# Parallel Simulation of Random Walks in the Semi-Streaming Model

Benjamin W. Priest

May 1, 2019

**Abstract**

The identification of important vertices or edges is a ubiquitous problem in the analysis of graphs. There are many application-dependent measures of importance, such as centrality indices (e.g. degree centrality, closeness centrality, betweenness centrality, and eigencentrality) and local triangle counts, among others. Traditional computational models assume that the entire input fits into working memory, which is impractical for very large graphs. The distributed memory model and streaming model are popular solutions to this problem of scale. In the distributed memory model a collection of processors partition the graph and must optimize communication in addition to execution time. The data stream model assumes only sequential access to the input, which is handled in small chunks. Data stream algorithms use sublinear memory and a small number of passes and seek to optimize update time, query time, and post processing time.

In this dissertation, we consider the application of distributed data stream algorithms to the sublinear approximation of several centrality indices, local triangle counts, and the simulation of random walks. We pay special attention to the recovery of *heavy hitters* - the largest elements relative to the given index.

The first part of this dissertation focuses on serial graph stream algorithms. We present new algorithms providing streaming approximations of degree centrality and a semi-streaming constant-pass approximation of closeness centrality. We achieve our results by way of counting sketches and sampling sketches.

The second part of this dissertation considers vertex-centric distributed graph stream algorithms. We develop hybrid pseudo-asynchronous communication protocols tailored to managing communication on distributed graph algorithms with asymmetric computational loads. We use this protocol as a framework to develop distributed streaming algorithms utilizing cardinality sketches. We present new algorithms for estimating local neighborhood sizes, as well as vertex- and edge-local triangle counts, with special attention paid to heavy hitter recovery. We also utilize reservoir sampling and $\ell_p$ sampling sketches to optimize the semi-streaming simulation of many random walks in parallel in distributed memory. We use these algorithms to approximate $K$-path centrality as a proxy for recovery the top-$k$ betweenness centrality elements.

## 1 Introduction

Many modern computing problems focus on complex relationship networks arising from real-world data. Many of these complex systems such as the Internet, communication networks, logistics and transportation systems, biological systems, epidemiological models, and social relationship networks map naturally onto graphs [WF94]. A natural question that arises in the study of such networks is how to go about identifying "important" vertices and edges. How one might interpret importance within a graph is contingent upon its domain. Accordingly, investigators have devised a large number of importance measures that account for different structural properties. These measures implicitly define an ordering on graphs, and typically only the top elements vis-á-vis the ordering are of analytic interest.

However, most traditional RAM algorithms scale poorly to large datasets. This means that very large graphs tend to confound standard algorithms for computing various important orderings. Newer computational models such as the data stream model and the distributed memory model were introduced to address these scalability concerns. The data steam model assumes only sequential access to the data, and permits a sublinear amount of additional working memory. The time to update, query, and post process this data structure, as well as the number of passes and amount of additional memory are the important resources to optimize in the data stream model. The data stream model is a popular computational model for handling scalability in sequential algorithms. The distributed memory model partitions the data input across several processors, which may need to subsequently communicate with each other. The amount of communication is an important optimization resource. In practical terms, minimizing the amount of time processors spend waiting on their communication partners is also important.

Although both models have been applied to very large graphs independently, there is relatively little literature focusing on the union of the two models of computation. In this work we devise distributed data stream algorithms to approximate orderings of vertices and edges of large graphs. We focus in particular on recovering the heavy hitters of these orderings. We consider the sublinear approximation of classic centrality scores, as well as local and global triangle counts. We also describe space-efficient methods for sampling random walks and subtrees in scale-free vertex-centric distributed graphs, and their application to estimating some centrality indices.

## 2 Data Stream Models

**The data stream model**: A stream $\sigma = \langle a_1, a_2, \ldots, a_m \rangle$ is a sequence of elements in the universe $\mathcal{U}$, $|\mathcal{U}| = n$. We assume throughout that that the hardware has working memory storage capabilities $o(\min\{m, n\})$. We will use the notation $[p] = \{1, 2, \ldots, p-1, p\}$ for $p \in \mathbb{Z}_{>0}$ throughout for compactness. For $t \in [m]$, we will sometimes refer to the state of $\sigma$ after reading $t$ updates as $\sigma(t)$. A streaming algorithm $\mathcal{A}$ accumulates a data structure $\mathcal{S}$ while reading over $\sigma$. We will sometimes use the notation $\mathcal{D}(\sigma)$ to indicate the data structure state after $\mathcal{A}$ has accumulated $\sigma$. Authors generally assume $|\mathcal{D}| = \widetilde{O}(1) = O(\log m + \log n)$, where here the tilde suppresses logarithmic factors. We will also adopt the convention that, except where noted otherwise, we present space complexities in terms of machine words rather than bits. Except where noted otherwise, we will assume the base 2 logarithm in our presentation.

**The semi-streaming model**: Unfortunately, logarithmic memory constraints are not always possible. In particular, it is known that many fundamental properties of complex structured data such matrices and graphs require memory linear in some dimension of the data [M+11, McG09]. In such cases, the logarithmic requirements of streaming algorithms are sometimes relaxed to $O(n \operatorname{polylog} n)$ memory, where $\operatorname{polylog} n = \Theta(\log^c n)$ for some constant $c$. In the case of matrices, here $n$ refers to one of the matrix's dimensions, whereas for graphs $n$ refers to the number of vertices. This is usually known as the *semi-streaming model*, although some authors also use the term to refer to $O(n^{1+\gamma})$ for small $\gamma$ [FKM+05, M+05].

**Distributed sampling of random walks and simple paths via fast $\ell_p$-sampling sketches**: The sampling of random walks is a core subroutine in many graph algorithms. However, random walks suffer from sampling from high degree vertices in scale-free graphs. Very large vertex neighborhoods stored in RAM can overwhelm the space and computation constraints of a graph partitioning in a Pregel-like system, whereas each sampling incurs nontrivial communication overhead in a delegated subpartitioning. We address this problem by applying reservoir samplers and fast $\ell_p$ sampling sketches to high degree vertices in scale free graphs. While the adjacency neighborhoods of most vertices are small enough to be stored explicitly in a vertex-centric distributed graph, we record substreams of high degree vertices in fast memory, e.g. NVRAM. We make the following contributions:

1. **Semi-streaming simulation of $k$ random walks of length $t$ lower bound**. We prove that for $t, k = O(n^2)$, simulating $k$ $t$-step random walks in the insertion-only model within error $\frac{1}{3}$ requries $\Omega(n\sqrt{kt})$ space.

2. **Semi-streaming simulation of $k$ random walks of length $t$ upper bound**. We demonstrate an algorithm that can sample $k$ $t$-step random walks on an undirected graph in the insertion-only model within error $\varepsilon$ using $O\left(n\sqrt{kt}\frac{q}{\log q}\right)$ words of memory, where $q = 2 + \frac{\log(1/\varepsilon)}{\sqrt{kt}}$, with some assumptions placed upon the distribution of the starting vertices.

3. **Hybrid distributed semi-streaming simulation of $k$ random walks of length $t$.** We demonstrate an algorithm framework similar to that undergirding DegreeSketch that allows the easy generalization of the above algorithm to distributed memory. We also describe the method by which compute nodes can store adjacency substreams in fast memory to obtain more samples on the fly, using a methodology we call *playback*. Finally, we describe generalizations of the algorithm framework to

support the simulation of augmented random walks - e.g. random walks that use history to bias future hop probabilities.

## 3 Background and Notation

This chapter introduces the basic concepts and notation that we will use throughout this document. We will lay out the basic graph, probability, and linear algebraic definitions and notation conventions that will be referenced later We also describe centrality indices as a class of functions on graphs. Finally, we describe $k$-universal hash families, an important concept for many sketches, as well as some approximation definitions.

## 4 Graph Definitions and Notation

Throughout this document we will consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{w})$. We assume that $\mathcal{G}$ has no self loops, and that where $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$. For convenience of reference and indexing, we will often assume that $\mathcal{V} = [n]$ and $E = [m]$. We denote an edge connecting $x, y \in \mathcal{V}$ as $xy \in \mathcal{E}$. In general we will assume that $\mathcal{G}$ is an undirected graph, except where noted otherwise. When we want to specify a direction on an edge, we will use the tuple notation $(x, y)$ for $x, y \in \mathcal{V}$. If $\mathcal{G}$ is a weighted graph, then $\mathbf{w} \in \mathbb{R}^{\binom{n}{2}}$ is the vector of edge weights. For $x, y \in \mathcal{V}$, $\mathbf{w}_{xy} \in \mathbb{R}_{\geq 0}$ is the weight associated with the edge $xy$ if $xy \in \mathcal{E}$, and is zero otherwise. If $\mathcal{G}$ is unweighted, then $\mathbf{w}_e = 1$ for every $e \in \mathcal{E}$.

Let $A \in \mathbb{R}^{n \times n}$ be the *adjacency matrix* of $\mathcal{G}$, where $A_{x,y} = \mathbf{w}_{xy}$. We will adopt the convention that, if $\mathcal{G}$ is unweighted, then the columns of $A$ correspond to the out edges whereas the rows of $A$ correspond to the in edges. Hence, $A_{:,x}$ is the out adjacency vector of vertex $x$, and $A_{x,:}$ is its in adjacency vector.

