Graph Streaming Algorithms

January 5, 2017

1 Introduction

In this class, we have encountered streaming algorithms for computing properties of a sequence of inputs with limited memory. One area of interest is in graph streaming algorithms, where the edges of a graph arrives in a stream and the algorithm computes properties of the graph such as connectivity, bipartiteness or the size of a maximum matching using space sublinear in the number of edges, which is $O(n^2)$ for a graph with n vertices. It can be proved [1] that in an adversarial stream (an adversary gets to choose what order the edges arrive in), at least $\Omega(n \log n)$ space is needed to compute properties such as connectivity and bipartiteness. Therefore, most of the algorithms studied in this survey use $\tilde{O}(n)$ space, where $\tilde{O}$ hides $\log n$ factors. In fact, for the model mentioned above (insertion of edges, adversarial stream), we have seen algorithms using spaces that match these lower bounds, e.g., by maintaining the spanning forest, greedy matching, bipartite-labeling, etc. Here we will look at streaming algorithms for two modifications of this basic model:

1. Allow edge deletions in the stream in addition to insertions
2. Instead of an adversarial stream, the edges in the stream are ordered randomly

These modifications are interesting in many ways. First, they are more relevant to real-world applications, since in reality graphs change over time (such as one adding and removing friends on a social network). Also, in many applications the average-case result is more important than the worst-case result. Analyzing a random stream order will tell us more about how much better we can do than in the worst case, more specifically if we can do better than the $\Omega(n \log n)$ space lower bound. The techniques used in these algorithms are newsworthy by their own interests, too. To handle edge deletions, [1] uses dimensional reduction methods to compute a sketch of a graph which is sublinear in the number of edges. For randomized streams, [5] analyzed the problem from the perspective of property testing, treating the stream as random measurements on the graph.

The results presented in this survey build on each other. We start with the method of graph sketching and demonstrate how one can use it to solve the graph connectivity problem. Next we focus on finding an approximation to maximum matching. First we use the graph sketching techniques presented previously for an algorithm in the adversarial stream setting and then we focus on the random edge arrival setting. Finally we will conclude by drawing some connections to the techniques used in these streaming algorithms.

[1] This lower bound is proved in Homework 3

2 Graph Sketching

A graph can be described by a stream of m edges on n nodes coming one by one. More generally, we consider dynamic graph streams where edges can be both added and removed. To store the whole graph can take $O(n^2)$ space. We want to use only $\tilde{O}(n)$ space to analyze structures for dynamic graphs, such as connectivity, bipartiteness, matching, etc.

2.1 Dynamic Streaming Model

First, let's generalize the streaming model a bit to deal with edge deletions by introducing weights:

Definition 1 (Data Stream)**.** *A stream σ has the form $\langle (x_1, \Delta_1), \cdots, (x_t, \Delta_t) \rangle$, where each x_i is an element from universe $[n]$ and $\Delta_i \in \mathbb{R}$ is x_i's weight.*

Definition 2 (Frequency Vector)**.** *σ's frequency vector $\mathbf{f} \in \mathbb{R}^n$ is defined as $\mathbf{f}_x = \sum_{i:x_i=x} \Delta_i$ for each $x \in [n]$.*

Namely, $\mathbf{f}$ is the current state of σ, and σ describes changes in $\mathbf{f}$. Recall the streaming model we have studied in class where all $\Delta_i = 1$. The number of distinct elements in σ is the 0-norm of $\mathbf{f}$, and the heavy hitters in σ are x's with large $\mathbf{f}_x$. We can ask more problems about $\mathbf{f}$. What is its p-norm? What is the median? Can we sample an element according to $\mathbf{f}$?

2.2 Linear Sketches

An important class of algorithms in data streaming are linear sketches. Such algorithms maintain a low dimensional sketch of the input stream while preserving its relevant properties. Formally,

Definition 3 (Linear Sketch)**.** *A linear sketch $\mathcal{S}$ is a random linear projection of $\mathbf{f}$ from $\mathbb{R}^n$ to $\mathbb{R}^k$ for some $k \ll n$.*

Our goal is to be able to infer some information about $\mathbf{f}$ from its low dimensional sketch $\mathcal{S}(\mathbf{f})$. The linearity of $\mathcal{S}$ makes it easy to combine two streams σ_1 and σ_2. To get $\mathcal{S}(\mathbf{f}(\sigma_1 \circ \sigma_2))$, where $\circ$ denotes concatenation, we simply add $\mathcal{S}(\mathbf{f}(\sigma_1))$ and $\mathcal{S}(\mathbf{f}(\sigma_2))$. Then we can update the current sketch on a newly coming element, or divide the whole stream into pieces for distributed processing.

