as follows:

Given a bit string x of length n, Player A adds an edge (u, u') to the stream, and, for each j, $1 \leq j \leq n$, such that $x_j = 1$, Player A adds an edge (u, v_j). Player A then runs the hypothetical streaming algorithm for Bipartiteness on the stream, pausing when all edges have been read. Player A then passes the contents of the streaming algorithm's memory to Player B. Player B then takes the query i, initializes the streaming algorithm with the memory contents provided by Player A, and runs the streaming algorithm on the stream consisting of a single edge (v_i, u') effectively restarting the streaming algorithm from where Player A left off. Player B outputs 0 if the streaming algorithm indicates that the graph is bipartite, and outputs 1 otherwise.

To see that Player B outputs the correct result notice the graph described by the entire stream consists of vertices $V = \{u, u', v_1, v_2, \ldots, v_n\}$ and edges $E = \{(u, u'), (u', v_i)\} \cup \{(u, v_j) : x_j = 1, 1 \leq j \leq n\}$. (See Figure 1.) If $x_i = 1$, then (u, u'), (u', v_i), and (u, v_i) are all edges in the graph which implies that the graph is not bipartite. If $x_i = 0$, then none of the v_j's are adjacent to u'. In this case, the graph is bipartite with u on one side of the partition and all the other vertices

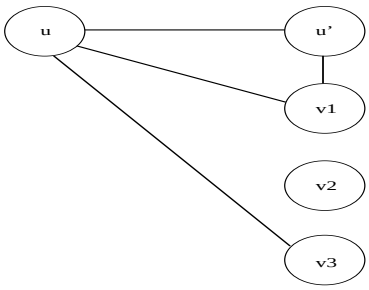

Figure 1: The graph constructed by the reduction of BVP to Bipartiteness when $x = 101$ and $i = 1$. Note that the graph is not bipartite because of the triangle $(u, u'), (u', v_1), (v_1, u)$.

on the other side. Therefore, the graph constructed is bipartite if and only if $x_i = 0$. Since the one-way communication complexity of BVP is n, Player A must have passed n bits of information to Player B in the worst case. Therefore, any streaming algorithm for bipartiteness must use n space in the worst case where n is the number of vertices in the graph.

Many natural graph problems share a characteristic with bipartiteness that leads to similar reductions from BVP and consequently the same lower bound ([FKM+05b]).

Definition 2.2 ([FKM+05b]) *A graph property P is* **balanced** *if there exists a constant $c > 0$ such that, for all sufficiently large n, there exists a graph $G(V, E)$ with $|V| = n$ and a vertex $u \in V$ such that:*

$$\min\left\{|V_{P,u}|, \left|\overline{V_{P,u}}\right|\right\} \geq cn$$

where $V_{P,u} = \{v \in V : G'(V, E \cup \{(u, v)\})$ has property $P\}$ and $\overline{V_{P,u}} = V - V_{P,u}$

Theorem 2.1 ([FKM+05b]) *In the 1-pass streaming model, testing a graph for any balanced graph property requires $\Omega(n)$ space where n is the number of vertices in the graph.*

In [FKM+05b], Feigenbaum et al point out that many graph properties such as including a vertex of a given degree, bipartiteness, and including a path between a given pair of vertices are balanced. They conclude that the 1-pass streaming model with an $o(n)$ space limitation does not provide enough power for graph problems. This motivates interest in the Semi-Streaming, W-Stream, and Stream-Sort models. We consider each of these in turn.

3 Semi-Streaming Model

In the Semi-Streaming model ([FKM+05a]), the algorithm is allowed to use $O(n \log^k n)$ space where n is the number of vertices and $k \geq 1$ is a constant. The following Semi-Streaming algorithms are presented in [FKM+05a]:

- a 1-pass, $O(n \log n)$ space algorithm for computing a bipartition for a graph if one exists,
- a 1-pass $O(n \log n)$ space algorithm for finding a maximal matching and, therefore, a 1/2-approximation to a maximum matching,
- for any $\epsilon, 0 < \epsilon < 1/3$, an $O(\frac{\log 1/\epsilon}{\epsilon})$-pass, $O(n \log n)$-space, $(2/3 - \epsilon)$-approximation algorithm for finding a maximum matching in a bipartite graph,
- a 1-pass, $O(n \log n)$-space, (1/6)-approximation algorithm for finding a weighted matching.

Additionally, it is mentioned that an algorithm from [UC00] can be adapted to give an $O(\log_{1+\epsilon/3} n)$-pass, $O(n \log n)$-space, $1/(2+\epsilon)$-approximation algorithm for weighted matching.