For $\mathcal{G}$ an unweighted graph, let $D \in \mathbb{R}^{n \times n}$ be a diagonal matrix, where $D_{x,x}$ is the *degree* or *valency* of vertex $x \in \mathcal{V}$, which can be computed as the row sum of the $x$th row of $A$. We define $L = D - A$ as the *Laplace Matrix* or *Laplacian* of $\mathcal{G}$.

Consider the signed vertex-edge incidence matrix, $B \in \mathbb{R}^{\binom{n}{2} \times n}$, given by

$$B_{xy,z} = \begin{cases} 1 & \text{if } xy \in \mathcal{E} \text{ and } x = z \\ -1 & \text{if } xy \in \mathcal{E} \text{ and } y = z \\ 0 & \text{else.} \end{cases} \tag{1}$$

Here we let $x$, $y$, and $z$ range over $\mathcal{V}$. Let $W \in \mathbb{R}^{n \times n}$ be a diagonal matrix such that $W_{x,y} = \sqrt{w_{xy}}$. Then if $G$ is undirected, we can alternatively write the Laplacian as

$$L = BWW^T B^T. \tag{2}$$

If $\mathcal{G}$ is unweighted, then we can simply write $L = BB^T$.

A *path* in $\mathcal{G}$ is a series of edges $(x_1x_2, x_2x_3, \ldots, x_{\ell-1}x_\ell)$ where the tail of each edge is the head of the following edge in the path. The length, alternatively weight, of a path is the sum of the weights of all of its edges. If G is unweighted, this is simply the number of edges in the path. We can equivalently identify the path with the series of vertices $(x_1, x_2, \ldots, x_\ell)$, where an edge links each $x_i, x_{i+1}$ pair. For vertices $x, y \in \mathcal{V}$, the *distance* $d_{\mathcal{G}}(x, y)$ between $x$ and $y$ in $\mathcal{G}$ is the length of the shortest path that begins at $x$ and ends with $y$. There may be more than one such path. If there is no path connecting $x$ to $y$ in $\mathcal{G}$, then we say that $d_{\mathcal{G}}(x, y) = \infty$. If the graph is clear from context, we may omit the subscripts and write $d(x, y)$. We call a path *simple* if it visits every vertex no more than once.

## 5 Centrality Indices

A centrality index is any map $\mathcal{C}$ that assigns to every $x \in \mathcal{V}$ a nonnegative score. The particulars of $\mathcal{C}$ are usually assumed to be conditioned only on the structure of $\mathcal{G}$. Consequently, we can identify the centrality index on $\mathcal{G}$ as a function $\mathcal{C}_{\mathcal{G}} : \mathcal{V} \to \mathbb{R}_{\geq 0}$. For $x \in \mathcal{V}$, we will call $\mathcal{C}_{\mathcal{G}}(x)$ the centrality score of $x$ in $\mathcal{G}$. Typically, for $x, y \in \mathcal{V}$, $\mathcal{C}_{\mathcal{G}}(x) > \mathcal{C}_{\mathcal{G}}(y)$ implies that $x$ is more important than $y$ in $\mathcal{G}$ with respect to the property that

$\mathcal{C}$ measures. We will generally drop the subscript from $\mathcal{C}$ when it is clear from context. It is important to note that if $\mathcal{G}$ changes, so may the mapping $\mathcal{C}$. At times, we will write $\mathcal{C}(\mathcal{G})$ or $\mathcal{C}(\mathcal{V})$ to denote the set of all centrality scores of the vertices in $\mathcal{G}$.

Researchers have considered more exotic centrality indices that rely on metadata, such as vertex and edge colorings [KKM+16]. Such notions of centrality are most likely out of scope for the research proposed by this document.

## 6 Vector and Matrix definitions and notation

For a vector $v \in \mathbb{R}^n$, we denote the $\ell_p$ norm as follows:

$$\|v\|_p = \left(\sum_{i=1}^{n} |v_i|^p\right)^{1/p}. \tag{3}$$

As $p \to 0$, this quantity converges to the special case of the $\ell_0$ norm:

$$\|v\|_0 = \sum_{i=1}^{n} v_i^0 = |\{i \in [n] \mid v_i \neq 0\}|. \tag{4}$$

Here we define $0^0 = 0$. Throughout this document we will mostly be concerned with the $\ell_0$ and $\ell_1$ norms of matrix rows and columns. The $p$-th frequency moment $F_p$ of a vector $v$ is related to its $\ell_p$ norm in the following way:

$$F_p(v) = \|v\|_p^p = \sum_{i=1}^{n} v_i^p. \tag{5}$$

We will also sometimes be interested in matrix norms. For $i \in [n]$ and $j \in [m]$, we will write the $i,j$th element of $M$ as $M_{i,j}$. We will also write the $i$th row and $j$th column of $M$ as $M_{i,:}$ and $M_{:,j}$, respectively. For a matrix $M \in \mathbb{R}^{n \times m}$, we define the Fröbenius norm as follows:

$$\|M\|_F = \left(\sum_{i=1}^{n} \sum_{j=1}^{m} M_{i,j}^2\right)^{1/2}. \tag{6}$$

Given $A \in \mathbb{R}^{n \times d}$, let $A = U\Sigma V^T$ be its singular value decomposition (SVD), where $\Sigma \in \mathbb{R}^{n \times n}$ is a diagonal matrix and $U$ and $V$ are orthonormal. Set $A_k = U_k \Sigma_k V_k^T$, where $U_k$ and $V_k$ are the leading $k$ columns of $U$ and $V$, respectively, and $\Sigma_k \in \mathbb{R}^{k \times k}$ is a diagonal matrix whose entries are the first $k$ entries of $\Sigma$. $A_k$ is known to solve the optimization problem

$$\min_{\widetilde{A} \in \mathbb{R}^{n \times d} : \operatorname{rank}(\widetilde{A}) \leq k} \|A - \widetilde{A}\|_F.$$

That is, $A_k$ is the rank-$k$ matrix which has the smallest Fröbenius residual with $A$. This is also true of the spectral norm. We will sometimes denote the rank-$k$ truncated SVD of a product of matrices $A_1 \cdots A_n$ as $[A_1 \cdots A_n]_k$. We use the notation $A^+ = V\Sigma^{-1}U^T$ to denote the Moore-Penrose Pseudoinverse of $A$.

## 7 Sublinear Simulation of Many Random Walks

We described serial algorithms for the sampling of single random walks in Theorems **??** and **??**. We will first discuss a serial algorithm for the simultaneous sampling of $k$ random walks of length $t$, and then discuss a distributed version. Of course, the naïve approach is to simply increase the memory overhead of the algorithms corresponding to Theorem **??** and **??** by a factor of $k$ and simulate the random walks in parallel sparsified subgraphs. However, we will show that it is possible to reduce the dependence on $t$ and $k$ to $O(\sqrt{tk})$ in undirected graphs.

## 7.1 A Lower Bound

First, we show it is not possible to do better than $\sqrt{k}$. This result depends on a reduction from the well-known INDEX problem of communication theory. In the INDEX problem, two participants Alice and Bob communicate to identify the index of vector. Alice is given a vector $X \in \{0,1\}^n$, while Bob is given an index $i \in [n]$. Alice sends a message containing $s$ bits to Bob, who must then output $X_i$. We will require the following lemma:

**Lemma 7.1.** *Solving the* INDEX *problem with probability* $> \frac{1}{2}$ *requires that Alice send* $s = \Omega(n)$ *bits.*

We now prove the corresponding lower bound for the streaming simulation of parallel random walk sumulation, whose proof is inspired by that of Theorem 13 of [Jin18].

**Theorem 7.2.** *For* $t = O(n^2)$ *and* $k = O(n^2)$*, simulating* $k$ $t$*-step random walks on a simple undirected graph in the insertion-only model within error* $\varepsilon = \frac{1}{3}$ *requires* $\Omega(n\sqrt{kt})$ *space.*

*Proof.* We reduce from the INDEX problem. We assume that there is a streaming algorithm $\mathcal{A}$ that can perfectly simulate $k$ random walks on an insert-only graph stream consisting of $O(n)$ vertices from starting vertices $v_0^{(1)}, v_0^{(2)}, \ldots, v_0^{(k)}$. Alice will insert edges into $\mathcal{A}$ and then pass its state to Bob, who will insert more edge, perform the sampling, and approximately solve the INDEX problem.

Alice receives a vector $X = \{0.1\}^{n\sqrt{kt}}$ and encodes it in a graph as follows. Alice and Bob agree upon a graph representation $\mathcal{G} = \mathcal{V}_0 \cup \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{\frac{n}{\sqrt{kt}}}$, where $|\mathcal{V}_j| = 2\sqrt{kt}$ and the $V_j$s are mutually disjoint. For each $j > 0$, $V_j = A_j \cup B_j$, where $|A_j| = |B_j| = \sqrt{kt}$ are disjoint. Note that there are $O(n + \sqrt{kt}) = O(n)$ vertices in $\mathcal{G}$. There are $k$ agreed-up starting vertices $v_0^{(1)}, v_0^{(2)}, \ldots, v_0^{(k)} \in V_0$. Due to the pigeonhole principle some of these vertices collide, but we will show that this is not a problem in the analysis.

Alice divides $X$ up into ranges of size $kt$. For the $j$th such range, she encodes the $kt$ bits into the possible edges between $A_j$ and $B_j$. Note that there are $|A_j||B_j| = kt$ such edges. If a bit is 1, she inserts the corresponding edge into $\mathcal{A}$. In total she so encodes $kt \cdot n/\sqrt{kt} = n\sqrt{kt}$ bits in this way, and then passes the state of $\mathcal{A}$ to Bob.