Recall the Count-Min Sketch Algorithm which we have seen in class. It first chooses l independent random hash functions $h_1, \cdots, h_l$: $[n] \to [b]$, and then maintains l different b-dimensional vectors $CMS^1, \cdots, CMS^l$. Each $CMS^i_j = \sum_{x:h_i(x)=j} \mathbf{f}_x$. So they are random linear projections of $\mathbf{f}$, and thus Count-Min Sketch is a linear sketch from $\mathbb{R}^n$ to $\mathbb{R}^{lb}$. Specifically, it is used to estimate $\mathbf{f}_x$ within ϵn-additive error.

2.3 ℓ_0-Sampling

For a non-zero vector $\mathbf{x} \in \mathbb{R}^n$, an ℓ_0-sampler is an algorithm that takes in a stream of updates to $\mathbf{x}$ and in the end returns some element in the support of $\mathbf{x}$ uniformly at random.

Definition 4 (ℓ_0-Sampling)**.** *A δ-ℓ_0-sampler for a non-zero $\mathbf{x} \in \mathbb{R}^n$ fails with probability at most δ and otherwise returns some $i \in \text{supp}(\mathbf{x})$ with probability $\frac{1}{|\text{supp}(\mathbf{x})|}$.*

Due to Jowhari et al. [3], ℓ_0-sampling for the frequency vector $\mathbf{f}$ of a stream σ can be performed via linear sketches:

Lemma 1. *δ-ℓ_0-sampling can be performed in $O(\log^2 n \log \delta^{-1})$ space by a linear sketch based algorithm.*

We omit the full proof since it involves a lot of machinery out of the scope of this survey. Informally, the algorithm randomly chooses $\log n$ subsets of $[n]$, and restricts $\mathbf{x}$ to the coordinates in each subset. Some of the restricted version of $\mathbf{x}$ should be sparse in expectation, from which we easily identify elements in $\text{supp}(\mathbf{x})$. To store these subsets in a small space, the algorithm starts from a random seed in size $\text{polylog}(n)$, and uses a pseudorandom generator to get the random sets.

2.4 Sketching Graphs

Now we examine how to use linear sketches on graphs. One way to do it is to sketch the adjacency matrix A of a graph G. For each node v, $\mathbf{a}^v \in \{0,1\}^n$ denotes its neighbors, i.e. $\mathbf{a}^v_u = 1$ iff u is a neighbor of v. If we apply linear sketches for ℓ_0-sampling of $\mathbf{a}^v$, we can get a random neighbor of v by querying $\mathcal{S}(\mathbf{a}^v)$. When an edge is added or removed, we can easily update these sketches on account of linearity. However, if we repeatedly update and query $\mathcal{S}(\mathbf{a}^v)$, the random neighbors it returns are not independent (when there is no update, $\mathcal{S}$ just returns the same random neighbor). In particular, we are not allowed to use sketches adaptively. That is to say, we cannot query a neighbor u from $\mathcal{S}(\mathbf{a}^v)$, then remove u and query again to get another neighbor of v. Otherwise we could recover all neighbors of v from a $O(\log^2 n \log \delta^{-1})$ size sketch, which is clearly impossible. Therefore, we need to sketch graphs more carefully and cleverly, to avoid adaptively updating a sketch based on itself.