One of the key computational benefits of the Semi-Streaming model is that there is enough space to store the connected components of a graph using a disjoint-set data structure ([CLRS01]). Furthermore, the connected components can be computed in a single pass. In [FKM+05a], the connected components of the graph are used in a 1-pass algorithm to determine whether a graph is bipartite. The disjoint set data structure is augmented to include a color for each vertex, either Red or Blue. The algorithm proceeds as follows:

1. Initially, color each vertex Red.
2. For each edge (u, v) in the stream,
   (a) if u and v are in the same connected component and u and v have the same color, return failure. (The graph cannot be bipartite in this case since the color of any vertex in a connected component uniquely determines the color of every other vertex in that connected component. In this particular case, if u and v have previously been assigned the same color, the graph is not 2-colorable, i.e, bipartite.)
   (b) if u and v are in different connected components then
      i. if u and v have the same color, flip the color of all the vertices in v's component
      ii. merge the two connected components

3. return the coloring as the bipartition of the graph.

The algorithm runs in a single pass and uses $\alpha(m, n)$ time per edge where α is the inverse Ackermann function (see [CLRS01]).

Maintaining the connected components can also be used in an algorithm for computing the minimum spanning tree for a graph ([FKM+05a]). The algorithm is a streaming version of an algorithm which appears as a remark in [T83]. As we read the edges from the stream, we keep track of the connected components in memory. For each connected component, we also maintain a minimum spanning tree (MST), which can be stored in $O(n \log n)$ space since the total number of edges in all the spanning trees is $O(n)$. The complete algorithm is as follows:

1. For each edge (u, v) in the stream
   (a) if u and v are in different components then union the two components and create a minimum spanning tree for this new larger component by merging the two components' minimum spanning trees and adding edge (u, v).
   (b) else add (u, v) to the MST for the component (creating a cycle) and remove the heaviest edge on the cycle created.
2. Assuming the graph is connected, only one component remains. Return the corresponding spanning tree as the result.

3.1 Spanners

Just as the connected components are a useful representation of a graph that maintain connectivity information, a spanner is a representation of a graph that maintains approximate distance information. Formally, we have the following definition:

Definition 3.1 *An* (α, β)**-spanner** *for a graph* $G(V, E)$ *is a subgraph* $H(V, E')$ *such that* $E' \subseteq E$ *and for any vertices* $x, y \in V$, $d_G(x, y) \leq d_H(x, y) \leq \alpha d_G(x, y) + \beta$ *where* $d_G(x, y)$ *and* $d_H(x, y)$ *are the distances from* x *to* y *in graphs* G *and* H *respectively. When the additive constant* β *is 0, we simply refer to the spanner as an* α*-spanner.*

A Semi-Streaming algorithm to find a $(\log n)/(\log \log n)$-spanner is mentioned in [FKM+05a]. An algorithm for constructing a $(1 + \epsilon, \beta)$-spanner of size $O(n^{1+\delta})$ in a constant number of passes and using $O(n^{1+\delta} \log n)$ space appears in [EZ04]. A randomized Semi-Streaming algorithm that

constructs a $(2t+1)$-spanner for an unweighted graph in one pass appears in [FKM+05b]. With probability $1 - \frac{1}{n^{\Omega(1)}}$ the algorithm uses $O(tn^{(1+1/t)} \log^2 n)$ bits of space. The per edge processing time is $O(t^2 n^{(1/t)} \log n)$. Using this algorithm, the all-pairs distances in a graph can be $(2t+1)$-approximated. An extension to the spanner algorithm that provides a $((1+\epsilon)(2t+1))$-spanner for weighted undirected graphs is also presented in [FKM+05a]. The idea behind the algorithm is to make sure that for every edge $(u, v) \in E$ that is not included in the spanner, the distance between u and v is at most $2t+1$. This ensures that the spanner is a $(2t+1)$-spanner since, every edge in a shortest path in the original graph can be replaced by a path of length $(2t+1)$ in the spanner. The idea is to maintain trees of height $\leq \lfloor t/2 \rfloor$ that represent dense parts of the graph, connections between trees of height $= \lfloor t/2 \rfloor$, and miscellaneous edges to represent sparse parts of the graph. Since the height of each tree is $\lfloor t/2 \rfloor$, the distance between any two vertices within the same tree is $\leq t$. The key to the algorithm is to ensure that for any edge (u, v) in the graph, u and v are either in the same tree, in two trees that are connected by a single edge, or connected by a path including miscellaneous edges that is not too long. The complete algorithm has a randomized labeling procedure to achieve the desired results. The reader is referred to [FKM+05b] for details.