Bob receives the index $i \in [n\sqrt{kt}]$ and the state of $\mathcal{A}$ so far. $i$ corresponds to the $j$th partition, $\mathcal{V}_j$, for some $j$, and to some particular edge, say $(a,b) \in A_j \times B_j$. Bob inserts every edge in $\mathcal{V}_0 \times A_j$ into $\mathcal{A}$. Bob then queries $\mathcal{A}$ to perform $k$ random walks of length $t^* = O(t)$ to be determined.

Any random walk starting in $\mathcal{V}_0$ will occur inside of the bipartite subgraph $(A_j, \mathcal{V}_0 \cup B_j)$. In particular, every other hop will take a random walk through $A_j$. We will assume that the edge $(a,b)$ exists, i.e. Bob wants to output 1. It is impossible to productively bound the probability of then hopping from some vertex in $A_j$ to $b$ before hopping to $a$. However, we instead bound the probability of hopping to some vertex in $V_0$, then $a$, then $b$. We again assume the $p$th random walk output by $\mathcal{A}$ corresponds to the random variables $\left(v_0^{(p)}, v_1^{(p)}, \ldots v_{t^*}^{(p)}\right)$. Consider,

$$\begin{aligned}
\Pr\left[\left(v_{\ell+2}^{(p)}, v_{\ell+3}^{(p)}\right) = (a,b) \mid v_\ell^{(p)} \in A_j\right] &\leq \Pr\left[v_{\ell+1}^{(p)} \in V_0 \wedge v_{\ell+2}^{(p)} = a \wedge v_{\ell+3}^{(p)} = b \mid v_\ell^{(p)} \in A_j\right] \\
&= \Pr[v_{\ell+1}^{(p)} \in V_0 \mid v_\ell^{(p)} \in A_j] \cdot \Pr[v_{\ell+2}^{(p)} = a \mid v_{\ell+1}^{(p)} \in V_0] \cdot \Pr[v_{\ell+3}^{(p)} = b \mid v_{\ell+2}^{(p)} = a] \\
&\leq \frac{|V_0|}{|V_0| + |B_j|} \cdot \frac{1}{|A_j|} \cdot \frac{1}{|V_0| + |B_j|} \\
&= \frac{2}{9kt}. \qquad (7)
\end{aligned}$$

Thus, in ever four hops on the $p$th walk the edge $(a,b)$ will be passed with probability at least $\frac{2}{9kt}$. Call each of these events where $(a,b)$ is *not* passed a *miss*. Walk $p$ has $\geq \lfloor t^*/4 \rfloor$ opportunities to miss over the course of its simulation. As these walks are independently sampled, this is true of every walk. Say that a walk *fails* if it completes without passing $(a, b)$. Then we have the following:

$$\begin{aligned}
\Pr\left[\text{walker } p \text{ fails} \mid v_0^{(p)} \in V_0\right] &\leq \Pr\left[\text{walker } p \text{ passes at every opportunity} \mid v_0^{(p)} \in V_0\right] \\
&\leq \prod_{s=1}^{t^*} \mathbf{1}_{[s \mod 4=1]} \Pr\left[\text{walker } p \text{ passes} \mid v_s^{(p)} \in A_j\right] \\
&\leq \prod_{s=1}^{t^*} \mathbf{1}_{[s \mod 4=1]} \left(1 - \Pr\left[\left(v_{\ell+2}^{(p)}, v_{\ell+3}^{(p)}\right) = (a, b) \mid v_\ell^{(p)} \in A_j\right]\right) \\
&\leq \left(1 - \frac{2}{9kt}\right)^{\left\lfloor \frac{t^*}{4} \right\rfloor}. && \text{Eq. (7)}
\end{aligned}$$

Note that Bob will only output 0 if all independent walkers fail. We can now bound this probability with

$$\begin{aligned}
\Pr\left[\text{all walkers fail} \mid v_0^{(1)}, v_0^{(2)}, \dots v_0^{(k)} \in V_0\right] &= \prod_{p=1}^{k} \Pr\left[\text{walker } p \text{ fails} \mid v_0^{(p)} \in V_0\right] \\
&\leq \left(1 - \frac{2}{9kt}\right)^{k\left\lfloor \frac{t^*}{4} \right\rfloor}.
\end{aligned}$$

If we choose $t^* = 42t$, we can guarantee that the probability that all walkers fail is $< 0.1$ for all $t$ and $k$. Consequently, Bob is able to use $\mathcal{A}$ to output 1 if $X_i = 1$ with probability $> 0.9$. $\mathcal{A}$ can admit error up to $\varepsilon = \frac{1}{3}$ and maintain $0.9 - \varepsilon > 0.5$. Meanwhile, if $X_i = 0$ Bob will always output 0. Thus, Alice and Bob can solve the Index problem. So, by Lemma 7.1, $\mathcal{A}$ requires $\Omega(n\sqrt{kt})$ memory, and we have the result. □

## 7.2 A Serial Algorithm

We have shown asymptotic bounds on the space performance of multiple random walk simulation algorithms. We will now demonstrate a serial algorithm that nearly meets this bound. The algorithm is inspired by [Jin18], and depends upon the intuition discussed in Section **??**, wherein $O(\sqrt{kt})$ sample neighbors are kept for each vertex and careful attention is paid to accounting how many times some random walk simulation visits vertices not wholly stored in memory. In particular, a straighforward algorithm runs $k$ parallel instances of the single random walk simulation algorithm of [Jin18], maintaining $O(k\sqrt{t})$ sample neighbors for each vertex. We will demonstrate an algorithm that improves upon this performance by a factor of $\sqrt{k}$. Unfortunately, the proof of its correctness depends upon an odious assumption about the $k$ source vertices, which we will discuss below.

We will set a positive threshold integer $c$, to be determined later, and assume that an input graph $\mathcal{G}$ has degree distribution $\mathbf{d}$, where $\mathbf{d}[x]$ is the degree of $x \in \mathcal{V}$. We will notionally separate $\mathcal{V} = \mathcal{B} \cup \mathcal{S}$, where $\mathcal{B} = \{x \in \mathcal{V} \mid \mathbf{d}[x] > c\}$ is the set of *big* vertices and $\mathcal{S} = \{x \in \mathcal{V} \mid \mathbf{d}[x] \leq c\}$ is the set of *small* vertices. A directed edge $(x, y)$ is *important* if $y \in \mathcal{S}$, and *unimportant* otherwise. Let $\mathcal{E}'$ be the set of directed edges corresponding to edges in $\mathcal{E}$. Since $\mathcal{G}$ is undirected, for every $xy \in \mathcal{E}$, $(x, y), (y, x) \in \mathcal{E}'$. Then we can partition $\mathcal{E}' = \mathcal{E}_\mathcal{S} \cup \mathcal{E}_\mathcal{B}$, where $\mathcal{E}_\mathcal{S}$ is the set of important edges and $\mathcal{E}_\mathcal{B}$ is the set of unimportant edges. Note in particular that by definition $|\mathcal{E}_\mathcal{S}| = \sum_{x \in \mathcal{S}} \mathbf{d}[x] \leq |\mathcal{S}|c = O(nc)$. $\mathcal{E}_\mathcal{B}$, on the otherhand, may be quite large.

The core idea of the algorithm is to store $\mathcal{E}_\mathcal{S}$ directly, and sample $O(c)$ unimportant edges incident upon each $x \in \mathcal{V}$ while recording $\mathbf{d}$. Let $\mathcal{N}_\mathcal{S}$ be a dictionary data structure such that for $x \in \mathcal{V}$, $\mathcal{N}_\mathcal{S}[x] = \{(u, v) \in \mathcal{E}_\mathcal{S} \mid u = x\}$. Meanwhile, the sampled unimportant edges are stored in a dictionary data structure $\mathcal{N}_\mathcal{B}$ such that for $x \in \mathcal{V}$, $\mathcal{N}_\mathcal{B}[x]$ is a replacement reservoir sampler over the stream, and after accumulation $\mathcal{N}_\mathcal{B}[x] = \{(u, v) \in \mathcal{E}_\mathcal{B} \mid u = x \wedge (u, v) \text{ is sampled}\}$. While simulating a random hop from $x \in \mathcal{V}$ we toss a coin and with probability $\frac{|\mathcal{N}_\mathcal{S}[x]|}{\mathbf{d}[x]}$ sample from $\mathcal{N}_\mathcal{S}[x]$. We otherwise consume a sample from $\mathcal{N}_\mathcal{B}[x]$, which are sampled with replacement via a scheme like Lemma **??** and so each require at most $O(c)$ words of memory.

We need to maintain $\mathcal{N}_\mathcal{S}$ and $\mathcal{N}_\mathcal{B}$ when $x \in \mathcal{V}$ moves from $\mathcal{S}$ to $\mathcal{B}$ as the algorithm reads the edge list. This is as simple as removing $(y, x)$ from $\mathcal{N}_\mathcal{S}[y]$ for each $y \in \mathcal{V}$, which unfortunately requires a linear scan over $\mathcal{V}$. We can ameliorate this by maintaining a separate dictionary data structure $\mathcal{L}$ so that for $x \in \mathcal{S}$,

$\mathcal{L}[x] = \{y \mid (y,x) \in \mathcal{N}_\mathcal{S}[y]\}$. Thankfully, $|\mathcal{L}[x]| = O(c)$ for each $x \in \mathcal{S}$ by the definition of $\mathcal{S}$, which allows us to perform this procedure with $O(c)$ lookups to $\mathcal{N}_\mathcal{S}^{(O)}$.