3 Connectivity

In this section, we present a single-pass streaming algorithm to test connectivity of dynamic graphs using $\tilde{O}(n)$ space. It is based on constructing linear sketches for ℓ_0-sampling of a graph representation we now define:

Definition 5 (Graph Representation)**.** *Given a graph $G = (V, E)$, for each node $i \in V$, define a vector $\mathbf{a}^i \in \{-1, 0, 1\}^{\binom{n}{2}}$:*

$$\mathbf{a}^i_{\{j,k\}} = \begin{cases} 1, & \text{if } i = j < k \text{ and } (j,k) \in E \\ -1, & \text{if } j < k = i \text{ and } (j,k) \in E \\ 0, & \text{otherwise} \end{cases}$$

The non-zero entries of $\mathbf{a}^i$ correspond to i's incident edges. Moreover, for any subset of nodes $S \subset V$, the non-zero entries of $\sum_{i \in S} \mathbf{a}^i$ correspond to the edges across the cut $(S, V \backslash S)$. This nice property can be naturally used in linear sketches: if we want to merge two nodes i and j, by adding $\mathcal{S}(\mathbf{a}^i)$ and $\mathcal{S}(\mathbf{a}^j)$, we get $\mathcal{S}(\mathbf{a}^i + \mathbf{a}^j)$ which exactly corresponds to the linear sketch of supernode (i, j).

Consider a simple local algorithm to count the number of connected components in a graph G by merging nodes. At first, let $\hat{V} = V$ be the initial set of supernodes. In each stage, find an edge from each supernode to another supernode if it is not isolated. Then merge all the resulting connected supernodes into one supernode. Continue this process until every supernode is isolated, so that each of them represents a connected component.

Suppose G has $cc(G)$ connected components. Let's analyze how many stages we need before the graph collapses to $cc(G)$ supernodes. After a stage, each supernode either comes from an isolated supernode, which means it already represents a connected component; or is merged from at least two supernodes. Therefore, the difference between the number of supernodes and $cc(G)$ halves. So after $t = O(\log n)$ stages we are done.

Now we convert the above local algorithm to a streaming algorithm:

Algorithm 1 Connected Components

1: Maintain t sketches $\mathcal{S}_1, \cdots, \mathcal{S}_t$ for $\mathbf{a}^1, \cdots, \mathbf{a}^n$
2: Initialize $\hat{V} \leftarrow V$
3: **for** $i = 1, \cdots, t$ **do**
4: For each $\hat{v} \in \hat{V}$, sample an incident edge of $\hat{v}$ by querying $\mathcal{S}_i \left(\sum_{v \in \hat{v}} \mathbf{a}^v\right)$
5: Merge connected supernodes and update $\hat{V}$
6: **return** $\hat{V}$

Note that we need to use t different sketches for each stage, to avoid using sketches adaptively, which is not allowed as we have elucidated before. It suffices to set δ to be a small constant, e.g. $1/100$, for ℓ_0-sampling. The algorithm uses $O(n \log^3 n)$ space.

Theorem 1. *There exists a single-pass, $O(n \log^3 n)$-space algorithm for dynamic connectivity.*

4 Bipartiteness

The bipartiteness problem can be reduced to the problem of counting connected components, then we can solve it using Algorithm 1 in dynamic graph streams.

Definition 6. *Given a graph $G = (V, E)$, construct a graph $D(G) = (V', E')$ as follows: for each node $v \in V$, create two nodes v_1, v_2; for each edge $(u, v) \in E$, create two edges (u_1, v_2) and (u_2, v_1).*

Lemma 2. *G is bipartite if and only if $cc(D(G)) = 2 \cdot cc(G)$.*

Proof. Let $G_1, \cdots, G_k$ be the connected components in G. They correspond to $D(G_1), \cdots, D(G_k)$ which are disjoint in $D(G)$. We claim that if G_i is bipartite then $D(G_i)$ has two connected components; otherwise $D(G_i)$ is connected. Then $cc(D(G)) \leq 2 \cdot cc(G)$, with equality holds if and only if every G_i is bipartite, i.e. G is bipartite.

Suppose $G_i = (V_i, E_i)$. Pick any node $u \in V_i$. For each $v \in V_i$, there is a path p from u to v in G_i. In $D(G_i)$, p either connects (u_1, v_1) and (u_2, v_2), or connects (u_1, v_2) and (u_2, v_1), depending on the parity of its length. Hence $D(G_i)$ has at most two connected components, represented by u_1 and u_2.