3.2 Sparsification

In [FKM+05b], Feigenbaum et al suggest a general approach for designing algorithms for the Semi-Streaming model and improving the running time per edge using the idea of sparsification from [EGI+97]. Sparsification is a technique for designing dynamic algorithms to test graph properties. It relies on the property having a strong certificate. A *strong certificate* for a property P and graph G is a graph G' on the same set of vertices such that for any graph H, $G \cup H$ has property P if and only if $G' \cup H$ has property P. A strong certificate G' for property P and graph G is said to be *sparse* if G' has at most cn edges for some constant $c > 0$ where n is the number of vertices. Since the Semi-Streaming model provides enough space to store a sparse certificate and $O(n)$ edges, we can read $O(n)$ edges at a time while maintaining a sparse certificate for the property in question. This gives us the following theorem:

Theorem 3.1 ([FKM+05b]) *Let P be a property for which a sparse certificate can be found in $f(n, m)$ time where n is the number of vertices and m is the number of edges in the graph. There is a 1-pass Semi-Streaming algorithm that maintains a sparse certificate for P using $f(n, O(n))/n$ time per edge.*

This result and the work by Eppstein et al ([EGI+97]) leads to Semi-Streaming algorithms for bipartitenes, connected components, and minimum spanning tree with better time per edge than previous algorithms. It also results in new Semi-Streaming algorithms for 2, 3, and 4-vertex connectivity, 2, 3, and 4-edge connectivity, and for constant edge connected components.

4 W-Stream

While the Semi-Streaming model extends the streaming model by allowing more space, in the W-Stream model ([R03]), algorithms are allowed to write as well as read streams. In each pass, the last stream written (or, in the case of the first pass, the input stream) is read and a new stream is written. This certainly seems like a reasonable extension to the streaming model as long as we do not allow the streams to grow too large. Therefore, the size of the streams written will be restricted to be within a constant factor of the size of the original input stream. While adding the ability to write intermediate streams would appear to provide more power to the streaming model, a p-pass W-Stream algorithm using s space can be simulated by a p-pass streaming algorithm using $p \times s$ bits of space ([R03]). Thus, when the number of passes is relatively small, the ability to write intermediate streams does not provide significantly more computational power. For example, a $(\log n)$-pass, $(\log n)$-space, W-Stream algorithm can be simulated by a $(\log n)$-pass, $(\log^2 n)$-space, streaming algorithm. In [DFR06], however, Demetrescu et al prove that when the space is $O(1)$, there are problems that can be solved using intermediate streams that cannot be solved without intermediate streams no matter how many passes are allowed. They also suggest that the total number of items processed is a better measure of the computational complexity of algorithms in the W-Stream model than the number of passes. They prove that the following problem, called FORK, can be solved with an $O(\log n)$-space W-Stream algorithm while processing $O(n)$ items. An $O(\log n)$-space streaming algorithm, on the other hand, would require $\Omega(\log n)$ passes and, therefore, processes $\Omega(n \log n)$ items ([DFR06]).

Definition 4.1 ([DFR06]) *In the* **FORK** *problem, we are given two vectors x and y of n numbers each such that $x_1 = y_1$ but $x_n \neq y_n$. Our objective is to find an index i, called a fork, such that $x_i = y_i$ but $x_{i+1} \neq y_{i+1}$.*

While reading and writing a single stream as in the W-Stream model certainly seems like a reasonable extension to the streaming model it also seems to be a bit too restrictive. Why not allow multiple streams to be read and written simultaneously? For example, the FORK problem

which requires $p \times s = \Omega(\log^2 n)$ passes in W-Stream ([DFR06]), can be solved in $O(\log n)$ space and two passes if we are allowed to read and write two streams. In the first pass, simply write A to one stream and B to another. In the second pass, move through A and B in lock step looking for the first index i at which $A[i] \neq B[i]$. In [R03], Ruhl points out that these types of algorithms have been studied previously as tape-based algorithms ([K98]). Ruhl proposes an alternative model called *Streaming Networks* which is equivalent in power to the tape model and proves several results relating the power of Streaming Networks to the models discussed in this chapter. A complete discussion of the Streaming Network model is beyond the scope of this chapter.

In addition to the separation results above, two algorithms are presented in [DFR06] along with a detailed analysis of the tradeoff between the number of passes versus the amount of space required by the algorithms. The problems considered are finding the connected components of an undirected graph and directed single-source shortest path. For both problems, any W-Stream algorithm using s bits of memory requires $\Omega(n/s)$ passes ([DFR06]). (Throughout the remainder of this section, we take n to be the number of vertices and m to be the number of edges in the graph.)

4.1 Connected Components in W-Stream

A deterministic W-Stream algorithm for finding the connected components of an undirected graph that uses s space and $O((n \log n)/s)$ passes is presented in [DFR06]. The idea behind the algorithm is to repeatedly find the connected components of subgraphs of G where these subgraphs are small enough to fit into memory. Once such a connected component is found, it can be compressed into a single vertex for the next iteration. As groups of vertices get compressed, a representation of the original vertices within each connected component must be maintained. Therefore, there will be two parts to the intermediate streams. Part A will hold the edges from the compressed graph. Part B will contain a representation of the connected components found so far. This representation will consist of an edge from each original vertex v to the representative vertex for the component containing v.