Algorithm 1 describes this accumulation procedure in pseudocode. Algorithm 2 describes the simulation procedure. As we have described above, the algorithm flips a coin at each random hop to decide whether to jump to a vertex in $\mathcal{S}$ or $\mathcal{B}$. If a vertex $x$ is receives $> c$ queries to unimportant edges $\mathcal{N}_\mathcal{B}[x]$ during the simulation of a random walk (counting all the queries that occured in earlier walks), that walk simulation terminates with FAIL.

**Algorithm 1** Insert-Only Streaming $k$ Random Walk Accumulation

```
Input: $\sigma$ - insert-only edge stream
Output: $\mathcal{N}_\mathcal{S}$ - dictionary for edges in $\mathcal{E}_\mathcal{S}$
        $\mathcal{N}_\mathcal{B}$ - dictionary for sampled edges in $\mathcal{E}_\mathcal{B}$

    Functions:
 1: function InitVertex($x$)
 2:     if $\exists! \mathbf{d}[x]$ then
 3:         $\mathbf{d}[x] \leftarrow 0$
 4:         $\mathcal{L}[x] \leftarrow \emptyset$
 5:         $\mathcal{N}_\mathcal{S}[x] \leftarrow \emptyset$
 6:         $\mathcal{N}_\mathcal{B}[x] \leftarrow$ empty sampler
 7: function FeedSampler($x, y$)
 8:     if $\mathbf{d}[x] > c$ then
 9:         Feed $(x, y)$ into $\mathcal{N}_\mathcal{B}$
10:     else
11:         $\mathcal{N}_\mathcal{S}[x] \leftarrow \mathcal{N}_\mathcal{S}[x] \cup \{(x,y)\}$
12: function InsertArc($x, y$)
13:     InitVertex($x$), InitVertex($y$)
14:     $\mathbf{d}[y] \leftarrow \mathbf{d}[y] + 1$
15:     if $\mathbf{d}[y] = c + 1$ then
16:         for $u \in \mathcal{L}[y]$ do
17:             $\mathcal{N}_\mathcal{S}[u] \leftarrow \mathcal{N}_\mathcal{S}[u] \setminus (u, y)$
18:             FeedSampler($u, y$)
19:     if $\mathbf{d}[y] \leq c$ then
20:         $\mathcal{N}_\mathcal{S}[x] \leftarrow \mathcal{N}_\mathcal{S}[x] \cup \{(x,y)\}$
21:         $\mathcal{L}[y] \leftarrow \mathcal{L}[y] \cup \{x\}$
22:     else
23:         FeedSampler($x, y$)
    Accumulation:
24: for $xy \in \sigma$ do
25:     InsertArc($x, y$)
26:     InsertArc($y, x$)
27: return $\mathcal{N}_\mathcal{S}$, $\mathcal{N}_\mathcal{B}$
```

A set of $k$ walks $\left(v_0^{(j)}, v_1^{(j)}, \ldots, v_t^{(j)}\right)$, $j \in [k]$, *fails* at vertex $x$ in the $w$th walk if

$$\left|\left\{(i,j) \in [t] \times [w] \mid v_i^{(j)} = x \wedge (v_i^{(j)}, v_{i+1}^{(j)}) \in \mathcal{E}_\mathcal{B}\right\}\right| = c + 1.$$

It is at this point that $\mathcal{N}_\mathcal{B}[x]$ is queried for the $(c+1)$th time, but all of the samples have already been consumed. If no vertex fails, then by the correctness of reservoir sampling the set is returned perfectly, i.e. with probability equal to that of the true distribution. It suffices to show that the algorithm fails with probability at most $\frac{\varepsilon}{2}$, which is achieved by setting the capacity $c$.

**Algorithm 2** Insert-Only Streaming $k$ Random Walk Simulation

```
Input: $\mathcal{N}_{\mathcal{S}}$ - dictionary mapping vertices to outgoing edges in $\mathcal{E}_{\mathcal{S}}$
       $\mathcal{N}_{\mathcal{B}}$ - dictionary mapping vertices to outgoing sampled edges in $\mathcal{E}_{\mathcal{B}}$
       $\mathbf{d}$ - degree dictionary
       $v_0^{(1)}, v_0^{(2)}, \ldots, v_0^{(k)}$ - $k$ starting vertices $\in \mathcal{V}$
Output: $k$ Random Walks (length $t$ or ends in FAIL)

    Functions:
 1: function SimulateRandomWalk($v_0$)
 2:     for $i = 0, 1, \ldots, t-1$ do
 3:         $a \sim_U [\mathbf{d}[v_i]]$
 4:         if $a \leq |\mathcal{N}_{\mathcal{S}}[v_i]|$ then
 5:             $v_{i+1} \sim_U \mathcal{N}_{\mathcal{S}}[v_i]$
 6:         else
 7:             if $|\mathcal{N}_{\mathcal{B}}[v_i]| > 0$ then
 8:                 $v_{i+1} \leftarrow$ next item from $\mathcal{N}_{\mathcal{B}}[v_i]$
 9:                 $\mathcal{N}_{\mathcal{B}}[v_i] \leftarrow \mathcal{N}_{\mathcal{B}}[v_i] \setminus \{v_{i+1}\}$
10:             else
11:                 return $(v_0, v_1, \ldots, v_i)$, FAIL
12:     return $(v_0, v_1, \ldots, v_t)$
    Accumulation:
13: parallel for $j \in [k]$ do
14:     $\left(v_0^{(j)}, v_1^{(j)}, \ldots, v_t^{(j)}\right) \leftarrow$ SimulateRandomWalk $\left(v_0^{(j)}\right)$
15: return $\left(v_0^{(j)}, v_1^{(j)}, \ldots, v_t^{(j)}\right)$ for all $j \in [k]$
```

**Lemma 7.3.** *Suppose for every $x \in \mathcal{V}$, $\Pr\left[x \text{ fails} \mid v_0^{(1)} = x \wedge \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \leq \delta$. Then for any starting vertex $s \in \mathcal{V}$, $\Pr\left[\text{any vertex fails} \mid v_0^{(1)} = s \wedge \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \leq tk\delta$.*

*Proof.* Fix $s \in \mathcal{V}$. As before, say vertex $x$ is *chosen* if $v_i^{(j)} = x$ for some $(i,j) \in [t] \times [k-1]$. For any $x \in \mathcal{V}$,

$$\begin{aligned}
&\Pr\left[x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \\
&\quad = \Pr\left[x \text{ fails and } x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \\
&\quad = \Pr\left[x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right) \text{ and } x \text{ is chosen}\right] \cdot \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \\
&\quad \leq \Pr\left[x \text{ fails} \mid v_0^{(1)} = x, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \cdot \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \\
&\quad \leq \delta \cdot \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right]. \qquad (8)
\end{aligned}$$

We are now able to show that

$$\begin{aligned}
&\Pr\left[\text{a failure occurs} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \\
&\quad \leq \Pr\left[\bigvee_{x \in \mathcal{V}} x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] \\
&\quad \leq \sum_{x \in \mathcal{V}} \Pr\left[x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] && \text{Union bound} \\
&\quad \leq \delta \sum_{x \in \mathcal{V}} \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \dots, v_0^{(k)}\right)\right] && \text{Eq. ??} \\
&\quad \leq \delta kt && \leq kt \text{ vertices chosen.}
\end{aligned}$$

□

We now must show that a bound of the type supposed in Lemma 7.3 exists. We do so by setting $c$ appropriately. The analysis unfortunately depends upon $\mu$, the steady-state distribution of $\mathcal{G}$. $\mu$ corresponds to the left dominant eigenvector of $A$.

**Lemma 7.4.** *There is a parameter $c = O\left(\sqrt{kt} \cdot \frac{q}{\log q}\right)$, where $q = 2 + \frac{\log(1/\delta)}{\sqrt{kt}}$ such that*

$$\Pr\left[x \text{ fails} \mid v_0^{(1)} = x \wedge v_0^{(2)}, \dots, v_0^{(k)} \sim \mu\right] \leq \delta$$

*for all $x \in \mathcal{V}$.*

*Proof.* Assume that $\mathbf{d}_{\mathcal{B}}[x] = |\{y \mid (x,y) \in \mathcal{E}_{\mathcal{B}}\}|$. Furthermore, certainly for any $x \in \mathcal{V}$,

$$\Pr\left[x \text{ fails} \mid v_0^{(1)} = x \wedge v_0^{(2)}, \dots, v_0^{(k)} \sim \mu\right] \leq \Pr\left[x \text{ fails} \mid v_0^{(1)} = x \wedge v_0^{(2)}, \dots, v_0^{(k)} \sim \mu \wedge (v_0, v_1) \in \mathcal{E}_{\mathcal{B}}\right].$$

We can rewrite this probability in terms of the sum of probabilities of all series of random walks in which $x$ fails. Recall that $x$ fails if and only if $\left|\left\{(i,j) \in [0,t-1] \times [k] \mid v_i^{(j)} = x \wedge \left(v_i^{(j)}, v_{i+1}^{(j)}\right) \in \mathcal{E}_{\mathcal{B}}\right\}\right| > c$. We imagine we simulate the random walks in lockstep in reverse order, i.e. we sample $v_1^{(k)}, v_2^{(k-1)}, \dots, v_1^{(1)}$, followed by $v_2^{(k)}, v_2^{(k-1)}, \dots, v_2^{(1)}$, and so on. Assume that $x$ fails on the $\ell$th step of the $w$th walk. When we perform this simulation, we keep only the shortest prefix of each walk sampled at the time that $x$ fails, i.e. we keep $\left(v_0^{(p)}, v_1^{(p)}, \dots, v_\ell^{(p)}\right)$ of the $p$th walk where $p \in [w,k]$, and $\left(v_0^{(p)}, v_1^{(p)}, \dots, v_{\ell-1}^{(p)}\right)$ where $p \in [w-1]$. Specifically, the edge $\left(v_{\ell-1}^{(w)}, v_\ell^{(w)}\right)$ is the $(c+1)$st unimportant edge sampled outgoing from $x$, so $v_{\ell-1}^{(w)} = x$.