If there is a an odd cycle c in G_i, let u be a node in c. c will connect u_1 and u_2, thus $D(G_i)$ is connected. Conversely, if u_1 and u_2 are connected in $D(G_i)$, it corresponds to an odd cycle in G_i. □

Theorem 2. *There exists a single-pass, $O(n \log^3 n)$-space algorithm for dynamic bipartiteness.*

5 Streaming Algorithms for Matching

There are efficient streaming algorithms for a variety of problems, and recently a lot of attention has been put into solving the maximum matching problem in a streaming model. A maximum matching in a graph is the largest set of edges without common vertices. In a non-streaming setting, for bipartite graphs it can be reduced to max flow, and for general graphs there is a classic algorithm by Edmonds [2] solving it in polynomial time. We have also seen in class a linear-time 1/2-approximation algorithm by greedily picking edges without common vertices, which finds a maximal matching. This algorithm is easy to implement in a streaming model without edge deletions, by keeping track of the vertices covered by previous selected edges and selecting each new edge in the stream that does not include a covered vertex. However there is no known single pass algorithm that achieves a better approximation than the greedy algorithm, and there are lower bounds [4] showing that one cannot do better than $1-1/e \approx 0.632$ approximation with $\tilde{O}(n)$ space. However when we consider dynamic streams (allowing edge deletions), [6] showed that there are no constant factor approximation algorithms using $\tilde{O}(n)$ space.

We summarize the results about matching in the following table; the results that we will focus on in the next few sections are marked with asterisks (*).

Deletion	Passes	Input	Space	Result
No	Single	Adversarial	$\tilde{O}(n)$	1/2 approximation (best result so far)
Yes	Single	Adversarial	$\tilde{O}(n)$	Impossible
Yes	Multiple	Adversarial	$\tilde{O}(n)$	1/2 approximation*
No	Single	Random	polylog(n)	polylog(n) approximation*

In our survey we will first present a multiple-pass dynamic streaming algorithm [1] (allowing edge insertions and deletions) achieving a constant-factor approximation using $\tilde{O}(n)$ space. This algorithm makes use of the graph sketching tools developed in Section 2. Next we will consider the random arrival model (as opposed to the adversarial streaming model) and present a single-pass streaming algorithm [5] with a polylog(n) approximation factor but only using polylog(n) space.

6 Matching in Adversarial Streams

Similar to the connectivity problem, this algorithm starts with a parallel algorithm to find maximal matchings as described in [7] and show that one can emulate it using linear sketches. We will first describe this algorithm and show how we can turn it into a streaming algorithm. Since we are memory limited, suppose there is memory for only $\eta = \tilde{O}(n)$ edges so we can only do computation on these many edges at a time. This algorithm works by making multiple passes over the stream, sampling a subset of η edges and finds a maximal matching among them. Then we remove all the

vertices included in the the matching and repeat until the remaining graph has less than η edges. Now we can directly run the greedy algorithm. Algorithm 2 describes this process:

Algorithm 2 Iterative sampling algorithm for computing maximal matching

```
Start with graph G = (V, E)
while |E| > η do
    E' ←Sample a subset of η edges, pick each edge with probability p = η/|E|
    M ← Maximal matching in the subgraph formed by E' found by greedy algorithm
    Remove the vertices associated with M from V
    E ← subgraph induced by the remaining vertices
M' ← Maximal matching in remaining graph
return Union of edges in M in each round and M'
```

How many rounds can there be? Suppose the initial graph is a complete graph and at each round we choose edges belonging to a star graph. Then only 2 vertices are removed per round, therefore in this worst case $\Omega(n)$ rounds are needed. However, this is not true in the average case. We can show that in fact with high probability the number of edges is reduced by 1/2 each round, and thus only $O(\log n)$ rounds are needed. This is proved by showing that:

1. If we randomly select each edge with probability $p = \frac{\eta}{|E|}$, then with high probability, any subgraph induced by any set of vertices with more than $|E|/2$ edges has at least one of its edges selected. Proof: For a particular subgraph of more than $|E|/2$ edges, the probability of no edge being selected is at most $(1-p)^{|E|/2} = (1-p)^{\eta/2p} \leq e^{-\eta/2}$. Using a union bound over all subgraphs, the above probability is at least $1 - e^{-n}$ if $\eta \geq 4n$.