The algorithm repeats the following until there are no more edges in the graph:

1. Read edges from Part A of the stream, storing each new vertex encountered in memory. As edges are read, maintain the connected components seen so far by maintaining a spanning forest. Continue reading edges until the internal memory is exhausted or until all the edges from Part A have been read. Let H be the subgraph induced by the edges that have been

read and stored in memory.

2. Read all remaining edges from Part A of the stream if any. As each such edge (u, v) is read, determine whether u and v are part of the same connected component in H. Let

$$c(x) = \begin{cases} \text{vertex representing the component in } H \text{ containing } x & \text{if } x \in H \\ x & \text{otherwise} \end{cases}$$

If $c(u) \neq c(v)$, write $(c(u), c(v))$ as an edge in the new stream for the compressed graph. In this step, all the connected components are compressed down to their representatives. Any vertex in G that is adjacent to a vertex in a connected component from H will now be adjacent to the representative of the component.

3. At this point the connected components of H have been found and the compressed graph has been written to the new stream. It is possible that some vertices that were previously representatives of connected components in G are no longer representatives since their components could have been merged into a larger component. If this is the case, we need to update Part B of the stream so that every vertex in such a connected component points to the representative of the new larger component. To do this, read the edges from Part B of the stream. As each edge (u, v) is read, write $(u, c(v))$ to Part B of the new stream and, if $u \in H$, mark u in memory as having been written to Part B of the new stream.

4. The entire stream has been read and most of the connected component information has been written to the new stream. However, there may be vertices in H whose connected component information have not been written to the new stream – vertices in H that were not included in the connected component information for G after the previous iteration. To remedy this, find each vertex $u \in H$ that was not marked as having been written to Part B of the new stream in Step 3 and write $(u, c(u))$ to Part B of the new stream.

Using the reduction of connected components to bipartiteness from [R03], this algorithm can be extended to provide an $O((n \log n)/s)$-pass, s-space, W-Stream algorithm for bipartiteness. It is an open question whether this W-Stream algorithm for connected components can be extended to find a minimum spanning tree as was done with the Semi-Streaming algorithm for connected components.

4.2 Single-Source Shortest Paths

A randomized W-Stream algorithm for the directed single source shortest path problem restricted to instances in which the edge weights are positive integers bounded by some number C appears in [DFR06]. With s space, the algorithm makes $p = O(Cn \log^{3/2} n/\sqrt{s})$ passes. The distances returned are correct with high probability and the size of each intermediate stream is $O(m + n\sqrt{s/\log n})$. The idea behind the algorithm is that, given any constant l, a shortest path between any two vertices can be viewed as a sequence of shortest paths between intermediate pairs of vertices where each of the intermediate shortest paths has length $\leq l$. If we can find shortest paths of length up to l for all pairs of vertices, we can splice these shortest paths together to get shortest paths of length $> l$. There is no known algorithm for finding shortest paths between all pairs of vertices even for limited path lengths in the W-Stream model so instead a random subset of vertices is chosen to serve as the intermediate sources. A streaming version of Dijkstra's algorithm can be run on each of these intermediate sources. We will explain this streaming version of Dijkstra's algorithm since the ideas used may be applicable to other problems.

4.2.1 Finding distances up to l from a single source using Dijkstra's algorithm.

Recall that in each iteration of the main loop in Dijkstra's algorithm, the vertex u with minimum estimated distance from the source is extracted from the priority queue. Each edge (u, v) out of u is then "relaxed", i.e. the estimated distance to v is reduced to the distance of the shortest path to v over the edge (u, v) if this distance is smaller than the previous estimated distance from the source to v. (See [CLRS01] for a complete description of Dijkstra's algorithm). The difficulty in implementing Dijkstra's algorithm when the space is restricted to $o(n)$ is that the priority queue cannot be stored in memory. Instead the priority queue must be stored on the intermediate streams created during each pass through the data. To simplify the exposition of the algorithm, let's first consider the case where we want to find all shortest paths of length $\leq l$ from a single source, a. If we have the priority queue stored in the stream, extracting the vertex with minimum priority can be done in one pass. However, relaxing all the edges coming out of that vertex would be problematic if the priority queue were simply appended to the stream of edges. (Since relaxing a single edge would require a pass through the entire stream, the entire algorithm would require at least m passes through the stream in the worst case.) An alternative idea, would be to store a copy of the priority queue after each edge as a linear array containing the priority of each vertex. In

addition, we would have to store a boolean value with each priority to indicate whether or not the vertex has previously been removed from the queue. In other words, the stream would look like:

$$e_1\,[(p_1, f_1), \ldots, (p_n, f_n)]\; e_2\,[(p_1, f_1), \ldots, (p_n, f_n)]\; \ldots\; e_m\,[(p_1, f_1), \ldots, (p_n, f_n)]$$

where e_i is the i-th edge, p_j is the priority of $v_j \in V$, and f_j indicates whether v_j has been removed from the queue.