In the following, let

$$\Gamma_{i,j}\begin{pmatrix}\left(v_0^{(k)},v_2^{(k)}\ldots,v_i^{(k)}\right),\\ \vdots\\ \left(v_0^{(j)},v_2^{(j)}\ldots,v_{i-1}^{(j)}\right),\\ \vdots\\ \left(v_0^{(1)},v_2^{(1)}\ldots,v_{i-1}^{(1)}\right)\end{pmatrix} = \Pr_{\mu}\left[v_0^{(2)},\ldots,v_0^{(k)}\right]\left(\prod_{i^*=0}^{i-1}\prod_{j^*=1}^{k}\frac{1}{\mathbf{d}\left[v_{i^*}^{(j^*)}\right]}\right)\prod_{j^*=j+1}^{k}\frac{1}{\mathbf{d}\left[v_i^{(j^*)}\right]} \tag{9}$$

be the probability that $v_0^{(2)},\ldots,v_0^{(k)}$ are sampled from $\mu$ and go on to sample the walks $\left(v_0^{(p)},v_1^{(p)}\ldots,v_i^{(p)}\right)$ for $p>j$ and $\left(v_0^{(p)},v_1^{(p)}\ldots,v_{i-1}^{(p)}\right)$ for $p\le j$. We will drop the parameterization in $\Gamma_{i,j}$ below for clarity. Consider the sum of probabilities of such random walks, which is given by

$$\begin{aligned}
&\Pr\left[x\text{ fails}\mid v_0^{(1)}=x\wedge v_0^{(2)},\ldots,v_0^{(k)}\sim\mu\wedge(v_0,v_1)\in\mathcal{E}_{\mathcal{B}}\right]\\
&=\sum_{i=1}^{t}\sum_{j=k}^{1}\sum_{\substack{\left(v_0^{(k)},v_2^{(k)}\ldots,v_i^{(k)}\right)\\ \vdots\\ \left(v_0^{(j)},v_2^{(j)}\ldots,v_i^{(j)}\right)\\ \vdots\\ \left(v_0^{(1)},v_2^{(1)}\ldots,v_{i-1}^{(1)}\right)}}\mathbf{1}\left[\substack{v_0^{(1)}=v_{i-1}^{(j)}=x\wedge\left(v_0^{(1)},v_1^{(1)}\right),\left(v_{i-1}^{(j)},v_i^{(j)}\right)\in\mathcal{E}_{\mathcal{B}}\wedge\\ \left(\left|\left\{(i^*,j^*)\in[0,i-2]\times[k]\mid v_{i^*}^{(j*)}=x\wedge\left(v_{i^*}^{(j^*)},v_{i^*+1}^{(j^*)}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\right.\\ \left.+\left|\left\{j^*\in[j,k]\mid v_i^{(j*)}=x\wedge\left(v_{i-1}^{(j^*)},v_i^{(j^*)}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|=c+1\right)}\right]\frac{1}{\mathbf{d}_{\mathcal{B}}[x]}\Gamma_{i,j}\\
&=\sum_{i=1}^{t}\sum_{j=k}^{1}\sum_{\substack{\left(v_0^{(k)},v_2^{(k)}\ldots,v_i^{(k)}\right)\\ \vdots\\ \left(v_0^{(j)},v_2^{(j)}\ldots,v_{i-1}^{(j)}\right)\\ \vdots\\ \left(v_0^{(1)},v_2^{(1)}\ldots,v_{i-1}^{(1)}\right)}}\mathbf{1}\left[\substack{v_0^{(1)}=v_{i-1}^{(j)}=x\wedge\left(v_0^{(1)},v_1^{(1)}\right)\in\mathcal{E}_{\mathcal{B}}\wedge\\ \left(\left|\left\{(i^*,j^*)\in[0,i-2]\times[k]\mid v_{i^*}^{(j*)}=x\wedge\left(v_{i^*}^{(j^*)},v_{i^*+1}^{(j^*)}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\right.\\ \left.+\left|\left\{j^*\in[j+1,k]\mid v_i^{(j*)}=x\wedge\left(v_{i-1}^{(j^*)},v_i^{(j^*)}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|=c\right)}\right]\Gamma_{i,j}
\end{aligned} \tag{10}$$

Recall that $v_{\ell-1}^{(w)}=x$ is the point at which we assume the simulation fails. In Eq. (10) we threw away the $\ell$th step of the $w$th walk. At this point we have simulated the $k$th through $(w+1)$th walks up to $\ell$ steps, and all other walks up to $\ell-1$ steps. Assume that $v_s^{(p)'}=v_{\ell-s}^{(p)}$ for $p\in[w+1,k]$ and $s\le\ell$. Then the walk $\left(v_0^{(p)'},v_1^{(p)'}\ldots,v_\ell^{(p)'}\right)$ is the reverse of the walk $\left(v_0^{(p)},v_1^{(p)}\ldots,v_\ell^{(p)}\right)$ for such $p$. Similarly assume that $v_s^{(p)'}=v_{\ell-1-s}^{(p)}$ for $p\in[2,w-1]$ and $s\le\ell-1$. Then the walk $\left(v_0^{(p)'},v_1^{(p)'}\ldots,v_{\ell-1}^{(p)'}\right)$ is also the reverse of the walk $\left(v_0^{(p)},v_1^{(p)}\ldots,v_{\ell-1}^{(p)}\right)$. Finally, assume that $v_s^{(1)'}=v_{\ell-1-s}^{(w)}$ and $v_s^{(w)'}=v_{\ell-1-s}^{(1)}$ for $s\le\ell-1$. Then $\left(v_0^{(1)'},v_1^{(1)'}\ldots,v_{\ell-1}^{(1)'}\right)$ is the reverse of $\left(v_0^{(w)},v_1^{(w)}\ldots,v_{\ell-1}^{(w)}\right)$ and $\left(v_0^{(w)'},v_1^{(w)'}\ldots,v_{\ell-1}^{(w)'}\right)$ is the reverse of $\left(v_0^{(1)},v_1^{(1)}\ldots,v_{\ell-1}^{(1)}\right)$. This yields a family of random walks of a the same form as the original walks, as $v_0^{(1)'}=v_{\ell-1}^{(w)'}=x$. Let $\Gamma'_{i,j}$ be defined like Eq. 9, but parameterized by these reversed walks. This allows us to set the summation Eq. (10) equal to

$$\sum_{i=1}^{t}\sum_{j=k}^{1}\sum_{\substack{\left(v_0^{(k)'},v_2^{(k)'}\dots,v_i^{(k)'}\right)\\ \vdots \\ \left(v_0^{(j)'},v_2^{(j)'}\dots,v_{i-1}^{(j)'}\right)\\ \vdots \\ \left(v_0^{(1)'},v_2^{(1)'}\dots,v_{i-1}^{(1)'}\right)}} \mathbf{1}\left[\substack{v_0^{(1)'}=v_{i-1}^{(j)'}=x\wedge\left(v_{i-1}^{(j)'},v_i^{(j)'}\right)\in\mathcal{E}_{\mathcal{B}}\wedge \\ \left(\left|\left\{(i^*,j^*)\in[1,i-1]\times[k]\mid v_{i^*}^{(j*)'}=x\wedge\left(v_{i^*}^{(j^*)'},v_{i^*-1}^{(j^*)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\right. \\ \left.+\left|\left\{j^*\in[j+1,k]\mid v_i^{(j*)}=x\wedge\left(v_i^{(j^*)'},v_{i-1}^{(j^*)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|=c\right)}\right]\Gamma'_{i,j} \tag{11}$$

$$= \Pr_{\substack{\left(v_0^{(1)'},v_2^{(1)'}\dots,v_{t-1}^{(1)'}\right)\\ \vdots \\ \left(v_0^{(k)'},v_2^{(k)'}\dots,v_{t-1}^{(k)'}\right)}}\left[\left|\left\{(i^*,j^*)\in[t-1]\times[k]\mid v_{i^*}^{(j^*)'}=x\wedge v_0^{(2)'},\dots,v_0^{(k)'}\sim\mu\wedge\left(v_{i^*}^{(j^*)'},v_{i^*-1}^{(j^*)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\geq c\right].$$

The last equality follows because we have transformed the summation into the sum of probabilities all sets of $k$ independent random walks that involve transferring from a vertex in $\mathcal{E}_{\mathcal{B}}$ to $x$ at least $c$ times.