2. If an edge is selected by random sampling at the start of a round, after that round it cannot be in the subgraph G'. This is because at least one of its vertices is included in the maximal matching.

Then we can see that with high probability the resulting subgraph has at most $|E|/2$ edges, thus $O(\log n)$ rounds are needed.

How do we modify this algorithm for a streaming setting? Each round in algorithm 2 corresponds to a pass over the stream of edges. After each pass we have η edges in memory, from which we can compute a maximal matching and store its value in a counter. We can also remember the vertices to include in the subgraph in the next round since it only takes $O(n)$ space. All that is left is to sample a subset of η edges at random after each pass on the stream. Define e to be the indicator vector of edges. We maintain η different sketches of e in each pass, and at the end of the pass sample an edge from each sketch for a total of η edges sampled. Here the chance of an edge not being selected is $(1 - 1/|E|)^{\eta} \leq e^{-\eta/|E|} = e^{-p}$, and the rest of the analysis is same as described above. Since each sketch takes $\log^2 n$ space, the total space needed is $\tilde{O}(n)$.

7 Matching in Random Streams

When the edges of a graph are provided in a random stream (which contains each edge of the graph in random order), we can estimate the size of the maximum matching to a polylogarithmic factor

in polylogarithmic space. In this section we will present the work [5], but we will not go into all the details of the proof since it is rather involved and require material outside the scope of this survey.

7.1 Non-streaming algorithm

Let's begin by describing a simple non-streaming algorithm for approximating the size of the maximum matching of a graph. We will then turn it into a streaming algorithm.

7.1.1 A condition for matchings

Let's consider a graph $G = (V, E)$. We'd like to find (approximately) the size of the largest matching in the graph. It is sufficient to check whether there is a matching of size U (up to a polylogarithmic factor, so we say YES if there is such a matching, and NO if the maximum matching is of size, say, $< \frac{U}{\log n}$). Then we can just binary search to find the size of the largest matching.

How do we check if there is a matching of size U? Intuitively, we might hope that if a graph is very dense, it would have a large matching, and vice versa. For example, a complete graph of size n has a matching of size n, the largest possible. On the other hand, a graph with no edges certainly does not have any nonempty matching.

This is a good start, but it's possible that a graph may have many edges, yet they are all concentrated on a few vertices of massive degree, so there is no large matching. For example, if one vertex is connected to all other vertices (a "star" graph), there are $n - 1$ edges, yet the maximum matching is only of size 1.

However, if a graph is both dense, *and* has low degree, then there is a large matching. To be precise:

Lemma 3 (Density Lemma)**.** *If a graph has $\geq dU$ edges, and every vertex has degree at most d, there is a matching of size $\Omega(U)$.*

Why? Let's construct the matching. We simply pick an arbitrary edge and add its two vertices to the matching. We have increased the size of our matching by 1, and decreased the number of edges by at most $2d$ (because each vertex has degree $\leq d$). Repeating this until we run out of edges, we have a matching of size at least $\frac{dU}{2d} = \frac{U}{2}$.

So, given a graph G, we can look for a subgraph G' that is "dense and low degree" in order to find a large matching. The converse is of course also true: if there is a matching of size U in G, it is itself a graph with $\frac{U}{2}$ edges and degree exactly 1, i.e. dense and low degree.

7.1.2 The algorithm

How can we turn the previous observation into an algorithm? After all, we cannot search over all possible subgraphs. Let us think. Since we're only looking for an approximation algorithm, it should be safe to just check some degrees, e.g. graphs of maximum degree n, $\frac{n}{2}$, $\frac{n}{4}$, etc. down to 1. We're looking for a subgraph with bounded degree but many edges. So, let's just throw away any vertices with too high of a degree, and count all the edges among the rest. By doing this iteratively, we can check a series of graphs with lesser and lesser degree, and see if any of them have enough edges.