If we relax edge $e_1 = (u, v)$, we can update the priority for v in the copy of the priority queue that comes immediately after e_1. Now suppose $e_4 = (u, w)$ is the next edge in the stream coming out of u. This edge needs to be relaxed. As with edge e_1, we can relax e_4 by updating the priority of w in the copy of the priority queue coming immediately after e_4. Note that the various copies of the priority queues are now out of sync. However, each priority queue maintains the necessary invariant that the priority of any vertex is greater than the actual length of the shortest path to that vertex. The extract-min operation can still find the vertex with the minimum priority in one pass because the vertex on the queue with the minimum priority can be found regardless of whether it also appears with a larger priority elsewhere in the stream.

The algorithm as described would require intermediate streams of size mn. Notice, however, that the only priority that can be updated in the copy of the priority queue immediately following edge (u, v) is v. Therefore, we only need to keep a single priority after each edge in the stream. This reduces the size of the intermediate stream to $O(m)$.

We can optimize the algorithm further in terms of the number of passes by allowing the extract-min operation to find multiple vertices in the case there are ties for the minimum priority. In other words, the extract-min operation will create a pool of vertices in memory that consists of up to k vertices that have minimum priority where k is such that the space constraint is obeyed. In this way, the number of extract-min passes is bounded by $n/k + l$ since there can be at most n/k times in which the priority does not change from one pool to the next, and at most l times when the priority does change from one pool to the next (assuming the weights are positive integers). Since there is one relaxation pass for each extract-min pass, the total number of passes is $O(n/k + l)$.

4.2.2 Finding distances up to l from multiple sources using Dijkstra's algorithm.

To be useful in solving the single source shortest path problem by spicing together paths of length $\leq l$, the algorithm above must be extended to handle a set of sources A. This can be accomplished easily enough by maintaining $|A|$ priority queues in the same manner as was done with one source.

In other words, in the intermediate stream, after each edge we would write a priority for each source. If the number of vertices k allowed in each pool for the extract-min operation is $s/(|A| \log n)$, the total number of passes is $O(n/k + l) = O(n|A| \log n/s + l)$. The size of the intermediate streams is $O(m|A|)$. The size of the intermediate streams can be optimized to $O(m + n|A|)$ by preprocessing the stream to create many groups of contiguous edges in the stream where the endpoints of the edges in each group are the same. The priority queue information for a vertex is then maintained only after each group rather than after each edge. For complete details, see [DFR06].

4.2.3 The complete Single Source Shortest Path algorithm

Using the algorithm above as a subroutine, the general single source problem can be solved when edge weights are positive integers $\leq C$ for some number C as follows:

1. Randomly choose a set A of $\sqrt{s/\log n}$ vertices including the true source a_0 to serve as the intermediate sources.

2. Use the algorithm above to compute distances up to $l = (\alpha C n \log^{3/2} n)/\sqrt{s}$ from each of the vertices in A to every other vertex in the graph where α is any constant > 1. Let $d(a_i, v)$ be the distance from source $a_i \in A$ to vertex $v \in V$. This distance information is stored in memory.

3. Build a new graph G^* on the vertices in A such that there is an edge between a_i and a_j if and only if Step 2 found $d(a_i, a_j) \leq l$. Set the weight of this edge to $d(a_i, a_j)$. G^* can be created in one pass and stored in memory as an adjacency matrix using $|A|^2 \log n = s$ space. This graph gives us a concise representation of the shortest paths computed in Step 2 between the sources in A. These are the paths that we will splice together to create paths of length $> l$ in the original graph.

4. Compute the shortest paths from source a_0 in G^* using any single source shortest path algorithm. Let $d^*(a_0, a_i)$ be the distance from a_0 to a_i in G^* for each i.

5. For any vertex v for which we did not compute a shortest path $\leq l$ in step 2, we set its distance to $\min_{a \in A}\{d^*(a_0, a) + d(a, v)\}$.

All distances $\leq l$ are computed correctly in step 2 since a_0 is one of the vertices in A. Distances of length $> l$ could be computed incorrectly. For example, if a poor choice of random vertices

for A resulted in a vertex $v \in V$ being a distance of more than l away from each $a \in A$, the distance computed for v would be ∞. Demetrescu et al prove, however, that with probability at least $1 - 1/n^{\alpha-1}$, each distance $> l$ is computed correctly in Step 5. The number of passes required is $O(\alpha C n \log^{3/2} n/\sqrt{s})$.