Recall that $\left(v_i^{(j)'},v_{i-1}^{(j)'}\right)\in\mathcal{E}_{\mathcal{B}}$ if and only if $v_{i-1}^{(j)'}\in\mathcal{E}_{\mathcal{B}}$. Thus, for any $i\in[1,t-1]$ and $j\in[k]$ with fixed prefix set $\left(v_0^{(j)'},v_2^{(j)'}\dots,v_{i-1}^{(j)'}\right)$, we have that

$$\begin{aligned}\Pr\left[v_i^{(j)'}=x\wedge\left(v_i^{(j)'},v_{i-1}^{(j)'}\right)\in\mathcal{E}_{\mathcal{B}}\mid\left(v_0^{(j)'},v_2^{(j)'}\dots,v_{i-1}^{(j)'}\right)\right]&\leq\mathbf{1}_{\left[v_{i-1}^{(j)'}\in\mathcal{B}\right]}\cdot\frac{1}{\mathbf{d}_{v_{i-1}^{(j)'}}}\\&<\frac{1}{c}.\end{aligned}$$

Moreover, the probability of this event is independent of all of the other walks, and there are $k(t-1)$ opportunities in a particular suite of random walks where it could occur, with at most $c$ occurrences, which could be in any of $\binom{k(t-1)}{c}$ combinations of opportunities. Ergo, we can bound the probability that $x$ fails in this way via

$$\begin{aligned}&\Pr\left[\left|\left\{(i,j)\in[1,t-1]\times[k]\mid v_i^{(j)'}=x\wedge v_0^{(2)'},\dots,v_0^{(k)'}\sim\mu\wedge\left(v_i^{(j)'},v_{i-1}^{(j)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\geq c\right]\\&\quad\leq\Pr\left[\left|\left\{(i,j)\in[1,t-1]\times[k]\mid v_i^{(j)'}=x\wedge\left(v_i^{(j)'},v_{i-1}^{(j)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\geq c\right]\\&\quad\leq\binom{k(t-1)}{c}\left(\frac{1}{c}\right)^c\\&\quad\leq\left(\frac{ek(t-1)}{c}\right)^c\left(\frac{1}{c}\right)^c\\&\quad<\left(\frac{ekt}{c^2}\right)^c.\end{aligned}$$

We can now set $c=\left\lceil 4\sqrt{kt}\frac{q}{\log q}\right\rceil$, where $q=2+\frac{\log(1/\delta)}{\sqrt{kt}}$ is greater than 2. Moreover, note that $\frac{q}{\log^2 q}>\frac{1}{4}$. Then we have that

$$c\log\left(\frac{c^2}{ekt}\right)\geq\frac{4q\sqrt{kt}}{\log q}\log\left(\frac{16q^2}{e\log^2 q}\right)>\frac{4q\sqrt{kt}}{\log q}\log\left(\frac{4q}{e}\right)>4q\sqrt{kt}>\log(1/\delta).$$

Hence, $\left(\frac{ekt}{c^2}\right)^c<\delta$. Thus, we have shown that $\Pr\left[x\text{ fails}\mid v_0^{(1)}=x\wedge v_0^{(2)},v_0^{(3)}\dots,v_0^{(k)}\sim\mu)\right]<\delta$ by setting $c=O\left(\frac{q\sqrt{kt}}{\log(1/\delta)}\right)$.

□

We are now able to prove the correctness of the Algorithm.

**Theorem 7.5.** *There is a randomized algorithm that can simulate $k$ $t$-step random walks where each source is drawn with replacement from $\mu$ in a single pass over an insert-only stream defining an undirected graph within error $\varepsilon$ using $O\left(n\sqrt{kt}\frac{q}{\log q}\right)$ words of memory, where $q = 2 + \frac{\log(1/\varepsilon)}{\sqrt{kt}}$.*

*Proof.* The result follows from Lemma 7.3 and Lemma 7.4, where $\delta = \frac{\varepsilon}{2kt}$. □

Unfortunately this dependence upon $\mu$ is a heavy hammer, as the steady state distribution is expensive to compute. Indeed, random walk simulation is often attempted in applications as a means of estimating $\mu$! Das Sarma et al. demonstrate an algorithm that can sample from $\mu$ using $\widetilde{O}(n + M)$ space and $O(\sqrt{M})$ passes, where $M$ is the mixing time of $\mathcal{G}$ [SGP11]. However, utilizing this method to sample starting vertices for our algorithm is clearly overkill. So, Theorem 7.5 proves an upper bound, but only given a severe restriction. There may be a less odious assumption on the sources of the random walks that would allow us to prove a version of Lemma 7.4, effectively proving the upper bound for a more practically accessible distribution of source vertices.

Jin proved extensions of the single random walk algorithm to handle multigraphs and turnstile streams [Jin18]. The analogous extensions could be made to Algorithms 1 and 2 easily, but their proofs of correctness would still depend upon $\mu$. Thus, an improvement upon the constraints of Lemma 7.4 also translates to analogous results for streaming multigraphs and turnstile graphs.

## 7.3 A Distributed Algorithm

We will describe a distributed generalization of the serial algorithm. We will follow the conventions we have established in earlier chapters, and assume that vertices are partitioned over a universe of processors $\mathcal{P}$ via some unknown parition function $f : \mathcal{V} \to \mathcal{P}$. Like the distributed algorithms in Chapter **??**, we will assume a mailbox abstraction on communication implemented using an asynchronous communication protocol like that described in Chapter **??**. Accordingly, we assume there is a distributed dictionary mapping vertices to send and receive buffers given by $\mathcal{S}$ and $\mathcal{R}$, respectively. We also assume that switching between execution, send, and receive contexts is arbitrary.

We will describe a distributed version of the serial algorithm described in Section 7.3. As the same operations will be performed, Theorem 7.5 will apply to this algorithm as well. We will additionally bound the amount of communication used. It is worth noting that by setting $k = 1$ we will also describe a distributed version of Jin's single-source random walk simulation algorithm.

As in the serial algorithm, we will maintain dictionary data structures $\mathbf{d}$, $\mathcal{N}_\mathcal{S}$, $\mathcal{N}_\mathcal{B}$, and $\mathcal{L}$. Recall that for $x \in \mathcal{V}$, $\mathbf{d}[x]$ is its degree, $\mathcal{N}_\mathcal{S}[x] = \{(u, v) \in \mathcal{E}_\mathcal{S} \mid u = x\}$ is its outgoing important edges to $\mathcal{E}_\mathcal{S}$, $\mathcal{N}_\mathcal{S}[x] = \{(u, v) \in \mathcal{E}_\mathcal{S} \mid u = x\}$ is its unimportant edge sampler, which can eventually query up to $\sqrt{kt}$ edges, and $\mathcal{L}[x] = \{y \mid (y, x) in \mathcal{N}_\mathcal{S}[x]\}$ is its lookup to neighbors owning important edges. For each dictionary, we will assemble their values relative to $x \in \mathcal{V}$ locally on its owner processor $f(x) \in \mathcal{P}$.

Algorithm 3 describes the procedure of accumulating these data structures in a pass over a partitioned stream $\sigma$. It is similar to Algorithm 1, except that the endpoints of a read edge $(x, y)$, might be owned by different processors, say $X$ and $Y$. Accordingly, the function InsertArc$(x, y)$ is split, where $Y$ determines $\mathbf{d}[y]$ and then sends a message to $X$, which determines how it will update the data structures relative to $x$. Each edge thus generates at most 4 messages of fixed size, so $O(m)$ communication is used.

Algorithm 4 describes the procedure of simulating $k$ random walks once $\mathbf{d}$, $\mathcal{N}_\mathcal{S}$, and $\mathcal{N}_\mathcal{B}$ are accumulated. It is like a mobile version of Algorithm 2 because each random walker must traverse $\mathcal{P}$ as it hops from vertex to vertex.

Each of the $k$ random walks generates at most $t$ messages. The first message contains 1 vertex, the second 2 vertices, until the final message which contains $t - 1$ vertices, assuming that no failures occur. The total words communicated then is the sum of a simple arithmetic series and uses $O(t^2)$ communication. Thus, $O(kt^2)$ communication is used overall.

Algorithms 3 and 4 generalize the $k$ random walk simulation algorithm to a distributed algorithm. Again, if $k = 1$ the algorithm generalizes the insert-only algorithm of [Jin18]. Furthermore, a distribution of this type (where $k = 1$ and the appropriate changes are made) also serves to generalize the simulation of a single random walk on multigraphs and turnstile streams. Furthermore, $k$ instances of these algorithms can run distributed in parallel, affording the sampling of $k$ independent random walks. However, such an algorithm