Concretely, our algorithm is as follows:

Algorithm 3 Checking Matching of size U

Let $G_0 \leftarrow G$
Let $d_1, d_2, d_3, ..., d_k = n, \frac{n}{2}, \frac{n}{4}, ..., 1$, so that $k \approx \log n$
for $i = 1, \cdots, k$ **do**
    Starting with G_{i-1}, discard all vertices with degree $> d_i$, and any edges connected to them
    Let $G_i = (V_i, E_i)$ be the resulting graph
    if $|E_i| \geq U d_i$ **then**
        return YES
return NO

So, we start by seeing if there are at least Un edges in the original graph (which would guarantee a matching of size U). If not, we throw away vertices of degree $> \frac{n}{2}$, and check if there is a matching of at least $\frac{Un}{2}$ edges. We continue on in this way until we are left with vertices of degree at most 1, and check if there are at least U edges. Note that this graph is just some vertices that are matched with each other or disconnected, so the matching is exactly the number of edges. Now let's prove that this works.

Theorem 3. *If Algorithm 3 returns YES, G has a matching of size $\Omega(U)$. Conversely, if a graph has a matching of size M, Algorithm 3 will return YES when asked if G has a matching of size $U = \frac{M}{8 \log k}$.*

Let's do the forward direction first. If the algorithm returns YES, it found a subgraph of degree $\leq d_i$ with Ud_i edges. By Lemma 3, this means there is a matching of size $\Omega(U)$, so we are immediately done.

Now let's do the reverse direction. Suppose that there is a matching M of size U (i.e. with U vertices and $\frac{U}{2}$ edges) in G. The vertices in the matching have various degrees. If almost all of them have low degrees - for example, if the matching M is disconnected from the rest of the graph - the algorithm will not touch them until the very end, and the matching will be plain to see (the graph will be exactly this matching, plus some disconnected vertices). On the other hand, if they have high degrees, the algorithm may discard many vertices of the matching, and thus destroy the matching. But, if the algorithm discards many vertices because they have high degree, this proves that there is a dense subgraph, which again shows that there is a large matching.

To be more precise, if there is a matching of size U, the algorithm will report YES when asked to check for a matching of size $\frac{U}{8k} \approx \frac{U}{8 \log n}$. There are two cases. In the first case, in some iteration i of the algorithm, we discard at least $\frac{U}{4k}$ vertices. Each vertex has degree $> d_i$, and therefore, we throw away at least $\frac{Ud_i}{8k}$ edges (since each edge must have been connected to at least one discarded vertex). In other words, the algorithm will report YES on iteration i. On the other hand, suppose that the first case never happens. Then, after k iterations, we will have discarded at most $\frac{U}{4}$ vertices. Then, we have discarded at most $\frac{U}{4}$ edges of M. Therefore, $\frac{U}{4}$ edges of M remain, and this is enough for the algorithm to return YES on the last iteration. Therefore, whenever there is a matching of size U, the algorithm finds it (up to a log factor).

7.2 Streaming algorithm

To simplify things, let's suppose that instead of a random permutation of the graph's edges, we have a stream in which each edge is a randomly sampled edge from the graph, independent of previous samples.

Now, how do we turn the previous algorithm, which stores the entire graph in memory, into a streaming algorithm that only uses polylogarithmic memory? The previous algorithm just does a series of checks of the form, "Are there many edges in G_i?" The number of edges in G_i corresponds to the probability that a random edge is in G_i. So we get some random edges from the stream, and for each of them, check whether it is in G_i or not.

How do we perform this check? An edge (v, w) is in G_i iff both v and w are in G_i (since, when we discard a vertex, we also discard its attached edges). Thus, the basic operation is to check whether a vertex is in V_i or not.

v is in V_i iff it has not been discarded in any previous V_j. Suppose v survives $V_1, \cdots, V_{j-1}$ (i.e. is not discarded in any of them). Then, it will be discarded in V_j if its degree in V_{j-1} is too large (that is, $> d_j$), so we just need to check its degree in V_{j-1}. We can do this by again randomly sampling some edges, and checking if these are edges connected to v in V_{j-1}. If the fraction of such edges we see is $> \frac{d_j}{m}$, this indicates that v's degree is probably larger than d_j. Note that to perform this check, we look at an edge (v, w), and we need to know if w is also in V_{j-1} (so that the edge has not already been discarded), so we need to make a recursive call.