5 The Stream-Sort Model

It is argued in [R03] and [ADR+04] that the crucial limitation of the streaming model is not in its inability to write intermediate streams but in its inability to write sorted intermediate streams. They prove that a p-pass, s-bit, W-Stream algorithm can be simulated by a p-pass, $(p \times s)$-bit, streaming algorithm and that sorting a stream can be accomplished by a p-pass, s-space, W-Stream algorithm only if $p \times s \geq$ the size of the stream ([R03]). The latter result is a consequence of a result from [BJK+04]. Since sorting large data sets can be done efficiently with today's hardware ([V01]), adding a sorting primitive to the W-stream model seems appropriate. This leads to the Stream-Sort model. In each pass through the data, we can either produce an intermediate stream as in the W-Stream model or sort the stream according to some partial order on the items in the stream. The partial order must be computable on a Turing machine M with an s-bit memory, i.e., M is given two items and returns the order of the items. M does not maintain an internal state between comparisons. On the other hand, we explicitly allow the local memory to be maintained between streaming passes.

In [R03] and [ADR+04], it is suggested that an algorithm in the Stream-Sort model be considered efficient if the number of streaming and sorting passes is $O(\log^k n)$ for some constant k. Efficient Stream-Sort graph algorithms for undirected s-t-connectivity, directed s-t-connectivity, bipartiteness, minimum spanning tree, maximal independent set, tree contraction, detecting cycles in an undirected graph, and minimum cut appear in [R03] and [ADR+04]. Each of the algorithms is randomized. As with the models discussed in the previous sections, we present a Stream-Sort algorithm for connected components, which is from [R03] and [ADR+04].

5.1 Connected Components in Stream-Sort

Although the algorithm presented in [R03] and [ADR+04] is for s-t-connectivity, we amend it slightly to output the complete connected components of a graph. The final stream created by this algorithm will have the connected components listed as pairs of vertices and component labels

similar to the format used in [DFR06] and discussed in Section 4.

The algorithm repeats the following until there are no more edges in the graph.

1. Assign a random $3 \log n$ bit integer to each vertex in the graph.
2. Label each vertex with the minimum number among those assigned to itself and its neighbors.
3. Merge all vertices that receive the same label.
4. If a representative of a connected component is merged then update the representative for all vertices in the corresponding component.

The idea is that by assigning random labels to each vertex and merging vertices based on neighboring labels, the number of vertices in the graph decreases by a constant factor in expectation during each iteration ([ADR+04]). This implies the expected number of iterations is $O(\log n)$. As we will see below, each iteration requires a constant number of passes through the stream.

Assume the input stream is given as a list of vertices followed by a list of edges and that each edge (u, v) appears as both (u, v) and (v, u) since the edges are undirected. The edges are assumed to be in no particular order. The computational benefit of sorting is that it gives us the ability to group together information that may be distributed in a variety of places in the stream. For example, when looking for the minimum label among a vertex v's neighbors, we need to find all the edges incident on v and the labels of those vertices, yet the edges incident on v do not necessarily appear near each other in the stream.

The implementation of this algorithm in the Stream-Sort model is described below. For concreteness, we provide an example for the first iteration on a graph consisting of a straight line through four vertices in Table 1.

Repeat the following until there are no more edges in the graph:

1. In one pass over the stream, replace each vertex v_i by a vertex-label pair (v_i, l_i) where l_i is the random number assigned to v_i. If this is the first iteration, append the initial connected component information with each vertex serving as its own representative.
2. For each vertex, we need to find the lowest label assigned to itself or its neighbors. To do this efficiently, we would like to group the labels for all of a vertex's neighbors together in the stream. We proceed as follows:

(a) Sort the stream so that the list of edges emanating from v_i appear right after the vertex label (v_i, l_i) for each vertex v_i. In addition, move the connected component information to the end of the stream.

(b) In one pass over this new stream, add an "edge label" (l_i, u) following each edge (v_i, u). This edge label indicates that u is adjacent to a vertex labeled l_i.

(c) Sort again to group these edge labels with the vertex label for the endpoints of the original edges. In other words, vertex label (v_j, l_j) will be followed by the list of edge labels (l_i, v_j) for edges coming into v_j. Put the actual edges at the end of the stream to get them out of the way.

(d) Now in one linear pass over the stream determine the smallest number assigned to v_i or its neighbors for each i. This can be done in one pass since each vertex label (v_i, l_i) is followed by a list consisting of an edge label (l_j, v_i) for each vertex v_j adjacent to v_i. In other words, v's label and the labels of all its neighbors are now grouped together in the stream. This number becomes the new label for v_i Update each vertex-label pair in the stream to reflect the new labeling.

3. Update the connected component information as follows:

(a) Sort the stream so that the list of component pairs (v_i, v_r) appears immediately after the vertex label (v_r, l_r) for the representative vertex v_r.

(b) Update the representative in each component pair to be the new label for that representative. In other words, the component pair (v_i, v_r) becomes (v_i, l_r). Note that the labels become the vertices in the next iteration.