**Algorithm 3** Insert-Only Streaming Distributed $k$ Random Walk Accumulation

```
Input: $\boldsymbol{\sigma}$ - insert-only edge stream
       $\mathcal{P}$ - universe of processors
       $\mathcal{S}$ - distributed dictionary mapping $\mathcal{P}$ to send queues
       $\mathcal{R}$ - distributed dictionary mapping $\mathcal{P}$ to receive queues
       $f$ - function mapping $\mathcal{V} \to \mathcal{P}$
Output: $\mathcal{N}_\mathcal{S}$ - distributed dictionary of edges in $\mathcal{E}_\mathcal{S}$
        $\mathcal{N}_\mathcal{B}$ - distributed dictionary of sampled edges in $\mathcal{E}_\mathcal{B}$
        $\mathbf{d}$ - distributed dictionary of degrees

    Functions:
 1: function InitVertex($x$)
 2:     if $\exists ! \mathbf{d}[x]$ then
 3:         $\mathbf{d}[x] \leftarrow 0$
 4:         $\mathcal{L}[x] \leftarrow \emptyset$
 5:         $\mathcal{N}_\mathcal{S}[x] \leftarrow \emptyset$
 6:         $\mathcal{N}_\mathcal{B}[x] \leftarrow$ empty sampler
 7:     function FeedSampler($x, y$)
 8:         if $\mathbf{d}[x] > c$ then
 9:             Feed $(x, y)$ into $\mathcal{N}_\mathcal{B}$
10:         else
11:             $\mathcal{N}_\mathcal{S}[x] \leftarrow \mathcal{N}_\mathcal{S}[x] \cup \{(x, y)\}$
    Send Context $P \in \mathcal{P}$:
12: while $\mathcal{S}[P]$ is not empty do
13:     $(\xi, (x, y)) \leftarrow \mathcal{S}[P]$.pop()
14:     if $\xi =$ Edge then
15:         $W \leftarrow f(y)$
16:     else
17:         $W \leftarrow f(x)$
18:     $\mathcal{R}[W]$.push $(\xi, (x, y))$
    Receive Context $P \in \mathcal{P}$:
19: while $\mathcal{R}[P]$ is not empty do
20:     $(\xi, (x, y)) \leftarrow \mathcal{R}[P]$.pop()
21:     if $\xi =$ Edge then
22:         InitVertex($y$)
23:         $\mathbf{d}[y] \leftarrow \mathbf{d}[y] + 1$
24:         if $\mathbf{d}[y] = c + 1$ then
25:             for $u \in \mathcal{L}[y]$ do
26:                 $\mathcal{S}[P]$.push (Promote, $(u, y)$)
27:         if $\mathbf{d}[y] \leq c$ then
28:             $\mathcal{S}[P]$.push (Small, $(x, y)$)
29:             $\mathcal{L}[y] \leftarrow \mathcal{L}[y] \cup \{x\}$
30:         else
31:             $\mathcal{S}[P]$.push (Big, $(x, y)$)
32:     else if $\xi =$ Promote then
33:         $\mathcal{N}_\mathcal{S}[x] \leftarrow \mathcal{N}_\mathcal{S}[x] \setminus (x, y)$
34:         FeedSampler($x, y$)
35:     else if $\xi =$ Small then
36:         $\mathcal{N}_\mathcal{S}[x] \leftarrow \mathcal{N}_\mathcal{S}[x] \cup \{(x, y)\}$
37:     else if $\xi =$ Big then
38:         FeedSampler($x, y$)
    Accumulation $P \in \mathcal{P}$:
39: for $xy \in \boldsymbol{\sigma}_P$ do
40:     $\mathcal{S}[P].push$(Edge, $(x, y)$)
41:     $\mathcal{S}[P].push$(Edge, $(y, x)$)
42: return $\mathcal{N}_\mathcal{S}$, $\mathcal{N}_\mathcal{B}$
```

**Algorithm 4** Insert-Only Streaming Distributed $k$ Random Walk Simulation

**Input:** $\mathcal{N}_{\mathcal{S}}$ - dictionary mapping vertices to outgoing edges in $\mathcal{E}_{\mathcal{S}}$
$\mathcal{N}_{\mathcal{B}}$ - dictionary mapping vertices to outgoing sampled edges in $\mathcal{E}_{\mathcal{B}}$
$\mathbf{d}$ - degree dictionary
$v_0^{(1)}, v_0^{(2)}, \ldots, v_0^{(k)}$ - $k$ starting vertices $\in \mathcal{V}$
**Output:** $k$ Random Walks (length $t$ or ends in FAIL)

```
    Send Context $P \in \mathcal{P}$:
 1: while $\mathcal{R}[P]$ is not empty do
 2:     $(v_0, v_1, \ldots, v_j) \leftarrow \mathcal{R}[P]$.pop()
 3:     $Q \leftarrow f(v_j)$
 4:     $R[Q]$.push$(v_0, v_1, \ldots, v_j)$
    Receive Context $P \in \mathcal{P}$:
 5: while $\mathcal{R}[P]$ is not empty do
 6:     $(v_0, v_1, \ldots, v_i) \leftarrow \mathcal{R}[P]$.pop()
 7:     $a \sim_U [\mathbf{d}[v_i]]$
 8:     if $a \leq |\mathcal{N}_{\mathcal{S}}[v_i]|$ then
 9:         $v_{i+1} \sim_U \mathcal{N}_{\mathcal{S}}[v_i]$
10:     else
11:         if $|\mathcal{N}_{\mathcal{B}}[v_i]| > 0$ then
12:             $v_{i+1} \leftarrow$ next item from $\mathcal{N}_{\mathcal{B}}[v_i]$
13:             $\mathcal{N}_{\mathcal{B}}[v_i] \leftarrow \mathcal{N}_{\mathcal{B}}[v_i] \setminus \{v_{i+1}\}$
14:         else
15:             return $(v_0, v_1, \ldots, v_i)$, FAIL
16:     if $i + 1 = t$ then
17:         return $(v_0, v_1, \ldots, v_{i+1})$
18:     else
19:         $S[P]$.push$(v_0, v_1, \ldots, v_{i+1})$
    Execution $P \in \mathcal{P}$:
20: parallel for $j \in [k]$ do
21:     if $f\left(v_0^{(j)}\right) = P$ then
22:         $\mathcal{R}[P]$.push$\left(v_0^{(j)}\right)$
```

uses $O\left(nk\sqrt{t}\frac{q'}{\log q'}\right)$ words of memory, where $q' = 2 + \frac{\log(1/\varepsilon)}{\sqrt{t}}$. This is why we want a better version of Lemma 7.4.

## 7.4 A Distributed Algorithm with Playback

We have so far discussed single-pass algorithms for sublinearly simulating random walks. A trivial $t$-pass algorithm certainly exists, where one maintains $k$ reservoir samplers with one element. Starting from a set of $k$ sources, each sampler samples from its source's neighbors in each pass. This allows us to iteratively simulate $k$ independent random walks. Moreover, since we know from which neighborhoods we should sample, there is no error and we use only $O(k)$ words of memory. There is an obvious tradeoff between memory and passes at play.

For general purpose serial algorithms, one pass is generally preferable. However, consider the scenario where one might want to simulate a *great* number of random walks, but possibly not all at once. It would be quite useful to remember $\mathcal{N}_{\mathcal{E}}$ at least, while retaining the ability to refresh $\mathcal{N}_{\mathcal{B}}$ when necessary.

While rather awkward in serial, the distributed model makes this approach not only sensible but rather convenient. Assume that every processor has access to some fast long-term memory bank $\mathcal{M}$- e.g. NVRAM, to which it can write and read. Each processor might have its own memory bank, or there may be many that partition $\mathcal{P}$. In either case $\mathcal{M}$ refers to a distributed data structure in the convention we have used throughout this document. Assume that $P \in \mathcal{P}$ can allocate a portion of its memory bank, $\mathcal{M}_P[x]$, allocated to $x \in \mathcal{V}_P$.

Algorithm 5 generalizes the FeedSampler function of Algorithm 3 to include playback. The other contexts of the accumulation procedure are unchanged. While processing an insert $(x, y)$ to $\mathcal{N}_{\mathcal{E}}[x]$ for some $x \in \mathcal{B}$ that it owns, $P$ also writes $(x, y)$ to $M[x]$. After accumulation is finished, $\mathcal{M}[x] = \{(u, v) \in \mathcal{E}_{\mathcal{B}} \mid u = x\}$ is a stream recorded in fast storage, but not held in working memory.

**Algorithm 5** Insert-Only Streaming Distributed $k$ Random Walk Accumulation with Playback

**Input:** $\mathcal{M}$ - distributed dictionary mapping $\mathcal{P}$ to memory banks
**Output:** $\mathcal{N}_{\mathcal{S}}$ - distributed dictionary of edges in $\mathcal{E}_{\mathcal{S}}$
$\mathcal{N}_{\mathcal{B}}$ - distributed dictionary of sampled edges in $\mathcal{E}_{\mathcal{B}}$
**d** - distributed dictionary of degrees

```
Functions:
1: function FeedSampler($(x, y)$)
2:     if $\mathbf{d}[x] > c$ then
3:         Feed $(x, y)$ into $\mathcal{N}_{\mathcal{B}}$
4:         Append $(x, y)$ to $\mathcal{M}[x]$
5:     else
6:         $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \mathcal{N}_{\mathcal{S}}[x] \cup \{(x, y)\}$
```

Say that $\mathcal{N}_{\mathcal{B}}[x]$ runs out of samples during the simulation phase. Rather than outputting FAIL, the algorithm can instead refresh it by taking another pass over $\mathcal{M}[x]$. This avoids both taking another pass over all of $\boldsymbol{\sigma}$ and outputting FAIL. Algorithm 6 generalizes the receive context of Algorithm 4 with this behavior.

Say, for example, that we run this algorithm where we sample $O(k^\alpha\sqrt{t})$ neighbors per vertex, where $\alpha \in [0, 1]$ is a real number. However, where simulation would have resulted in FAIL, we now take another pass over the relevant substream and record a new set of $O(k^\alpha\sqrt{t})$ neighbors for the offending vertex. We will call such an event a *playback*.