To sum up, our algorithm is as follows:

Algorithm 4 Check-Level(v, i)

```
1: //Check whether v is in V_i.
2: //If yes, return i. Otherwise return the largest j such that v is in V_j.
3: for j = 1, ..., i do
4:    //We know v is in V_{j-1}, check if it is in V_j
5:    Set count ← 0
6:    for l = 1, ..., R_j do
7:       Sample an edge e
8:       if e is of the form (v, w) and Check-Level(w, j − 1) = j − 1 then
9:          count ← count + 1
10:   //If v's degree in V_{j-1} is too high, it should be discarded
11:   if count/R_j ≥ d_j/m then
12:      return j − 1
13: return i
```

R_j indicates the number of samples required to accurately test for membership in V_j, which we will analyze in the next section. The Check-Level algorithm is clearly polylogarithmic space, because it has a constant number of local variables, and the depth of the stack is at most k, which is logarithmic in n.

7.3 Sampling complexity

If we had an infinite stream of samples, the algorithm above is clearly correct, as it is approximating the same quantities as Algorithm 3, and the approximations will be sufficiently accurate given enough samples. However, we only have m samples. Therefore, we need to figure out how many samples we need to carry out the algorithm.

7.3.1 Chernoff bounds

In order to approximate some quantity, e.g. the degree of a vertex in V_i, it is sufficient to get a constant approximation. This is because in Algorithm 3, even if we discard some vertices at a slightly wrong subgraph (off by one), it will affect the approximation by only a constant factor.

How many edges do we need to sample in order to get this constant approximation? Using a straightforward application of the Chernoff bound we saw in class, we know that it is sufficient for the expected value to be $\Theta(\log n)$. Of course, if we are looking for some rare property, we will need to sample many edges, and if we are looking for some common property, we only need to sample a few edges; but in either case, the threshold that we are checking against (i.e. the expected value) should be logarithmic in n.

7.3.2 Check-Level

In order to check that a vertex v is in V_{i+1}, we need to sample some edges. For each neighbor w that we find, we need to check whether it is in V_i. This in turn is done recursively by sampling edges and checking more neighbors, and so on. It seems that this could consume an exponential number of samples, which is clearly too much. However, in reality, this cannot occur. v's degree in V_i cannot be too large, or else it would have had large degree in V_{i-1}, and it would have already been discarded.

Concretely, let R_{i+1} be the number of samples needed to check if a vertex v is in V_{i+1}. First, we need to test whether v is in V_i, which takes R_i samples. Then, we take $C\frac{m}{d_{i+1}}$ samples of edges, and for each neighbor w, we check whether it is in V_i. In order for v to be in V_i in the first place, it must have had sufficiently small degree in V_0, V_1, ..., V_{i-1}. That is, in graph V_j, it must have had degree at most d_{j+1}. Thus, when we sample our edges, we expect to find at most $C\frac{m}{d_{i+1}} \cdot \frac{d_{j+1}}{m} = C\frac{d_{j+1}}{d_{i+1}}$ edges of this type.

For each neighbor in V_j, we will spend R_j time running Check-Level. Combining this with the maximum number of neighbors in each subgraph, we get the recursive relation

$$R_{i+1} = R_i + C\frac{m}{d_{i+1}} + CR_1\frac{d_1}{d_{i+1}} + ... + CR_i\frac{d_i}{d_{i+1}}$$

Observe that the right-hand side of this has a series of terms $\frac{d_j}{d_{i+1}}$. Since the d_j form a geometric series, if we make the gap between them large enough, we may get a reasonable bound on R_{i+1} (in the local algorithm, the gap was just a constant 2). Indeed, if we set the gap to say, polylogarithmic in n, then induction on R_{i+1} shows that $R_{i+1} = O\left(\frac{m}{d_{i+1}}\right) = O\left(\frac{m}{\text{polylog}(n)}\right)$. Note that it does not cause any problems to set the gap to polylogarithmic; we only need the gap to be at least constant (so k, the number of subgraphs, is logarithmic) and at most polylogarithmic (so the ratio between the d_i, which influences our approximation ratio, is polylogarithmic).