4. We need to update the edges so that edge (v_i, v_j) is replaced by edge (l_i, l_j) in the compressed graph. This can be done as follows:

(a) Sort the stream so that edges emanating from v_i appear right after vertex label (v_i, l_i) and the connected component information appears at the end of the stream.

(b) In one pass over this stream, produce a new stream of edges where each edge (v_i, u) is replaced by (l_i, u). This updates the first component of each edge in the compressed graph for the next iteration.

(c) Now sort this stream so that each vertex label (v_j, l_j) is followed by the list of edges (l_i, v_j) coming into v_j.

(d) In one pass over this stream, produce a new stream of edges where each edge (l_i, v_j) is replaced by (l_i, l_j). This updates the second component of each edge in the compressed graph for the next iteration. Also remove self-loops of the form (l_i, l_i) and replace each vertex label by the label alone – the labels becoming the vertices for the next iteration. In a second pass, remove duplicate vertices.

In the end, the remaining labels in the stream represent the connected components and each pair (v_i, l_i) in the connected components portion of the stream indicates the connected component in which v_i is a member. In the example from Table 1, notice that after the first iteration, we have determined that v_1, v_2, and v_3 are in the same connected component which is represented by the new vertex 10. In the next iteration, we will find that v_4 is also in this same connected component (although the representative could be 30 depending on how the random numbers are assigned).

Since the expected number of iterations is $O(\log n)$ and each iteration requires a constant number of passes, the expected number of passes is $O(\log n)$. Since the only information required to be stored in memory at any one time is a small number of labels and vertices, the total amount of space used is also $O(\log n)$. Recall that in W-Stream this problem requires $\Omega(n/s)$ passes using s space. Having the ability to sort, therefore, provides a significant computational advantage for this problem. This algorithm is used in [R03] and [ADR+04] as a subroutine in Stream-Sort algorithms for bipartiteness, directed s-t-connectivity, and minimum spanning tree.

6 Summary and Conclusions

We have discussed a number of graph algorithms for the three main extensions to the streaming model – Semi-Streaming, W-Stream, and Stream-Sort. In each model there is a need to maintain some compressed representation of the information contained in the graph. For example, maintaining connectivity information has proven extremely useful in all three models. In [R03], a number of open problems for the Stream-Sort model are listed including developing algorithms for breadth first search, depth first search, topological sort, and directed connectivity. The W-Stream algorithm from [DFR06] for single source shortest path discussed above can be used for breadth-first search and can achieve a sub-linear number of passes using sublinear space. (By taking $s = \sqrt{n}\log^3 n$ for example, the number of passes required would be $n^{3/4}$.) It would be interesting to see whether sorting can be used effectively to bring this down to polylog passes and space.

Interest in the streaming model of computation is not likely to subside anytime soon especially

Table 1: A trace of the first iteration of the connected components algorithm for the graph consisting of a straight line through four vertices, v_1, v_2, v_3, v_4. Since ordered pairs are being used to describe edges, labelings, and component information, we will differentiate between each of these types of pairs. We use $e(v_1, v_2)$ to represent the edge (v_1, v_2), $C(v_1, v_r)$ to represent that v_1 is in the connected component represented by v_r, and $L_V(v_1, l_1)$ and $L_E(l_1, v_1)$ to represent the vertex-label and edge-label pairs used in the algorithm. To save space, let $[E]$ be the complete list of edges unchanged from the previous step and $[C]$ be the complete list of connected component pairs unchanged from the previous step.