We can bound the number of playbacks that are likely to occur. Consider a particular vertex $x \in \mathcal{V}$, and assume that while simulating $k$ random walks of length $t$, the algorithm triggers $\omega\left(k^{1-\alpha}\right)$ playbacks on $x$. Then the algorithm considers $\omega\left(k\sqrt{t}\right)$ samples from neighbors of $x$ in $\mathcal{B}$. This means that there is at least one simulated random walk $w$ that consumes $\omega(\sqrt{t})$ of these samples. As there are no failures by design, these random walks are perfectly simulated. That means that, should that $w$ have been simulated

**Algorithm 6** Insert-Only Streaming Distributed $k$ Random Walk Simulation

```
Input: $\mathcal{M}$ - dictionary mapping vertices to external incident edge streams in $\mathcal{E}_\mathcal{B}$
       $\mathcal{N}_\mathcal{S}$ - dictionary mapping vertices to outgoing edges in $\mathcal{E}_\mathcal{S}$
       $\mathcal{N}_\mathcal{B}$ - dictionary mapping vertices to outgoing sampled edges in $\mathcal{E}_\mathcal{B}$
       $\mathbf{d}$ - degree dictionary
Output: $k$ Random Walks (length $t$ or ends in FAIL)

  Receive Context $P \in \mathcal{P}$:
 1: while $\mathcal{R}[P]$ is not empty do
 2:     $(v_0, v_1, \ldots, v_i) \leftarrow \mathcal{R}[P]$.pop()
 3:     $a \sim_U [\mathbf{d}[v_i]]$
 4:     if $a \leq |\mathcal{N}_\mathcal{S}[v_i]|$ then
 5:         $v_{i+1} \sim_U \mathcal{N}_\mathcal{S}[v_i]$
 6:     else
 7:         if $|\mathcal{N}_\mathcal{B}[v_i]| = 0$ then
 8:             Refresh $\mathcal{N}_\mathcal{B}[v_i]$ by taking a pass over $\mathcal{M}[x]$
 9:         $v_{i+1} \leftarrow$ next item from $\mathcal{N}_\mathcal{B}[v_i]$
10:         $\mathcal{N}_\mathcal{B}[v_i] \leftarrow \mathcal{N}_\mathcal{B}[v_i] \setminus \{v_{i+1}\}$
11:     if $i + 1 = t$ then
12:         return $(v_0, v_1, \ldots, v_{i+1})$
13:     else
14:         $S[P]$.push$(v_0, v_1, \ldots, v_{i+1})$
```

by the single source algorithm using the same bits of randomness, it would have failed. We have shown that $\omega(k^{1-\alpha})$ playbacks occurring on one vertex implies that some walk is simulated that would have failed using the single source algorithm. Thus, the probability that $\omega(k^{\alpha})$ playbacks occur on a single vertex is bounded by the probability that one of $k$ independent single source instantiations of the algorithm with the same starting vertices fails. We have proven the following Lemma.

**Theorem 7.6.** *Let $\delta$ be the probability that a single source streaming random walk simulation of length $t$ fails given a parameterization. Further assume that the distributed $k$-source streaming algorithm with playback using the same parameterization accumulates $O(k^{\alpha}\sqrt{t})$ neighbors per vertex. Then the distributed algorithm will generate $O(k^{1-\alpha})$ playbacks on each vertex with probability at least $(1-\delta)^k$.*

This means that we are unlikely to take too many passes over the memory banks, which has the added benefit of allowing us to partially skirt the limitations of Lemma 7.4 in the distributed case. In particular, if $\alpha = \frac{1}{2}$ we will accumulate $O(\sqrt{kt})$ neighbors per vertex per pass. It is likely that we will take $O(\sqrt{k})$ passes over the unimportant edges $\mathcal{E}_\mathcal{B}$, while using $O\left(n\sqrt{kt}\frac{q}{\log q}\right)$ words of memory. Meanwhile, using $k$ single source simulators in parallel requires a single pass over all of $\boldsymbol{\sigma}$ using $O\left(nk\sqrt{t}\frac{q'}{\log q'}\right)$ words of memory, where $q' = 2 + \frac{\log 1/\varepsilon}{\sqrt{t}}$. Further, using a single source simulator in a series requires $\Theta(k)$ passes over all of $\boldsymbol{\sigma}$ using $O\left(n\sqrt{t}\frac{q'}{\log q'}\right)$ words of memory. So, we are able to provide a middle ground in terms of memory and pass efficiency.

Say that in a particular simulation, the most active vertex triggers $O(k^{\alpha})$ playbacks. For practical graphs, it is likely that most of the other vertices will trigger $o(k^{\alpha})$ playbacks, which indicates much less time spent in I/O and communication than taking $\Theta(k^{\alpha})$ passes over $\boldsymbol{\sigma}$.

## 7.5 Simulating Augmented Random Walks

Many applications call for the simulation of random sequences of vertices that are generalizations of random walks. For example, one might want to prefer to follow edges to vertices two hops in the past so as to bias toward closing triangles. Alternatively, one might want to avoid vertices visited up to a certain number of hops in the past so as to bias toward exploration. One might want to include a probability of hopping back to a previously visited vertex, e.g. restarting from the source. Furthermore, any of these augmentations might want to be biases with respect to the length of the walk thus far, e.g. the probability of return-to-source grows as the length of the walk increases.

For the purposes of our analysis in Section **??**, we will focus on only one of these cases, namely history-avoiding random walks. In the unitary case of sampling from a stream while avoiding a subset of possible items, the solution is as simple as ignoring stream indices that match the list of forbidden items. This approach works for insert-only, weighted, and turnstile streams.

However, the sampling of full random walks is more involved. The histories to be avoided are not known ahead of time. The simplest serial algorithm is to sample from $v_0$'s adjacency set in one pass, and in subsequent passes to sample from $v_i$'s adjacency set while avoiding edges returning to $\{v_0, v_1, \ldots, v_{i-1}\}$. However, this requires $t$ passes over the entire graph. This approach can be made somewhat less wasteful via the simulation of $k$ random walks in parallel, as the space complexity and update times increase by a factor of $k$ and the pass complexity remains the same.

One might be tempted, for unweighted simple graphs, to simply sample many vertices ahead of time and upon simulation ignore the samples that match a history to be ignored. However, we may not know which, if any of $\{v_0, v_1, \ldots, v_{i-1}\}$ are actually adjacent to $v_i$. If $\{v_0, v_1, \ldots, v_{i-1}\} \not\subseteq \mathcal{N}_{\mathcal{S}}[v_i]$, then we are unable to flip a coin with probability $\frac{|\mathcal{N}_{\mathcal{S}}[v_i] \setminus \{v_0, v_1, \ldots, v_{i-1}\}|}{|\{x \in \mathcal{B} | (v_i, x) \in \mathcal{E}_{\mathcal{B}}\} \setminus \{v_0, v_1, \ldots, v_{i-1}\}|}$, because we do not know the size of the denominator without taking another pass over $M[v_i] = \{x \in \mathcal{B} \mid (v_i, x) \in \mathcal{E}_{\mathcal{B}}\}$. Hence, even for unweighted simple graphs, we must pass over $M[x]$ each time we sample from non-source $x$.

History-avoiding random walks also introduce a second notion of failure, where a simulation writes itself into a proverbial corner and finds no valid options for the next hop. We will call such an event a *dead end*. Note, however, that near the end of a single source random walk simulation $O(t)$ potential vertices must be avoided in order to avoid a failure. It is not at all a safe assumption that every vertex have $\omega(t)$ degree in most practical graphs for nontrivial $t$. Indeed, it is impossible to guarantee that dead ends do not occur with bounded probability in the simulation of history-avoiding random walks.

However, a high-performance computing approach with playback of the sort described in Section 7.4 provides a possible way forward. Simulating $k$ parallel history-avoiding random walks can then proceed, where for each history $(v_0, v_1, \ldots, v_i)$ the processor in question takes another pass over $M[v_i]$, avoiding any neighbors in $\{v_0, v_1, \ldots, v_{i-1}\}$. Only in the situation where $\{v_0, v_1, \ldots, v_{i-1}\} \subseteq \mathcal{N}_{\mathcal{B}}[v_i]$ and the coin flip determines sampling from $\mathcal{N}_{\mathcal{B}}$ can a simulation avoid executing this playback. In the worst case, certainly no more than $O(kt)$ playbacks will occur for a particular vertex. There may be strategies for cleverly batching playbacks for multiple samples from the same vertex, as $O(k)$ different random walks are being simulated in parallel and might all visit some $x \in \mathcal{V}$. However, there is no way to guarantee that these visits happen close together in time, short of implementing a Pregel-like synchronous communication protocol. Unfortunately, it is then much more difficult to assign a better bound on the number of playbacks that will occur for the simulation of history-avoiding random walks in general. Furthermore, each simulation now can terminate in a dead end, the probability of which is highly dependent upon graph structure and independent of the sublinear approximation scheme.

## References

[FKM+05] 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.

[Jin18] Ce Jin. Simulating random walks on graphs in the streaming model. *arXiv preprint arXiv:1811.08205*, 2018.

[KKM+16] Chanhyun Kang, Sarit Kraus, Cristian Molinaro, Francesca Spezzano, and VS Subrahmanian. Diffusion centrality: A paradigm to maximize spread in social networks. *Artificial Intelligence*, 239:70–96, 2016.

[M$^+$05] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[M$^+$11] Michael W Mahoney et al. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

[McG09] Andrew McGregor. Graph mining on streams. In *Encyclopedia of Database Systems*, pages 1271–1275. Springer, 2009.

[SGP11] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM (JACM)*, 58(3):13, 2011.

[WF94] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.