7.3.3 Checking edges

As previously stated, once we know how to run Check-Level, we just run it on both vertices for a random sampling of edges. For each edge (u, v), if it turns out that u is discarded in V_i and v in V_j, we consume $\min(R_i, R_j)$ samples, because we can run the two tests in parallel and terminate once we discard either u or v (which is the point at which we discard (u, v)). Furthermore, once we find a sufficient number of edges in any subgraph, we can terminate. As before, our thresholds for "sufficiently many edges" are only polylogarithmic, so using the $O\left(\frac{m}{\text{polylog}(n)}\right)$ bound from the previous section, we need to use at most m edges.

Finally, as in the previous case, we run this algorithm with geometrically increasing values of U in order to find the maximum matching.

7.4 Random permutations

In reality, we do not have access to a random stream, but a random permutation of the graph's edges. In the permutation, every edge appears exactly once, whereas in a random stream, each new edge is chosen independently of the rest of the stream. However, intuitively, it seems like a random permutation of some items should be fairly "close" to a uniformly random stream of those same items. It can thus be shown that the algorithm still works when the stream is a random permutation.

8 Discussion and Conclusion

From previous sections we have seen two techniques that allows us to turn non-streaming algorithms into streaming algorithms, namely linear sketches of the graph adjacency matrix and treating the stream as random samples of edges. In this section, we will discuss the limitations of each technique, what class of non-streaming algorithms they would work for, as well as some open questions.

Linear sketches enables us to work with dynamic streams, since with linearity we can update each sketch dynamically as edges are inserted and removed. However, this comes with the caveat that we can only use each sketch to sample once otherwise we would lose independence. There is an interesting connection between this and parallel algorithms. The amount of storage space needed is dependent on how many rounds of computation the algorithm needs to perform as we need a separate sketch per round. In the connectivity example, we need to run $\log n$ rounds of merging and sampling, therefore we also need $\log n$ sketches to sample from. We can also think of the connectivity algorithm implemented in the map-reduce model. Each round can be parallelized as a map-reduce step. In fact, the maximal matching algorithm is adapted from a similar algorithm [7] developed for map-reduce. Therefore one interesting extension could be to consider map-reduce graph algorithms and turn them into dynamic streaming algorithms.

Treating the stream as i.i.d. samples of edges allows us to turn local algorithms into a streaming algorithm by estimating values used in the algorithm from random samples, but it works only in the random edge arrival model. In fact, this assumption is crucial as the concentration bounds in the proof depend on the samples being close to uniformly random, otherwise in an adversarial setting concentration may not occur. Property testing algorithms solve a decision problem with a number of queries smaller than the instance size, usually by randomly sampling. This technique

suggests that one can transform property testing algorithms into graph streaming algorithms by treating the stream as random samples from the graph. Another open question is whether there are algorithms for adversarial streams that can achieve polylog approximation to maximum matching in polylog space. It is known that a constant approximation better than 0.632 is not possible in polylog space, but as the authors in [5] noted, current techniques are insufficient in proving this.

Although we presented results for specific problems, we think that the techniques used to turn non-streaming algorithms into streaming ones are quite general in that they can be used for large classes of algorithms, and can potentially be applied to other problems.

References

[1] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[2] J. Edmonds. *Paths, Trees, and Flowers*, pages 361–379. Birkhäuser Boston, Boston, MA, 1987. ISBN 978-0-8176-4842-8. doi: 10.1007/978-0-8176-4842-8_26. URL `http://dx.doi.org/10.1007/978-0-8176-4842-8_26`.

[3] H. Jowhari, M. Sağlam, and G. Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 49–58. ACM, 2011.

[4] M. Kapralov. Improved lower bounds for matchings in the streaming model. *CoRR*, abs/1206.2269, 2012. URL `http://arxiv.org/abs/1206.2269`.

[5] M. Kapralov, S. Khanna, and M. Sudan. Approximating matching size from random streams. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, Proceedings, pages 734–751. Society for Industrial and Applied Mathematics, Dec. 2013. ISBN 978-1-61197-338-9.

[6] C. Konrad. Maximum matching in turnstile streams. *CoRR*, abs/1505.01460, 2015. URL `http://arxiv.org/abs/1505.01460`.

[7] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A Method for Solving Graph Problems in MapReduce. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989505.