Step	Stream
Initial	(v_1) (v_2) (v_3) (v_4) $e(v_1, v_2)$ $e(v_2, v_1)$ $e(v_2, v_3)$ $e(v_3, v_2)$ $e(v_3, v_4)$ $e(v_4, v_3)$
1	$L_V(v_1, 20)$ $C(v_1, v_1)$ $L_V(v_2, 10)$ $C(v_2, v_2)$ $L_V(v_3, 30)$ $C(v_3, v_3)$ $L_V(v_4, 40)$ $C(v_4, v_4)$ $[E]$
2a	$L_V(v_1, 20)$ $e(v_1, v_2)$ $L_V(v_2, 10)$ $e(v_2, v_1)$ $e(v_2, v_3)$ $L_V(v_3, 30)$ $e(v_3, v_2)$ $e(v_3, v_4)$ $L_V(v_4, 40)$ $e(v_4, v_3)$ $[C]$
2b	$L_V(v_1, 20)$ $e(v_1, v_2)$ $L_E(20, v_2)$ $L_V(v_2, 10)$ $e(v_2, v_1)$ $L_E(10, v_1)$ $e(v_2, v_3)$ $L_E(10, v_3)$ $L_V(v_3, 30)$ $e(v_3, v_2)$ $L_E(30, v_2)$ $e(v_3, v_4)$ $L_E(30, v_4)$ $L_V(v_4, 40)$ $e(v_4, v_3)$ $L_E(40, v_3)$ $[C]$
2c	$L_V(v_1, 20)$ $L_E(10, v_1)$ $L_V(v_2, 10)$ $L_E(20, v_2)$ $L_E(30, v_2)$ $L_V(v_3, 30)$ $L_E(10, v_3)$ $L_E(40, v_3)$ $L_V(v_4, 40)$ $L_E(30, v_4)$ $[C]$ $[E]$
2d	$L_V(v_1, 10)$ $L_V(v_2, 10)$ $L_V(v_3, 10)$ $L_V(v_4, 30)$ $e(v_1, v_2)$ $e(v_2, v_1)$ $[C]$ $[E]$
3a	$L_V(v_1, 10)$ $C(v_1, v_1)$ $L_V(v_2, 10)$ $C(v_2, v_2)$ $L_V(v_3, 10)$ $C(v_3, v_3)$ $L_V(v_4, 30)$ $C(v_4, v_4)$ $[E]$
3b	$L_V(v_1, 10)$ $C(v_1, 10)$ $L_V(v_2, 10)$ $C(v_2, 10)$ $L_V(v_3, 10)$ $C(v_3, 10)$ $L_V(v_4, 30)$ $C(v_4, 30)$ $[E]$
4a	$L_V(v_1, 10)$ $e(v_1, v_2)$ $L_V(v_2, 10)$ $e(v_2, v_1)$ $e(v_2, v_3)$ $L_V(v_3, 10)$ $e(v_3, v_2)$ $e(v_3, v_4)$ $L_V(v_4, 30)$ $e(v_4, v_3)$ $[C]$
4b	$L_V(v_1, 10)$ $e(10, v_2)$ $L_V(v_2, 10)$ $e(10, v_1)$ $e(10, v_3)$ $L_V(v_3, 10)$ $e(10, v_2)$ $e(10, v_4)$ $L_V(v_4, 30)$ $e(30, v_3)$ $[C]$
4c	$L_V(v_1, 10)$ $e(10, v_1)$ $L_V(v_2, 10)$ $e(10, v_2)$ $e(10, v_2)$ $L_V(v_3, 10)$ $e(10, v_3)$ $e(30, v_3)$ $L_V(v_4, 30)$ $e(10, v_4)$ $[C]$
4d	(10) (30) $e(30, 10)$ $e(10, 30)$ $C(v_1, 10)$ $C(v_2, 10)$ $C(v_3, 10)$ $C(v_4, 30)$

in the data mining community. The algorithm for finding coherent threads in search results from [GKS+05] for example, is mentioned to be efficiently implementable in the Stream-Sort model. In addition, Ruhl points out in [R03], that the implementation of the PageRank algorithm discussed in [PBM+99] can be done efficiently in the Stream-Sort model. As more people investigate the streaming model, consideration of graph algorithms in streaming model variants will become increasingly common.

Acknowledgments

I would like to thank Mike Eckmann for many helpful comments and suggestions.

References

[ADR+04] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. In *FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, pages 540–549, Washington, DC, USA, 2004. IEEE Computer Society.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[BJK+04] Z. Bar-Yossef, T. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[BKS02] Ziv Bar-Yosseff, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[BGW03] Adam L. Buchsbaum, Raffaele Giancarlo, and Jeffery R. Westbrook. On finding common neighborhoods in massive graphs. *Theor. Comput. Sci.*, 299(1-3):707–718, 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[DFR06] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 714–723, New York, NY, USA, 2006. ACM Press.

[EZ04] Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1 + \epsilon, \beta)$-spanners in the distributed and streaming models. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 160–168, New York, NY, USA, 2004. ACM Press.

[EGI+97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification: a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[FKM+05b] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: the value of space. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 745–754, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[FKM+05a] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[GKS+05] R. Guha, Ravi Kumar, D. Sivakumar, and Ravi Sundaram. Unweaving a web of documents. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 574–579, New York, NY, USA, 2005. ACM Press.

[HRR00] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science: '"External Memory Algorithms"*, volume 50, pages 107–118, 2000.

[K98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[KN96] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, December 1996.

[MP91] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980.

[M03] S. Muthukrishnan. Data streams: algorithms and applications. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 413–413, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[PBM+99] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford Digital Libraries, November 1999.

[R03] Matthias Ruhl. *Efficient Algorithms for New Computational Models*. Ph.D. Thesis, MIT, Cambridge, MA, USA, 2001.

[T83] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[UC00] Ryuhei Uehara and Zhi-Zhong Chen. Parallel approximation algorithms for maximum weighted matching in general graphs. *Information Processing Letters*, 76(1–2):13–17, 2000.

[V01] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[Y79] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing(preliminary report). In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 209–213, New York, NY, USA, 1979. ACM Press.